

StockSharp Documentation

DE — offline documentation

About StockSharp

About StockSharp

StockSharp (S#) offers **free** trading programs for global markets (American, European, Asian, Russian, stocks, futures, options, Bitcoins, forex, etc.). Users can trade manually or through automated trading, including algorithmic trading robots, conventional or HFT (High-Frequency Trading).

Supports 90+ brokers, exchanges, and data sources: [Connectors](#).

S# is compatible with any broker that supports our connections.

| The installation of **all** programs is uniformly performed through the [Installer](#) utility.

Designer

[image: StockSharpTitle 0]

[Designer](#) is a universal algorithmic strategies application that simplifies strategy creation:

- Visual designer for strategy creation using mouse clicks.
- Integrated [C#](#) editor.
- Simple creation of custom indicators.
- Built-in debugger.
- Connection capabilities to multiple electronic boards and brokers.
- Compatibility with all world platforms.
- Ability to share schemas with your team.

[More...](#)

Hydra

[image: StockSharpTitle 1]

[Hydra](#) is an application for the automatic downloading of historical and real-time market data:

- Supports numerous sources [Connectors](#).
- High compression ratio (2 bytes per trade, 7 bytes per order book).
- Handles any data type (candles, ticks, order books, order-logs, options, news, and more).
- API access to stored data.
- Export capabilities to CSV, Excel, XML, or databases.
- CSV import functionality.
- Scheduled tasks and auto-sync over the Internet between running Hydra instances.

[More...](#)

Terminal

[image: Terminal main 00]

[Terminal](#) is a trading and charting application (trading terminal):

- Enables trading directly from charts with a click.
- Supports arbitrary timeframes.
- Features various candle types: Volume, Tick, Range, Renko.
- Includes cluster and box charts.

Shell

The Shell provides a ready-made graphical framework that can be quickly customized to your needs and comes with fully open source code in C#:

- Complete source code included.
- Supports all StockSharp platform connections: FIX/FAST, Crypto Exchanges (+30 currently), etc.
- Designer schema support.
- Flexible user interface.
- Strategy testing tools (statistics, equity, reports).
- Save and load strategy settings.
- Concurrent strategy execution.
- Detailed strategy performance insights (orders, transactions, positions, revenue, logs, etc.).
- Scheduled strategy launches.

API

[API](#) is a library for the professional development of trading robots in C#. It is designed for those who are programming in Visual Studio and are professional programmers in algorithmic trading.

Our Products:

- [Designer](#) - Universal algorithmic strategy designer.
- [Hydra](#) - Market data downloading program.
- [API](#) - Library for developing trading robots in [C#](#).
- [Terminal](#) - Trading terminal.
- [Shell](#) - Ready-made graphical framework for strategies with source codes.
- [MATLAB](#) - Integration of MATLAB with trading systems. Trade from MATLAB scripts.

[Download](#)

Recommended Content

[Reference Materials](#)

Installer

Installer

Installer - is the control center for all programs and applications from **StockSharp**, allowing you to:

- Download and install [Designer](#), [Terminal](#), [Hydra](#)
- Install [MetaTrader](#) plugin-connectors
- Purchase and install paid components such as [Shell](#), [MATLAB](#)
- Purchase and unpack source codes for programs like [Designer](#).
- Install programs together with purchased connectors
- Monitor the program update process
- Download the [API](#) library and automatically track its updates

[image: first win installer]

Recommended content

[Setup](#)

Setup

Setup

Install and remove apps

Install and remove apps

Update apps

Update apps

Setup

First Run

1. To install [Installer](#), go to the [Download](#) page:
[image: Installer installation]
2. Download the [Installer](#) distribution.
3. Run the installation file **stocksharp_setup.exe** and follow the installer instructions.
4. Sometimes, Windows does not start the installation immediately and displays a warning:
[image: Installerzip Properties]
5. In this case, click on the **More info** link in the warning window, after which the following window will appear:
[image: Installerzip Properties]
Clicking the **Run anyway** button starts the installation of [Installer](#).
6. [Installer](#) will then unpack. Wait for the process to complete.
7. During the first launch you need to enter your **StockSharp** login and password.
[image: log In installer]
You can sign in either by entering your credentials directly or via social network authorization. If you registered on the StockSharp website through a social network, you will not have a password.
8. After logging in, the program window will open: *[image: first win installer]*

Watch the [video tutorial](#)

See also

[Installing and Removing Programs](#)

Install and remove apps

Install and remove apps

The [Installer](#) program allows you to select the application type to find the product you need.

[image: choose installer]

To install the required application, you need:

1. Select an application, click **Install**, agree to the license agreement and click **Continue**.*[image: download installer]*
2. After that, you need to select the installation path.
IMPORTANT! The folder into which the program will be installed must be empty.
Click **Continue**.*[image: direction installer]*
3. Select **Run** and wait for the installation to complete.

After the installation is complete, the program is ready to use.

To uninstall the program, select **Uninstall** and click the **Continue** button.

[image: del installer]

To restore, select **Restore** and click **Continue**.

[image: repair installer]

Watch [video tutorial](#)

Recommended content

[Update apps](#)

Update apps

Update apps

[Installer](#) keeps track of all software updates and updates itself automatically. Therefore, there is no need to uninstall it after installation.

To check for available updates manually, you should click on the **Updates** button in the right corner of the program window.

[image: force update installer]

If updates are available for the program, [Installer](#) will notify you accordingly.

Then you need to click the button.

[image: update installer]

[Installer](#) is not closed by clicking on the "X" in the program window, but is closed using the toolbar.

To do this, go to the menu by right-clicking and click **Close**.

Watch [video tutorial](#).

Recommended content

[Install and remove apps](#)

Console installer

The `Installer.Console` application is a cross-platform version of StockSharp Installer. It allows you to download, update and remove products without using the graphical interface. The tool runs on any OS where the [.NET 6](#) runtime is available.

Running

1. Install the .NET 6 SDK or runtime for your platform.
2. Download `StockSharp.Installer.Console.zip` from the [Download page](#).
3. Unpack the archive and run the utility from the command line:

```
dotnet StockSharp.Installer.Console.dll <Command> [product] [dir] [options]
```

<Command> is one of the following:

- `Install` – install a product.
- `Update` – update an installed product.
- `Repair` – repair existing installation.
- `Remove` – uninstall a product.
- `License` – show license for a product.
- `Licenses` – list available licenses.
- `HddId` – print hard drive identifier.
- `Products` – list available products.
- `Updates` – show available updates.
- `Installed` – list installed programs.
- `Sign` – sign a DLL file.

The optional `[product]` parameter is the product ID from the [Store](#). You can find this ID on the product page—for example on the [Hydra server page](#)—or by running `StockSharp.Installer.Console.exe Products -s hydra`. `[dir]` specifies the installation directory.

Options

The utility accepts the following options:

- `-s, --search` – filter products by name.
- `-r, --run` – automatically run an application (for example, `StockSharp.Hydra.Server.exe`) after installation.
- `-c, --cache` – use the NuGet cache.
- `-f, --force` – force update check ignoring the configured interval.
- `-p, --pre` – allow installing pre-release versions.

- -e, --noerror – suppress any errors.
- -b, --backup – backup previous settings before repair or update.
- -l, --clear – clear the target directory before installation.
- -t, --fw – specify the target .NET framework.
- -d, --data – remove application data folder.
- -i, --in – DLL to sign.
- -o, --out – result DLL after signing.

Example command:

```
dotnet StockSharp.Installer.Console.dll Install 1269 /home/user/stocksharp -p -r
StockSharp.Hydra.Server.exe
```

This installs product **1269** (used here just as an example) into the specified directory, allows pre-release builds and launches `StockSharp.Hydra.Server.exe` after completion.

Signing

The `Sign` command digitally signs a robot DLL. Use it when distributing your own API-based robot so that all recipients, even on the [free plan](#), can run it.

Compile the robot beforehand and add the `[assembly: ProductId(9)]` attribute from the `StockSharp.Configuration` namespace. Sign the DLL (for example, `MyRobot.dll`) rather than the `.exe` file.

Example:

```
StockSharp.Installer.Console.exe Sign -i "MyRobot.dll"
```

Rebuilding the project removes the signature, so sign the assembly at the end of development when no further compilation is expected.

Designer

Designer

Designer is a program that allows you to create trading strategies, test them on history and manage strategies for real trading. Designer offers the following ways of creating strategies:

1. [Using visual designer](#), it does not require programming skills. The process of creating a strategy is accomplished by combining cubes and connections (lines). At the same time, the whole process of work is visually represented.
2. [Using code](#), it is for experienced programmers who prefer to work with the code. Strategies written in **C#**, **F#** or **Python** work much faster than strategies written in a visual designer. Also, such strategies are not limited in capabilities during creation, and it is possible to write any algorithm, unlike cubes. The process of creating the strategy runs directly in **Designer** or the development environment in **C#** (the most popular of the development environments is **Microsoft Visual Studio**), using the library for professional development of trading algorithms in **C#** and [API](#).

[image: StockSharpTitle 0]

Recommended content

[Installing Designer](#)

Getting Started

Getting Started

Trading Strategy Constructor

Trading Strategy Constructor

Variable Logic Mathematics

Variable Logic Mathematics

Trading Strategy Debugger

Trading Strategy Debugger

Protecting Positions

Protect Positions

Indicators and Formulas

Indicators and Formulas

Bollinger Bands and MACD

Bollinger Bands and MACD

Multi Timeframe

Multi Timeframe

Time

Time

Connecting Designer to Hydra

Connecting Designer to Hydra

Optimization of Trading Strategies

Optimization of Trading Strategies

Live Trading

Live Trading

Composite Cubes

Composite Cubes

Compressing Candles from Ticks

Compressing Candles from Ticks

Flag

Flag

Maximum Volume Strategy

Maximum Volume Strategy

Financial Indices

Financial Indices

Pair Trading

Pair Trading

Indices and Pair Trading

Indices and Pair Trading

Resources for Learning

Resources for Learning

This documentation is intended to acquaint you with the key resources provided by StockSharp for learning and working with the "Designer" program. These materials will help users quickly start working and effectively use the program.

Educational Videos

On the StockSharp YouTube channel, you can find tutorial videos that demonstrate how to work with "Designer", including setting parameters, creating strategies, and testing them.

- "Designer" Playlist on YouTube: [View Playlist](#)

Strategy Examples

The Strategy Gallery on the official StockSharp website features various trading strategies developed using "Designer". These examples can serve as an excellent starting point for creating your own strategies.

- "Designer" Strategy Gallery: [Visit the Strategy Gallery](#)

GitHub Repository

Source codes for strategies and other components used in "Designer" are available on GitHub. This resource is ideal for studying code examples and adapting them to your needs.

- StockSharp Repository on GitHub: [Visit the Repository](#)

Support Chat

For interacting with other users and getting quick help with using "Designer", join our chat on Telegram.

- "Designer" Chat in Telegram: [Join the Chat](#)

These resources provide everything necessary for effective learning and use of the "Designer" program. We recommend taking advantage of all the provided materials to achieve the best results in trading.

Installing Designer

Installing Designer

[Install](#) [and remove apps](#)

[Update to the new version](#)

Update to the new version

[Update apps](#)

Recommended content

[Installing Designer](#)

Quick start

Quick start

On first run **Designer** will prompt you to open the [Download market-data](#) window. Also, historical data can be downloaded by the [Hydra](#) program (codenamed Hydra) designed for automatically downloading market data (instruments, candles, tick trades and order books, etc.) from various sources and storing them in a local storage. About downloading and storing historical data is described in detail in the [Market data storage](#) section.

When you click the **Download securities** button, the [Download instruments](#) window appears. To download the instrument, you should enter the instrument code, instrument type, select the data source and click **OK**. **Designer** will query the data source for available instruments. All found instruments will appear in the **All securities** panel. By default, it is specified as a data source in **Designer**. You can also use trading terminals as a data source. How to configure the connection to terminals is described in the [Connections settings](#) section.

[image: Designer Quick start 01]

To obtain historical data on the instrument, you should select the necessary instrument from the **All securities** list, set the historical data period, select the type and Time Frame of the candles and press the **Start** button. All data will be stored in the [Market data storage](#).

[image: Designer Quick start 02]

After obtaining the historical data, choose one of the demonstration strategies. Double-click the [Schemas](#) panel in the **Strategy** folder selects the **SMA** strategy example, after that the Sma tab appears in the workspace. After switching to the strategy, in the bar, the **Emulation** tab will automatically open, which contains the main elements for strategy creation, debugging, and testing ([Strategies](#), [Getting started](#))

[image: Designer Quick start 03]

In the **Emulation** tab, you need to set a test period, and select [Market data storage](#) in the **Market Data** field.

Clicking the *[image: Designer Quick start 04]* icon in the **Security** field opens the **Select security** window. You should select the required instrument in this window.

[image: Designer Quick start 05]

After selecting any block in the **Designer** panel, the properties of this block appear in the **Properties** panel. In the **Properties** panel of the **Candles** block, you can set the type and Time Frame of [Candles](#).

After clicking the **Start** button, the trading emulation will start.

[image: Designer Quick start 06]

Ribbon

Ribbon

The main user interface element of [Designer](#) is the **Ribbon**, which runs along the top of the application window. The ribbon allows for quick access to necessary commands. Commands are organized into logical groups, collected on tabs. To navigate to the desired tab, simply click on its title (name). Each tab is associated with the type of action being performed.

1. The **Common** tab, which opens by default after starting, contains elements that may be needed at the initial stage of work. From the **Common** tab, you can open [Connection Settings](#), [Scheme Panel](#), [Logs Panel](#), [Portfolios](#), [Board Editor](#), [Creating a Repository of Historical Data](#). Also, on the **Common** tab, you can add, open, delete, import, and export strategies. If you want to share your strategy with the community, you can do so by clicking the *Publish* button. The adjacent button - *Available Strategies*, opens algorithms published by you and other users. On the right, there are service buttons for calling help, as well as for contacting us. You can report a problem or write to us in Telegram chat.

[image: Designer Tape 00]

2. The **Backtest** tab automatically opens when selecting a strategy on the [Scheme](#) panel. The **Backtest** tab contains the main elements for creating, debugging, testing, and optimizing strategies ([Creating a Strategy](#), [Example of Backtesting](#)). Also, on this tab, the strategy is launched for real trading, and necessary components for your strategy are selected: chart, order book, trades, etc.

[image: Designer Tape 01]

3. The **Live** tab is intended specifically for real trading. Details on connection settings are written in [Connection Settings](#). Real trading using [Designer](#) is described in [Live Trading](#).

[image: Designer Tape 02]

4. The **Cloud** tab. [Designer](#) is designed to work with cloud services. It allows viewing completed *Cloud Tasks*, getting information on instruments available in the Cloud, and setting up remote work with channels and robots.

[image: Designer Tape 03]

See Also

[Workspace](#)

Workspace

Workspace

The **Workspace** in [Designer](#) is a windowed interface with panel docking similar to that used in Microsoft Visual Studio. By clicking the header of the panel with the left mouse button and holding it, you can lock the panel in a convenient place. By dragging the edges of the panel, you can adjust its height or width:

[image: Designer Workspace 00]

Recommended content

[Schemas](#)

Schemas

Schemes Panel

To open the **Schemes** panel, you need to click on the **Schemes** button in the **Common** tab. The **Schemes** panel contains a tree of scripts, grouped into folders by purpose. Strategy schemes and custom blocks are no different. They are edited using a common editor, [Strategy Designer](#). However, to avoid confusion between them, they are divided into two independent lists and stored in different folders (strategies in the **Backtest** folder, custom blocks in the **Custom Blocks** folder). Selecting a scheme for editing is done by double-clicking the required item in the list. The selected scheme will then open in the designer for viewing and editing. Below is a description of the folders in the **Schemes** panel:

[image: Designer Panel Circuits 00]

1. The **Backtest** folder contains trading strategies created both as schemes from a set of elements and connections between them, and from code. You can add a new strategy by pressing the **Add** *[image: Designer Panel Circuits 01]* button in the **Common** tab and selecting **Strategy**. Or by right-clicking on the **Backtest** folder in the **Schemes** panel, and pressing the **Add** *[image: Designer Panel Circuits 01]* button in the dropdown menu. In the window that opens, select how exactly you want to create a strategy.

[image: Designer Panel Circuits 04]

Strategies can be created using a visual designer without coding or using the built-in source code editor. Additionally, external DLL files with strategies written in Microsoft Visual Studio can be connected. Detailed information about **Strategies** is described in the section [Using blocks](#).

2. The **Own elements** folder contains elements that represent a complete functionality and can be used in various schemes or in one scheme multiple times with different property values. Such sets of elements can be extracted into a separate block, which will then be used like any standard element. **Custom block** is a regular scheme that is saved/loaded/edited like any strategy scheme. Add a new composite element by pressing the **Add** *[image: Designer Panel Circuits 01]* button in the **Common** tab and selecting **Custom Blocks**. Or by right-clicking on the **Custom Blocks** folder in the **Schemes** panel, and pressing the **Add** *[image: Designer Panel Circuits 01]* button in the dropdown menu. When adding new custom blocks, they are automatically added to the **Element Palette**, in the **Custom Blocks** group, and can be used in creating other strategy schemes and custom blocks. Detailed information about **Custom Blocks** is described in the section [Creating composite elements](#).
3. The **Live** folder contains strategies added for trading. Strategies that are launched are marked with an icon *[image: Designer Panel Circuits 02]*, and those that are stopped are marked with an icon *[image: Designer Panel Circuits 03]*. How to add strategies to the **Live** folder and how to launch them is described in the section [Live trading](#).
4. The **Indicators** folder contains your own indicators for trading strategies, written by yourself. New indicators cannot be created with schemes, only code and external DLL files are available. The use of custom indicators in schemes is available through the [Indicator](#) block when selecting the type of indicator.
5. The **Remote** folder contains strategies located on a remote server.

See Also

[Logs Panel](#)

Logs

Logs Panel

To simplify monitoring the operation of the program, there is a special **Logs** panel. This panel displays messages from all sources, strategies, and connections. In the **Logs** panel, the nesting of sources is shown as a tree. Each parent node contains messages of all nested sources and so on, down to the lowest level. For strategies, this hierarchy allows you to see child strategies. To open the **Logs** panel, you need to press the **Logs** button in the **Common** tab.

[image: Designer Panel Logs 01]

See Also

[Portfolios](#)

Portfolios

Portfolios

To work with portfolios there is the **Portfolios** panel. You can open it by clicking the Portfolios button in the **Common** tab.

[image: Designer Panel A Portfolios 00]

The **Portfolios** panel is a table that shows the basic data on the portfolio, and the current positions for these portfolios. By double-clicking on the portfolio or position, the **Creating portfolio** window or the **Creating position** window appears in which you can edit the selected portfolio or position. To create a new portfolio or a new position, you should click the *[image: Designer Panel Circuits 01]* button and select the appropriate item in the drop-down menu.

[image: Designer Panel A Portfolios 01]

When you click the *[image: Designer Alert Bell]* button, the [Notification settings](#) window opens.

Recommended content

[Strategy gallery](#)

Boards

Boards

In the **Board editor** panel, you can create **Boards** and **Exchanges**, and view or customize existing ones.

[image: Designer Boards]

In **S#**, instruments from different sources use a unified identifier consisting of the instrument code and the board code. The syntax is **[instrument code]@[board code]**. For example, for **AAPL** shares of the **NASDAQ** exchange, the identifier is **AAPL@NASDAQ**. Each instrument is attached to a specific board on which it is traded. However, the instrument can be traded on different boards. In this case, the board codes will be different. For each board, you can set up a work schedule with working days and weekends.

The exchange may have several boards with different terms of trade (session time, commission, etc.). But each board is attached to a specific exchange. Therefore, in the **Board editor**, when you select a board, the information about the exchange will automatically change to the exchange on which the board is located. For each exchange, you can set the exchange's code, country, Russian-language and English-language names.

Risk management

Risk Management

On the [Testing Properties](#) and [Live Trading Properties](#) panels, you can set the risk control settings.

In the Risks window, it's necessary to select a **Risk Rule**, configure the triggering condition for the **Risk Rule**, and the action (Close positions, Stop trading, Cancel orders) that will be executed when the condition for the **Risk Rule** occurs.

It is possible to use multiple risk rules of the same type with different actions. For example, in the screenshot below, if the order volume is 20, then the actions to cancel orders and stop trading are executed.

[image: Designer Risk Rule]

List of Risk Rules

List of Risk Rules

- **P/L** - a risk rule that monitors the size of profit/loss.
- **Position** - a risk rule that monitors the size of the position.
- **Position (Time)** - a risk rule that monitors the lifespan of a position.
- **Commission** - a risk rule that monitors the size of the commission.
- **Slippage** - a risk rule that monitors the amount of slippage.
- **Order Price** - a risk rule that monitors the price of an order.
- **Order Volume** - a risk rule that monitors the volume of an order.
- **Order (Frequency)** - a risk rule that monitors the frequency of placing orders.
- **Error in Registration/Cancellation of Order** - a risk rule that monitors the number of errors during registration/cancellation of orders.
- **Trade Price** - a risk rule that monitors the price of a trade.
- **Trade (Volume)** - a risk rule that monitors the volume of a trade.
- **Trade (Frequency)** - a risk rule that monitors the frequency of making trades.
- **Error** - a risk rule that monitors the number of any errors.

Commissions

Commissions

In the [Backtesting settings](#) panel, you can set the commission monitoring settings.

In the **Commissions** window, you should select the type of commission, set the commission value and the condition under which the commission will be charged.

[image: Designer Commission Rule]

List of commission types

List of commission types

- **Orders (number)** – the commission for the number of orders.
- **Order** – the commission for the order.
- **Order (volume)** – the commission for the volume in the order.
- **Trade (number)** – the commission for the number of trades.
- **Trade (price)** – the commission for the trade price.
- **Trade** – the commission for the trade.
- **Trade (volume)** – the commission for the volume in the order.
- **Security** – the instrument commission.
- **Security (type)** – the instrument type commission.
- **Turnover** – the commission for turnover.
- **Board** – the board commission.

Components

Components

During testing or trading, there arises a need to fully control the process. For complete control and analysis, [Designer](#) includes graphical components that can be added by selecting them in the **Ribbon** under the **Components** group:

[image: Designer Components]

- [Chart](#) - represents a candlestick chart and/or indicators. Contains all the [Chart Panel](#) cubes present in the strategy.
- [P/L](#) - represents a chart of Profit/Loss (unrealized), Profit/Loss (realized), and commissions.
- [Trades](#) - represents a table with trades, displaying complete information about all the strategy's trades.
- [Orders](#) - represents a table with orders, displaying complete information about all the strategy's orders.
- [Positions](#) - represents a table of positions for the strategy's instruments.
- [Statistics](#) - represents a table, grouped by categories: P/L, Trades, Positions, Orders. It contains data about the strategy's performance.
- [Trade Tape](#) - represents a table with trades, displaying complete information about all the trades for selected instruments.
- [Position Chart](#) - represents a Position chart.
- [Properties](#) - represents a table of [Testing Properties](#) or a table of [Live Trading Properties](#).
- [Scheme](#) - represents the schema designer.
- [Source Code](#) - represents a window for creating or editing strategies based on source code.
- [DLL](#) - a window for viewing DLL metadata. In case of creating a strategy based on a ready DLL.
- [Options Position Chart](#) - represents an options position chart.
- [Options Board](#) - represents an options table.
- [Depth](#) - represents a table of limit orders for buying and selling.
- [Order Log](#) - represents a table with orders, displaying complete information about all orders for selected instruments.

Chart

Chart

The **Chart** component contains all the **Chart panel** cubes that are in the strategy. For example, if the strategy uses two **Chart panel** cubes, then in the **Chart** component 2 charts are displayed, as in the figure.

[image: Designer Panel graphics 04]

The top left corner of each chart shows all the graphical elements added to the chart. If you clear the check box on the *[image: Designer Schedule 00]* graphic element, the element will be removed from the chart. Click on the *[image: Designer Schedule 01]* button will open the settings for the graphic element. Also, you can set up the graphic element in the [Chart](#) cube properties.

[image: Designer Schedule 02]

In the graphical element settings, you can set the required chart style: Japanese candles, bars, box chart, cluster profile, etc.

[image: Designer Schedule 04]

For a box chart candles can be additionally grouped, the grouping order is set in the fields: the multiplier of the 2nd timeframe, the multiplier of the 3rd timeframe.

[image: Designer Schedule 05]

There is a toolbar above the chart, where you can select auto scroll, auto-zoom, legend modes and other general chart settings. You can also select the elements to draw on the chart: lines, levels, pointers, rectangle, text.

[image: Designer Schedule 03]

Recommended content

[Orders](#)

Orders

Orders

The **Orders** component is a table with orders, which displays complete information about all the orders of the strategy.

[image: Designer Applications 00]

Recommended content

[Trades](#)

Trades

Trades

The **Trades** component is a table with trades, which displays complete information about all trades of the strategy:

[image: Designer The transaction 00]

Recommended content

[P&L equity](#)

Order book

Order book

The **Order book** component is a table of limit orders for the purchase and sale. At the top of the component, you should select the data source from which the data will be received. The source of the data is the [Order book panel](#) cube.

In the strategy, you can use several different types of order books (or from different instruments) simultaneously, and display them in different panels.

[image: Designer Panel Market Depth]

Recommended content

[Positions](#)

Positions

Positions

The **Positions (chart)** component is a chart of the Position. The top left corner of the chart shows all the graphic elements added to the chart. If you clear the *[image: Designer Schedule 00]* check box on the graphic element, the element will be removed from the chart. Clicking the *[image: Designer Schedule 01]* button will open the settings of the graphic element.

[image: Designer Chart Position 00]

The **Positions** component is a table of positions for the strategy instruments.

[image: Designer Table Position 00]

Recommended content

[Statistics](#)

Statistics

Statistics

The **Statistics** component is a table grouped by categories: P/L, Trades, Positions, Orders. Statistics help to assess the correctness and performance of the strategy. For example, the sample strategy SMA cannot have more than 1 position. Statistics also help to evaluate the strategy's performance by other qualitative parameters, such as maximum drawdown, relative drawdown, recovery factor, etc.

[image: Designer Statistics 00]

See Also

[Debugging](#)

P&L equity

P&L equity

The P/L component is a chart of the Profit/Loss (unrealized), Profit/Loss (realized) and commission. In the upper left corner of the chart, all the graphic elements added to the chart are shown, if you remove the *[image: Designer Schedule 00]* check box on the graphic element, the element is also removed from the chart. Clicking the *[image: Designer Schedule 01]* button will open the settings for the graphic element.

[image: Designer Schedule PU 00]

Recommended content

[Live execution sample](#)

Backtesting settings

The **Properties** panel is minimized on the right side of the strategy tab by default. This panel is a table of emulation or Live trade properties. When you select a specific property, a detailed description of this property appears at the bottom of the table. All properties are grouped into groups:

[image: Designer Properties emulation 00]

Settings

- **Market data** – the data storage.
- **Storage format** – the storage format.
- **Data type** – the data type.
- **Time frame** – to use candles with the specified time frame.
- **Maximum quote volume in generated depth** – the maximum quote volume in generated order book.
- **Interval** – the time interval.
- **Unrealized P/L** – the interval of recalculation of the unrealized profit.
- **Trades** – which trades to use.
- **Marked depth** – which order books to use.
- **Order log** – to use the order log.
- **Number of strategies** – the number of simultaneously tested strategies.
- **Logging level** – the logging level.
- **Combine on touch** – during emulation, combine orders when the trade price touches the order price (i.e., equal to the order price).
- **Marked depth (lifetime)** – the maximum time during which the order book is in the emulator. If during this time there was no update, the order book is deleted. This property can be used to remove old order books when there are holes in the data.
- **Errors percentage** – the percentage value of the error in registering new orders. The value can be from 0 (there will be no errors) to 100.
- **Latency** – the minimum value of the registered order latency.
- **Reregistering** – whether order reregistering is supported as a single trade.
- **Buffering period** – to send responses in a single packet at intervals. The network latency and the buffered operation of the exchange core is emulated.
- **Order ID** – the number from which the emulator will generate identifiers for orders.
- **Trade ID** – the number from which the emulator will generate identifiers for trades.
- **Transaction** – the number from which the emulator will generate identifiers for order transactions.
- **Spread size** – the spread size in the price steps. It is used in specifying the spread for the generation of an order book from tick trades.
- **Depth of book** – the maximum depth of the order book that will be generated from the ticks.

- **Number of volume steps** – the number of volume steps for which the order is larger than the tick trade. It is used for testing on tick trades.
- **Portfolios interval** – the interval for portfolio recalculation. If the interval is zero, no recalculation is performed.
- **Change time** – to change time for orders and trades into exchange time.
- **Time zone** – information about the time zone where the exchange is located.
- **Price shift** – the price shift from the last trade, specifying the limits of the maximum and minimum prices for the next session.
- **Add extra volume** – add extra volume to the order book when orders with large volume are being registered.
- **Commissions** – the commission (brokerage, exchange, etc.).

Logging

- **Logging level** – the logging level for this element.

Setting

- **Risk management** – the risk management settings.

Diagram parameters

- **Security** - the security.
- **Portfolio** - the portfolio.

If you do not fill in the **Diagram parameters**, when emulating, the instrument will be used from the **Instrument** field of the **Emulation** tab, and as a portfolio, the test portfolio will be used by default.

Recommended content

[Chart](#)

Live settings

Live settings

The **Properties** panel is minimized on the right side of the strategy tab by default. This panel is a table of emulation properties or Live settings. When you select a specific property, a detailed description of this property appears at the bottom of the table. All properties are grouped into groups:

[image: Designer Properties Live 00]

Logging

- **Logging level** – the logging level for this element.

Initialization

- **Data type** - the data type.
- **Time frame** - to use candles with the specified time frame.
- **Days of history data** - the number of days of history data to initialize the strategy.

Settings

- [Risk management](#) - settings of risk management.

Diagram parameters

- **Security** - the security.
- **Portfolio** - the portfolio.

If you do not fill in the **Diagram parameters**, when emulating, the instrument will be used from the **Instrument** field of the **Emulation** tab, and as a portfolio, the test portfolio will be used by default.

Recommended content

[Chart](#)

Tick trades

Tick trades

The **Trades feed** component is a table with trades, which displays complete information about all trades for the selected instruments.

The **Trades feed** has a filter to select the necessary instruments. You can also configure notifications for events related to the selected instruments in the [Notification settings](#) window.

[image: Designer Tape Trades]

Using visual designer

Using Blocks

Using blocks does not require programming skills. The process of creating a strategy involves combining blocks and connections (lines), visually representing the entire workflow.

You can add a new strategy by pressing the **Add** *[image: Designer Panel Circuits 01]* button in the **Common** tab and choosing **Strategy**. Or, by right-clicking on the **Strategies** folder in the **Scheme** panel, and pressing the **Add** *[image: Designer Panel Circuits 01]* button in the dropdown menu:

[image: Designer The creation of a strategy 00]

After pressing the **Add** *[image: Designer Panel Circuits 01]* button, a window will appear with a choice of content type to create the strategy on:

[image: Designer The creation of a strategy 02]

To create a strategy out of blocks, you need to select the **Scheme** tab. You can also choose a template that will be used as the initial scheme.

After pressing **OK**, a new strategy will appear in the **Strategies** folder of the **Scheme** panel. A new tab with the strategy will appear in the workspace, and upon navigating to it, the **Backtest** tab will automatically open in the ribbon. On the **Scheme** panel, right-clicking on the strategy opens a menu that allows you to rename the strategy, among other actions.

[image: Designer The creation of a strategy 01]

The strategy tab consists of the **Scheme** panel ([Strategy Designer](#)), as well as other tabs that represent the [graphical components](#) of the strategy, necessary for displaying the results of testing the strategy created in the **Scheme** area. Detailed information about strategy testing is described in the section [Example of backtesting](#).

See Also

[Strategy Designer](#)

Diagram panel

Strategy Designer

The main process of designing a strategy and its component elements takes place in the **Scheme** panel by combining blocks and connecting lines. The Scheme panel consists of the panels: **Palette**, **Designer**, and **Properties**.

[image: Designer Designer schemes strategies and component elements 00]

Palette Panel

The **Palette** panel contains blocks from which strategies are created. All elements in the palette are divided into categories, described in the section [Description of blocks](#). To add a block to the **Designer** panel, right-click on the required block and, without releasing the button, drag it to the **Designer** panel. After that, the element will be automatically selected, and its parameters will be shown in the window for editing the block's properties.

Designer Panel

The **Designer** panel is where the entire process of creating a strategy occurs through the combination of blocks and connections (lines). It visually represents the strategy scheme. Detailed information about creating a strategy is described in the section [Creating an algorithm from blocks](#).

Properties Panel

The **Properties** panel displays the parameters of the selected block on the **Designer** panel. When a block is selected on the **Designer** panel, its frame is colored black.

[image: Designer The Properties Panel 00]

The **Properties** panel can be displayed in two modes: *Basic settings* and *Advanced settings*.

By default, when building a scheme, the properties are initially displayed in *basic settings* mode. To switch to *advanced settings* mode, you need to click on the corresponding title.

In *basic settings* mode, only the most necessary properties of the block are displayed. For example, for the [candles](#) block, the timeframe, the flag for receiving only formed candles, the flag for the possibility of constructing candles from a smaller timeframe, and the flag for subscribing to candles on signal will be displayed.

In *advanced settings* mode, all properties of the block available for change and setting will be displayed.

[image: Designer The Properties Panel 00]

All blocks contain a set of predefined properties, which become visible in *advanced settings* mode:

- **Name** – the name of the element, displayed in the designer.
- **Logging level** – the logging level for this element.
- **Parameters** – display the element's parameters in higher-level elements.
- **Sockets** – display the element's sockets in higher-level elements.

Detailed information about the properties of each block is described in the section [Description of blocks](#).

See Also

Elements

Inside each cube, an icon is displayed that characterizes it, as well as a name that can be changed to a user-defined one in the **Properties** panel. A tooltip for the cube shows a description of what this cube is for. When you select a cube with a mouse, you can view its properties in the **Properties** panel and, if necessary, change some parameters.

[image: Designer Description of cubes 00]

To the left and to the right of the cube, the colored boxes show the incoming (left) and outgoing (right) parameters.

Parameters are needed to fill the cube with information while the strategy is running. For example, for the **Candles** cube, the instrument for which you want to build a candle is passed to the input, and the built candles are returned at output. Which, in turn, can be used as an input parameter for the **Chart** element. Or to pass them to a method that determines the candle size.

[image: Designer Description of cubes 01]

The color denotes the type of data that is passed in the parameters. Different parameters in different cubes can receive and pass different and incompatible data types. The description of each parameter is indicated in the tooltip. To exclude a lot of errors when connecting parameters of different types, each parameter has its own data type, which differs in color. The following color set is used to indicate the parameters:

- **Black** – any type of data, usually used as a signal to perform certain actions within the element.
- **Dark green** – the instrument.
- **Dark cyan** – the order book.
- **Cyan** – the quote (a pair of price and volume).
- **Orange red** – candles and candle state.
- **Dark goldenrod** – the indicator value.
- **Olive** – the order.
- **Pale violet red** – order error.
- **Dark olive green** – own trade.
- **Dodger blue** – the flag value (indicates the state and has two values: up (true) and down (false)).
- **Medium sea green** – a numerical value, can be set as a number or percentage.
- **Dark slate blue** – values that can be compared (for example, a numeric value, a string, an indicator value, etc.).
- **Brown** – the portfolio.
- **Deep pink** – options.
- **Beige** – the side.
- **Dark khaki** – the trade.
- **Dark blue** – the strategy.
- **Chocolate** – the date.

- **Coral** – the time.
- **Saddle brown** – the position.
- **Chartreuse** – the order state.
- **Gainsboro** – the Black-Scholes model.
- **Tan** – the basket Black-Scholes model.
- **Purple** – the text string.

Thus, you can connect the parameters of the same colors (the same data types), except for the following types of parameters:

1. The **black** parameter can accept any data. Most often such parameters are used to pass signals for any actions inside the cube. For example, the [Variable](#) cube stores a certain value and sends it to the output when it receives a signal.
2. The **dark slate blue** parameter can receive on the input various comparable data types. For example, numeric, indicator values, strings, etc.

It should be noted that the parameter types can depend on the cube properties. For example, for the [Converter](#) cube, the input parameter type is automatically determined by the data source cube data type for [Converter](#). When you create a link, the color of the square on the element automatically changes.

Output parameters usually allow several outgoing connections to different cubes, the input parameters generally allow one connection, except for the [Combination](#) cube, which allows you to merge the data stream from different cubes into one. The number of concurrent connections for a parameter is specified in the source code of the cube.

Cubes for building schemas are divided into several categories, each of the categories is intended for use in a certain part of the schema.

Recommended content

[Chart](#)

Chart

Chart

[image: Designer Panel graphics 00]

The cube is designed to display data on the chart in the [Chart](#) panel. Each element corresponds to one area on the chart and contains a set of properties with which you can define a set of axes on the chart, a set of graphic elements and the parameters of their display. Adding any graphic element is done by pressing the *[image: Designer Panel graphics 01]* button, deletion by pressing the *[image: Designer Panel graphics 02]* button. When adding a graphic element (a series of candles, indicator, trades, etc.) in the settings, a special input parameter is created in the cube, through which the data to display will be passed to this cube, and as a result will be displayed in the [Chart](#) panel. When you click the *[image: Designer Panel graphics 03]* button, the child graphic elements or their settings will be opened.

[image: Designer Panel graphics 04]

The [Chart](#) panel displays all the data received in the **Chart panel** cubes. For different series of candles, it is necessary to take different **Chart panel** cubes. At that, the charts will be placed one above the other, but only the corresponding graphic elements (candle series, indicator, trades, etc.) will be displayed on each one. The example shows that the top chart displays candles, two indicators and trades, which correspond to four input parameters of the top cube. Only candles are shown on the bottom chart. For more information about the [Chart](#) panel, see [Chart](#) section.

Recommended content

[Crossing](#)

Crossing

Crossing

[image: Designer Crossing 00]

This element is used for tracking the position of two values relative to each other. For example, to determine the moment of crossing between two lines.

The comparison is made regarding the values on two sockets **Up** and **Down**.

Incoming Sockets

- **Up** – values that allow comparison (for example, a numerical value, indicator value, etc.).
- **Down** – values that allow comparison (for example, a numerical value, indicator value, etc.).

Outgoing Sockets

- **Flag** – true if **Up** is greater than **Down**, otherwise false.

[image: Designer Crossing 01]

An example of using the Crossing block to track the crossings of two [SMA indicators](#). Two Crossing blocks are used, and each outputs true separately depending on when the long SMA is greater than the short one and when it is less.

See Also

[Value Delay](#)

Delay value

Value Delay

[image: Designer Delay 00]

This component is used to delay the transmission of a value for a specified number of iterations.

Input Sockets

- **Trigger** – A signal (any value except `False`) that initializes the internal counter to start the delay countdown.
- **Input** - Any incoming value (except for [unfinished candles](#) or [non-final indicator values](#)) that decreases the internal counter. When the counter reaches zero, it is deactivated, and the outgoing socket is activated. If the counter was not activated by the **Trigger**, incoming values are ignored.

Output Sockets

- **Signal** – Outputs a signal when the counter reaches zero, indicating the end of the delay.

Parameters

- **Duration** - Specifies the delay duration in iterations.

[image: Designer Delay 01]

See Also

- [Comparison](#)

Comparison

Comparison

[image: Designer Comparison 00]

An element is used to compare two input objects.

Incoming sockets

- **Value 1** – values that can be compared (for example, a numeric value, a string, an indicator value, etc.).
- **Value 2** – values that can be compared (for example, a numeric value, a string, an indicator value, etc.).

Outgoing sockets

- **Flag** – the flag value (indicates the state and has two values: up (true) and down (false)).

Parameters

- **Left** – the parameter whose value will be used on the left side of the comparison.
- **Operator** – the condition by which you need to compare the passed values.
- **Right** – the parameter whose value will be used on the right side of the comparison.

When you change the operator, the cube name automatically changes to the operator name.

Recommended content

[Indicator](#)

Indicator

Indicator

[image: Designer Indicator 00]

This block is used for calculating indicator values.

Incoming Sockets

- **Any Data** – a specific type of data based on which the selected indicator should be calculated (depending on the indicator, this could be a numerical value, candle, etc.).

Outgoing Sockets

- **Indicator** – the calculated indicator value, which can be used for display on the chart panel or further calculations.

Parameters

- **Indicator Type** - a parameter used to select the desired indicator, and several additional parameters that correspond to the selected indicator type. The set of these parameters changes when the selected indicator type changes.
- **Final** - pass only [final values](#) of the indicator.
- **Formed** - pass only values when the indicator is fully [formed](#).

[image: Designer Indicator 01]

See Also

[List of Indicators Logical Condition](#)

Logical condition

Logical Condition

[image: Designer The logical condition 00]

This element is used to compute a logical formula with two arguments.

Incoming Sockets

- **Flag** – the flag value (characterizes the state and has two values: raised (true) and lowered (false)).
- **Flag** – the flag value (characterizes the state and has two values: raised (true) and lowered (false)).

Outgoing Sockets

- **Flag** – the flag value (characterizes the state and has two values: raised (true) and lowered (false)).

Parameters

- **Operator** - a predefined set of logical formulas AND, OR, Exclusive OR.

See Also

[Formula](#)

Formula

Formula

[image: Designer Universal formula 00]

A cube is used to calculate a mathematical formula with an arbitrary number of arguments. You can choose a formula from the available list or write your own. In the case of writing your own formula, the number of incoming sockets is determined automatically.

Incoming sockets

- **Value** – the value with which you can perform mathematical operations (for example, a number or indicator). The number of input values depends on the formula.

Outgoing sockets

- **Result** – calculated value of the mathematical formula.

Parameters

- **Formula** – predefined set of mathematical formulas.

Together with the standard mathematical operators, you can use the following functions:

- **abs(a)** - returns the absolute value of a number.
- **acos(a)** - returns the angle which cosine is equal to the specified number.
- **asin(a)** - returns the angle which sine is equal to the specified number.
- **atan(a)** - returns the angle which tangent is equal to the specified number.
- **ceiling(a)** - returns the smallest integer that is greater than or equal to a specified number.
- **cos(a)** - returns the cosine of the specified angle.
- **exp(a)** - returns the value of **e** raised to the specified power.
- **floor(a)** - returns the largest integer that is less than or equal to the specified number.
- **log(a)** - returns the natural logarithm (with base **e**) of the specified number.
- **log10(a)** - returns the logarithm with base 10 of the specified number.
- **max(a, b)** - returns the larger of two decimal numbers.
- **min(a, b)** - returns the smaller of two decimal numbers.
- **pow(a, b)** - returns the specified number raised to the specified power.
- **sign(a)** - returns an integer indicating the sign of the specified number.
- **sin(a)** - returns the sine of the specified angle.
- **sqrt(a)** - returns the square root of the specified number.
- **tan(a)** - returns the tangent of the specified angle.
- **truncate(a)** - calculates the integer part of the specified number.

Recommended content

Combination

Combination

[image: Designer Association 00]

The cube is used to combine the same type of data coming from different cubes into one output parameter for passing to another element or to divide the received data into several elements.

Incoming sockets

- **Any data** - specifies the type of data received and passed.

Outgoing sockets

- **Any data** - specifies the type of data received and passed.

Parameters

- **Type** - specifies the type of data received and passed.

Recommended content

[Comparison](#)

Prev value

Prev value

[image: Designer The previous value 00]

The cube is used to obtain the previous value passed to the input, with the specified offset.

Incoming sockets

- **Any data** - a certain type of data received and passed.

Outgoing sockets

- **Any data** - a certain type of data received and passed, entered at the input with the offset specified in the parameters.

Parameters

- **Data type** - the data type that can be entered through the input socket. If you change this parameter, the data type for the parameters is automatically changed.
- **Offset** - the size of the offset, the output value for how many changes will lag behind the input one.

Recommended content

[Random](#)

Random

Random Value

[image: Designer Random 00]

This block is used to generate a random value.

Incoming Sockets

- **Trigger** - the signal (any value except `False`) that determines the moment when it is necessary to pass a generated random value through the outgoing socket.

Outgoing Sockets

- **Value** - a random value.

Parameters

- **Min** - the minimum permissible value boundary.
- **Max** - the maximum permissible value boundary.

[image: Designer Random 01]

See Also

[Change in P/L](#)

Flag

Flag

[image: Designer Flag]

The "Flag" component is used to manage a binary flag, which can be set or reset based on incoming signals.

Input Sockets

- **Trigger:** Accepts any value except `False`. Sets the flag upon receiving the first appropriate value. If the flag is already set, subsequent signals are ignored until it is reset.
- **Reset:** Accepts any value except `False`. Resets the flag, allowing it to respond to subsequent triggers.

Output Sockets

- **Signal:** Outputs a signal when the flag is set.

Synchronization

[image: Designer Sync 00]

The Synchronization block is designed to accumulate and synchronize data from various sources (for example, candles from different instruments, different time frames, combinations of candles and transactions) and subsequently issue them when a certain amount is accumulated. This block is useful for creating custom indexes or arbitrage.

Input Sockets

- **Incoming:** When a new data source is connected, a corresponding outgoing socket and a new incoming socket are automatically created. The number of incoming values is unlimited.

Parameters

- **Interval:** Sets the time after which data must be updated or deleted. If an incoming value arrives with a time exceeding the previous value plus the interval, old data is cleared, and a new batch of data accumulation begins.
- **Clear items:** If this option is activated, data are cleared after their accumulation for all connected incoming sockets, or data are accumulated until data from the next time interval appears.

Usage Examples

1. Creating a custom index for multiple stocks, where it is necessary to consider different time series from various data sources.
2. Arbitrage between different markets using synchronized time data to identify temporal price differences.

[image: Designer Sync 01]

P&L strategy

P&L strategy

[image: Designer StrategyPnLDiagramElement 00]

Documentation eng RemoteManager

Outgoing sockets

- **P/L unrealized** – is the numerical value of the unrealized profit/loss.
- **P/L realized** – is the numerical value of the realized profit/loss.
- **Commission fee** – is the numerical value of the commission fee.

Recommended content

[Prev Value](#)

Trades by strategy

Trades by strategy

[image: Designer The transaction strategy 00]

The cube is used to obtain all the strategy trades.

Incoming sockets

- **Instrument** - the instrument for which you need to obtain trades. If the instrument is not passed, then trades by all strategy instruments are transferred to the output.

Outgoing sockets

- **Trades** - trades arising from the passed instrument. They can be used both for displaying on the chart using the **Chart panel** element, and for position protection using the **Position protection** element.

Candles

Candles

[image: Designer Candles 00]

This block is used to construct candles for a specified instrument.

Incoming Sockets

Incoming Sockets

- **Security** – the instrument for which the candles are to be built with given parameters.

Outgoing Sockets

Outgoing Sockets

- **Candles** – the constructed candles.

Parameters

Parameters

- **Series** – the type of candle series and parameters for the specified type;
- **Only Formed** – pass to the output only fully formed candles or any change;
- **Smaller Timeframe** – construct candles from a smaller timeframe;
- **Subscribe on Signal** – subscribe to data only after receiving a trigger.

See Also

[Level 1](#)

Level 1

Level 1

[image: Designer Level 1 00]

This block is used to receive **Level1** data for an instrument.

Incoming Sockets

Incoming Sockets

- **Instrument** – the instrument for which **Level1** data needs to be received.

Outgoing Sockets

Outgoing Sockets

- **Change** – **Level1** values of the tracked parameter.

Parameters

Parameters

- **Value** – the **Level1** parameter that needs to be tracked.
- **Subscribe on Signal** – subscribe to data only after a trigger arrives.

See Also

[Ticks](#)

Ticks

Ticks

[image: Designer Ticks]

This block is used to receive **Tick** data for an instrument.

Incoming Sockets

Incoming Sockets

- **Instrument** – the instrument for which **Level1** data needs to be received.

Outgoing Sockets

Outgoing Sockets

- **Tick** – a tick transaction.

Parameters

Parameters

- **Subscribe on Signal** – subscribe to data only after a trigger arrives.

See Also

[Index](#)

Index

Index

[image: Designer Index 00]

The cube is used to create your own index.

Outgoing sockets

Outgoing sockets

- **Security** - the calculated index, represented as a **Security**.

Parameters

Parameters

- **Index** - the mathematical formula of a combination of several instruments (for example, $(AAPL@NASDAQ+10)*(abs(20/GOOG@NYSE))$).
- **Ignore errors** - the set flag indicates that errors will be ignored when calculating the index.
- **Calculate extended information** - the set flag indicates that in calculating the index, in addition to the basic information (Total volume, Opening price, Closing price, Highest price, Lowest price), the extended information (Total trade turnover, Opening volume, Closing volume, Maximum volume, Minimum volume) will be calculated.

Available mathematical formulas are similar to the cube [Formula](#).

Recommended content

[Variable](#)

Variable

[image: Designer Variable 00]

The cube is used to store values and pass the previously stored value further along the chain of elements.

Incoming sockets

Incoming sockets

- **Any data** – the value of the selected type, which will be stored instead of the default value.
- **Trigger** – the signal (any value except `False`) that determines the point at which it is necessary to pass the stored value through the output socket.

Outgoing sockets

Outgoing sockets

- **Any data** – the value of the selected type of passed data.

Parameters

Parameters

- **Data type** - the type of stored data within the variable, the type of the input and output parameters depends on the selected data type.
- **Value** - the default value that is stored in the variable. This value is used if no other values were received to the element input.
- **Raise on start** - when the checkbox is selected, the value will be passed when the strategy is started.

If the **Instrument** or **Portfolio** data type is selected, the default value may be missing. In this case, if the **Parameters** flag is set in the properties, when the strategy is executed, these data will be taken from the corresponding properties of the strategy.

Recommended content

[Indexer](#)

Working time

Working Time

[image: Designer Working time 00]

This block is used to determine the working time for the strategy. For example, to define when trading is taking place for a specific instrument or when the strategy is allowed to trade.

Incoming Sockets

- **Any Data** - the block accepts any value but takes the timestamp from it, which is then compared to the block's parameters.

Outgoing Sockets

- **Flag** - a flag that determines whether the timestamp meets the block's parameters (true) or not (false).

Parameters

- **Time From** - the start time of the working time.
- **Time To** - the end time of the working time.

The block can be used to determine when trading is conducted for multiple instruments from different trading platforms.

[image: Designer Working time 01]

See Also

[Is Trading Allowed](#)

Trade allow

Is Trading Allowed

[image: Designer TradeAllowedDiagramElement 00]

This block is used to check if trading is currently allowed. The following conditions are checked:

- All strategy subscriptions to market data must be in [Online](#) state (receiving real-time data).
- All indicators must be [formed](#).
- In the case of [live trading](#), the incoming trigger value must have a timestamp greater than the strategy's start time.

Incoming Sockets

- **Trigger** - the signal that determines the moment when the check should be performed.

Outgoing Sockets

- **Flag** - a flag that determines whether the trading session is active.

See Also

[Current Time](#)

Current time

Current Time

[image: Designer time 00]

This block is used for obtaining the current time.

Outgoing Sockets

- **Output** - the current time.

Modify position

Modify Position

[image: Designer position modify 00]

The "Modify Position" component is used to change a trading position based on specified conditions.

Input Sockets

- **Security:** The instrument for which the position will be modified.
- **Trigger:** Signal to activate the modification of the position.
- **Portfolio:** The portfolio within which the operation occurs.
- **Volume** (optional): The volume for "Increase" and "Decrease" operations. Not used for "Reverse" and "Close Position".
- **Last Price** and **Last Volume:** For "VWAP" and "Iceberg" algorithms, data about the last price and volume of the transaction are required.
- **Cancel:** Signal to cancel the position setup, for example, due to a timeout.

Output Sockets

- **Order:** Information about the placed order.
- **Transaction:** Information about the transaction conducted on the order.
- **Balance:** This socket transmits information about the portion of the position that was not realized at the end of the position modification operation. The value returned by the socket indicates the result of the operation:
 - 0 means that the component successfully completed the position modification operation, and all planned actions were executed.
 - -1 indicates that the component did not start the position modification due to a mismatch of the current position and the specified condition (for example, if the current position is different from zero and the condition was "OpenPosition").
 - Any value greater than 0 signals that the process of modifying the position was interrupted before its completion. This could happen due to cancellation through the scheme logic or an error during order registration.

Parameters

- **Condition:** Position Modification Conditions:
 - None: Performs no actions.
 - OpenPosition: Opens a position in the specified direction.
 - ClosePosition: Closes the current position.
 - Decrease: Reduces the size of the current position.
 - Increase: Increases the size of the current position.
 - Reverse: Closes the current position and opens a new one in the opposite direction.
- **Direction:** Specifies the direction for "OpenPosition" and "None", and serves as an optional filter for other conditions.

- **Algorithm:** Options include "Market Order", "VWAP", "Iceberg".
- **Part:** The fraction of the total volume that will be divided into smaller segments when using algorithms like "VWAP" or "Iceberg".

If the component receives a trigger while it has already started changing the volume, it ignores the new trigger. If the conditions for modification are incompatible with the current state of the position (for example, attempting to "OpenPosition" when a position is already open), the component immediately returns -1 through the output socket **Balance**, indicating that the operation is not required and was not started.

Note

For low-level order management, the [Order Registration](#) component can be used. For higher-level position management, this "Modify Position" component is recommended.

See Also

- [Order Registration](#)

Current position

Position

[image: Designer Position 00]

The element is used to obtain information on the position change for the specified instrument and portfolio.

Incoming sockets

Incoming sockets

- **Instrument** - the instrument for which you want to obtain a position.
- **Portfolio** - the portfolio for which you want to obtain a position.

Outgoing sockets

Outgoing sockets

- **Position** - the numeric value of the position on the instrument or the current amount of available funds on the account. This value is generated when either the position or the funds change and after the strategy is started.

Parameters

Parameters

- **Money** - when the flag on the input is set, the element only accepts the portfolio, the money position for the selected portfolio is passed to the output of the element.

Recommended content

[Protect Position](#)

Protect position

Position Protection

[image: Designer Protect positions 00]

[image: Designer Protect positions 01]

This block is used for automatically protecting open trades with stop-loss and take-profit.

Incoming Sockets

Incoming Sockets

- **Own trade** – the trade that needs to be protected with a stop-loss and take-profit.
- **Price** – the current price (can be taken from a candle, the last tick, etc.). Required for tracking the current price of the instrument and activating protective orders.

Outgoing Sockets

Outgoing Sockets

- **Take-profit** – an order to lock in profits.
- **Stop-loss** – an order to limit losses.
- **Own transaction** – a transaction created by one of the above orders.

Parameters

Take and Stop Parameters

- **Value** - the value of the take or stop.
- **Trailing** – is trailing protection used.
- **Timeout** - the timeout value after which protection is forcibly triggered at the market price.
- **Market orders** – use market orders (without price) for quick position closing.

[image: Designer Protect positions 02]

Incoming transactions CANNOT be transactions of the entire strategy (the [Strategy Trades](#) block) as this will lead to incorrect calculation of the current position: the protection transactions will also become strategy transactions. The **Position Protection** block should receive transactions from the **Transaction** output socket of the [Order Registration](#) and [Modify Position](#) cubes, or similar components that directly change the position.

Register order

Order Registration

[image: Designer Position opening 00]

The "Order Registration" component is used for placing trading orders for a selected instrument.

Input Sockets

- **Instrument** – The selected instrument for the order.
- **Price** – Specifies the price for a limit order.
- **Trigger** – Activation signal for the order, accepts any value except False.
- **Volume** – The quantity of instruments for the order.
- **Portfolio** – The portfolio within which the order will be placed.

Output Sockets

- **Order** – Information about the placed order.
- **Error** – Information about any error during order registration.
- **Transaction** – Information about the transaction made on the order.
- **Cancellation** – Signal that the order has been cancelled.
- **Executed** – Signal that the order has been fully executed.
- **Completed** – Signal that combines events of error, cancellation, or full execution of the order.

Parameters

- **Direction** – Determines whether the order is for buying or selling.
- **Market Order** – Indicates whether the order is a market order.
- **Zero Price** – If the price is set to zero, the order is registered as a market order.
- **Lifetime** – The duration for which a limit order remains active.

Conditional Order Setup

Conditional Order – An order with additional conditions determining the timing of placement in the trading system based on the current market situation.

[image: Designer Conditional Application]

- **Connection** – The connection where the order will be placed.
- **Stop Order Type** – The type of stop order.
- **Result** – The outcome of the executed stop order.
- **Instrument Identifier** – The identifier for the instrument for stop orders with conditions related to another instrument.
- **Stop Price Condition** – The stop price condition. Used for orders like "Stop price for another instrument."

- **Stop Price** – The stop price that sets the condition for triggering the stop order.
- **Stop-Limit Price** – Similar to the Stop Price but used only for "Take-profit and stop-limit" type orders.
- **Stop-Limit at Market Price** – Indicates if the "Stop-Limit" order executes at the market price.
- **Condition Check Interval** – The time interval for checking the conditions of the order only within the specified period (if null, then no checks). Used for "Take-profit and stop-limit" and "Take-profit and stop-limit by order" types.
- **Conditional Order Execution Identifier** – The identifier for the conditional order based on execution.
- **Direction of Conditional Order by Execution** – The direction of the conditional order based on execution.
- **Activation on Partial Execution** – Partial execution of the order is considered. An "on-execution" order will be activated upon partial execution of the condition order.
- **Executed Volume** – Takes the executed volume of the order as the quantity for placing the stop order. The quantity of securities in an "on-execution" order is taken as the executed volume of the condition order.
- **Price of Linked Order** – The price of the linked limit order.
- **Withdrawal on Partial Execution** – Indicates the withdrawal of the stop order upon partial execution of the linked limit order.
- **Offset from Maximum** – The amount of offset from the maximum (minimum) price of the last transaction.
- **Protective Spread** – The size of the protective spread.
- **Take-Profit at Market Price** – Indicates if the "Take-Profit" order executes at the market price.

Note

Working with orders is a low-level method of managing positions. For higher-level management, it is recommended to use the "Modify Position" component, described in [Modify Position](#).

See Also

[Modify Position](#)

Cancel order

Order Cancellation

[image: Designer Cancellations 00]

This block is used to cancel an order for an instrument.

Incoming Sockets

Incoming Sockets

- **Trigger** – the event that triggers the order cancellation.
- **Order** – the signal used to determine the moment when an order needs to be canceled.

Outgoing Sockets

Outgoing Sockets

- **Order** – the canceled order, which can be used to retrieve transactions for it using the **Transactions** element, as well as to display it on the chart using the **Chart Panel** block.
- **Error** – an error in cancelling the order (for example, the order has already been executed or canceled earlier).

See Also

[Mass Orders Cancel](#)

Modify order

Order Movement

[image: Designer Moving applications 00]

This block is used for modifying an order for an instrument.

Incoming Sockets

Incoming Sockets

- **Trigger** - the signal that determines when to move an order.
- **Order** - the order that will be modified.
- **Price** - numerical value of the new price.
- **Volume** - numerical value of the new volume.

Outgoing Sockets

Outgoing Sockets

- **Order** - the modified order, which can be used to obtain transactions for it using the **Transactions by Order** element and for display on the chart using the **Chart Panel** block.
- **Error** - an error in moving the order.
- **Trade** - the trade for the placed order.

Parameters

- **Zero Price** – a zero price registers a market order.

See Also

[Cancel Order](#)

Mass cancel orders

Mass Order Cancellations

[image: Designer Mass Cancellations 00]

This block is used for canceling all orders for an instrument.

Incoming Sockets

Incoming Sockets

- **Trigger** - the signal that determines the moment when it's necessary to cancel orders.
- **Portfolio** – the portfolio for which all orders need to be canceled.
- **Security** – the instrument for which all orders need to be canceled.

Outgoing Sockets

Outgoing Sockets

- **Result** - a flag signaling the success of the operation.

Parameters

Parameters

- **Direction** – the direction of the orders being canceled (buy or sell), acts as a cancellation signal for the order.

See Also

[Order Registration](#)

Indexer

Indexer

[image: Designer Indexer 00]

The element is used to get an element from a collection with the specified index.

Incoming sockets

Incoming sockets

- **Any data** – the collection of elements.

Outgoing sockets

Outgoing sockets

- **Any data** - the element from the collection with the index specified in the parameters.

Parameters

Parameters

- **Index** - the index of the required element.

Recommended content

[Converter](#)

Converter

[image: Designer Converter 00]

The cube is used to convert complex objects into simple data types. For example, it allows you to get the value of the price step for the instrument.

Incoming sockets

Incoming sockets

- **Any data** - a certain type of complex objects that are received.

Outgoing sockets

Outgoing sockets

- **Any data** - the value of the selected property for the received object.

Parameters

Parameters

- **Data type** - the type of data to be input. The type of the input parameter depends on the selected value.
- **Property** – the tree of selected type properties, which can be obtained at the output of the cube. The set of available properties depends on the selected data type and appears only after selecting the required data type.

Notification

Notification

[image: Designer Notice 00]

The cube sends a notification when data arrives on its input socket. The incoming value is converted to text via ToString. You can connect a [Variable](#) to send fixed text, attach trade or candle streams to see their details, or use [String format](#) and [String concat](#) cubes to prepare a custom message.

Incoming sockets

Incoming sockets

- **Message** - data to send. Any value is accepted and converted to a string.

Parameters

Parameters

- **Type** - type of message (pop-up window, e-mail, sms, etc.). The types of notifications are described in the [Notification settings](#) section.
- **Telegram** - channel used for Telegram notifications.
- **Header** - the header of the message.

Recommended content

[String format](#) [String concat](#) [Notification settings](#)

String concat

String concat

[image: Designer String Concat 00]

The cube concatenates several incoming values into a single text string according to a template with placeholders in curly braces. Each placeholder name adds an input socket with the same name. You can reference nested properties through dots and specify formatting after a colon.

Incoming sockets

Incoming sockets

- Created dynamically from placeholder names. Each socket accepts data of any type.

Outgoing sockets

Outgoing sockets

- **Text** – the concatenated and formatted string.

Parameters

Parameters

- **Template** – string concatenation and formatting template. Editing the template updates the list of input sockets.

Examples

- Template Price: {price:0.00}, Qty: {qty} with price = 10.5 and qty = 2 produces Price: 10.50, Qty: 2.
- Template {time:HH:mm:ss} - {trade.Price} with sockets time and trade (trade.Price = 100) produces 09:15:00 - 100.
- Template {side} {volume} @ {trade.Price} with sockets side = Buy, volume = 1, trade.Price = 100 produces Buy 1 @ 100.

Recommended content

[String format Notification](#)

String format

String format

[image: Designer String Format 00]

The cube converts an incoming value of any type into a text string. The conversion is performed according to a template with placeholders in curly braces. Each placeholder refers to the whole value (`{0}`) or to one of its properties (`{Price}`, `{Trade.Price}`, etc.). You can specify a format after a colon to control how numbers, dates or other objects appear in text.

Incoming sockets

Incoming sockets

- **Input** – the value to be formatted. The socket accepts data of any type.

Outgoing sockets

Outgoing sockets

- **Text** – the result of applying the template to the incoming value.

Parameters

Parameters

- **Template** – string formatting template applied to the incoming value. The default template is `{0}`, meaning the value is inserted without additional formatting. Placeholders may contain property names and format strings, for example `Price: {0:0.00}` or `{Price:0.00}`.

Examples

- Template `Price: {0:0.00}` with input `10.5` produces `Price: 10.50`.
- Template `{Price} - {Volume}` with input trade object `{ Price = 100, Volume = 2 }` produces `100 - 2`.
- Template `{Trade.Price} - {Trade.Volume}` with input object `{ Trade = { Price = 100, Volume = 2 } }` produces `100 - 2`.
- Template `Time: {Time:HH:mm:ss}` with input object with `Time = 2024-05-01T09:15:00` produces `Time: 09:15:00`.

Recommended content

[String concat Notification](#)

Order book

Order book

[image: Designer Order Book 00]

The cube is used to get the order book by the specified instrument.

Incoming sockets

- **Instrument** - the instrument for which you need to get an order book.

Outgoing sockets

- **Order book** - changes of the order book from the moment you subscribe to receive the data (the incoming of the instrument to the input of the element).

Recommended content

[Grouped order book](#)

Grouped order book

Grouped order book

[image: Designer Grouped order book 01]

[image: Designer Grouped order book 00]

The cube is used to obtain a grouped order book.

Incoming sockets

- **Order book** - the order book to be grouped.

Outgoing sockets

- **Order book** - the grouped order book.

Parameters

- **Price range** - the price range in which orders will be grouped.

Recommended content

[Sparse order book](#)

Order book panel

Order Book Panel

[image: Designer Order Book Panel]

The cube is designed to display the order book in a specialized graphical component [Order Book](#).

The [Order Book](#) component should be added from the [Components](#) group of the **Simulation** or **Trading** tabs. More details about the [Order Book](#) component can be found in the section [Order Book](#).

Incoming Sockets

- **Order Book** - the order book that needs to be displayed.
- **Order** - an order, the volume of which needs to be displayed in the *Own Volume* column.
- **Order Error** - an error in the registration or cancellation of an order, for which animation needs to be shown.

See Also

[Grouped Order Book](#)

Sparse order book

Sparse order book

[image: Designer Sparse order book 01]

[image: Designer Sparse order book 00]

The cube is used to obtain a sparse order book for the specified instrument.

Incoming sockets

- Order book - the order book that needs to be thinned.

Outgoing sockets

- Order book - the sparse order book.

Parameters

- **Price Range** - The range of prices, with steps at which the order book will be thinned.

Recommended content

[IV book](#)

Truncated order book

Truncated order book

[image: Designer MarketDepthTruncateDiagramElement 00]

The cube is used to obtain an order book truncated to a specified depth.

Incoming sockets

- **Order book** is an order book that needs to be truncated.

Outgoing sockets

- **Order book** is the truncated order book.

Parameters

- **Depth** – the maximum depth of the order book.

Recommended content

[IV book](#)

IV book

IV book

[image: Designer Glass implied volatility 00]

The cube is used to calculate an order book of implied volatility.

Incoming sockets

Incoming sockets

- **Model** – the calculation model (for example, Black-Scholes).
- **Order book** – the order book.

Outgoing sockets

Outgoing sockets

- **Order book** - the values of the order book of implied volatility.

Recommended content

[Option desk](#)

Option desk

Option desk

[image: Designer Options Board 00]

The cube is used to display the option desk.

To display the **Option desk**, you need to add the **Option desk** graphic component.

Incoming sockets

Incoming sockets

- **Model** – the calculation model (for example, Black-Scholes).

[image: Designer Options Board 01]

[image: Designer Options Board 02]

Recommended content

[Chart positions](#)

Chart positions

Chart positions

[image: Designer Graph options positions 00]

The cube is used to display the **Chart of options positions**.

To display the **Chart of options positions**, you need to add the **Chart of options positions** graphic component.

Incoming sockets

Incoming sockets

- **Model** – the calculation model (for example, Black-Scholes).
- **Price of the underlying asset** - the price of the underlying asset.

[image: Designer Graph options positions 01]

[image: Designer Graph options positions 02]

Recommended content

[Black Scholes](#)

Greeks

Greeks

[image: Designer Greek 00]

This block is used to calculate the main "Greeks": Delta, Gamma, Vega, Theta, Rho at the current moment.

Incoming Sockets

Incoming Sockets

- **Model**  the calculation model (for example, Black-Scholes).
- **Price of the Underlying Asset**  the price of the underlying asset.
- **Maximum Deviation**  the maximum deviation.

Outgoing Sockets

Outgoing Sockets

- **Result**  the result of calculating the main "Greeks": Delta, Gamma, Vega, Theta, Rho at the current moment.

Parameters

Parameters

- **Value**  can take the type of "Greek" Delta, Gamma, Vega, Theta, Rho and determines what value will be output from the block.

See Also

[Hedging](#)

Black Scholes

Black-Scholes

[image: Designer Black Sols 00]

This block is used to create an option pricing model.

Incoming Sockets

Incoming Sockets

- **Option**  the strike for which the model needs to be created.

Outgoing Sockets

Outgoing Sockets

- **Model**  the pricing model (for example, Black-Scholes).

Parameters

Parameters

- **Black Model**  a flag indicating whether to create a Black-Scholes model.

See Also

[Hedging](#)

Basket

Basket

[image: Designer Black Basket 00]

This block is used to create an option pricing model.

Incoming Sockets

Incoming Sockets

- **Options** ♦ the strikes for which the model needs to be created.

Outgoing Sockets

Outgoing Sockets

- **Model** ♦ the pricing model (for example, Black-Scholes).

Parameters

Parameters

- **Black Model** ♦ a flag indicating whether to create a Black-Scholes model.

[image: Designer Black Basket 01]

See Also

[Black-Scholes](#)

Hedging

Hedging

[image: Designer Hedging 00]

The cube is used to hedge positions on options.

Incoming sockets

Incoming sockets

- **Model** – the calculation model (for example, Black-Scholes).
- **Instrument** – the instrument, the underlying asset.
- **Volume** - the numerical value of the volume.
- **Position by underlying asset** – the position by the underlying asset.
- **Flag** – the signal (flag) that initiates the hedging process.

Outgoing sockets

Outgoing sockets

- **Order** – the registered order that can be used to obtain trades on it by using the Trades element by the order and displaying it on the chart using the Chart panel cube

Parameters

Parameters

- **Hedging type** - the hedging type, it can take the values Delta, Gamma, Vega, Theta, or Rho.

Recommended content

[Options quoting](#)

Options quoting

Options quoting

[image: Designer Quoting 00]

The cube is used to quote options according to the specified parameters.

Incoming sockets

Incoming sockets

- **Model** – the calculation model (for example, Black-Scholes).
- **Volume** - the volume of quoting.

Outgoing sockets

Outgoing sockets

- **Order** - the registered order that can be used to obtain trades on it by using the **Trades** element by the order and displaying it on the chart using the **Chart panel** cube.

Parameters

Parameters

- **Quoting** - the parameter by which quoting will be performed. It can take the values **Volatility** (the quoting volume will follow the specified volatility limits) or **Theoretical price** (the quoting volume will follow the specified limits of the theoretical price).
- **Direction** - the direction of quoting can take the values of Purchase and Sell.
- **Minimum** – the minimum value of volatility or theoretical price.
- **Maximum** - the maximum value of volatility or theoretical price.

Recommended content

[Derivatives](#)

Strikes

[image: Designer Derivatives 00]

The cube is used to obtain a list of options by a specified filter.

Incoming sockets

Incoming sockets

- **Instrument** – the instrument, the underlying asset.

Outgoing sockets

Outgoing sockets

- **Options** - the list of options by the underlying asset.

Parameters

Parameters

- **Option type** – the option type can be Call option or Put option.
- **Expiry date** - expiry date of option.
- **Strike (less)** – the shift left (less) from the central strike. If the price is not set, then all strikes with a lower central value will be used. Shift is calculated in the strike steps, for example, if the strike step is 500 c.u., then the shift equal to 3 will be 1,500 c.u.
- **Strike (more)** – the shift to the right (more) from the central strike. If the price is not set, then all strikes with a larger central value will be used. Shift is calculated in the strike steps, for example, if the strike step is 500 c.u., then the shift equal to 3 will be 1,500 c.u.

Recommended content

[Crossing](#)

Lines

Lines

The strategy in the designer is a schema of a set of elements and links between them, called connections. Each connection goes from the output parameter of one cube to the input parameter of another cube. Usually, all connection lines are colored gray, but when you point to the cube to which they belong, the lines are painted black.

[image: Designer Line 00]

Each connection can be highlighted by pointing over it and clicking the left mouse button. The selected connection will be marked with circles at the ends of the line, taking for which, you can redirect the line. If you press the Del button on the selected line, it will be deleted.

You can connect to each other the parameters of the same colors (the same data types), except for the following types of parameters:

- The **black** parameter can accept any data. Most often, these parameters are used to pass signals for any actions within the element. For example, the [Variable](#) element stores some value and, when it receives a signal, it passes the value to the output.
- The **green** parameter can accept different compared data types. For example, numeric, indicator values, strings, etc.

Recommended content

[Event model](#)

Event model

Event model

The approach of building schemas in [Designer](#) is based on the generation and subsequent processing of events. When building a strategy, it is not known when an event of the market data change will occur, but you can subscribe to this event and process it accordingly.

Each [Designer](#) cube that has an output parameter is the event generator. And cubes having an input parameter can subscribe to the event generated by the outgoing parameter. Subscribing for an event is nothing more than creating a connection line between two cubes.

For example, the [Order book](#) cube generates an order book change event. It is not known in advance when a change will occur. Having created a connection line between the [Order book](#) cube and the [Converter](#) cube, a subscription is made to the order book change for further processing with the [Converter](#) cube, etc.:

[image: Designer Event model 00]

Recommended content

[First strategy](#)

First strategy

First strategy

To create schemas of strategies and composite elements, and to test the obtained strategies on historical data, you can use an example of the strategy of moving average (SMA). It allows you to go through a complete cycle from creating a strategy to its testing and debugging. The strategy of moving average (SMA) can be found in the **Strategies** folder of the **Schemas** panel.

1. Creating a new strategy from the cubes as described in [Using Code](#). To add a new strategy by click the **Add** *[image: Designer Panel Circuits 01]* button in the **Common** tab and selecting **Strategy**. Or by right-click the **Strategy** folder in the **Schemas** panel, and to click the **Add** *[image: Designer Panel Circuits 01]* button in the drop-down menu.

[image: Designer The creation of a strategy 00]

After clicking the **Add** *[image: Designer Panel Circuits 01]* button in the **Strategy** folder of the **Schemas** panel, a new strategy will appear. In the workspace, a new tab with a strategy appears, when you switch to it, the **Emulation** tab will automatically open in the ribbon. In the **Emulation** tab, you can change the name of the strategy and give it a brief description.

[image: Designer The creation of a strategy 01]

2. For convenient work it is necessary to open and fix the **Palette** and **Properties** panels of the **Schemas** area by clicking the *[image: Designer Algorithm creation of cubes 13]* button. The result is a window of the following type.

[image: Designer Algorithm creation of cubes 00]

3. The essence of the strategy of moving average (SMA) is as follows:
 - There are two moving averages with different calculation periods, a long SMA and a short SMA. In the example, the **Indicator** cube of the long SMA is called Long SMA with a period of 80 candles, a short SMA is called Short SMA with a period of 10 candles.
 - When a short moving average crosses a long one bottom-up, open a long position.
 - When a short moving average crosses a long one top-down, open a short position.
 - If there is an opposite position at the moment of receiving a signal to open a position, turn around the position.
4. For all strategies, you need an instrument and a portfolio, which will be used for trades. You should add them from the **Palette** panel to the **Designer** panel. In the example, the **Variable** cube with the **Instrument** type is called the Instrument, the **Variable** cube with the **Portfolio** type is called the Portfolio. Set the **Parameters** check box of Instrument and Portfolio cubes. When the check box is selected, the cube will take value from the strategy settings. If you do not select the check box, you should manually enter the instrument and portfolio values. If you leave the Value field of the **Variable** cube empty and do not set check box of parameters, the strategy during testing will give an error out about the unset value of the **Variable** cube.

[image: Designer Algorithm creation of cubes 01]

If you need to use several instruments or portfolios in the strategy, then for each cube you should uncheck the **Parameters** box and set the value of the instrument or portfolio.

[image: Designer Algorithm creation of cubes 02]

[image: Designer Algorithm creation of cubes 03]

5. After adding the instrument and portfolio, you should add two **Indicator** cubes, select the SMA type,

name the first Long SMA, set the period of 80 candles, name the second Short SMA, set the period of 10 candles.

[image: Designer Algorithm creation of cubes 04]

6. For the indicators to work, it is necessary to pass to them a series of candles. To do this, you need to create the [Candles](#) cube. In the example, only formed candles with a timeframe of 5 minutes are used.

[image: Designer Algorithm creation of cubes 05]

7. After adding the indicators, you need to add two cubes that define the intersections of the indicators. These are the [Crossing](#) cubes from the composite elements. The first cube is called Crossing Up. It defines the intersection from bottom to top. The Short SMA indicator is passed to the upper input of the cube, and the Long SMA indicator to the lower input. The CurrComparison operator is set to a value larger, the PrevComparison operator is set to less than or equal to. The second cube is called Crossing Down, it defines the intersection from top to bottom. The Short SMA indicator is passed to the upper input of the cube, and the Long SMA indicator to the lower input. The CurrComparison operator is set to a value less, the PrevComparison operator is set to greater than or equal to.

[image: Designer Algorithm creation of cubes 06]

8. It is worth adding the [Chart](#) for a visual display of candles, indicators and trades. We will add to the [Chart](#) the elements of the display, candles, two indicators and trades.

[image: Designer Algorithm creation of cubes 07]

9. As a source of trades for display on the chart the **Trades** cube of the strategy is used. The example is called Strategy trades.

[image: Designer Algorithm creation of cubes 08]

10. To open a position, add two [Register order](#) cubes. The first cube is for the purchase by a market order. The following are passed to the input of this cube: the **Instrument**, the signal for opening a position from the Crossing Up intersection cube, the **Portfolio** and the volume of the order. The second cube is for sale by a market order. The following are passed to the input of this cube: the **Instrument**, the signal for opening a position from the Crossing Down intersection cube, the **Portfolio** and the volume of the order.

[image: Designer Algorithm creation of cubes 09]

11. By connecting the above elements with lines ([Lines](#)), a schema is obtained without taking into account the current position of the strategy. In such condition, it will gain an excessive amount of lots.

[image: Designer Algorithm creation of cubes 10]

To control the position, you need to add the [Position](#), cube to the input of which **Instrument** and **Portfolio** are passed.

[image: Designer Algorithm creation of cubes 11]

To process the current position, you can use the ready-made schema described in the [Get current position](#) section. This schema determines the actual value of the required volume for the registering in the order. And if it is necessary to reverse the position, it will give twice the value of the portfolio.

12. As a result, the completed strategy looks like:

[image: Designer Algorithm creation of cubes 12]

Recommended content

[Composite elements](#)

Composite elements

Composite elements

When composing schemas, there are often sets of elements that form a complete functionality and can be used in different schemas or in one schema many times with different values of properties. Such sets of elements can be put into a separate composite element, which will be used further as any ordinary cube.

A composite element is a usual schema that is saved/loaded/edited as any strategy schema.

When you add a composite element to a schema, all unconnected parameters of all inner cubes are automatically added to it. Unconnected parameters at the input of cubes are added as input, unconnected parameters at the output as output. Each added parameter is named in the same way as the source element and its parameter. In addition, for this element, the properties of all elements for which the **Parameters** property was specified are added.

We will consider the use of composite elements in the example of the moving average intersection strategy, which illustrates the use of the [Crossing](#) composite element several times. The strategy can open a long position when the short moving average crosses the long one bottom-up, and a short position when the short moving average crosses the long one top-down. The schema of the part of the moving averages intersection strategy, where the moment of moving averages intersection is determined, is shown in the figure below:

[image: Designer Creating a composite elements 00]

Because moving average intersection differs only in its possible direction (short crosses from top to bottom or from bottom to top), the part of the schema that determines the moment of intersection can be taken out into a separate composite element. When you add this element to the schema, you specify the properties that define the moving average intersection algorithm. The schema of the composite element by which the intersection is determined is shown in the figure below:

[image: Designer Crossing 01]

The diagram of the composite element consists of simple elements and is based on memorizing the current values (Prev In 1 and Prev In 2) and comparing the pairs, current (CurrComparison) and previous (PrevComparison) values with each other. Because each of the input values is used in two elements of the diagram, the elements of [Combination](#) (In 1, In 2) are placed at the input of the composite element, and they allow one input to be divided into two elements and pass the input value to the [Comparison](#) and [Prev value](#) elements. When a new value arrives at the input, the current values are compared and a new value is passed to the [Prev value](#) element, from which the previous value for the current input is passed, then the previous values are compared. If both conditions are fulfilled, which is checked using the And [Logical condition](#), then the value of the raised flag is passed to the output of the composite element, which can be used as a trigger for further action.

For the CurrComparison and PrevComparison cubes, the **Parameters** flag of the **Common** properties group is set. Therefore, the properties of these cubes were taken into the properties of the composite element [Crossing](#), which will be further specified when using a composite element in the strategy schema.

[image: Designer Crossing 00]

Recommended content

[Display candles on chart](#)

Display candles on chart

Display candles on chart

To output candles by an instrument to a chart, the following schema can be used:

[image: Designer The conclusion of the candles on the chart 00]

For the [Variable](#) cube, the **Instrument** data type is selected. If the instrument is not specified, but the **Parameters** flag of the **Common** properties group is set, it will be taken from the strategy and passed to the [Candles](#) cube. For the [Candles](#) cube, the settings for building 5-minute candles and passing only fully formed candles are specified.

For the [Chart](#) cube, one graphic element with a candle type was added, for which the input parameter was automatically added.

After adding the necessary graphic elements to the chart panel, the connection of the [Candles](#) and [Chart](#) elements is added, through which the built candles will be passed for output to the chart.

Recommended content

[Get best price for instrument](#)

Get best price for instrument

Get best price for instrument

To register a purchase order at the current best price for the instrument, the following schema can be used:

[image: Designer Get the best rates for the tool 00]

For the [Variable](#) cube, the **Instrument** data type is selected. If the instrument is not specified, but the **Parameters** flag of the **Common** property group is set, then it will be taken from the strategy and passed to the [Order book](#) cube. The [Order book](#) cube, after receiving the current instrument from the variable, passes the order book changes by the selected instrument through the output parameter. When receiving order book changes, the [Converter](#) cube chooses the current value of the best purchase price from them.

Recommended content

[Get current position](#)

Get current position

Get current position

To get the volume required for flipping the current position to the opposite position, the schema from the SMA strategy example can be used:

[image: Designer Determination of the volume position 00]

The **Instrument** data type is selected for the **Variable** cube. If the instrument is not specified, but the **Parameters** flag of the **Common** group is set, then it will be taken from the strategy, and then it will be passed to the **Position**.

For the **Position** cube, the position property is also not specified, but the **Parameters** flag of the **Common** group is set, which means that the position will be obtained for the portfolio that is specified in the strategy settings.

After passing the instrument and changing the position, using the mathematical function with one argument ($\text{abs}(\text{pos})$), the absolute value is calculated and a signal is given for the cube of variable (2). This cube contains a factor of 2, to pass the stored value through the output parameter, after which, using a mathematical formula with two arguments ($\text{abs}(\text{pos}) * 2$), their product is calculated. Next, using the composite cube Conditional operator ($\text{pos} == 0 ? 1 : \text{pos}$), the actual value of the required volume is determined, which can differ from the current position value multiplied by 2. For example, at the time of the strategy start, when no orders have been executed yet. In this case, the Conditional statement element returns a default value of 1. Because one output parameter can be connected only once with the input parameter of another element, to pass the same value between the formula and the conditional operator, an additional **Combination** cube is added.

Recommended content

[Get order book price level](#)

Get order book price level

Get order book price level

To get the required purchase string from the order book, the following schema can be used:

[image: Designer Event model 00]

The **Instrument** data type is selected for the [Variable](#) cube. If the instrument is not specified, but the **Parameters** flag of the **Common** property group is set, then it will be taken from the strategy. For the [Converter](#) cube, the data type and the corresponding field of the cubes collection for Bids purchase are selected. The indexer cube gets the required element from the collection of the best prices for the purchase. To get a certain price or volume value at a level, you can use the [Converter](#) cube.

Recommended content

[Gallery](#)

Using code

Programming Languages

[Designer](#) supports three programming languages for creating trading strategies:

- [C#](#)
- [F#](#)
- [Python](#)

Extensive strategy samples in C# and Python are available in the [AlgoTrading API folder](#). All of them work with the [Designer](#).

C#

C# is the main language for developing trading strategies. All components, examples and core code are written in C#. C# provides full access to the entire [S#](#) platform architecture.

F#

F# is a functional programming language from Microsoft. Trading strategies in F# have full access to all API capabilities and can be used for both testing and live trading.

Python

Python integration allows writing trading strategies using the Python programming language, making it easier for Python developers to work with the platform.

Using C#

Creating strategies from code is for users who prefer working with C# code. Such strategies are not limited in capabilities unlike diagrams, and any algorithm can be described.

The process of creating a strategy takes place directly in [Designer](#) or a C# development environment (the most popular of which are **Visual Studio** and **JetBrains Rider**), using a library for professional development of trading robots in C# and [API](#).

You can add a new strategy by pressing the **Add** *[image: Designer Panel Circuits 01]* button on the **Common** tab and choose **Strategy**. Or, by right-clicking on the **Strategies** folder in the **Scheme** panel, and pressing the **Add** *[image: Designer Panel Circuits 01]* button in the dropdown menu:

[image: Designer The creation of a strategy 00]

After pressing the **Add** *[image: Designer Panel Circuits 01]* button, a window will appear with a choice of content type to create the strategy on:

[image: Designer_Creation_of_element_containing_source_code_00]

To create a strategy from C# code, you need to select the second tab. You can also choose a template that will be used as the initial code.

After pressing **OK**, a new strategy will appear in the **Strategies** folder of the **Scheme** panel, similarly to when creating a strategy from [diagrams](#). And similar actions for deleting or renaming the strategy.

But instead of a diagram, a C# code editor will be displayed:

[image: Designer_Creation_of_element_containing_source_code_01]

The code editor tab consists of **Source Code** and **Error List** panels. The **Source Code** panel contains the C# code editor itself. At the top, there is a toolbar where highlighting such things as **Current Line**, **Line Number**, etc., can be turned on or off. To increase the font size, you can use the CTRL+MouseWheel combination.

The **Error List** panel is a table with a list of errors in the code, double-clicking on a line will automatically move the cursor in the **Source Code** panel to the error location.

When editing the code, an icon *[image: Designer The creation of the cube containing the source code 03]* will appear in the bottom right corner of the **Error List** panel, indicating that change tracking has begun. The code compiles at the moment when the code stops changing.

Running the strategy on [backtest](#), on [live](#), and other operations are similar to the strategy from diagrams.

First strategy

C# Strategy Example

Creating a strategy from source code will be demonstrated using the SMA strategy example - similar to the SMA strategy example assembled from cubes in the [Creating Algorithm from Cubes](#) section.

This section will not describe C# language constructs (or [Strategy](#), which is used as the base for creating strategies), but will mention specific features for working with code in the **Designer**.

Strategies created in the **Designer** are compatible with strategies created in the [API](#) due to using the common base class [Strategy](#). This makes running such strategies outside the **Designer** significantly easier than running [schemes](#).

1. Strategy parameters are created using a special approach:

```
_candleTypeParam = this.Param(nameof(CandleType), DataType.TimeFrame(TimeSpan.FromMinutes(1)));
_long = this.Param(nameof(Long), 80);
_short = this.Param(nameof(Short), 20);

private readonly StrategyParam<DataType> _candleTypeParam;

public DataType CandleType
{
    get => _candleTypeParam.Value;
    set => _candleTypeParam.Value = value;
}

private readonly StrategyParam<int> _long;

public int Long
{
    get => _long.Value;
    set => _long.Value = value;
}

private readonly StrategyParam<int> _short;

public int Short
{
    get => _short.Value;
    set => _short.Value = value;
}
```

When using the [StrategyParam](#) class, the settings save and restore approach is automatically used.

2. When creating indicators and subscribing to market data, you need to bind them so that incoming data from the subscription can update indicator values:

```
// ----- create indicators -----

var longSma = new SMA { Length = Long };
var shortSma = new SMA { Length = Short };

// -----

// --- bind candles set and indicators ---

var subscription = SubscribeCandles(CandleType);

subscription
    // bind indicators to the candles
```

```
.Bind(longSma, shortSma, OnProcess)
// start processing
.Start();
```

- When working with charts, it's important to consider that when running the strategy [outside the Designer](#), the chart object might be absent.

```
var area = CreateChartArea();

// area can be null in case of no GUI (strategy hosted in Runner or in own console app)
if (area != null)
{
    DrawCandles(area, subscription);

    DrawIndicator(area, shortSma, System.Drawing.Color.Coral);
    DrawIndicator(area, longSma);

    DrawOwnTrades(area);
}
```

- Start position protection through [StartProtection](#) if required by the strategy logic:

```
// start protection by take profit and-or stop loss
StartProtection(TakeValue, StopValue);
```

- The strategy logic itself, implemented in the OnProcess method. The method is called by the subscription created in step 1:

```
private void OnProcess(ICandleMessage candle, decimal longValue, decimal shortValue)
{
    LogInfo(LocalizedStrings.SmaNewCandleLog, candle.OpenTime, candle.OpenPrice, candle.HighPrice,
    candle.LowPrice, candle.ClosePrice, candle.TotalVolume, candle.SecurityId);

    // in case we subscribed on non finished only candles
    if (candle.State != CandleStates.Finished)
        return;

    // calc new values for short and long
    var isShortLessThenLong = shortValue < longValue;

    if (_isShortLessThenLong == null)
    {
        _isShortLessThenLong = isShortLessThenLong;
    }
    else if (_isShortLessThenLong != isShortLessThenLong)
    {
        // crossing happened

        // if short less than long, the sale, otherwise buy
        var direction = isShortLessThenLong ? Sides.Sell : Sides.Buy;

        // calc size for open position or revert
        var volume = Position == 0 ? Volume : Position.Abs().Min(Volume) * 2;

        var priceStep = GetSecurity().PriceStep ?? 1;

        // calc order price as a close price + offset
        var price = candle.ClosePrice + (direction == Sides.Buy ? priceStep : -priceStep);

        if (direction == Sides.Buy)
            BuyLimit(price, volume);
        else
            SellLimit(price, volume);

        // store current values for short and long
```

```
    _isShortLessThenLong = isShortLessThenLong;  
  }  
}
```

External DLLs

External DLLs

When creating content from **Source Code**, along with [API](#) libraries, you can use other C# libraries, including your own. These libraries must be compatible with .NET 6.0. To add a reference to a library, you need to click on the **References** button in the **Backtest** tab. After that, the **Extra references** window will open, where you can add or remove the necessary libraries:

[image: Designer Add Dll Lib]

Creating Your Own Cube

Creating Your Own Cube

Similar to creating a [cube from a diagram](#), you can create your own cube based on C# code. Such a cube will be more functional than a cube from a diagram.

To create a cube from code, it needs to be created in the **Own elements** folder:

[image: Designer_Source_Code_Elem_00]

In the example provided below, the cube inherits from the [DiagramExternalElement](#) class, and looks as follows:

```
/// <summary>
/// Sample diagram element demonstrates input and output sockets usage.
///
/// https://doc.stocksharp.com/topics/Designer_Combine_Source_code_and_standard_elements.html
/// </summary>
public class EmptyDiagramElement : DiagramExternalElement
{
    private readonly DiagramElementParam<int> _minValue;

    public EmptyDiagramElement()
    {
        // example property to show how to make parameters

        _minValue = AddParam("MinValue", 10)
            .SetBasic(true) // make parameter visible in basic mode
            .SetDisplay("Parameters", "Min value", "Min value parameter description", 10);
    }

    // output sockets are events marked with DiagramExternal attribute

    [DiagramExternal]
    public event Action<Unit> Output1;

    [DiagramExternal]
    public event Action<Unit> Output2;

    // input sockets are method parameters marked with DiagramExternal attribute

    // uncomment to get Process method called every time when new arg received
    // (no need wait when all input args received)
    //public override bool WaitAllInput => false;

    [DiagramExternal]
    public void Process(CandleMessage candle, Unit diff)
    {
        var res = candle.ClosePrice + diff;

        if (diff >= _minValue.Value)
            Output1?.Invoke(res);
        else
            Output2?.Invoke(res);
    }

    public override void Start()
    {
        base.Start();

        // add logic before start
    }

    public override void Stop()
```



```

{
    base.Stop();

    // add logic after stop
}

public override void Reset()
{
    base.Reset();

    // add logic for reset internal state
}
}

```

In this code, the cube has two incoming sockets and two outgoing sockets. Incoming sockets are defined by applying the [DiagramExternalAttribute](#) attribute to the method:

```

[DiagramExternal]
public void Process(CandleMessage candle, Unit diff)

```

Outgoing sockets are defined by applying the attribute to an event. In the example of the cube, there are two such events:

```

[DiagramExternal]
public event Action<Unit> Output1;

[DiagramExternal]
public event Action<Unit> Output2;

```

Therefore, there will also be two outgoing sockets.

Additionally, it shows how to make a property for the cube:

```

_minValue = AddParam("MinValue", 10)
    .SetBasic(true) // make parameter visible in basic mode
    .SetDisplay("Parameters", "Min value", "Min value parameter description", 10);

```

Using the [DiagramElementParam](#) class automatically employs the approach to save and restore settings.

The **MinValue** property is marked as basic, and it will be visible in the [Basic properties](#) mode.

The commented [WaitAllInput](#) property is responsible for the timing of the method call with incoming sockets:

```

//public override bool WaitAllInput => false;

```

If uncommented, the **Process** method will always be called as soon as at least one value arrives (in the example case, this is either a candle or a numerical value).

To add the resulting cube to the diagram, you need to select the created cube in the palette in the **Own elements** section:

[image: Designer_Source_Code_Elem_01]

Cubes from C# code cannot be used in strategies created in C# code. They can only be used in strategies created [from cubes](#).

See Also

[Creating an Indicator](#)

Create own indicator

Creating an Indicator in C#

Creating your own indicator in [API](#) is described in the section [Custom Indicator](#). Such indicators are fully compatible with **Designer**.

To create an indicator, on the **Scheme** panel you need to select the **Indicators** folder, right-click and in the context menu select **Add**:

[image: Designer_Source_Code_Indicator_00]

The indicator code will look like this:

```
/// <summary>
/// Sample indicator demonstrating to save and load parameters.
///
/// Changes input price on +20% or -20%.
///
/// See more examples https://github.com/StockSharp/StockSharp/tree/master/Algo/Indicators
///
/// Doc https://doc.stocksharp.com/topics/Designer\_Creating\_indicator\_from\_source\_code.html
/// </summary>
public class EmptyIndicator : BaseIndicator
{
    private int _change = 20;

    public int Change
    {
        get => _change;
        set
        {
            _change = value;
            Reset();
        }
    }

    private int _counter;
    // formed indicator received all necessary inputs for be available for trading
    private bool _isFormed;

    protected override bool CalcIsFormed() => _isFormed;

    public override void Reset()
    {
        base.Reset();

        _isFormed = default;
        _counter = default;
    }

    protected override IIndicatorValue OnProcess(IIndicatorValue input)
    {
        // every 10th call try return empty value
        if (RandomGen.GetInt(0, 10) == 0)
            return new DecimalIndicatorValue(this);

        if (_counter++ == 5)
        {
            // for example, our indicator needs 5 inputs for become formed
            _isFormed = true;
        }

        var value = input.GetValue<decimal>();
```

```

// random change on +20% or -20% current value

value += value * RandomGen.GetInt(-Change, Change) / 100.0m;

return new DecimalIndicatorValue(this, value)
{
    // final value means that this value for the specified input
    // is not changed anymore (for example, for candles that changes with last price)
    IsFinal = RandomGen.GetBool()
};
}

// persist our properties to save for further the app restarts

public override void Load(SettingsStorage storage)
{
    base.Load(storage);
    Change = storage.GetValue<int>(nameof(Change));
}

public override void Save(SettingsStorage storage)
{
    base.Save(storage);
    storage.SetValue(nameof(Change), Change);
}

public override string ToString() => $"Change: {Change}";
}

```

This indicator receives an incoming value and makes an arbitrary deviation based on the **Change** parameter value.

The description of the indicator methods is available in the section [Custom Indicator](#).

To add the created indicator to the diagram, you need to use the [Indicator](#) cube, and in it, specify the necessary indicator:

[image: Designer_Source_Code_Indicator_01]

The **Change** parameter, previously set in the indicator code, is shown in the properties panel.

Indicators from C# code cannot be used in strategies created in C# code. They can only be used in strategies created [from cubes](#).

Draw order book

Drawing Order Book

A strategy from code can draw data on the [Order Book](#) panel similarly to the [Order Book](#) cube. For this, the following code needs to be written.

1. Create a descendant of the [IOrderBookSource](#) interface, which **Designer** uses to identify the source. In the example case, the [OrderBookSource](#) class is used, which is the default implementation of the interface:

```
private static readonly OrderBookSource _bookSource = new OrderBookSource("SMA");
```

2. Override the [OrderBookSources](#) property:

```
public override IEnumerable<IOrderBookSource> OrderBookSources  
=> new[] { _bookSource };
```

Thus, the strategy will indicate to external code (in this case, the [Order Book](#) panel) which sources of order books are available. The multitude of sources occurs in the case when the strategy works with several order books (different instruments, or order books with various modifications, such as, for example, [thinned order book](#)).

3. Add the initialization of the subscription to the order book in the strategy code. In the case of SmaStrategy, it is added to the end of the [OnStarted](#) method:

```
var bookSubscription = new Subscription(DataType.MarketDepth, Security);  
  
bookSubscription  
    .WhenOrderBookReceived(this)  
    .Do(book =>  
    {  
        // drawing order book  
        DrawOrderBook(bookSubscription, _bookSource, book);  
    })  
    .Apply(this);  
  
Subscribe(bookSubscription);
```

In the Do handler, a call is made to the [DrawOrderBook](#) method, which sends the order book for drawing.

4. Add the [Order Book](#) panel and select the source created in code:

[image: Designer_Source_Code_OrderBook_00]

5. After launching the strategy for testing, the order book will be filled with data:

[image: Designer_Source_Code_OrderBook_01]

Using Visual Studio or Rider

Using Visual Studio or Rider

When using **Visual Studio** and **JetBrains Rider** to create code that will run in **Designer**, the only option is constant [file importing](#). This is due to the special file structure of files created by **Designer**, making it impossible to directly modify the files, even in the case of strategies created in C#. Therefore, if external programs are needed for writing code, it is recommended to use the approach with [dll files](#).

F#

Using F#

Creating strategies from code is for users who prefer working with F# code. Such strategies are not limited in capabilities unlike diagrams, and any algorithm can be described.

The process of creating a strategy takes place directly in [Designer](#) or a **F#** development environment (the most popular of which are **Visual Studio** and **JetBrains Rider**), using a library for professional development of trading robots in **F#** and [API](#).

You can add a new strategy by pressing the **Add** *[image: Designer Panel Circuits 01]* button on the **Common** tab and choose **Strategy**. Or, by right-clicking on the **Strategies** folder in the **Scheme** panel, and pressing the **Add** *[image: Designer Panel Circuits 01]* button in the dropdown menu:

[image: Designer The creation of a strategy 00]

After pressing the **Add** *[image: Designer Panel Circuits 01]* button, a window will appear with a choice of content type to create the strategy on:

[image: Designer_Creation_of_element_containing_source_code_00]

To create a strategy from F# code, you need to select the second tab. You can also choose a template that will be used as the initial code.

After pressing **OK**, a new strategy will appear in the **Strategies** folder of the **Scheme** panel, similarly to when creating a strategy from [diagrams](#). And similar actions for deleting or renaming the strategy.

But instead of a diagram, an F# code editor will be displayed:

[image: Designer_Creation_of_element_containing_source_code_01]

The code editor tab consists of **Source Code** and **Error List** panels. The **Source Code** panel contains the F# code editor itself. At the top, there is a toolbar where highlighting such things as **Current Line**, **Line Number**, etc., can be turned on or off. To increase the font size, you can use the CTRL+MouseWheel combination.

The **Error List** panel is a table with a list of errors in the code, double-clicking on a line will automatically move the cursor in the **Source Code** panel to the error location.

When editing the code, an icon *[image: Designer The creation of the cube containing the source code 03]* will appear in the bottom right corner of the **Error List** panel, indicating that change tracking has begun. The code compiles at the moment when the code stops changing.

Running the strategy on [backtest](#), on [live](#), and other operations are similar to the strategy from diagrams.

First strategy

F# Strategy Example

Creating a strategy from source code will be demonstrated using the SMA strategy example - similar to the SMA strategy example assembled from cubes in the [Creating Algorithm from Cubes](#) section.

This section will not describe F# language constructs (or [Strategy](#), which is used as the base for creating strategies), but will mention specific features for working with code in the **Designer**.

Strategies created in the **Designer** are compatible with strategies created in the [API](#) due to using the common base class [Strategy](#). This makes running such strategies outside the **Designer** significantly easier than running [schemes](#).

1. Strategy parameters are created using a special approach:

```
// --- Strategy parameters: CandleType, Long, Short, TakeValue, StopValue ---

// Parameter for the candle type
let candleTypeParam =
    this.Param<DataType>(nameof(this.CandleType), DataType.TimeFrame(TimeSpan.FromMinutes 1.0))
    .SetDisplay("Candle type", "Candle type for strategy calculation.", "General")

// Parameter for the long SMA
let longParam =
    this.Param<int>(nameof(this.Long), 80)

// Parameter for the short SMA
let shortParam =
    this.Param<int>(nameof(this.Short), 30)

// Parameter for take profit
let takeValueParam =
    this.Param<Unit>(nameof(this.TakeValue), Unit(0m, UnitTypes.Absolute))

// Parameter for stop loss
let stopValueParam =
    this.Param<Unit>(nameof(this.StopValue), Unit(2m, UnitTypes.Percent))

// Flag to track SMA crossing
let mutable isShortLessThenLong : bool option = None

// ----- Public properties -----

/// <summary>The candle type used by the strategy.</summary>
member this.CandleType
    with get () = candleTypeParam.Value
    and set value = candleTypeParam.Value <- value

/// <summary>The period for the long SMA indicator.</summary>
member this.Long
    with get () = longParam.Value
    and set value = longParam.Value <- value

/// <summary>The period for the short SMA indicator.</summary>
member this.Short
    with get () = shortParam.Value
    and set value = shortParam.Value <- value

/// <summary>Take profit value.</summary>
member this.TakeValue
    with get () = takeValueParam.Value
    and set value = takeValueParam.Value <- value
```

```

/// <summary>Stop loss value.</summary>
member this.StopValue
    with get () = stopValueParam.Value
    and set value = stopValueParam.Value <- value

```

When using the [StrategyParam](#) class, the settings save and restore approach is automatically used.

2. When creating indicators and subscribing to market data, you need to bind them so that incoming data from the subscription can update indicator values:

```

// ----- Create indicators -----
let longSma = SMA()
longSma.Length <- this.Long

let shortSma = SMA()
shortSma.Length <- this.Short
// -----

// ----- Subscribe to the candle flow and bind indicators -----
let subscription = this.SubscribeCandles(this.CandleType)

// Bind our indicators to the subscription and assign the processing function
subscription
    .Bind(longSma, shortSma, fun candle longV shortV -> this.OnProcess(candle, longV, shortV))
    .Start() |> ignore

```

3. When working with charts, it's important to consider that when running the strategy [outside the Designer](#), the chart object might be absent.

```

// ----- Configure chart -----
let area = this.CreateChartArea()

// area can be null in case there is no GUI (e.g., Runner or console app)
if not (isNull area) then
    // Draw candles
    this.DrawCandles(area, subscription) |> ignore

    // Draw indicators
    this.DrawIndicator(area, shortSma, System.Drawing.Color.Coral) |> ignore
    this.DrawIndicator(area, longSma) |> ignore

    // Draw own trades
    this.DrawOwnTrades(area) |> ignore

```

4. Start position protection through [StartProtection](#) if required by the strategy logic:

```

this.StartProtection(this.TakeValue, this.StopValue)

```

5. The strategy logic itself, implemented in the OnProcess method. The method is called by the subscription created in step 1:

```

member private this.OnProcess
(
    candle: ICandleMessage,
    longValue: decimal,
    shortValue: decimal
) =
    // Log candle information
    this.LogInfo(
        LocalizedStrings.SmaNewCandleLog,
        candle.OpenTime,
        candle.OpenPrice,
        candle.HighPrice,

```



```

    candle.LowPrice,
    candle.ClosePrice,
    candle.TotalVolume,
    candle.SecurityId
)

// Skip if the candle is not finished
if candle.State <> CandleStates.Finished then
    ()
else
    // Determine if short SMA is less than long SMA
    let shortLess = shortValue < longValue

    match isShortLessThenLong with
    | None ->
        // First time: just remember the current relation
        isShortLessThenLong <- Some shortLess
    | Some prevValue when prevValue <> shortLess ->
        // A crossing has occurred
        // If short < long, that means Sell, otherwise Buy
        let direction =
            if shortLess then
                Sides.Sell
            else
                Sides.Buy

        // Calculate volume for opening a new position or reversing
        // If there is no position, use Volume; otherwise, double
        // the minimum of the absolute position size and Volume
        let vol =
            if this.Position = 0m then
                this.Volume
            else
                (abs this.Position |> min this.Volume) * 2m

        // Get price step for the limit price
        let priceStep =
            let step = this.GetSecurity().PriceStep
            if step.HasValue then step.Value else 1m

        // Set the limit order price slightly higher/lower than the current close price
        let limitPrice =
            match direction with
            | Sides.Buy -> candle.ClosePrice + priceStep
            | Sides.Sell -> candle.ClosePrice - priceStep
            | _ -> candle.ClosePrice // should not occur

        // Send limit order
        match direction with
        | Sides.Buy -> this.BuyLimit(limitPrice, vol) |> ignore
        | Sides.Sell -> this.SellLimit(limitPrice, vol) |> ignore
        | _ -> ()

        // Update the tracking flag
        isShortLessThenLong <- Some shortLess
    | _ ->
        // Do nothing if no crossing
        ()

```

Creating Your Own Cube

Creating Your Own Cube

Similar to creating a [cube from a diagram](#), you can create your own cube based on F# code. Such a cube will be more functional than a cube from a diagram.

To create a cube from code, it needs to be created in the **Own elements** folder:

[image: Designer_Source_Code_Elem_00]

In the example provided below, the cube inherits from the [DiagramExternalElement](#) class, and looks as follows:

```
/// <summary>
/// Sample diagram element demonstrating input and output sockets usage.
///
/// See more details:
/// https://doc.stocksharp.com/topics/designer/strategies/using_code/fsharp/
/// creating_your_own_cube.html
/// </summary>
type EmptyDiagramElement() as this =
    inherit DiagramExternalElement()

    // Example property showing how to create parameters
    let minValueParam =
        this.AddParam<int>("MinValue", 10)
        .SetBasic(true) // make the parameter visible in basic mode
        .SetDisplay("Parameters", "Min value", "Min value parameter description", 10)

    // Output sockets are events marked with DiagramExternal attribute
    let output1Event = new Event<Unit>()
    let output2Event = new Event<Unit>()

    [<CLIEvent>]
    [<DiagramExternal>]
    member this.Output1 = output1Event.Publish

    [<CLIEvent>]
    [<DiagramExternal>]
    member this.Output2 = output2Event.Publish

    // Uncomment the following property if you want the Process method
    // to be called every time when a new argument is received
    // (no need to wait for all input args to be received).
    //
    // override this.WaitAllInput
    //     with get () = false

    // Input sockets are method parameters marked with DiagramExternal attribute

    [<DiagramExternal>]
    member this.Process(candle: CandleMessage, diff: Unit) =
        let res = candle.ClosePrice + diff

        if diff >= minValueParam.Value then
            // Trigger the first output event
            output1Event.Trigger(res)
        else
            // Trigger the second output event
            output2Event.Trigger(res)

    override this.Start() =
        base.Start()
        // Add logic before start if needed
```

```

override this.Stop() =
    base.Stop()
    // Add logic after stop if needed

override this.Reset() =
    base.Reset()
    // Add logic for resetting internal state if needed

```

In this code, the cube has two incoming sockets and two outgoing sockets. Incoming sockets are defined by applying the [DiagramExternalAttribute](#) attribute to the method:

```

[<DiagramExternal>]
member this.Process(candle: CandleMessage, diff: Unit) =

```

Outgoing sockets are defined by applying the attribute to an event. In the example of the cube, there are two such events:

```

let output1Event = new Event<Unit>()
let output2Event = new Event<Unit>()

[<CLIEvent>]
[<DiagramExternal>]
member this.Output1 = output1Event.Publish

[<CLIEvent>]
[<DiagramExternal>]
member this.Output2 = output2Event.Publish

```

Therefore, there will also be two outgoing sockets.

Additionally, it shows how to make a property for the cube:

```

let minValueParam =
    this.AddParam<int>("MinValue", 10)
    .SetBasic(true) // make the parameter visible in basic mode
    .SetDisplay("Parameters", "Min value", "Min value parameter description", 10)

```

Using the [DiagramElementParam](#) class automatically employs the approach to save and restore settings.

The **MinValue** property is marked as basic, and it will be visible in the [Basic properties](#) mode.

The commented [WaitAllInput](#) property is responsible for the timing of the method call with incoming sockets:

```

// override this.WaitAllInput
//     with get () = false

```

If uncommented, the **Process** method will always be called as soon as at least one value arrives (in the example case, this is either a candle or a numerical value).

To add the resulting cube to the diagram, you need to select the created cube in the palette in the **Own elements** section:

[image: Designer_Source_Code_Elem_01]

Cubes from F# code cannot be used in strategies created in F# code. They can only be used in strategies created [from cubes](#).

See Also

Create own indicator

Creating an Indicator

Creating your own indicator in [API](#) is described in the section [Custom Indicator](#). Such indicators are fully compatible with **Designer**.

To create an indicator, on the **Scheme** panel you need to select the **Indicators** folder, right-click and in the context menu select **Add**:

[image: Designer_Source_Code_Indicator_00]

The indicator code will look like this:

```
/// <summary>
/// Sample indicator demonstrating how to save and load parameters.
/// Changes the input price by +20% or -20%.
///
/// See more examples:
/// https://github.com/StockSharp/StockSharp/tree/master/Algo/Indicators
///
/// Documentation:
/// https://doc.stocksharp.com/topics/designer/strategies/using_code/fsharp/
create_own_indicator.html
/// </summary>
type EmptyIndicator() as this =
    inherit BaseIndicator()

    // Internal fields
    let mutable changeValue = 20
    let mutable counter = 0
    let mutable isFormedValue = false

    /// <summary>
    /// The percentage value (+/-) used to modify the input price.
    /// </summary>
    member this.Change
        with get () = changeValue
        and set value =
            changeValue <- value
            this.Reset()

    /// <summary>
    /// Defines if the indicator has formed (became ready for trading).
    /// </summary>
    override this.CalcIsFormed() = isFormedValue

    /// <summary>
    /// Resets the indicator to its initial state.
    /// </summary>
    override this.Reset() =
        base.Reset()
        isFormedValue <- false
        counter <- 0

    /// <summary>
    /// The main logic to process input values.
    /// </summary>
    override this.OnProcess(input: IIndicatorValue) : IIndicatorValue =
        // every 10th call try to return an "empty" value
        if RandomGen.GetInt(0, 10) = 0 then
            // empty value still contains just time, no actual data
            DecimalIndicatorValue(this, input.Time)
        else
            // increment counter on each call
```

```

counter <- counter + 1

// after 5 inputs, indicator is considered formed
if counter = 5 then
    isFormedValue <- true

let mutable value = input.ToDecimal()

// random change by a factor of +/- Change%
let randomFactor = decimal (RandomGen.GetInt(-changeValue, changeValue)) / 100m
value <- value + (value * randomFactor)

// return final indicator value
let result = DecimalIndicatorValue(this, value, input.Time)
// randomly mark it as final or not
result.IsFinal <- RandomGen.GetBool()
result

/// <summary>
/// Load indicator settings from a given <see cref="SettingsStorage"/>.
/// </summary>
override this.Load(storage: SettingsStorage) =
    base.Load(storage)
    this.Change <- storage.GetValue<int>(nameof(this.Change))

/// <summary>
/// Save indicator settings to a given <see cref="SettingsStorage"/>.
/// </summary>
override this.Save(storage: SettingsStorage) =
    base.Save(storage)
    storage.SetValue(nameof(this.Change), this.Change)

/// <summary>
/// A string representation that includes the current <see cref="Change"/> value.
/// </summary>
override this.ToString() =
    sprintf "Change: %d" this.Change

```

This indicator receives an incoming value and makes an arbitrary deviation on the set parameter **Change** value.

The description of the indicator methods is available in the section [Custom Indicator](#).

To add the created indicator to the diagram, you need to use the [Indicator](#) cube, and in it, specify the necessary indicator:

[image: Designer_Source_Code_Indicator_01]

The **Change** parameter, previously set in the indicator code, is shown in the properties panel.

Indicators from F# code cannot be used in strategies created in F# code. They can only be used in strategies created [from cubes](#).

Using Python

Creating strategies from code is for users who prefer working with Python code. Such strategies are not limited in capabilities unlike schemes, and any algorithm can be implemented.

The strategy creation process takes place directly in [Designer](#) or in a **Python** development environment (the most popular development environments are **Visual Studio** and **JetBrains Rider**), using the library for professional development of trading robots in **Python** and [API](#).

You can add a new strategy by clicking the **Add** button *[image: Designer Panel Circuits 01]* in the **General** tab and selecting **Strategy**. Or by right-clicking on the **Strategies** folder in the **Schemes** panel and clicking the **Add** button *[image: Designer Panel Circuits 01]* in the dropdown menu:

[image: Designer The creation of a strategy 00]

After clicking the **Add** button *[image: Designer Panel Circuits 01]*, a window will appear with a choice of content type on which to create the strategy:

[image: Designer_Creation_of_element_containing_source_code_00]

To create a strategy from Python code, select the second tab. You can also choose a template that will be used as initial code.

After clicking **OK**, a new strategy will appear in the **Strategies** folder of the **Schemes** panel, similar to creating a strategy from a [scheme](#). Actions for deleting or renaming the strategy are also similar.

But instead of a scheme, a Python code editor will be shown:

[image: Designer_Creation_of_element_containing_source_code_01]

The code editor tab consists of **Source Code** and **Error List** panels. The **Source Code** panel contains the Python code editor itself. At the top is a toolbar where you can enable or disable highlighting of things like **Current Line**, **Line Number**, etc. To increase the font size, you can use the CTRL+MouseWheel combination.

The **Error List** panel is a table with a list of code errors; double-clicking on a line will automatically move the cursor in the **Source Code** panel to the error location.

When editing code, an icon *[image: Designer The creation of the cube containing the source code 03]* will appear in the bottom right corner of the **Error List** panel, indicating that change tracking has begun. Code compilation occurs when the code stops changing.

Running the strategy on [backtest](#), on [live](#), and other operations work similarly to strategies created from schemes.

Limitations

[Designer](#) uses IronPython, which has the following limitations:

- Compatibility with Python version 3.4
- Partial numpy support through a special .NET implementation (usage example can be found [here](#))
- Lack of support for other popular libraries written in C (pandas, scipy, etc.)
- Limited support for asynchronous programming
- Inability to use certain built-in Python modules due to dependency on CPython-specific implementations
- Performance may be lower compared to CPython for some operations

It is recommended to consider these limitations when developing trading strategies in Python within [Designer](#).

First strategy

Python Strategy Example

Creating a strategy from source code will be demonstrated using the SMA strategy example - similar to the SMA strategy example assembled from cubes in the [Creating Algorithm from Cubes](#) section.

This section will not describe Python language constructs (or [Strategy](#), which is used as the base for creating strategies), but will mention specific features for working with code in the **Designer**.

Strategies created in the **Designer** are compatible with strategies created in the [API](#) due to using the common base class [Strategy](#). This makes running such strategies outside the **Designer** significantly easier than running [schemes](#).

1. Before starting Python development, you need to import .NET dependencies through the clr module, which is provided by IronPython for .NET interaction:

```
import clr

clr.AddReference("System.Drawing")
clr.AddReference("StockSharp.Messages")
clr.AddReference("StockSharp.Algo")
```

Main libraries that need to be connected:

- clr module - imported to enable Python code interaction with .NET Framework
- System.Drawing - library needed for working with colors and graphics
- StockSharp.Messages - contains basic S# messages and data types
- StockSharp.Algo - contains basic S# algorithmic components, including the Strategy base class

It's important to connect libraries at the beginning of the file, before using any types from these libraries. After connecting the libraries, you can import specific types through regular Python import.

2. Strategy parameters are created using a special approach:

```
def __init__(self):
    super(sma_strategy, self).__init__()
    self._isShortLessThenLong = None

    # Initialize strategy parameters
    self._candleTypeParam = self.Param("CandleType", DataType.TimeFrame(TimeSpan.FromMinutes(1))) \
        .SetDisplay("Candle type", "Candle type for strategy calculation.", "General")

    self._long = self.Param("Long", 80)
    self._short = self.Param("Short", 30)

    self._takeValue = self.Param("TakeValue", Unit(0, UnitTypes.Absolute))
    self._stopValue = self.Param("StopValue", Unit(2, UnitTypes.Percent))

    @property
    def CandleType(self):
        return self._candleTypeParam.Value

    @CandleType.setter
    def CandleType(self, value):
        self._candleTypeParam.Value = value

    @property
    def Long(self):
        return self._long.Value
```

```

@Long.setter
def Long(self, value):
    self._long.Value = value

@property
def Short(self):
    return self._short.Value

@Short.setter
def Short(self, value):
    self._short.Value = value

@property
def TakeValue(self):
    return self._takeValue.Value

@TakeValue.setter
def TakeValue(self, value):
    self._takeValue.Value = value

@property
def StopValue(self):
    return self._stopValue.Value

@StopValue.setter
def StopValue(self, value):
    self._stopValue.Value = value

```

When using the [StrategyParam](#) class, the settings save and restore approach is automatically used.

3. When creating indicators and subscribing to market data, you need to bind them so that incoming data from the subscription can update indicator values:

```

# Create indicators
longSma = SMA()
longSma.Length = self.Long
shortSma = SMA()
shortSma.Length = self.Short

# Bind candles set and indicators
subscription = self.SubscribeCandles(self.CandleType)
# Bind indicators to the candles and start processing
subscription.Bind(longSma, shortSma, self.OnProcess).Start()

```

4. When working with charts, it's important to consider that when running the strategy [outside the Designer](#), the chart object might be absent.

```

# Configure chart if GUI is available
area = self.CreateChartArea()
if area is not None:
    self.DrawCandles(area, subscription)
    self.DrawIndicator(area, shortSma, Color.Coral)
    self.DrawIndicator(area, longSma)
    self.DrawOwnTrades(area)

```

5. Start position protection through [StartProtection](#) if required by the strategy logic:

```

self.StartProtection(self.TakeValue, self.StopValue)

```

6. The strategy logic itself, implemented in the OnProcess method. The method is called by the subscription created in step 1:

```

def OnProcess(self, candle, longValue, shortValue):
    """
    Processes each finished candle, logs information, and executes trading logic on SMA crossing.

    :param candle: The processed candle message.
    :param longValue: The current value of the long SMA.
    :param shortValue: The current value of the short SMA.
    """
    self.LogInfo("New candle {0}: {6} {1};{2};{3};{4}; volume {5}", candle.OpenTime, candle.OpenPrice,
    candle.HighPrice, candle.LowPrice, candle.ClosePrice, candle.TotalVolume, candle.SecurityId)

    # If candle is not finished, do nothing
    if candle.State != CandleStates.Finished:
        return

    # Determine if short SMA is less than long SMA
    isShortLessThenLong = shortValue < longValue

    if self._isShortLessThenLong is None:
        self._isShortLessThenLong = isShortLessThenLong
    elif self._isShortLessThenLong != isShortLessThenLong:
        # Crossing happened
        direction = Sides.Sell if isShortLessThenLong else Sides.Buy

    # Calculate volume for opening position or reverting
    volume = self.Volume if self.Position == 0 else Math.Min(Math.Abs(self.Position), self.Volume) * 2

    # Get price step (default to 1 if not set)
    priceStep = self.GetSecurity().PriceStep or 1

    # Calculate order price with offset
    price = candle.ClosePrice + (priceStep if direction == Sides.Buy else -priceStep)

    if direction == Sides.Buy:
        self.BuyLimit(price, volume)
    else:
        self.SellLimit(price, volume)

    # Update state
    self._isShortLessThenLong = isShortLessThenLong

```

7. A mandatory requirement for Python strategies is to override the virtual method [CreateClone](#). This method is needed to create new strategy instances required by [Designer](#) when optimizing strategy parameters or during testing. Without overriding it, these operations will be impossible.

```

def CreateClone(self):
    """
    !! REQUIRED!! Creates a new instance of the strategy.
    """
    return sma_strategy()

```

Failure to override the CreateClone method will make it impossible to test the strategy on historical data or perform parameter optimization. When creating a strategy in Python, always add this method.

Creating Your Own Cube

Creating Your Own Cube

Similar to creating a [cube from a scheme](#), you can create your own cube based on Python code. Such a cube will be more functional than a cube created from a scheme.

To create a cube from code, you need to create it in the **Custom Cubes** folder:

[image: Designer_Source_Code_Elem_00]

In the example below, the cube inherits from the [DiagramExternalElement](#) class and looks as follows:

```
import clr

# Add references to the required StockSharp assemblies
clr.AddReference("StockSharp.Messages")
clr.AddReference("StockSharp.Diagram.Core")

# Import necessary types from .NET and StockSharp
from System import Action
from StockSharp.Messages import Unit
from StockSharp.Messages import ICandleMessage
from StockSharp.Diagram import DiagramExternalElement

from designer_extensions import diagram_external

# Custom diagram element class that demonstrates input and output sockets usage
class empty_diagram_element(DiagramExternalElement):
    """
    Sample diagram element demonstrating input and output sockets usage.

    https://doc.stocksharp.com/topics/designer/strategies/using_code/python/creating_your_own_cube.html
    """

    def __init__(self):
        super(empty_diagram_element, self).__init__()

        # Example property to show how to add parameters to the diagram element
        # This parameter is named "MinValue" with a default value of 10
        self._minValue = self.AddParam("MinValue", 10)\
            .SetBasic(True)\
            .SetDisplay("Parameters", "Min value", "Min value parameter description", 10)

        # Initialize output event handlers as empty lists
        # Subscribers can assign callable methods to these handlers
        self._output1_handlers = []
        self._output2_handlers = []

        # Output sockets are events marked with the DiagramExternal attribute
        @diagram_external
        def add_Output1(self, handler: Action[Unit]):
            """
            Subscribe to the Output1 event.

            :param handler: A callable method to be invoked when Output1 is triggered.
            """
            self._output1_handlers.append(handler)

        def remove_Output1(self, handler):
            """
            Unsubscribe from the Output1 event.

            :param handler: The callable method to be removed from the Output1 subscribers.
```

```

"""
if handler in self._output1_handlers:
    self._output1_handlers.remove(handler)

@diagram_external
def add_Output2(self, handler: Action[Unit]):
    """
    Subscribe to the Output2 event.

    :param handler: A callable method to be invoked when Output2 is triggered.
    """
    self._output2_handlers.append(handler)

def remove_Output2(self, handler):
    """
    Unsubscribe from the Output2 event.

    :param handler: The callable method to be removed from the Output2 subscribers.
    """
    if handler in self._output2_handlers:
        self._output2_handlers.remove(handler)

# Uncomment the following property if you want the Process method
# to be called every time when a new argument is received
# (no need to wait for all input args to be received).
#
# @property
# def WaitAllInput(self):
#     return False

@diagram_external
def Process(self, candle: ICandleMessage, diff: Unit) -> None:
    """
    Input sockets are method parameters marked with the DiagramExternal attribute.
    Processes a candle and a diff value, then invokes output events based on the logic.

    :param candle: CandleMessage input representing a candlestick.
    :param diff: Unit representing the difference value to be processed.
    """
    # Calculate the result as the sum of the candle's close price and the diff value
    res = candle.ClosePrice + diff

    # Invoke Output1 if diff is greater than or equal to the MinValue parameter,
    # otherwise invoke Output2
    if diff >= self._minValue.Value:
        for handler in self._output1_handlers:
            handler(res)
    else:
        for handler in self._output2_handlers:
            handler(res)

def Start(self):
    """
    Called when the diagram element starts. Add any pre-start logic here.
    """
    super(empty_diagram_element, self).Start()
    # Add custom logic to be executed before the element starts

def Stop(self):
    """
    Called when the diagram element stops. Add any post-stop logic here.
    """
    super(empty_diagram_element, self).Stop()
    # Add custom logic to be executed after the element stops

def Reset(self):
    """
    Called when the diagram element resets. Add any reset logic here.
    """

```

```

super(empty_diagram_element, self).Reset()
# Add custom logic to reset the internal state of the element

```

In this code, the cube has two incoming sockets and two outgoing sockets. Incoming sockets are defined by applying the `@diagram_external` decorator to a method:

```

@diagram_external
def Process(self, candle: ICandleMessage, diff: Unit) -> None:

```

Outgoing sockets are defined by applying the `@diagram_external` decorator to an event (event subscription operation `add_NNN`). In the cube example, there are two such events:

```

# Output sockets are events marked with the DiagramExternal attribute

@diagram_external
def add_Output1(self, handler: Action[Unit]):
    """
    Subscribe to the Output1 event.

    :param handler: A callable method to be invoked when Output1 is triggered.
    """
    self._output1_handlers.append(handler)

def remove_Output1(self, handler):
    """
    Unsubscribe from the Output1 event.

    :param handler: The callable method to be removed from the Output1 subscribers.
    """
    if handler in self._output1_handlers:
        self._output1_handlers.remove(handler)

@diagram_external
def add_Output2(self, handler: Action[Unit]):
    """
    Subscribe to the Output2 event.

    :param handler: A callable method to be invoked when Output2 is triggered.
    """
    self._output2_handlers.append(handler)

def remove_Output2(self, handler):
    """
    Unsubscribe from the Output2 event.

    :param handler: The callable method to be removed from the Output2 subscribers.
    """
    if handler in self._output2_handlers:
        self._output2_handlers.remove(handler)

```

Therefore, there will also be two outgoing sockets.

Additionally, it shows how to create a property for the cube:

```

self._minValue = self.AddParam("MinValue", 10)\
    .SetBasic(True)\
    .SetDisplay("Parameters", "Min value", "Min value parameter description", 10)

```

When using the [DiagramElementParam](#) class, the settings save and restore approach is automatically used. The **MinValue** property is marked as basic, and it will be visible in [Basic Properties](#) mode.

The commented [WaitAllInput](#) property determines when the method with incoming sockets is called:

```
# @property
# def WaitAllInput(self):
#     return False
```

If you uncomment the property, the **Process** method will be called whenever at least one value arrives (in the example case, either a candle or a numeric value).

To add the resulting cube to the scheme, you need to select the created cube in the palette under the **Custom Cubes** section:

[image: Designer_Source_Code_Elem_01]

Cubes from Python code cannot be used in strategies created in Python code. They can only be used in strategies created [from cubes](#).

See Also

[Creating an Indicator](#)

Create own indicator

Creating an Indicator

Creating a custom indicator in [API](#) is described in the [Custom Indicator](#) section. Such indicators are fully compatible with **Designer**.

To create an indicator, you need to select the **Indicators** folder on the **Schemes** panel, right-click, and select **Add** from the context menu:

[image: Designer_Source_Code_Indicator_00]

The indicator code will look like this:

```
import clr
import random

clr.AddReference("StockSharp.BusinessEntities")
clr.AddReference("StockSharp.Algo")

from StockSharp.Algo.Indicators import BaseIndicator, DecimalIndicatorValue
from indicator_extensions import *

class empty_indicator(BaseIndicator):
    """
    Sample indicator demonstrating saving and loading parameters.

    Doc https://doc.stocksharp.com/topics/designer/strategies/using\_code/python/create\_own\_indicator.html

    Changes input price on +20% or -20%.
    """
    def __init__(self):
        super(empty_indicator, self).__init__()
        self._change = 20
        self._counter = 0
        self._isFormed = False

    @property
    def Change(self) -> int:
        return self._change

    @Change.setter
    def Change(self, value):
        self._change = value
        self.Reset()

    def CalcIsFormed(self):
        """Determines if the indicator has received sufficient inputs to be considered formed."""
        return self._isFormed

    def Reset(self):
        """Resets the indicator's state and internal counters."""
        super(empty_indicator, self).Reset()
        self._isFormed = False
        self._counter = 0

    def OnProcess(self, input):
        """
        Processes the incoming indicator value and applies a random change.

        :param input: The incoming indicator value.
        :return: A new DecimalIndicatorValue after applying changes.
        """
        # Every 10th call, try to return an empty value
```



```

if random.randint(0, 10) == 0:
    return DecimalIndicatorValue(self, input.Time)

if self._counter == 5:
    # For example, our indicator needs 5 inputs to become formed
    self._isFormed = True
    self._counter += 1

value = to_decimal(input)

# Apply random change of +/- _change percent to the current value
value += value * random.randint(-self._change, self._change) / 100.0

result = DecimalIndicatorValue(self, value, input.Time)
# Mark value as final based on a random decision
result.IsFinal = bool(random.getrandbits(1))
return result

def Load(self, storage):
    """
    Loads the indicator parameters from persistent storage.

    :param storage: The settings storage to load from.
    """
    super(empty_indicator, self).Load(storage)
    self.Change = storage.GetValue("Change", self.Change)

def Save(self, storage):
    """
    Saves the indicator parameters to persistent storage.

    :param storage: The settings storage to save to.
    """
    super(empty_indicator, self).Save(storage)
    storage.SetValue("Change", self.Change)

def __str__(self):
    return f"Change: {self.Change}"

def ToString(self):
    return str(self)

```

This indicator receives an incoming value and makes a random deviation based on the specified **Change** parameter value.

Description of indicator methods is available in the [Custom Indicator](#) section.

To add the created indicator to the scheme, you need to use the [Indicator](#) cube, and then set the desired indicator within it:

[image: Designer_Source_Code_Indicator_01]

The **Change** parameter, previously set in the indicator code, is shown in the properties panel.

Indicators from Python code cannot be used in strategies created in Python code. They can only be used in strategies created [from cubes](#).

Using DLL

Using DLL

Using ready-made DLLs is familiar to those who want to work continuously in **Visual Studio** and **JetBrains Rider** environments. This approach offers several advantages over writing [code](#) inside **Designer**:

- Enhanced code editor compared to the built-in editor inside **Designer**.
- Recompiling code automatically updates content inside **Designer**.
- Possibility to split code into several files (in the case of the [code](#) approach, only OneFile-OneStrategy variant is possible).
- Using the [debugger](#).

Creating a Project in Visual Studio

1. To create a strategy in **Visual Studio**, you need to create a project:

[image: Designer Creating a DLL cube in Visual Studio 00]

2. Next, you need to write the strategy code. For a quick start, you can copy the SmaStrategy code, which is created as a template in [strategy from code](#):

[image: Designer Creating a DLL cube in Visual Studio 03]

3. To compile the code, it is necessary to include the NuGet package [StockSharp.Algo](#), where the base class for all strategies [Strategy](#) is located:

[image: Designer Creating a DLL cube in Visual Studio 04]

If the strategy uses charting interfaces, it is necessary to include the NuGet package [StockSharp.Charting.Interfaces](#). These interfaces do not contain the logic of the actual chart and are only needed for compiling the code. In the case of launching the strategy in **Designer**, real data rendering on the chart will occur through these interfaces.

4. After creating the strategy, the project needs to be built by pressing **Build Solution** in the **Build** tab.

[image: Designer Creating a DLL cube in Visual Studio 01]

5. In Visual Studio by default, the project is built into the ...\\bin\\Debug\\net6.0 folder.

[image: Designer Creating a DLL cube in Visual Studio 02]

Adding DLL to Designer

1. Adding a strategy from a DLL is similar to creating a strategy from [code](#). But at the content type definition stage, you need to choose **DLL**:

[image: Designer_Creation_Strategy_Dll_00]

2. In the window, you need to specify the path to the assembly (must be compatible with .NET 6.0), and choose the type. The latter is necessary because one DLL can contain several strategies (or [cubes with indicators](#)). After clicking **OK**, the strategy will be added to the **Scheme** panel and is ready for use:

[image: Designer_Creation_Strategy_Dll_01]

3. Launching the strategy on [backtest](#), on [live](#), and other operations - works similarly to strategy from diagrams and code:

[image: Designer_Creation_Strategy_Dll_02]

Create element and indicator

Creating a Cube and Indicator

The algorithm for creating a cube or indicator from a DLL assembly is similar to the process from code (see the creation of a [cube](#) and [indicator](#)) except for the content selection stage. Similarly, when adding a [strategy from DLL](#), you can choose both the cube type and the indicator type if they are created in the connected DLL:

[image: Designer_Import_Element_00]

When creating an indicator, to compile the code it is necessary to include the NuGet package [StockSharp.Algo](#), where the base class for all indicators [Strategy](#) is located.

[image: Designer_Import_Indicator_00]

When creating a cube, to compile the code it is necessary to include the NuGet package [StockSharp.Diagram.Core](#), where the base class for all cubes [DiagramExternalElement](#) is located.

When adding connected cubes or indicators to the diagram, you need to follow the steps described in the sections for a [cube](#) or [indicator](#).

See Also

[Debugging a DLL Cube with Visual Studio](#)

Debug DLL in Visual Studio

Debugging a DLL with Visual Studio

Visual Studio provides a mechanism for attaching to running processes using the Visual Studio debugger. The Visual Studio debugger is described in more detail in the documentation [Attach to running processes](#). The debugging process will be demonstrated using the example of a strategy added in the section [Using DLL](#).

1. To attach to a process and start debugging a DLL strategy, it needs to be loaded into memory. The DLL is loaded into memory after [adding the strategy](#). Once the DLL is loaded into memory, you can attach to the process.

[image: Designer_Creation_Strategy_DLL_01]

2. In Visual Studio, select **Debug -> Attach to Process**.

[image: Designer Debugging DLL cube using Visual Studio 00]

3. In the **Attach to Process** dialog box, find the **Designer.exe** process in the **Available processes** list that you want to attach to.

[image: Designer Debugging DLL cube using Visual Studio 01]

If the process is running under a different user account, you need to check the **Show processes from all users** checkbox.

4. It's important that the **Attach to** window specifies the code type that needs to be debugged. The default **Auto** parameter tries to determine the code type to be debugged, but it does not always correctly identify the code type. To manually set the code type, you need to do the following steps.
 - In the **Attach to** field, click **Select**.
 - In the **Select Code Type** dialog box, click the **Debug these code types** button and select the types for debugging.
 - Click OK.

[image: Designer Debugging DLL cube using Visual Studio 02]

5. Click the **Attach** button.
6. In Visual Studio, set breakpoints in the code. If the breakpoints are red and filled with red *[image: Designer Debugging DLL cube using Visual Studio 03]* (and Studio is in debugging mode), it means that the exact version of the DLL was loaded. If the breakpoints are red and filled with white *[image: Designer Debugging DLL cube using Visual Studio 04]* (and Studio is in debugging mode), it means that the wrong version of the DLL was loaded.
7. In the example, the breakpoint is set in the first line of the method **public void ProcessCandle(Candle candle)**. When the strategy runs in [Designer](#), as soon as candle values start being passed to the DLL, Visual Studio will stop at the breakpoint. From there, you can track the execution of the code:

[image: Designer Debugging DLL cube using Visual Studio 05]

When the code is stopped under the debugger, all processes inside the **Designer** program are suspended. If the program is connected to real trading, then in the case of a long stop under the debugger, disconnections will occur.

See Also

[Exporting Strategies](#)

DLL panel

DLL Panel

The DLL panel displays the meta-information of a .NET assembly:

[image: Designer_Panel_DLL]

In the panel, you can see the file creation time, its version, and path. The creation time is the easiest way to determine which assembly version is being used by **Designer**.

When recompiling the assembly through an [external program](#), the metadata will be automatically updated

Indicators

See [List of indicators](#)

Creating Your Own Indicator

Creating Your Own Indicator

Designer provides two ways to create your own indicator:

- through [code](#)
- by [importing a dll](#)

User interface

User interface

To run the test on the history, you shall select a strategy, the scheme of which will be tested on the history. The strategy is selected on the [Schemas](#) panel in the strategy folder, by double clicking on the strategy of interest. When you select a strategy for the workspace, a new tab with a strategy appears, when you switch to this tab, the **Emulation** tab will automatically open in the Ribbon.

[image: Designer Interface Backtesting 00]

On the **Emulation** tab, you can change the strategy name and give it a brief description.

To run the test on the history, on the **Emulation tab** specify the path to the historical data in the Market Data field and set the testing period. The strategy for testing is started by clicking the **Start button** *[image: Designer Interface Backtesting 01]*. After starting the strategy for testing, the **Pause** *[image: Designer Interface Backtesting 02]* button, suspending the testing and the **Stop** *[image: Designer Interface Backtesting 03]* button, completely stopping the testing, become active. When editing the strategy, the **Undo(Ctrl+Z)** *[image: Designer Interface Backtesting 04]*, canceling the last action, **Redo(Ctrl+Y)** *[image: Designer Interface Backtesting 05]*, returning undo, **Refresh(Ctrl+R)** *[image: Designer Interface Backtesting 06]*, completely updating the scheme, buttons will be useful. Also, from the **Emulation tab** you can use the **Debugger** ([Debugging](#)) or run strategy **Optimization**.

The selected strategy tab contains the following panels by default:

- The **Scheme** panel, in which the main working process on the strategy and its components design is performed, by combining cubes and connecting lines. The Scheme is described in detail in the [Diagram panel](#) section.
- Panel of information elements, containing the **Chart, Orders, Trades, Statistics** and other components. You can add required component, selecting it on the **Emulation** tab in the **Components** group.
- The **Properties** panel is collapsed by default on the right side of the strategy tab. On the **Properties** panel, you can arrange the **Emulation** general settings. For example, the **Market-data storage format** can be set to **BIN** or **CSV**, depending on the file format of the selected storage. The data type can be Ticks or Candles. If Ticks is selected, candles will be formed from the ticks specified in the [Backtesting settings](#).

Recommended content

[Backtesting settings](#)

Getting started

Getting started

For instance, the SMA strategy example will be considered.

To run the test on the history, you shall select a strategy, the scheme of which will be tested on the history. The strategy is selected on the [Schemas](#) panel in the strategy folder, by double clicking on the strategy of interest.

Prior to testing, it is necessary to load market data (instruments, candles, tic trades and/or order books). The way to do that is described in the [Market data storage](#) section.

When switching to the tab with a strategy, the **Emulation** tab automatically opens in the **Ribbon**. It is necessary to set the testing period on this tab. In the market-data field specify the required storage ([Market data storage](#)), in the instrument field specify the required instrument.

The following parameters will be used in the example with the SMA strategy.

1. AAPL@NASDAQ instrument
2. Standard storage \Documents\StockSharp\Designer\Storage
3. Storage format - CSV
4. Type of data, taken from the storage - Ticks
5. Order book - generated
6. Order book depth - 5
7. Spread size - 2
8. Candles with time-frame of 30 s
9. Volume - 100

It is required to set the selected parameters up:

[image: Designer An example of backtesting 00]

[image: Designer An example of backtesting 01]

After setting up all the required parameters, start up the strategy testing by clicking the *[image: Designer Interface Backtesting 01]* button.

During or after the testing you can view charts and tables with the testing information.

[image: Designer An example of backtesting 02]

The graph shows that the trades take place at the intersection of moving averages, as planned by the strategy. It can also be seen that orders are satisfied within several trades. It happens because of generated order book use, which increases the testing reality. The fact that orders are satisfied within several trades can be seen from the Trades tables, Statistics, and the Positions chart.

[image: Designer An example of backtesting 03]

It can be seen on the **Positions chart** that the strategy has decreased the operated volume. This happened because the generated order book has a depth of 5 and, as a result, the entire order book depth was insufficient to satisfy the order of 200 lots. Since the strategy only reverses the position, every time the depth of the order book was not sufficient to satisfy the order, the order size was reduced.

[image: Designer An example of backtesting 04]

The P/L chart indicates that the strategy is unprofitable with such parameters.

Recommended content

[Live execution](#)

Debugging

Debugging

In the process of strategy testing, it often becomes necessary to check what data enters the input of a given cube or is passed at its output. To do that, the [Designer](#) features the **Debugger**.

[image: Designer Debug 00]

The following buttons are located in the **Debugger** group of the **Emulation** Ribbon:

- *[image: Designer Debug 01]* **Add breakpoint** – to add a breakpoint at the selected element. Elements, for which a breakpoint is added, are highlighted with a red border.
- *[image: Designer Debug 02]* **Delete breakpoint** – to delete a breakpoint.
- *[image: Designer Debug 03]* **Next element** – when the breakpoint is triggered, moves to the next element of the scheme.
- **Step to out** – when the breakpoint is triggered, moves to the output of the current element, is used to check the values, passed at the element output.
- *[image: Designer Debug 04]* **Step in** – when the breakpoint is triggered, moves inside the composite element. Automatically opens the composite element scheme and stops at the element to which the data is first transmitted.
- *[image: Designer Debug 05]* **Step out** – when the breakpoint is triggered and is within the composite element, it exits one level up where an open composite element is used.
- *[image: Designer Debug 06]* **Continue** – Continues until the next breakpoint is triggered.

Recommended content

[Break points](#)

Break points

Break points

To add a breakpoint, select the cube and click the *[image: Designer Debug 01]***Add Breakpoint** button. The breakpoints are highlighted with a red circle:

[image: Designer Stopping point 00]

When the breakpoint actuates on an element, its background changes to light red. The element is automatically highlighted in the properties window, where its properties and the values of the input and output parameters are displayed. If you point over the input or output parameter with the mouse, a prompt appears with the parameter value. An example of viewing the values at the input and output of the composite element during the execution of the strategy is shown in the figure below:

[image: Designer Stopping point 01]

Breakpoints can be added both before the launch of the testing process and during the testing of the strategy on history.

If you click the *[image: Designer Debug 01]***Breakpoints** button, a window will appear in which all breakpoints will be displayed. For each of them, an additional triggering condition can be set. For example, you can set the **True** value for the logical signal. In this case, the breakpoint will stop only if the signal value is **True**.

[image: Designer Stopping point 02]

Recommended content

[Step by step execution](#)

Step by step execution

Step by step execution

To track step-by-step execution, the breakpoint shall be placed on the cube from which you want to start step-by-step execution. After stopping on the required cube, click the  **Next element** button. Designer will consistently move from cube to cube, showing the strategy step-by-step execution. If it is necessary to enter the **Composite element**, then when step-by-step execution reaches the **Composite element**, click the  **Step in** button. And the next step of the step-by-step execution will be inside the **Composite element**. To exit the Composite element, click the  **Step out** button.

Recommended content

[Optimization](#)

Cloud backtesting

Cloud Testing

To test strategies in the cloud, it's necessary to first find all the instruments of interest. For this, in the **Designer**, you need to open the instrument search panel available for testing in the **Cloud** tab:

[image: Designer_Backtest_Cloud_01]

When you enter the instrument name in the search field and click **Search** (or press **Enter**), the StockSharp server returns suitable search results. The date ranges of historical data will also be indicated to the right of the instrument names.

This procedure needs to be done only once for each new instrument. After that, the found instruments will be saved locally on the disk, and upon restarting the **Designer**, they will already be loaded from the local storage. This step is necessary because specifying the instrument is required when launching the strategy (as well as when specifying instruments directly in the [Variable](#) block).

After this, you need to return to the strategy and enable the cloud option in the **Backtest** tab:

[image: Designer_Backtest_Cloud_00]

By starting the test, the strategy will be sent to the StockSharp cloud instead of being tested locally:

[image: Designer_Backtest_Cloud_02]

After the testing is completed, the report with results will be shown in the task waiting tab:

[image: Designer_Backtest_Cloud_03]

If you want to view the history of testing in the cloud, as well as see the current active tasks, open the **Tasks** panel in the **Cloud** tab:

[image: Designer_Backtest_Cloud_04]

Brute-Force

Brute-force

To switch to strategy optimization mode, click the **Optimization** button on the **Emulation** tab. The optimization example will be considered using the SMA strategy created [from cubes](#).

[image: Designer Optimization 00]

A tab named Optimization + 'Strategy Name' will open in the workspace. The **Optimization** tab is divided into two areas, **Properties**, **Optimization Result**:

[image: Designer Optimization 02]

- The **Properties** area consists of tabs with several tables. The first is the strategy parameters, which are [iterated over](#). The second - settings for [genetics](#). The third - system settings of the optimizer. For example, you can change the number of threads and cores involved in the optimization there.
- The **Optimization Result** area is a table, each row of which is the result of testing the strategy with unique parameters. Also, in the **Optimization Result** area, there is a progress bar showing the optimization progress, elapsed time, and estimated time until the end of optimization. Additionally, there is a tab for displaying the results in the form of a [3D chart](#).

Setting the parameters for iteration results in more than 1000 iterations. After starting the optimizer, the progress at the top above the results will show data on the planned number of iterations, how many have already been completed, and how much time is approximately needed until completion:

[image: Designer Optimization 03]

See also

[Example of backtesting](#)

Optimization Parameters

Optimization Parameters

Optimization is performed on strategy parameters that have the following types:

- Numerical (integer and fractional)
- Time ([TimeSpan](#))
- Boolean value (True-False)
- [Unit](#) value

By default, all parameters with these types will be represented in the [optimizer parameters table](#). If it is necessary to exclude some parameter from the optimization, then:

- In the case of a [diagram](#), it is necessary to select the required cube, open its property, switch to **Advanced settings** and turn off the **Parameter** checkbox:

[image: Designer Optimization 01]

- In the case of [code](#), you need to write code when defining a parameter and change the [CanOptimize](#) property:

```
_long = this.Param(nameof(Long), 80);  
_short = this.Param(nameof(Short), 20);  
  
// turn off param for optimization  
_long.CanOptimize = false;
```

After changing the available optimization parameters, it is necessary to reopen the [optimization panel](#).

Genetics

The **Designer** supports optimization both by the [brute force method](#) and based on genetic algorithms. Genetic optimization significantly speeds up the process of finding optimal parameters.

To enable **Genetic** optimization, you need to:

- switch the mode:
[image: Designer_Optimization_Genetic_00]
- set the optimization parameters:
[image: Designer_Optimization_Genetic_01]
- as the target function (Fitness), you can specify an extended formula:
[image: Designer_Optimization_Genetic_02]

For example, make calculations not only by **Profit** but also relative to its **Maximum Drawdown**. Available mathematical functions are similar to the [Formula](#) block.

Optimization through genetics is not deterministic. Therefore, determining the exact number of iterations and, consequently, the necessary total time is impossible, unlike [brute-force search](#).

3D chart

3D Report

Optimization results can be viewed as a 3D chart. To do this, in the results panel, you need to switch the tab to Chart:

[image: Designer Optimization Chart 00]

The chart can display in a 3-dimensional projection, meaning it can only accommodate 3 dimensions. Therefore, if the optimization is conducted over 4 or more parameters, to view the results, you need to change the dimensions for X and Y.

The dimension for Z must always be numerical, unlike X and Y, where dimensions can be, for example, by [instruments](#).

Portfolio optimization

Portfolio optimization

The [Designer](#) has the ability to test one strategy on various securities at the same time. To do that, click the [Optimization](#) button.

[image: Designer Portfolio testing]

In the optimization parameters, set the **Security** parameter, select from the list the securities for which you want to perform testing. In the **Ribbon**, set the testing period and the market-data storage. After clicking the **Start** button, the testing will start on all securities simultaneously.

Cloud optimization

Cloud Optimization

The process of optimization in the cloud is similar to [cloud testing](#). The report results are visually similar to local optimization:

[image: Designer Optimization Cloud 00]

Market data storage

Market data storage

The historical data storage is designed to load market data (instruments, candles, tick trades and order books) from various sources and store them in a local or remote storage. [Designer](#) can use sources of both historical and real-time data ([Connectors](#)). Further, the stored information is available for use by trading strategies.

To open the **Market data** tab, switch to the **Common** tab and click the **Market data** button. The **Market data** area is divided into three areas. The left area contains the list of all received instruments, from all sources that have ever been connected. The central areas contain the active instruments. Using these instruments, you can download or view the downloaded history. The right area displays the available data for the instrument selected in the central zone, and you can also download data using the selected instrument.

[image: Designer Repository of historical data 00]

Recommended content

[Getting started](#)

Download instruments

Download instruments

Clicking the **Download securities** button opens the **Download securities** window. To download securities, you have to enter their code, type, select data source and click the **OK** button. [Designer](#) starts searching securities in the source. All found securities are added to the securities list on the **All securities** panel. If a storage is selected as the source, and the storage already contains downloaded securities, [Designer](#) will find all securities available in the storage. It may be useful, when the history on securities is already downloaded and copied to the storage folder.

[image: Designer Download the tools 00]

Clicking the **Add connection** button opens the [Connections settings](#) window.

Recommended content

[Create instrument](#)

Create instrument

Create instrument

Clicking the **Create security** *[image: Designer Creation tool 00]* button opens the **Editing** window. To create a security, you should fill the security properties and click the **OK** button:

[image: Designer Creation tool 01]

Securities from different sources have unified identifiers. This is done so that the trading robot code does not depend on the connection type ([Connectors](#)). The following syntax is used for the security identifier - **[security code]@[board code]**. For example, for Apple shares on the NASDAQ stock exchange the identifier will be **AAPL@NASDAQ**.

Recommended content

[Edit instrument](#)

Edit instrument

Edit instrument

See

[Editing instruments](#)

[Index](#)

Index

See

[Index](#)

Continuous futures

Continuous futures

See

[Continuous futures](#)

Download market-data

Download market-data

To get the historical data for the security, select the required security from the **All securities** list, move it to the **Active** securities list. Set the historical data period, select type and Time Frame for candles and click the **Start** button. All data will be saved in the [Market data storage](#).

[image: Designer Quick start 02]

Getting started

Getting started

In order to create historical data storage, click the *[image: Designer Creating a repository of historical data 00]* button on the **Market data** tab. By clicking the *[image: Designer Creating a repository of historical data 01]* button, you can change the current storage parameters. By clicking the *[image: Designer Creating a repository of historical data 02]* button, you can delete the current storage from the list of storages.

[image: Designer Creating a repository of historical data 03]

Historical data storage can be local or remote.

Local storage - when all data is stored on local computer. To set the local storage, it is sufficient to set the path to the folder with stored data.

Remote storage may be located on a remote computer. To set the remote storage you have to set the remote storage address, login and password, if necessary.

You can reproduce the remote storage on a local machine, using the [Hydra](#) software (code name Hydra), designed for automatic loading of market data (instruments, candles, tick trades and order books etc.) from various sources and for storing them in the local storage. To do that, switch [Hydra](#) into the server mode.

[image: Designer Creating a repository of historical data 04]

After that, in [Designer](#), create a new storage by clicking the *[image: Designer Creating a repository of historical data 00]* button. In the storage settings, in the address field, specify "net.tcp://localhost:8000". Click OK. When using [Hydra](#) as a remote storage, don't forget, that [Hydra](#) shall be started up and configured accordingly.

[image: Designer Creating a repository of historical data 05]

After adding a new storage, it can be selected in the **Storage** drop-down list.

[image: Designer Creating a repository of historical data 06]

You also have to select the format of storage files - BIN or CSV. Data can be stored in two formats: in a special binary BIN format, which provides the maximum compression ratio, or in text format CSV, which is convenient when analyzing data in other programs. The BIN format is preferred when there is a need to save space on disk. The CSV format is preferred when there is a need to adjust data manually. CSV is easily edited by standard notepad, MS Excel, etc.

Recommended content

[Download instruments](#)

Instrument extended info

Instrument extended info

Any necessary information about a security (for example, country, city, website etc.) may be extended info.

[image: Designer ExtensionInfo securities]

For more information about extended information, see the [Extended instrument info](#) documentation on [Hydra](#).

Symbol mapping

Symbol mapping

The same instrument in different trading systems can be called differently. In **Designer**, it is possible to match the instrument and the connections through which this instrument will be traded, and indicate how it is identified in the external trading system. This is useful when trading one instrument on different trading boards or through different connections (or brokers). Also, it allows receiving data from one connection and making trades through another connection.

To match securities and connections, click the **Securities and connections** button in the **All securities** panel.

[image: Designer Security mapping 00]

In the opened window, click the *[image: Designer Creation tool 00]* button to add a new line.

In the **Connection** column select a connection from the drop-down list. In the **Security code** and **Board code** columns specify the security and board codes, as they are specified in the **Designer**. In the **Adapter code** and **Adapter board** columns specify the security and board codes, as they are specified in the external trading system.

[image: Designer Security mapping 01]

Getting started

Getting started

To add a strategy to **Live trading**, in the [Schemas](#) panel right-click the required strategy from the **Strategies** folder and select *[image: Designer Live trade 00]* **Live**. The strategy will be added to the **Live** folder of the **Schemas** panel.

[image: Designer Live trade 01]

Also, you can add a strategy to Live trade by clicking the *[image: Designer Live trade 00]* **Live** button on the **Backtest** tab.

[image: Designer Live trade 02]

The Trade folder contains the strategies, added for running in Live-trade. The running strategies are marked with the *[image: Designer Panel Circuits 02]*, sign, and the stopped strategies - with the *[image: Designer Panel Circuits 03]* sign.

Recommended content

[User interface](#)

User interface

Interface

After adding a strategy to the **Live** folder, by double-clicking the added strategy, a tab titled "Live [Strategy Name]" will open. Upon navigating to this tab, the **Live** tab will automatically open in the **Ribbon**. On the **Live** tab, you can specify the instrument and portfolio with which the strategy will work. By pressing the **Start** button - initiate Live trading for the strategy; by pressing the **Stop** button - halt it.

[image: Designer Interface Live trade 00]

The strategy tab contains the Strategy Designer for schemes and component elements, similar to the one described in [Strategy Designer](#). Additionally, the tab includes the [Live Trading Properties](#) panel, which by default is collapsed and attached to the right side of the tab.

Adding a strategy to **Live** involves copying it from the original code (in the case of using [schemes](#) or [code](#)). Therefore, changes to the algorithm inside the **Live** copy do not affect the original. When launching the strategy, if there's a discrepancy between **Live** and the original, a warning will be displayed:

[image: Designer Interface Live trade 01]

- **Yes** means to apply changes from the original to the **live** copy.
- **No** means to ignore the difference and launch the **live** copy without applying changes.
- **Cancel** means not to launch anything.

Changes in the **Live** copy should be minimal, aimed at testing and subsequent transfer to the original. Otherwise, there's a risk of losing changes if the **Live** copy is updated to the original's version.

See Also

[Connection Settings](#)

Strategies dashboard

Strategies dashboard

Clicking the *[image: Designer Panel strategies 00]* button on the **Live** tab opens the **Live** panel.

[image: Designer Panel strategies 01]

The **Live** panel is a table, displaying all strategies added to the **Live**. On the **Live** panel, you can view the current state of the strategy, and run or stop the strategy using the *[image: Designer Panel Circuits 02]*, *[image: Designer Panel Circuits 03]* buttons.

[image: Designer Panel strategies 02]

- The first column is responsible for starting/stopping the strategy.
- The second column is for strategy settings.
- **Online** indicates whether all strategy indicators are formed and all subscriptions have transitioned to an **Online state**.
- **Trading** allows you to change available operations. For example, prohibit opening new positions or increasing existing ones.
- **Position** shows the current position for the strategy's instrument. Clicking the cross button will close the position.
- The other columns display the current trading statistics of the strategy.

Double-clicking on a selected row will open the panel with the strategy.

Recommended content

[Live execution sample](#)

Live execution sample

Live execution sample

To run an example in **Live** you will need:

1. Testing terminal **IB Trader Workstation (TWS) Demo** from [Interactive Brokers](#), you can get it on the manufacturer's website.
2. Set the IB TWS Demo terminal for work with [Designer](#). See the **IB TWS Setting demo** in the [Interactive Brokers](#) section.
3. Set connection to the IB TWS Demo in [Designer](#) and connect.
4. Download history for the required instrument. For example, the **AAPL@NASDAQ** instrument will be used. The strategy will use candles with a time frame of 5 seconds, and history will not be needed, but such a history will be sufficient to demonstrate the possibility.

[image: Designer Example of Live trading 00]

5. Set and run the strategy.

The following parameters will be used in the example with the SMA strategy.

- **AAPL@NASDAQ** instrument
- Standard storage **\Documents\StockSharp\Designer\Storage**
- Storage format - **CSV**
- Type of data, taken from the storage - **Ticks**
- Candles with time-frame of 5 s
- Volume - 100
- History days - 2

[image: Designer Example of Live trading 01]

After setting up all the required parameters, start up the strategy testing by clicking the *[image: Designer Panel Circuits 02]* Start button.

After clicking the *[image: Designer Panel Circuits 02]* Start button, the chart will start displaying the entire downloaded history for 2 days:

[image: Designer Example of Live trading 02]

After downloading the entire history from the [Market data storage](#) and the Table of anonymous trades from the terminal, the strategy will start trading.

The chart below shows the trades from [Designer](#).

[image: Designer Example of Live trading 03]

Recommended content

[Market data storage](#)

Running Strategies Outside of Designer

Running Strategies Outside of Designer

Strategies created in [Designer](#) can be run both in your own programs built on the [API](#) and in the [Runner](#) or [Shell](#) applications.

- [Runner](#) is faster and consumes less memory than [Designer](#), so strategies will perform better. This approach is ideal for running strategies on a server. More details can be found in [Export from Designer](#).
- [Shell](#) is provided as source code. This approach is ideal for distributing strategies with a unique interface specially created for the strategies. More details can be found in [Running a Strategy Created in Designer](#).

Connections settings

Connections settings

1. When clicking the arrow next to the **Connect** *[image: Designer The quick access toolbar 00]* button, the **Connection settings** button appears:

[image: Designer The connection settings 00]

2. When clicking the **Connection settings** button, the **Connection settings** window opens which is divided into 3 areas:

[image: Designer The connection settings 01]

3. Clicking the *[image: Designer The connection settings 02]* button activates the drop-down list of supported connectors. The selected connectors are displayed in the area No. 1. The *[image: Designer The connection settings 03]* button removes the selected connector. The first column of the area No. 1 displays whether the connector will connect when you click the **Connect** *[image: Designer The connection settings 04]* button or will not connect *[image: Designer The connection settings 05]*. These values are set using the *[image: Designer The connection settings 06]* and *[image: Designer The connection settings 07]* buttons.

The **Market data** and **Trades** fields display whether the connector supports market data and work with orders.

4. When the **Auto-connect** check box is selected, the connection automatically connects when **Designer** starts.
5. The area No. 2 displays the selected connector settings. When the *[image: Designer The connection settings 09]* is clicked, the program redirects to the documentation ([API](#)) on adjusting the selected connector.

Advanced settings are needed in special cases, and changing them is not recommended.

6. Clicking the **Check** button will check the connection of the current connector. If the check fails, a window with an error will appear, describing the reason for the unsuccessful connection. If the check is successful, a window will appear:

[image: Designer The connection settings 10]

7. The area No. 3 displays data types, which will be received or transmitted by the connector. By clicking the *[image: Designer The connection settings 11]*/ *[image: Designer The connection settings 12]* checkbox you can enable/disable transmission or reception for this type of data.

8. Clicking the **Network settings** button will open the **Proxy-server settings** window:

[image: Designer The connection settings 13]

Recommended content

[Strategies dashboard](#)

Connectors settings

Connectors settings

Some connectors require pre-configuration. Below are links to pages with their settings.

Stock, Futures and Options

Connectors	Connector settings	Graphical configuration
Alpaca	Configuration Alpaca	Graphical configuration Alpaca
AlphaVantage	Configuration AlphaVantage	Graphical configuration AlphaVantage
BarChart	Configuration BarChart	Graphical configuration BarChart
Blackwood (Fusion)	Configuration Blackwood (Fusion)	Graphical configuration Blackwood (Fusion)
CQG	Configuration CQG	Graphical configuration CQG COM
CQG	Configuration CQG	Graphical configuration CQG Continuum
E*TRADE	Configuration E*TRADE	Graphical configuration E*TRADE
IEX	Configuration IEX	Graphical configuration IEX
Interactive Brokers	Configuration Interactive Brokers	Graphical configuration Interactive Brokers
IQFeed	Configuration IQFeed	Graphical configuration IQFeed
ITCH	Configuration ITCH	Graphical configuration ITCH
OpenECry	Configuration OpenECry	Graphical configuration OpenECry
PolygonIO	Configuration PolygonIO	Graphical configuration PolygonIO
Quandl	Configuration Quandl	Graphical configuration Quandl
QuantFEED	Configuration QuantFEED	Graphical configuration QuantFEED
Rithmic	Configuration Rithmic	Graphical configuration Rithmic
Sterling	Configuration Sterling	Graphical configuration Sterling
Tradier	Configuration Tradier	Graphical configuration Tradier
Xignite	Configuration Xignite	Graphical configuration Xignite
Yahoo	-----	Graphical configuration Yahoo

Forex

Connectors	Connector settings	Graphical configuration

MetaTrader	-----	-----
DXtrade	Configuration DXtrade	Graphical configuration DXtrade
cTrader	Configuration cTrader	Graphical configuration cTrader
DukasCopy	-----	Graphical configuration DukasCopy
FXCM	Configuration FXCM	Graphical configuration FXCM
LMAX	Configuration LMAX	Graphical configuration LMAX
Oanda	Configuration Oanda	Graphical configuration Oanda

Cryptocurrencies

Connectors	Connector settings	Graphical configuration
Bibox	Configuration Bibox	Graphical configuration Bibox
Binance	Configuration Binance	Graphical configuration Binance
Binance History	Configuration Binance History	Graphical configuration Binance History
BingX	Configuration BingX	Graphical configuration BingX
Bitbank	Configuration Bitbank	Graphical configuration Bitbank
Bitget	Configuration Bitget	Graphical configuration Bitget
Bitexbook	Configuration Bitexbook	Graphical configuration Bitexbook
Bitfinex	Configuration Bitfinex	Graphical configuration Bitfinex
Bithumb	Configuration Bithumb	Graphical configuration Bithumb
Bitmart	Configuration Bitmart	Graphical configuration Bitmart
BitMax	Configuration BitMax	Graphical configuration BitMax
BitMEX	Configuration BitMEX	Graphical configuration BitMEX
BitStamp	Configuration BitStamp	Graphical configuration BitStamp
Bittrex	Configuration Bittrex	Graphical configuration Bittrex
BitZ	Configuration BitZ	Graphical configuration BitZ
WEX (BTC-e)	Configuration WEX (BTC-e)	Graphical configuration WEX (BTC-e)
ByBit	Configuration ByBit	Graphical configuration ByBit
Bybit History	Configuration Bybit History	Graphical configuration Bybit History
BW	Configuration BW	Graphical configuration BW
CEX.IO	Configuration CEX.IO	Graphical configuration CEX.IO
Coinbase	Configuration Coinbase	Graphical configuration Coinbase
CoinBene	Configuration CoinBene	Graphical configuration CoinBene

CoinCap	Configuration CoinCap	Graphical configuration CoinCap
Coincheck	Configuration Coincheck	Graphical configuration Coincheck
CoinEx	Configuration CoinEx	Graphical configuration CoinEx
CoinExchange	Configuration CoinExchange	Graphical configuration CoinExchange
Coinigy	Configuration Coinigy	Graphical configuration Coinigy
CoinHub	Configuration CoinHub	Graphical configuration CoinHub
Cryptopia	Configuration Cryptopia	Graphical configuration Cryptopia
Deribit	Configuration Deribit	Graphical configuration Deribit
Digifinex	Configuration Digifinex	Graphical configuration Digifinex
EXMO	Configuration EXMO	Graphical configuration EXMO
FatBTC	Configuration FatBTC	Graphical configuration FatBTC
Gate.io	Configuration Gate.io	Graphical configuration Gate.io
GDAX	Configuration GDAX	Graphical configuration GDAX
Gopax	Configuration Gopax	Graphical configuration Gopax
HitBTC	Configuration HitBTC	Graphical configuration HitBTC
Hotbit	Configuration Hotbit	Graphical configuration Hotbit
Huobi	Configuration Huobi	Graphical configuration Huobi
Idax	Configuration Idax	Graphical configuration Idax
Kraken	Configuration Kraken	Graphical configuration Kraken
Kucoin	Configuration Kucoin	Graphical configuration Kucoin
Latoken	Configuration Latoken	Graphical configuration Latoken
LBank	Configuration LBank	Graphical configuration LBank
Liqui	Configuration Liqui	Graphical configuration Liqui
Livecoin	Configuration Livecoin	Graphical configuration Livecoin
MEXC	Configuration MEXC	Graphical configuration MEXC
OKCoin	Configuration OKCoin	Graphical configuration OKCoin
OKEx	Configuration OKEx	Graphical configuration OKEx
OKEx History	Configuration OKEx History	Graphical configuration OKEx History
Poloniex	Configuration Poloniex	Graphical configuration Poloniex
Quoinex	Configuration Quoinex	Graphical configuration Quoinex
TradeOgre	Configuration TradeOgre	Graphical configuration TradeOgre
Upbit	Configuration Upbit	Graphical configuration Upbit

Yobit	Configuration Yobit	Graphical configuration Yobit
Zaif	Configuration Zaif	Graphical configuration Zaif
ZB	Configuration ZB	Graphical configuration ZB

DEX

Connectors	Connector settings	Graphical configuration
Aster	Configuration Aster	Graphical configuration Aster
edgeX	Configuration edgeX	Graphical configuration edgeX
Hyperliquid	Configuration Hyperliquid	Graphical configuration Hyperliquid
Ligther	Configuration Ligther	Graphical configuration Ligther
Paradex	Configuration Paradex	Graphical configuration Paradex

Common

Connectors	Connector settings	Graphical configuration
CSV	-----	Graphical configuration CSV
FIX protocol	Configuration FIX	Graphical configuration FIX
FAST protocol	Configuration FAST	Graphical configuration FAST
RSS	-----	Graphical configuration RSS

Simulator

Designer allows running the created strategies in the **Simulation** mode, to customize the **Simulation**, the following actions shall be performed:

1. When clicking the arrow next to the **Connect** *[image: Designer The quick access toolbar 00]* button, the **Emulator settings** button appears:

[image: Designer The connection settings 00]

2. When clicking the **Emulator settings** button, the **Emulator settings** window opens:

[image: Designer Properties emulation 00]

1. Simulator

- **Use emulator** – Use emulator.
- **Instruments** – Instruments.

2. Settings

- **Combine on touch** - During emulation, combine trades when the trade price touches the order price (i.e., equals the order price).
- **Market depth (lifetime)** - Maximum time, during which the order book is in the emulator. If during this time there was no update, the order book is erased. This property can be used to remove old order books when there are holes in the data.
- **Errors percentage** - The percentage value of the error when registering new orders. Value may be from 0 (no errors at all) to 100.
- **Latency** - The minimum value of delay for registered orders.
- **Reregistering** - Is re-registration of orders in the form of a single trade supported?
- **Buffering period** - Send responses in batches in a single package. The network delay and the buffered work of the stock exchange core are emulated.
- **Order ID** - The number, starting from which, the emulator will generate the identifiers for orders.
- **Trade ID** - The number, starting from which, the emulator will generate the identifiers for trades.
- **Transaction** - The number, starting from which, the emulator will generate the identifiers for order transactions.
- **Spread size** - Spread size in price increments. Used in determining the spread for the order book generation from tick trades.
- **Depth of book** - The maximum depth of the order book, that will be generated from ticks.
- **Number of volume steps** - The number of volume steps, for which the order is larger than the tick trade. Used when testing tick trades.
- **Portfolios interval** - Portfolio recalculation interval. If the interval equals zero, no recalculation is performed.
- **Change time** - Change time for orders and trades with the stock exchange time.
- **Time zone** - Information on the stock exchange time zone.
- **Price shift** - Price shift from the last trade, determining boundaries of maximal and minimal prices for

the next session.

- **Add extra volume** - Add extra volume to the book order when registering large volume orders.

Recommended content

[Chart](#)

Strategy gallery

Strategy Gallery

The **Strategy Gallery** provides the opportunity to download ready-made trading strategies to your computer. You can open the **Strategy Gallery** by clicking on the **Strategy Gallery** button in the **Common** tab.

To download a strategy to your computer, you need to:

- select the strategy of interest and click the Download button:
[image: Designer The gallery of strategies 00]
- the strategy will be added to the strategy tree in the **Backtest** section:
[image: Designer The gallery of strategies 01]

See Also

[Publishing Your Strategy](#)

Publish own strategy

Publishing Your Strategy

You can publish your strategy by clicking on the strategy with the left mouse button on the [Schemes](#) panel and selecting **Publish**:

[image: Designer_publish_00]

After clicking the **Publish** button, a window will open with the choice of export type. For more details, see the section [Exporting Strategies](#).

After choosing the export type, the [Installer](#) program is activated with the publishing parameters ([Installer](#) must be launched in advance):

[image: Designer_publish_01]

Fields that must be filled in:

- Name
- Description
- Nuget package identifier. This parameter is needed to set the link to the product in the store. For example, in the address <https://stocksharp.com/store/runner/>, the word **runner** is specified through this parameter.

Access to the **Free** or **Paid** level is only granted after contacting via email info@stocksharp.com. By default, the **Private** level is available, and it allows publishing strategies only in a private format (for selected users):

After clicking the **Save** button, the strategy will be sent to the StockSharp server.

When publishing updates, it is not required to enter all parameters again. Instead of entering product parameters, a window will appear for entering a note for the update:

[image: Designer_publish_02]

After clicking the **OK** button, a window will appear indicating a successful update:

[image: Designer_publish_03]

Export

Export

The Designer allows you to export any type of data: strategies, blocks, and indicators. There are several ways to export:

- In the **Schemes** panel, right-click on the strategy, block, or indicator. In the menu that appears, select **Export**.
- On the **Common** tab, press the **Export** button:

[image: Designer Export strategies 00]

After pressing **Export**, depending on the content type, a window will appear:

- for a **scheme**:
[image: Designer Export strategies 01]
 - scheme - export the scheme as is. The **Standalone** mode is required for schemes that use their own elements or indicators. In this case, all inner elements will be exported within the strategy diagram.
 - code - convert the scheme into C# code.
 - DLL - compile the scheme into a DLL. Suitable if you need to keep the code confidential.
- for **code**:
[image: Designer Export strategies 02]
 - scheme - export the code as a JSON file, which will include both the code itself and the references needed for compiling this code.
 - code - export the code as is.
 - DLL - compile the code into a DLL. Suitable if you need to keep the code confidential.
- for a **dll** a file selection window will appear.

See Also

[Running Strategies Outside of Designer](#)

Import

Import

The Designer allows you to import any type of data: strategies, blocks, and indicators. There are several ways to import:

- In the **Schemes** panel, right-click on the strategy folder, and in the menu that appears, select **Import**.
- On the **Common** tab, press the **Import** button, and in the menu that appears, select **Strategy**, **Own element**, or **Indicator**:

[image: Designer Import strategies 00]

If data of the same name already exists, a window will appear offering to overwrite or add with a different name:

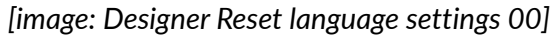
[image: Designer Import strategies 01]

Reset settings

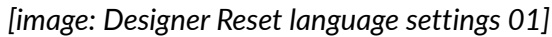
Reset settings

CAUTION!!! Prior to resetting please read this paragraph through!

On the **Quick access panel**, located by default at the top of the [Designer](#), window, there is a **Settings**  button. Clicking the  button allows changing the program launch mode, interface language or totally reset the [Designer](#) settings.



When **Reset settings** is selected, the confirmation window opens.



After clicking the **OK** button all [Designer](#) settings will be reset to default values. The directory with the [Designer](#) will be cleared completely, **all created strategies and downloaded instruments and other information stored in the settings directory will be DESTROYED.**

Recommended content

[Settings directory](#)

Settings directory

Settings directory

The following directories are important for work with the [Designer](#):

1. The directory, where the [Designer](#), was installed, from this folder you can launch [Designer](#) by running the **Designer.exe** file. Or update Designer, by running the **Designer.Update.exe** file. Deleting this directory will delete Designer, however, the Designer settings will not be deleted.
2. The [Designer](#) settings directory is located in the folders of user's documents ...\\StockSharp\\Designer\\ (for example, c:\\Users\\User\\Documents\\StockSharp\\Designer\\). Deleting this directory will cause all Designer settings to be reset to the default, **all created strategies and downloaded instruments and other information stored in the settings directory will be DESTROYED**.

[image: Designer Directory and edit the data manually 00]

This directory contains the following folders and files:

- The **Compositions** folder, containing in the form of XML files all cubes, contained in the Composite elements folder on the [Schemas](#) panel. Deleting files from this directory will delete the **Composite element** from the **Composite elements** folder on the Schemas panel. You should not manually edit the files in this directory, this can lead to a malfunction of the corresponding **Composite elements** cube.
- The **LiveStrategies** folder, containing in the form of XML files all cubes, contained in the **Trading** folder on the [Schemas](#) panel. Deleting files from this directory will delete the strategy from the Trading folder on the Schemas panel. You should not manually edit the files in this directory, this can lead to a malfunction of the corresponding strategy.
- The **Logs** folder, containing all [Designer](#) logs, which simplifies the Designer troubleshooting.
- The **SourceCode** folder, containing in the form of XML files all cubes, contained in the SourceCode folder on the [Schemas](#) panel. Deleting files from this directory will delete the **SourceCode** cube from the **SourceCode** folder on the **Schemas** panel. You should not manually edit the files in this directory, this can lead to a malfunction of the corresponding **SourceCode** cube.
- The **Strategies** folder, containing in the form of XML files all cubes, contained in the Strategies folder on the [Schemas](#) panel. Deleting files from this directory will delete the strategy from the **Strategies** folder on the Schemas panel. You should not manually edit the files in this directory, this can lead to a malfunction of the corresponding strategy. If you manually add a strategy file to this folder and reboot Designer, the strategy will appear in the **Strategies** folder on the **Schemas** panel.
- The **Storage** folder - market data, downloaded by [Designer](#) into the corresponding [Market data storage](#). The folder is created at the Market data storage creation. By default, the path to this folder is specified. Deleting this folder will delete all the downloaded market data from the corresponding storage. If the storage contains CSV format files, then they can be edited with a standard notepad or MS Excel. The BIN files cannot be edited manually.
- The **exchange.csv** and **exchangeboard.csv** files contain the list of **Stock Exchanges**, list of instrument codes and trading modes. These files can be edited with standard notepad or MS Excel.
- The **security.csv** file contains all received and created securities by all sources. Deleting this file will delete all securities in [Designer](#). Adding new instruments is described in [Download instruments](#), [Create instrument](#) sections. This file can be edited with standard notepad or MS Excel.
- The **portfolio.csv** and **position.csv** files contain all received and created portfolios and current position on them. Deleting this file will delete all corresponding data in [Designer](#). If information on portfolios is received by Designer at each connection, information on the positions can be lost irrevocably. These files can be edited with standard notepad or MS Excel.

- The **settings.json** file contains current settings. This file is created by [Designer](#) when settings were changed or when closing the program. Deleting this file will reset current settings to the default values. You should not manually edit this file, this can lead to a malfunction of [Designer](#).

If it is necessary to edit individual files or reset the Designer settings, it will be useful to backup both individual files and the entire directory.

Recommended content

[Update to the new version](#)

Terminal

Terminal

Terminal is a free trading terminal. Its main advantage is the simultaneous connection to a variety of trading boards, including those where there is no own terminal.

[image: Terminal main 00]

Advantages and features of Terminal:

1. Supports many sources: [Connectors](#).
2. Trade from the chart.
3. Arbitrary time frames.
4. Support for Volume, Tick, Range, PnF, Renko Candles.
5. Clusters and box plots.
6. Any markets: stocks, futures, Forex, bitcoins.

Recommended content

[Installing Terminal](#)

Installing Terminal

Installing Terminal

[Install and remove apps](#)

[Update to the new version](#)

Update to the new version

[Update apps](#)

Recommended content

[Installing Terminal](#)

Quick start

Quick start

When you start [Terminal](#) for the first time, the following window will be displayed:

[image: Terminal Quick start 00]

You must select the start mode and click **OK**.

After selecting the start mode, the terminal window opens with the default set of components.

Let's go to the connection settings and select the required connection. How to set up the connection is described in the [Connection settings](#) section.

[image: Terminal Quick start 011]

The next step is to connect by clicking the **Connect** *[image: Designer The quick access toolbar 00]* button.

Clicking the **Add** *[image: Designer Creation tool 00]* button on the chart panel we add a new chart area. Right clicking the mouse button on the chart area, we add candles to it by the instrument of interest to us. We can add indicators, own trades and orders to the chart. There is also a possibility to register orders from the chart. For more information on how to work with the chart, see the [Chart](#) section.

[image: Terminal Quick start 01]

Clicking the **Add** *[image: Designer Creation tool 00]* button on the Instrument panel adds the instruments you want to watch. The best price data will be displayed here. For more information on how to work with the Instrument panel, see the [Instruments](#) section.

[image: Terminal Quick start 02]

In the order book, after clicking the **Settings** *[image: Designer Schedule 01]* button, a panel appears where you can specify the **Instrument** and the **Portfolio** for the trades. Here you can also adjust the order book depth. For more information on how to work with an order book, see the [Order book](#) section.

[image: Terminal Quick start 03]

Let's register the first few orders. Orders can be registered either by clicking the **Buy/Sell** buttons or by clicking the cells in the **Bid/Offer** columns of the order book itself. The order panel displays all your orders. When you right-click an order, you will see a panel that lets you submit a new order, cancel, or change the selected order. For more information on how to work with the order panel, see the [Orders](#) section.

[image: Terminal Quick start 04]

In the Trades panel, you can view trades by instruments. For more information on how to work with the Trades panel, see the [Trades](#) section.

[image: Terminal Quick start 05]

If necessary, you can add additional components. All components are described in the [Components](#) section.

[image: Terminal Quick start 06]

Ribbon

Ribbon

The main element of the [Terminal](#) user interface is a **Ribbon** that is located along the top of the application window. Using the ribbon, you can quickly find the necessary commands. The commands are arranged in logical groups, collected on tabs. To go to the required tab, just click its name. Each tab is associated with the type of action being performed.

The **Common** tab, which is opened by default after startup, contains the elements that you might need at the beginning of the work. In the **Common** tab, you can add [Workspace](#) or open [Logs](#), [Portfolios](#), [Boards](#), or add [Components](#) . Also in the **Common** tab, you can select the theme of [Terminal](#).

[image: Terminal Tape 00]

Recommended content

[Workspace](#)

Workspace

Workspace

See

[Workspace](#)

Logs

Logs

See

[Logs](#)

Portfolios

Portfolios

See

[Portfolios](#)

Board editor

Board editor

See

[Boards](#)

Risk management

Risk management

In the **Risk** panel, you can set risk control settings.

At the bottom of the panel, you have to select the **Risk-rule**, configure the trigger condition for the **Risk-rule** and the action (Close positions, Stop trading, Cancel orders), which will be executed when the **Risk-rule** condition is triggered.

[image: Terminal Risk Rule]

The list of risk rules and their description is available in the section [Risk management](#).

Components

Components

During trading, you may need to fully control the process. For complete control and analysis, **Terminal** provides graphical components that you can add by selecting them in the **Components** group of the **Ribbon**:

[image: Designer Components]

- **Instruments** - is a table with instruments that displays information about all selected instruments.
- **Level 1** - is a table with a history of changing Level 1 for the selected instruments.
- **Buy/Sell** - is a panel group that displays the best price information for the selected instruments, and it allows you to buy or sell the required volume for the selected instruments.
- **Order book** - is a table of limit orders for purchase and sale.
- **Chart** - allows you to draw candles and indicators for the selected instrument.
- **P&L equity** - is a chart of Profit/Loss (unrealized), Profit/Loss (realized) and commission.
- **Trades** - is a table with trades, that displays complete information about all trades of the strategy.
- **Orders** - is a table with orders, that displays complete information about all orders of the strategy.
- **Conditional orders** - is a table with conditional orders, that displays complete information about all conditional orders of the strategy.
- **Positions** - is a table of the current Position.
- **Tick trades** - is a trade table that displays complete information about all trades for the selected instruments.
- **Positions** - is a chart of the Position.
- **News** - displays news received from connections.
- **Positions (options)** - is a graphic display of the position by options.
- **Option desk** - is a table of the main parameters of the selected options for the underlying instrument.
- **Smile volatility** - is a graphic representation of the expected volatility level for options with the same underlying asset and different strikes.
- **Order log** - is a table with orders, which displays complete information about all orders for the selected instruments.

Instruments

Instruments

The **Instruments** component is a table with instruments that displays information about all selected instruments.

To add a new instrument, click the *[image: Designer Creation tool 00]* button.

It is also possible to configure notifications for events of the selected instruments - [Notification settings](#).

[image: Terminal securities 00]

Recommended content

[Level 1](#)

Level 1

Level 1

The **Level 1** component is a table with a history of changing **Level 1** for the selected instruments.

Level 1 has a filter to select the necessary instruments. It is also possible to configure notifications for events of the selected instruments - [Notification settings](#).

[image: Terminal Level1 00]

Recommended content

[Buy/Sell](#)

Buy/Sell

Buy/Sell

The **Buy/Sell** component is a panel group that displays the best price information for the selected instruments, and also allows you to buy or sell the required volume.

To add a new instrument, click the *[image: Designer Creation tool 00]* button.

[image: Terminal Buy Sell 00]

Recommended content

[Order book](#)

Order book

Order book

The **Order book** component is a table of limit orders for the purchase and sale.

When you click the settings *[image: Designer The quick access toolbar 02]* button, a panel appears in which you can set the necessary instrument and portfolio, as well as configure the order book display options.

Orders can be registered either by clicking the **Buy/Sell** buttons or by clicking the cells in the **Bid/Offer** columns in the order book itself.

[image: Terminal Panel Market Depth]

Recommended content

[Positions](#)

Chart

Chart

The **Chart** component allows you to draw candles and indicators for the selected instrument.

[image: Terminal Panel graphics 00]

To add a new area, click the *[image: Designer Creation tool 00]* button.

To add a new element to the chart, right-click anywhere in the chart area and select the required element; available options include candles, indicators, orders, and own trades.

Clicking *[image: Designer Schedule 01]* opens the settings for the graphic element.

[image: Designer Schedule 02]

You can register orders from the chart. To do this, first specify the **Instrument** and **Portfolio** for which orders will be registered.

Orders for purchase will be registered by the combination of keys **Ctrl+Left mouse button**.

Orders for sale will be registered by the combination of keys **Ctrl+Right mouse button**.

[image: Terminal Panel graphics 01]

In the graphical element settings, you can set the required chart style: Japanese candles, bars, box chart, cluster profile, etc.

[image: Designer Schedule 04]

For a box chart, candles can also be grouped; the grouping order is set in the fields: the multiplier of the 2nd timeframe and the multiplier of the 3rd timeframe.

[image: Designer Schedule 05]

There is a toolbar above the chart, where you can select auto scroll, auto-zoom, legend modes and other general chart settings. You can also select the elements to draw on the chart: lines, levels, pointers, rectangle, text.

[image: Designer Schedule 03]

Recommended content

[Orders](#)

P&L chart

P&L chart

See

[P&L equity](#)

Trades

Trades

See

[Trades](#)

Orders

Orders

The **Orders** component is a table with orders, which displays complete information about all the orders. When you right click the mouse button on the order, a panel appears with which you can register a new order, cancel or change the selected order.

[image: Terminal Quick start 04]

When you click the **Order Registration** button, a window appears. To register a new order, you have to fill it out and click the **Send** button.

[image: Terminal OrderPanel 01]

When you click the **Change Order** button, a window appears. To change the order, you have to make the required changes and click the **Send** button.

[image: Terminal OrderPanel 02]

Recommended content

[Trades](#)

Conditional orders

Conditional orders

The **Conditional** orders component is a table with orders, which displays complete information about all conditional orders. When you right click the mouse button on the order, a panel appears with which you can register a new conditional order, cancel or change the selected conditional order.

[image: Terminal Orders conditional panel 01]

When you click the **Order Registration** button, a window appears. To register a new conditional order, you have to fill it out and click the **Send** button.

[image: Terminal Orders conditional panel 02]

When you click the **Change Order** button, a window appears. To change the conditional order, you have to make the required changes and click the **Send** button.

[image: Terminal Orders conditional panel 03]

Recommended content

[Trades](#)

Positions

Positions

See

[Positions](#)

Trades feed

Trades feed

See

[Tick trades](#)

News

News

The **News** component displays news received from connections.

To start receiving news, click the **Receive news** button.

The News panel allows you to configure notifications for selected events – see [Notification settings](#).

[image: Terminal news 00]

Recommended content

[Positions \(options\)](#)

Positions (options)

Positions (options)

Positions (options) is a graphic display of the position by options.

To display the position by options, you have to select the underlying asset and options based on the prices of which the position will be built.

In addition, you can specify a filter for the exact expiration date of options and filters for minimum/maximum strikes.

[image: Terminal Graph options positions 00]

Recommended content

[Option desk](#)

Option desk

Option desk

The **Option desk** component is a table of the main parameters of the selected options for the underlying instrument.

To display the option desk, you have to select the underlying asset and the necessary options.

In addition, you can specify a filter for the exact expiration date of options and filters for minimum/maximum strikes.

[image: Terminal option desk 00]

Recommended content

[Smile volatility](#)

Smile volatility

Smile volatility

The **Smile volatility** component is a graphic representation of the expected volatility level for options with the same underlying asset and different strikes.

To display the volatility smile, you have to select the underlying asset and options based on the prices of which the volatility smile will be built.

In addition, you can specify a filter for the exact expiration date of options and filters for minimum/maximum strikes.

[image: Terminal smile of volatility 00]

Recommended content

[Order log](#)

Order log

Order log

The **Order log** component is a table with orders that displays complete information about all orders for the selected instruments.

The **Order log** has a filter for selecting the necessary instruments. It is also possible to configure notifications for events of the selected instruments - [Notification settings](#).

[image: Terminal orderlog 00]

Recommended content

[Market data storage](#)

Connection settings

Connection settings

See

[Connections settings](#)

Market data storage

Market data storage

See

[Market data storage](#)

Indicators

See [List of indicators](#)

Notifications

Notifications

Notifications are a system for alerting a user about a pre-specified event that occurs.

Notifications can be specified for a wide range of trading events for the following objects: portfolio, position, Level 1, news. For example, exceeding the weighted average position price of a certain value or the appearance of news with a specific title.

Notifications can be of the following forms:

- **Window** - a small pop-up window with a message will appear in the corner of the screen.
- **Music** - the music will be played.
- **SMS** - the message will be sent by SMS.
- **Email** - the message will be sent by email.
- **Voice** - the message will be spoken by the computer-generated voice.
- **Log** - the message will be sent to the [Logs](#).
- **Disabled** - the notification will not be displayed.

To set a notification, you have to click the *[image: Designer Creation tool 00]* button in the notification panel ([Notification panel](#)) or click the *[image: Designer Alert Bell]* button directly in the panels that support notifications.

For information on how to configure notifications, see the [Notifications setup](#) section.

Notification panel

Notification panel

The **Notifications** panel is a table that displays information about all currently set notifications.

[image: Terminal Notifications Panel]

In the **Notifications** panel, you can edit *[image: Designer Creating a repository of historical data 01]* an existing notification, delete *[image: Designer Creating a repository of historical data 02]* it, or set a new notification *[image: Designer Creating a repository of historical data 00]*.

Notifications setup

Notifications setup

To set a notification, you have to click the *[image: Designer Creation tool 00]* button in the notification panel ([Notification panel](#)) or click the *[image: Designer Alert Bell]* button directly in the panels that support notifications ([Portfolios](#), [Trades feed](#), [News](#), [Level 1](#)).

[image: Designer Notifications Setting]

Notifications can be configured for changes in the following data types: Portfolio, Customer code, Broker, Depository, Server time, Trade, Data type, Cancellation, Order ID, Order ID (string), Order ID (board), Derivative, Derivative (string), Price, Volume (order), Volume (trade), Visible volume, Direction, Remainder, Order type, Status, Comment, Message to order, System order, Order expiration time, Execution condition, Order ID, Order ID (string), Price, Trade initiator, Open interest, Error, Condition, Uptrend, Commission, Delay, Slippage, ID (user), Currency, P/L, Position, Market Maker.

Notifications can be of the following forms:

- **Window** – a small pop-up window with a message will appear in the corner of the screen.
- **Melody** – a melody will be played.
- **SMS** – a message will be sent via SMS.
- **Email** – the message will be emailed.
- **Speech** – the message will be spoken by a computer-generated voice.
- **Log** – the message will be sent to the Designer logs window.
- **Off** – the notification will not be shown.

[Reset settings](#)

Reset settings

See [Reset settings](#).

Hydra

Hydra

[image: hydra main]

The [Hydra](#) program is intended for automatic downloading of market data (instruments, candles, tick trades and order books) from various sources and for storing it locally. Data can be saved in two formats: the special binary format [Hydra](#) (BIN), which provides the maximum compression rate, or the plain text format CSV, which is useful for analyzing the data in other programs. Later, the stored information becomes available for trading strategies (for more information about strategy testing see the [Backtesting](#) section). Access to data can be obtained directly through the [StorageRegistry](#) (for more details see the [Market-data storage](#)), or through the usual export to formats such as [Excel](#), XML or TXT (for more details see the [Installation and operation](#) section).

At the same time, [Hydra](#) can use both historical and real-time data sources (for example, connecting to [OpenECry](#) or [Rithmic](#) to obtain order books). This is possible through the extensible plug-in source model.

The plug-in model lets you develop your own sources. For details on creating and installing custom sources in [Hydra](#), see the [Creating source](#) section.

First start

First start

Sources samples

Source samples

Connection settings in [Designer](#), [Terminal](#), [Hydra](#).

You can find out more about connection settings via various connectors [here](#)

- [Oanda](#)
- [Interactive Brokers](#)
- [Bitmex](#)
- [Binance](#)

Oanda

Oanda

Interactive Brokers

Interactive Brokers

Bitmex

Bitmex

Binance

Binance

Instruments downloading

Downloading instruments

Building candles

Building candles

Building order books

Building order books

Saving format

Saving format

Indicators

Indicators

Analytics

Analytics

Import task

Import task

Export task

Export task

Converter task

Converter task

Installing Hydra

Installing Hydra

[Install](#) [and remove apps](#)

[Update to the new version](#)

Update to the new version

[Update apps](#)

Recommended content

[First start](#)

First start

First start

The first time you run it, the following window for selecting data sources appears. You can also open this window in the **Common** tab by selecting **Add => Sources**.

[image: hydra source add]

In the window, mark the required sources. You can use filters by region, board, data type, payment, real-time or not. When the selection is complete, click **OK**. Then the program will offer to enable the utilities. For more details on working with the utilities, see the [Utilities](#) section. Click **OK**.

[image: hydra first started utilities 00]

After that, the sources will be added to the left panel of the main application window.

[image: hydra Quick start 01]

Downloading a tool for uploading market data:

Before you start downloading market data, you need to set up the securities for which you need to get market data.

After adding market data sources, panels of added sources will be opened in the central part, where a list of securities is displayed. If the panel is closed, then to open it, double click the source logo in the list on the left side of the program.

For example, download the SBER@TQBR securities (stock of Sber Bank) from the ФИНАМ source.

[image: hydra choose market data]

| IMPORTANT! Data is downloaded only for the securities added to the securities list

1. Adding securities.

At the first start, the program will offer to download all the securities at once for the selected source. Further, the user will download the securities himself. Initially, the security database in [Hydra](#) is empty, there is only an auxiliary security **ALL@ALL**. When this security is selected, data will be downloaded for all available securities for this source.

To add a security, click on the **Add** *[image: hydra add]* button. Then a window for downloading the security will open. *[image: hydra securities]*

To download the securities, you have to click the corresponding **Download securities** button.

After that, a menu will appear on the screen, in which the user can select **Download all securities**.

[image: hydra securities choose all]

Or, for a number of sources, [configure](#) the securities you need to download.

After the securities are received, the window will look like this. *[image: hydra security full list]*

It will list all the securities available for adding. For a quick search, you can enter its name in the appropriate field.

To select a security, double-click on it and it will move to the right side of the list. *[image: hydra security full list 00]*

Then it will move to the right side of the table. *[image: hydra security full list 01]*

The selected instruments will be displayed in the **Securities** table, which is a tree-structured table. The main element of which is the security, the additional one is the market data types that will be received for this security.

2. For each selected security, you should select the market data types required for downloading.

If all the necessary security parameters are not set, the icon *[image: hydra zero]* will appear in the left column of the security line. *[image: hydra type market data choose]*

Let's select to download **Ticks** and **Candles Time Frame 5**.

At the bottom of the source window is a panel with buttons for setting up the data and instruments to be received. *[image: hydra Quick start 02 00]*

The following operations can be performed at this panel:

- Configure the amount of information received using the buttons: **Trades, Order Books, Candles, Order Log, Level 1, Own Transactions**. Lists of available market data types vary for different sources.
- Specify the required Time Frame for the loaded candles. Time Frame of the received candles is different for different sources.*[image: hydra Quick start 02]*
- Set the required period for downloading market data. The period can also be configured directly in the market data window. To do this, you should select the beginning and end of the period.

If the user does not specify the end date of the period, the program downloads all data available for the current date. If the source supports the transfer of market data in real time, then if there is no end date for the period, the market data will be downloaded in real time.

Let's set the period for which you need to download the market data.*[image: hydra Quick start 02 01]*

- Specify what the market data will be built from. If this parameter is not specified, then the candles available in the source will be received. If the user specifies the market data type, then candles will be built from the specified market data type. For example, candles can be built from the last trade price, the order book spread (usually for the Forex market), volatility or the best price.

This function is convenient to use if the source does not allow receiving data for candle plotting. In this case, candles are plotted based on the averaged data values.*[image: hydra candle build type]*

The user also has the ability to select **Custom type** of candles to customize the received data.

- After selecting a security, market data type and setting the period, you should click on the **Start** button. After that, the market data download will begin.

The work process can be observed in the special **Logs** tab, which is fixed at the bottom of the program. In addition, logs are stored in files in the local folder.

[image: hydra main start]

Also, the user can add **additional sources**.

After the market data is downloaded, the user can **view market data**, **plot candles**, save or **export in various formats**.

Watch **video tutorial**.

Data sources

Data sources

[Hydra](#) has connections to various data sources. In addition to receiving historical data, the program can receive market data from brokers, while market data is received in real time. Connection to the sources is carried out using the connectors provided to the user.

The user is provided with free connectors, and he can also purchase paid ones, which allow receiving market data from paid sources.

At the moment, [Hydra](#) supports the following sources:

Stock, Futures and Options

Connectors	Connector settings	Graphical configuration
Alpaca	Configuration Alpaca	Graphical configuration Alpaca
AlphaVantage	Configuration AlphaVantage	Graphical configuration AlphaVantage
BarChart	Configuration BarChart	Graphical configuration BarChart
Blackwood (Fusion)	Configuration Blackwood (Fusion)	Graphical configuration Blackwood (Fusion)
CQG	Configuration CQG	Graphical configuration CQG COM
CQG	Configuration CQG	Graphical configuration CQG Continuum
E*TRADE	Configuration E*TRADE	Graphical configuration E*TRADE
IEX	Configuration IEX	Graphical configuration IEX
Interactive Brokers	Configuration Interactive Brokers	Graphical configuration Interactive Brokers
IQFeed	Configuration IQFeed	Graphical configuration IQFeed
ITCH	Configuration ITCH	Graphical configuration ITCH
OpenECry	Configuration OpenECry	Graphical configuration OpenECry
PolygonIO	Configuration PolygonIO	Graphical configuration PolygonIO
Quandl	Configuration Quandl	Graphical configuration Quandl
QuantFEED	Configuration QuantFEED	Graphical configuration QuantFEED
Rithmic	Configuration Rithmic	Graphical configuration Rithmic
Sterling	Configuration Sterling	Graphical configuration Sterling
Tradier	Configuration Tradier	Graphical configuration Tradier
Xignite	Configuration Xignite	Graphical configuration Xignite

Yahoo	-----	Graphical configuration Yahoo

Forex

Connectors	Connector settings	Graphical configuration
MetaTrader	-----	-----
DXtrade	Configuration DXtrade	Graphical configuration DXtrade
cTrader	Configuration cTrader	Graphical configuration cTrader
DukasCopy	-----	Graphical configuration DukasCopy
FXCM	Configuration FXCM	Graphical configuration FXCM
LMAX	Configuration LMAX	Graphical configuration LMAX
Oanda	Configuration Oanda	Graphical configuration Oanda

Cryptocurrencies

Connectors	Connector settings	Graphical configuration
Bibox	Configuration Bibox	Graphical configuration Bibox
Binance	Configuration Binance	Graphical configuration Binance
Binance History	Configuration Binance History	Graphical configuration Binance History
BingX	Configuration BingX	Graphical configuration BingX
Bitbank	Configuration Bitbank	Graphical configuration Bitbank
Bitget	Configuration Bitget	Graphical configuration Bitget
Bitexbook	Configuration Bitexbook	Graphical configuration Bitexbook
Bitfinex	Configuration Bitfinex	Graphical configuration Bitfinex
Bithumb	Configuration Bithumb	Graphical configuration Bithumb
Bitmart	Configuration Bitmart	Graphical configuration Bitmart
BitMax	Configuration BitMax	Graphical configuration BitMax
BitMEX	Configuration BitMEX	Graphical configuration BitMEX
BitStamp	Configuration BitStamp	Graphical configuration BitStamp
Bittrex	Configuration Bittrex	Graphical configuration Bittrex
BitZ	Configuration BitZ	Graphical configuration BitZ
WEX (BTC-e)	Configuration WEX (BTC-e)	Graphical configuration WEX (BTC-e)
ByBit	Configuration ByBit	Graphical configuration ByBit
Bybit History	Configuration Bybit History	Graphical configuration Bybit History

BW	Configuration BW	Graphical configuration BW
CEX.IO	Configuration CEX.IO	Graphical configuration CEX.IO
Coinbase	Configuration Coinbase	Graphical configuration Coinbase
CoinBene	Configuration CoinBene	Graphical configuration CoinBene
CoinCap	Configuration CoinCap	Graphical configuration CoinCap
Coincheck	Configuration Coincheck	Graphical configuration Coincheck
CoinEx	Configuration CoinEx	Graphical configuration CoinEx
CoinExchange	Configuration CoinExchange	Graphical configuration CoinExchange
Coinigy	Configuration Coinigy	Graphical configuration Coinigy
CoinHub	Configuration CoinHub	Graphical configuration CoinHub
Cryptopia	Configuration Cryptopia	Graphical configuration Cryptopia
Deribit	Configuration Deribit	Graphical configuration Deribit
Digifinex	Configuration Digifinex	Graphical configuration Digifinex
EXMO	Configuration EXMO	Graphical configuration EXMO
FatBTC	Configuration FatBTC	Graphical configuration FatBTC
Gate.io	Configuration Gate.io	Graphical configuration Gate.io
GDAX	Configuration GDAX	Graphical configuration GDAX
Gopax	Configuration Gopax	Graphical configuration Gopax
HitBTC	Configuration HitBTC	Graphical configuration HitBTC
Hotbit	Configuration Hotbit	Graphical configuration Hotbit
Huobi	Configuration Huobi	Graphical configuration Huobi
Idax	Configuration Idax	Graphical configuration Idax
Kraken	Configuration Kraken	Graphical configuration Kraken
Kucoin	Configuration Kucoin	Graphical configuration Kucoin
Latoken	Configuration Latoken	Graphical configuration Latoken
LBank	Configuration LBank	Graphical configuration LBank
Liqui	Configuration Liqui	Graphical configuration Liqui
Livecoin	Configuration Livecoin	Graphical configuration Livecoin
MEXC	Configuration MEXC	Graphical configuration MEXC
OKCoin	Configuration OKCoin	Graphical configuration OKCoin
OKEx	Configuration OKEx	Graphical configuration OKEx
OKEx History	Configuration OKEx History	Graphical configuration OKEx History
Poloniex	Configuration Poloniex	Graphical configuration Poloniex

Quoinex	Configuration Quoinex	Graphical configuration Quoinex
TradeOgre	Configuration TradeOgre	Graphical configuration TradeOgre
Upbit	Configuration Upbit	Graphical configuration Upbit
Yobit	Configuration Yobit	Graphical configuration Yobit
Zaif	Configuration Zaif	Graphical configuration Zaif
ZB	Configuration ZB	Graphical configuration ZB

DEX

Connectors	Connector settings	Graphical configuration
Aster	Configuration Aster	Graphical configuration Aster
edgeX	Configuration edgeX	Graphical configuration edgeX
Hyperliquid	Configuration Hyperliquid	Graphical configuration Hyperliquid
Ligther	Configuration Ligther	Graphical configuration Ligther
Paradex	Configuration Paradex	Graphical configuration Paradex

Common

Connectors	Connector settings	Graphical configuration
CSV	-----	Graphical configuration CSV
FIX protocol	Configuration FIX	Graphical configuration FIX
FAST protocol	Configuration FAST	Graphical configuration FAST
RSS	-----	Graphical configuration RSS

Select source

Select source

To add a new market data source, you need to:

- Go to the **Common** tab
- Select **Add**
- Select **Sources**

[image: hydra source choose]

Then a list of sources will appear. The user can select several sources at once.

[image: hydra source add]

You can also create multiple instances of the same source. For example, multiple instances of **Interactive Brokers** that will save downloaded data to different folders.

[image: hydra two source]

In order for the source to start downloading data after clicking the **Start** button, it should be enabled. To do this, select the source icon in the left panel and use the *[image: hydra source on off]* button to switch it on or off. You can perform this operation before or after adding instruments for downloading.

Unnecessary sources can be deleted using the *[image: hydra delete]* button.

Source settings can be changed in the **Properties** panel on the right side.

[image: hydra setting source button]

Common settings for all sources can be found in the [Common connection settings](#) item.

Each connection has its own characteristics. The link [List of connectors](#) provides a list of connectors and their settings.

Watch [video tutorial](#)

Common connection settings

Common connection settings

Connection properties common to all sources.

[image: hydra settings connectionparams]

Connection

- **Connection** - Connection settings.
- **Title** - Title of the task.
- **Start date (ticks)** - From what date to start downloading data.
- **Work from** - Start time.
- **Work until** - End time.
- **Interval of work** - Operation interval.
- **Data directory** - The directory with data where the final files in StockSharp format will be saved.
- **Format** - Data format: BIN/CSV.
- **Max. errors** - The maximum number of errors after which the task will be stopped. The default is 0, which means ignoring the number of errors.
- **News** - Download news.
- **Dependency** - The task that must be completed before starting the current one.
- **Update securities** - Update securities when connected.

Instruments list

Setting up the security for downloading.

For a number of sources, it is possible to make settings for downloading the necessary securities.

Let's take an example of downloading securities from the **Interactive Brokers** source:

1. Select download securities and click on the **Extended condition** button.*[image: hydra choose securitiy]*
2. After that, a list of advanced settings for the downloaded security will open.*[image: hydra choose securitiy 00]*
3. You need to download a security that meets the following parameters: Stock of APPLE, currency - US Dollar. To do this, we set the security parameters as shown below and click **OK**.*[image: hydra choose securitiy 01]*

After that, the [Hydra](#) program will download all the securities that meet the specified parameters.
[image: hydra choose securitiy 02][image: hydra choose securitiy 03]

Setting allows the user to select various parameters for the downloaded security.

So, for example, the user can select:

- **Price and volume step**
- **Minimum and maximum volumes**
- **External ID** *[image: hydra choose securitiy 04]*
- For options, it allows setting the **Underlying Asset** and **Asset Type** (type of the underlying security).
[image: hydra choose securitiy 05]

Watch [video tutorial](#)

Custom candles

Custom candles

The user can select a **Custom type** of candles and independently select which candles will be built, while the candles will be built "on the fly", that is, immediately.

[image: hydra type candle 00 00]

Let's consider an example of such a building. **Bitmex** exchange does not provide a possibility to receive candles with a Time Frame of 10 minutes.

[image: hydra type candle 00 01]

The sequence of obtaining such candles:

1. Select **Custom** candles
2. In the settings, we specify **TF** candles and a period of 10 minutes
3. In the source, we specify from what the candles will be built - **Order Log** *[image: hydra type candle 00 02]*
4. We set the period. As you can see, next to the candle name appeared the **Generated** indication.*[image: hydra type candle 00 03]*
5. Click on start and the data starts downloading.*[image: hydra type candle 00 04]*
6. Let's go to the candles section and [see the downloaded data](#).*[image: hydra type candle 00 06]*

As you can see, the data has been successfully received.

Consider an example when we need to get a [RangeCandleMessage](#):

1. Select **Custom** candles.
2. In the settings, specify the Range candles and the volume 10.
3. In the source, we specify from what the candles will be built - **Ticks**.*[image: hydra type candle 00 07]*
4. We set the period.
5. Click on start and the data starts downloading.*[image: hydra type candle 00 08]*
6. Let's go to the candles section and [see the downloaded data](#).*[image: hydra type candle 00 09]*

Working with data

Working with data

After the market data is received, the user can start working with it.

The program allows the following actions:

- [View and export](#)
- [Candles generation](#)
- [Export data](#)

View and export

View and export

The received [Hydra](#) data can be viewed in special panels.

To do this, on the Common tab, click on one of the following buttons: [Ticks](#), [Order books](#), [Candles generation](#), [Order log](#), [Level 1](#), [News](#), [Transactions](#), [Option desk](#), [Indicators](#), [Positions](#).

Or right-click on the required data type, as shown in the figure, as well as double-clicking on the required data type.

[image: hydra view export]

Each panel contains a general interface of the settings as follows:

[image: hydra export 00]

- The top line indicates the market data storage and its format (BIN or CSV).
- The bottom line sets the period for which the data will be requested. When you click the **Select security** button, the security selection window will appear, in which you can select one or several securities. If several securities are selected, then during the subsequent export to Excel or CSV, the program will automatically sort the data of different securities into different files.
- If, when building a table with data, the amount of downloaded data exceeds the set limit, then a window will appear on the screen:*[image: hydra tick limit]*
It is necessary to increase the limit of downloaded data.
- If the data was received from sources whose time zone does not match the current time zone, you can adjust the time zone. After building, the data will be shown in the zone selected by the user. *[image: hydra TZ]*
- Since a number of sources do not provide a possibility to download some data, the program provides the [Build from](#) field. Using this field, the user can build market data from another type of market data. The same function can be used to build market data without additional download, using already existing data as a basis.
- After the above parameters are selected, you should click on the *[image: hydra find]* button.*[image: hydra candles tf]*

Using the context menu, you can configure various parameters of the table of market data values: row grouping, available columns, display format, etc.

[image: hydra export context]

Ticks

Ticks

In the window that appears, select the securities, the required time interval and click the *[image: hydra find]* button:

[image: hydra export trades]

To unload non-system trades, you should check the **Non-system** box.

The received values can be [exported to the required format](#).

Order books

Order books

In the window that appears, select the securities, the required time interval and click the *[image: hydra find]* button:

[image: hydra export depth]

Order books can be created from [Order Log and Level 1 files](#). In addition, you can adjust the depth and update period of the order book.

The received values can be [exported to the required format](#).

Candles

Candles

See [Candle generation](#) section.

Order log

Order log

In the window that appears, select the securities, the required time interval and click the *[image: hydra find]* button:

[image: hydra export order log]

The received values can be [exported to the required format](#).

Level 1

Level 1

In the window that appears, select the securities, the required time interval and click the *[image: hydra find]* button:

[image: hydra export level1]

You can also choose which types of changes should be exported. This is done using the drop-down list shown below:

[image: hydra export security change selected]

The received values can be [exported to the required format](#).

News

News

In the window that appears, select the securities, the required time interval and click the *[image: hydra find]* button:

[image: hydra export news]

The received values can be [exported to the required format](#).

Transactions

Transactions

In the window that appears, select the securities, the required time interval and click the *[image: hydra find]* button:

[image: hydra export executions]

The received values can be [exported to the required format](#).

Option desk

Option desk

In the window that appears, select the desired time range, choose the underlying asset, add options for it, and click the *[image: hydra find]* button:

If there is no history, but there is data on the spread, you can calculate the main Greeks (Delta, Gamma, Vega, Theta, Rho, Volatility (implied)), to do this you need to select them in the **Calculate Greeks** field.

[image: hydra option desk]

To view the **Volatility Smile**, you need to go to the **Volatility Smile** tab.

[image: hydra option smile of volatility]

Using the context menu, you can configure the parameters of your own transactions table: grouping of rows, available columns, display format, etc.

[image: hydra export context]

The found transactions can be exported to the [Excel](#), xml, sql, bin or txt formats. This is done using the drop-down list:

[image: hydra export]

Indicators

Indicators

In the panel that appears, select the instrument, time range of interest, specify the data type from which the indicator will be built, set the [indicator](#) and its parameters, and then click the *[image: hydra find]* button: :

[image: hydra export indicator]

To view the chart of values, just click the *[image: hydra candles]* button.

[image: hydra export indicator view]

The received values can be [exported to the required format](#).

Positions

Positions

In the window that appears, select the securities, the required time interval and click the *[image: hydra find]* button:

[image: hydra export position]

The received values can be [exported to the required format](#).

Candles generation

Candle Generation

Hydra allows you to generate various types of candles based on downloaded trades, which can subsequently be exported to Excel, XML, SQL, BIN, JSON, or TXT formats.

This allows you to use the generated data in any technical analysis programs (WealthLab, AmiBroker, etc.).

Candle Generation Process

1. On the **General** tab, click the **Candles** button, and the following window will open:

[image: hydra candles main]

2. In the opened window, you need to configure the candle generation parameters:
 - Select the desired candle type from the dropdown list (all [standard candle types](#) are supported)
 - Specify the necessary parameters for the selected candle type:
 - For [TimeFrameCandleMessage](#) - select **Timeframe**
 - For [VolumeCandleMessage](#) - specify **Volume**
 - For [TickCandleMessage](#) - specify **Number of ticks**
 - For [RangeCandleMessage](#) - specify **Range**
 - For [RenkoCandleMessage](#) - specify **Block size**
 - For [PnFCandleMessage](#) - specify **P&F Parameters**
 - Select the instrument for which candles will be generated
 - Specify a time range (if necessary)
 - Click the *[image: hydra find]* button to start generation

Example of Timeframe Candle Generation

To generate 5-minute candles for the SBER@TQBR instrument:

1. Select the candle type [TimeFrameCandleMessage](#)
2. Set **Timeframe** = 5 min
3. Select the SBER@TQBR instrument
4. Click the search button

After data generation, you will see the result:

[image: hydra candles tf]

Example of Volume Candle Generation

To generate volume candles:

1. Select the candle type [VolumeCandleMessage](#)
2. Specify the volume (for example, 100)
3. Select the instrument
4. In the **Build from** field, select **Ticks**
5. Click the search button

Generation result:

[image: hydra candles volume]

Data Sources for Building Candles

If market data could not be obtained directly from the source, you can generate candles by selecting in the **Build from** field the type of data from which they will be built:

- **Ticks** - building candles from tick data
- **Order Books** - building candles from order book data
- **Level1** - building candles from Level1 data
- **Smaller Timeframe** - building candles with a larger timeframe from candles with a smaller one

Examples of Different Building Options:

- 10-minute candles from ticks:
[image: hydra candles tf 10]
- 30-minute candles from 5-minute candles:
[image: hydra candles tf 01]

If you select **don't build** in the **Build from** field, only ready-made candles that were downloaded directly through the data source will be searched.

Visualization of Generated Candles

For graphical display of generated candles:

1. Click the *[image: hydra candles]* button
2. A chart with the built candles will open:
[image: hydra candles tf chart]
[image: hydra candles volume chart]

Adding Indicators to the Chart

Technical indicators can be added to the candle chart:

1. Open the context menu by right-clicking on the chart panel
2. Select the **Indicator** item and the desired indicator from the list
3. To display the indicator on a separate panel:
 - Add a new panel using the *[image: hydra add]* button
 - Select the desired indicator from the context menu

Example of a chart with added indicators:

[image: hydra candles ind chart]

Data Export

The obtained candle values can be [exported to various formats](#) for use in other programs.

Also see the [video tutorial](#) on building various types of candles

Export data

Export data

[Hydra](#) allows you to export the received market data in various formats, including [MetaStock data formats](#).

To export data, files in [Excel](#), xml, bin, txt, Json formats or SQL tables are used.

For export, you should select the required file format from the drop-down list:

[image: hydra export]

Then you have to select a folder and change the file name if necessary.

When exporting to text files (txt), a window appears in which you can specify the export template of the form:

;;;;;

Here, in curly braces, the properties to be exported and their order are indicated, separated by semicolons.

By clicking on the **Preview** button, you can see what data will be saved in the file.

[image: hydra export TSLab Meta Stock 1]

The user can add additional properties such as instrument code via the `property` property, or specify a Time Frame value.

You can add a header indicating the name of properties. In this case, the record will look like this.

[image: hydra export TSLab Meta Stock 2]

If you need to export in a format using colons, you should specify the default keyword as in the example above .

Watch [video tutorial](#)

Export into MetaStock

Export into MetaStock

To export data to MetaStock format files, select the Txt format from the drop-down list:

[image: hydra export]

When exporting to text format (Txt) files, a window appears:

[image: hydra export Meta Stock 2]

In this window, specify the export template, where the curly brackets indicate the properties to export and their order:

```
{SecurityId.SecurityCode},5,{OpenTime:yyyyMMdd},{OpenTime:HHmmss},{OpenPrice},{HighPrice},  
{LowPrice},{ClosePrice},{TotalVolume}
```

In the example, the five-minute candle timeframe is specified in the second position.

Also, the first line (Header) should be set in the file:

```
<TICKER>,<PER>,<DATE>,<TIME>,<OPEN>,<HIGH>,<LOW>,<CLOSE>,<VOL>
```

Any market data types

Any market data types

Hydra allows you to use alternative data types to obtain various kinds of market data.

This is necessary if the source does not allow downloading the required market data. So, for example, several types of market data can be used at once to build the **Order book**.

IMPORTANT! **Order book** can be built from the **Order Log** or **Level 1**, provided that these data types contain the best prices.

It is worth remembering that **Level 1** values can be downloaded from any source that provides real-time market data. **Level 1** can also be received by [converting](#) from the **Order book**.

To build it, you need to:

1. Select the period and instrument for which you want to get market data.*[image: hydra LEVEL 1 build depth data]*
2. Select the **Build from** field and select the required data type*[image: hydra type build data]*
IMPORTANT! If **Order Book**, **Order Log**, **Level 1** are selected as a source for building a candle, a selection of additional parameters appears.*[image: hydra ext proper build data]*
3. After setting the parameters, click the *[image: hydra candles]* button.*[image: hydra LEVEL 1 build depth data result]*

To build **Candles**, the option of building candles of a larger Time Frame from candles of a smaller Time Frame is also available.

For example, if there are candles with a Time Frame of 1 minute, you can build candles with a Time Frame of 5 minutes from them by selecting the appropriate type in the **Build from** line.

Watch [video tutorial](#)

Indicators

Indicators

See [List of indicators](#)

See [video](#)

Importing

Importing

[Hydra](#) allows you to import your own data saved in the .csv format. To import, open the **Import** tab, and select the type of exchange data that you want to import.

[image: hydra import menu]

The following types can be imported:

- [Candles](#)
- [Securities](#)
- [Trades](#)
- [Order books](#)
- [Order log](#)
- [Level 1](#)
- [News](#)
- [Own transactions](#)

Watch [video tutorial](#)

Candles

To import candles, select **Import => Candles** item from the main application menu.

[image: hydra import candles]

Candle import process

1. Common.

- **Data type** - type of imported data.
- **Filename** - full path to the CSV file.
- **Data directory** - folder where the final [S#](#) files will be saved.
- **File mask** - file mask that is used when scanning the directory. For example, candle_*.csv.
- **Column separator** - column separator. Tabulation is denoted by TAB.
- **Indent from the beginning** - number of lines from the beginning of file to be skipped (if they contain meta information).
- **Time zone** - time zone.
- **Interval** - frequency of data update.

Instruments

- **Extended information** - save extended imported fields in the extended information storage
- **Duplicates** - whether duplicate securities will be updated if they already exist.

2. Configure import parameters for [S#](#) fields.

- **S# field** - value of the S# field. (**Security**, **Board**, etc.).
- **Associations** - match the column value in file to the stocksharp type (if necessary).
- **Format** - data format. Typically used when importing date and time values (see [Trades](#)).
- **Use** - whether to use the data when importing.
- **Field order** - sequence in which the property columns of the imported item are arranged.

For example, if the imported file has the following template type:

```
{SecurityId.SecurityCode},{SecurityId.BoardCode},{OpenTime:yyyyMMdd},  
{OpenTime:default:HH:mm:ss},{OpenPrice},{HighPrice},{LowPrice},{ClosePrice},{TotalVolume}
```

Then the following setting will correspond to it:[image: hydra import prop candles]

Here:

The **Security** value corresponds to the with the serial number **0**.

[!TIP] In programming, the ordinal number of the first element is always 0

The **Board** value corresponds to the with serial number **1**. And so on.

- **By default** - default field value. For example, it can be used for repeated field values (**Security**, **or Board** when importing trades, order books, etc., see [Trades](#)), if the corresponding information is not in the data file.
- **Zero** - in some cases when saving data, some data properties may be saved as "0", which is an error. For example, the price value, for various reasons, can be equal to 0, which is not acceptable, and in the future will lead to an incorrect reading. This can lead to incorrect strategy operation that work with them and, as a consequence, an erroneous result. By checking the box, the user specifies that data in this section, if equal to 0, is written as empty, that is, it is absent. During further work, for example, testing, the user will see an error of no data, which will indicate an incorrect data import.

In fact, this is the protection of user from "broken" data, for more correct work.

The user can configure a large number of properties for the downloaded data. Based on the imported file template, you need to specify the property and assign it the required number in the sequence.

3. To preview the data, click the **Preview** button.*[image: hydra import preview candles]*
4. Click the **Import** button.

Instruments

To import instruments, select the **Import => Instruments** tab.

[image: hydra import securities]

Import process.

1. Import settings..

See [Candles](#) import.

2. Configure import parameters for **S#** fields.

See [Candles](#) import.

Let's consider an example of importing an security from a CSV file:

- The file from which you want to import data has the following template:

```
{SecurityId.SecurityCode};{SecurityId.BoardCode};{PriceStep};{SecurityType};{VolumeStep}
```

Here the values of and correspond to the values of **Security** and **Board**, respectively. Therefore, in the **Field order** field we assign the values 0 and 1, respectively.

- For the field, select the **Nominal** field from the **S# field** window and assign it the value 2.
- For the field, select the **Type** field from the **S# field** window - the security type (stock, currency, futures, etc.). We assign it the value 3.
- For the field, select the **Min volume (base)** field from the **S# field** window - the base or minimum security volume. We assign it the value 4
- The field setting window will look like this:*[image: hydra import prop securitiy]*

The user can configure a large number of properties for the downloaded data. Based on the imported file template, you need to specify the property and assign it the required number in the sequence.

3. To preview the data, click the **Preview** button.*[image: hydra import preview securitiy]*

4. Click the **Import** button.

Ticks

To import trades, select the **Import => Ticks** tab.

[image: hydra import trades]

Import process.

1. Import settings..

See [Candles](#) import.

2. Configure import parameters for **S#** fields.

See [Candles](#) import.

Let's consider an example of importing trades (ticks) from a CSV file:

- The file from which you want to import data has the following template:

```
{SecurityId.SecurityCode};{SecurityId.BoardCode};{ServerTime:default:yyyyMMdd};  
{ServerTime:default:HH:mm:ss.ffffff};{TradeId};{TradePrice};{TradeVolume};{OriginSide}
```

Here the values of and correspond to the values of **Security** and **Board**, respectively. Therefore, in the **Field order** field we assign the values 0 and 1, respectively.

- For the and fields, select the **Date** and **Time** fields from the **S# field** window, respectively. We assign the values 2 and 3.
- For the field, select the **Identifier** field from the **S# field** window - the trade identifier or the trade number. We assign it the value 4.
- For the field, select the **Price** field - the trade price from the **S# field** window. We assign it the value 5.
- For the field, select the **Volume** field from the **S# field** window - the trade volume. We assign it the value 6.
- For the field, select the **Initiator** field from the **S# field** window - the trade initiator (Seller or Buyer). We assign it the value 7.
- The field setting window will look like this:*[image: hydra import prop trade]*

The user can configure a large number of properties for the downloaded data. Based on the imported file template, you need to specify the property and assign it the required number in the sequence.

3. To preview the data, click the **Preview** button.*[image: hydra import preview trade]*

4. Click the **Import** button.

Order books

Order books

To import order books, select **Import => Order books** item from the main application menu.

[image: hydra import depths]

Import process.

1. Import settings..

See [Candles](#) import.

2. Configure import parameters for **S#** fields.

See [Candles](#) import.

Let's consider an example of importing an order book from a CSV file:

- The file from which you want to import data has the following template:

```
{SecurityId.SecurityCode};{SecurityId.BoardCode};{ServerTime:default:yyyyMMdd};  
{ServerTime:default:HH:mm:ss.ffffff};{Quote.Price};{Quote.Volume};{Side}
```

Here the values of and correspond to the values of **Security** and **Board**, respectively. Therefore, in the **Field order** field we assign the values 0 and 1, respectively.

- For the and fields, select the **Date** and **Time** fields from the **S# field** window, respectively. We assign the values 2 and 3.
- For the field, select the **Price** field from the **S# field** window - quote price. We assign it the value 4.
- For the field, select the **Volume** field from the **S# field** window - quote volume. We assign it the value 5
- For the field, select the **Direction** field from the **S# field** window - the trade direction (Buy or Sell). We assign it the value 6.
- The field setting window will look like this:[image: hydra import prop depth]

The user can configure a large number of properties for the downloaded data. Based on the imported file template, you need to specify the property and assign it the required number in the sequence.

3. To preview the data, click the **Preview** button.[image: hydra import preview depth]

4. Click the **Import** button.

Order log

Order log

To import the order log, select the **Import => Order log** item from the main application menu.

[image: hydra import orderlog]

Import process.

1. Import settings..

See [Candles](#) import.

2. Configure import parameters for **S#** fields.

See [Candles](#) import.

Let's consider an example of importing an Order Log from a CSV file:

- The file from which you want to import data has the following template:

```
{SecurityId.SecurityCode};{SecurityId.BoardCode};{ServerTime:default:yyyyMMdd};  
{ServerTime:default:HH:mm:ss.ffffff};{OrderId};{OrderPrice};{OrderVolume};{Side};  
{OrderState};{TimeInForce};{TradeId};{TradePrice}
```

Here the values of and correspond to the values of **Security** and **Board**, respectively. Therefore, in the **Field order** field we assign the values 0 and 1, respectively.

- For the and fields, select the **Date** and **Time** fields from the **S# field** window, respectively. We assign the values 2 and 3.
- For the field, select the **ID** field from the **S# field** window - order ID. We assign it the value 4.
- For the field, select the **Price** field from the **S# field** window - order price. We assign it the value 5
- For the field, select the **Volume** field from the **S# field** window - order volume. We assign it the value 6.
- For the field, select the **Direction** field from the **S# field** window - order direction (buy or sell). We assign it the value 7.
- For the field, select the **Action** field from the **S# field** window - the order state (active, inactive or error). We assign it the value 8.
- For the field select the **Time** in force from the **S# field** window - a condition of limit order execution. We assign it the value 9.
- For the field, select the **ID (trade)** field from the **S# field** window - the trade identifier. We assign it the value 10.
- For the field, select the **Price (trade)** field from the **S# field** window - the trade price. We assign it the value 11.
- The field setting window will look like this:[image: hydra import prop orderlog]

The user can configure a large number of properties for the downloaded data. Based on the imported file template, you need to specify the property and assign it the required number in the sequence.

3. To preview the data, click the **Preview** button.[image: hydra import preview orderlog]

4. Click the **Import** button.

Level 1

Level 1

To import Level 1 data, select **Import => Level 1** from the main application menu

[image: hydra import level1]

Import process.

1. **Import settings..**

See [Candles](#) import.

2. Configure import parameters for **S#** fields.

See [Candles](#) import.

Let's consider an example of importing an Level 1 from a CSV file:

- The file from which you want to import data has the following template:

```
{SecurityId.SecurityCode};{SecurityId.BoardCode};{ServerTime:default:yyyyMMdd};  
{ServerTime:default:HH:mm:ss.ffffff};{Changes:{BestBidPrice};{BestBidVolume};{BestAskPrice};  
{BestAskVolume};{LastTradeTime};{LastTradePrice};{LastTradeVolume}}
```

Here the values of and correspond to the values of **Security** and **Board**, respectively. Therefore, in the **Field order** field we assign the values 0 and 1, respectively.

- For the and fields, select the **Date** and **Time** fields from the **S# field** window, respectively. We assign the values 2 and 3.
- For the field, select the **Best buy price** field from the **S# field** window. We assign it the value 4.
- For the field, select the **Best buy volume** field from the **S# field** window. We assign it the value 5.
- For the field, select the **Best sale price** field from the **S# field** window. We assign it the value 6.
- For the field, select the **Best sale volume** field from the **S# field** window. We assign it the value 7.
- For the field, select the **Last trade time** field the **S# field** window. We assign it the value 8.
- For the field, select the **Last trade price** field from the **S# field** window. We assign it the value 9.
- For the field, select the **Last trade volume** field from the **S# field** window. We assign it the value 10.
- The field setting window will look like this:[image: hydra import prop level 1]

The user can configure a large number of properties for the downloaded data. Based on the imported file template, you need to specify the property and assign it the required number in the sequence.

3. To preview the data, click the **Preview** button.[image: hydra import preview level 1]

4. Click the **Import** button.

News

News

To import news, select **Import => News** from the main application menu.

[image: hydra import news]

News import process

1. Execute **CSV Import Settings**.
See [Candles](#) import.
2. Configure import parameters for [S#](#) fields.
See [Candles](#) import.
3. To preview the data, click the **Preview** button
4. Click the **Import** button.

Transactions

Transactions

To import your transactions, you need to select **Import => Own Transactions** from the main application menu.

[image: hydra import executions]

The process of importing your transactions

1. Execute **CSV Import Settings**.
See [Candles](#) import.
2. Configure import parameters for [S#](#) fields.
See [Candles](#) import.
3. To preview the data, click the **Preview** button
4. Click the **Import** button.

Settings

Settings

[Hydra](#) can be used in server mode, in this mode you can remotely connect to [Hydra](#) and get the existing data in the storage. You can connect to [Hydra](#) running in server mode from [Designer](#) (see [Getting started](#) in the [Designer](#) documentation for how to do this). You can also connect to [Hydra](#) via the [API](#) (see the [FIX/FAST connectivity](#) section for details).

In server mode, the [Hydra](#) program allows the user to work using one connection, with several programs at once. By setting the access key in the program settings, the user can simultaneously work with one source under one account.

Actually, the connection to the source occurs via [Hydra](#), to which, for example, [Designer](#), [Terminal](#) are simultaneously connected. This method allows you to avoid reconnection between programs, purchase of an additional connection. With such work, conflicts are excluded, the occurrence of which is due to the orders registering or trades from different programs. [Hydra](#) receives the signal and passes the result back to the program from which it was received, while the sequence of other work is not disturbed.

To enable [Hydra](#) server mode, select the **Server mode** tab in the top menu of the program.

[image: hydra server menu]

After that, click on the **Settings** button to open the server mode settings window.

[image: hydra server]

Hydra Server

- **FIX server** - switch [Hydra](#) to server mode, distributing live trading and historical data via the FIX protocol.

In this section, you configure the connection for working with sources:

1. **ConvertToLatin** - convert Cyrillic to Latin
2. **QuotesInterval** - quotes update period
3. **TransactionSession** - setting of a trading session. Setting up for trading via the [Hydra](#) program.

This setting allows you to configure Dialect of the FIX protocol, Sender and Recipient, Data format and other settings. See [FIXServer properties](#) for details.

4. **MarketDataSession** - settings for transfer of market data received using [Hydra](#). See [FIXServer properties](#) for details.
 5. **KeepSubscriptionsOnDisconnect** - keeping subscriptions when disconnected from the source.
 6. **DeadSessionCleanupInterval** - after what time interval the information will be cleared if the connection is disconnected.
- **Authorization** - authorization to gain access to the Hydra server
 - **Number of securities** - the maximum number of instruments that can be requested from the server
 - **Candles (days)** - the maximum number of days available to download candle history
 - **Ticks (days)** - the maximum number of days available to download the tick data history
 - **Order books (days)** - the maximum number of days available to download order book history
 - **OL (days)** - the maximum number of days available to download the OL data history
 - **Transactions (days)** - the maximum number of days available to download the transaction history
 - **Simulator** - turn on simulator mode

- **Security mapping** - enable the transfer mode of only specified instruments.

If you set **Authorization** other than **Anonymous**, the **Users** button will appear on the **Common** tab. After clicking it, the **Users** window appears.

[image: hydra users]

On the left side of the window, you can add a new user, and on the right, set access rights for them.

Emulation setup

Emulation setup

In server mode, the program allows you to enable emulation mode.

[image: hydra emulator start]

In emulation mode, the [Hydra](#) program allows you to perform the following functions:

- The program allows you to configure the keys for connecting to the source and simultaneously work with one connection in different programs ([Designer](#), [Terminal](#)).
- If the market data source allows you to download historical data, then they can be used simultaneously for testing.
- If the source allows you to receive data in real time, then the emulation mode allows you to emulate the trading mode. In this mode, data on user actions (orders registering, trades) are transferred directly to Hydra, while actions are recorded separately for each program. For example, when registering an order in Terminal, changes to it will be visible only for it, and in Designer they will not be recorded. This avoids conflicts between two programs running over the same connection.
- **IMPORTANT!** Trades conducted in emulation mode, trading and operations on them are emulated in real time, when the mode is off, actions will be performed on real trading.

This mode is used when [testing strategies](#).

Emulation settings.

[image: hydra emulator prop]

- **Match on touch** - when emulating trade matching, match orders when the trade price is equal to the order price.
- **Order book (time in force)** - the maximum period of order book in the emulator. If the order book has not been updated within the specified period, its value is erased. It is used to remove old order book data if there are gaps in the data.
- **Percentage of errors** - the percentage of errors in registering new orders (from 0 to 100).
- **Latency** - the minimum latency of registered orders.
- **Re-registration** - whether re-registration of orders will be supported as a single transaction.
- **Buffering period** - a parameter that is responsible for the period of sending whole packets in order to emulate network latency and buffer the work of the exchange core.
- **Order ID** - the number with which the emulator will generate identifiers for orders.
- **Trade identifier** - the number with which the emulator will generate identifiers for trades.
- **Transaction** - the number with which the emulator will generate identifiers for order transactions.
- **Spread size** - the spread size in price steps. It is used to determine the spread when generating the order book from tick trades.
- **Order book depth** - maximum order book depth generated by ticks
- **Number of volume steps** - the number of volume steps by which the order is larger than the tick trade. It is used in testing on tick trades.
- **Portfolio interval** - interval for recalculating data on portfolios. If the interval is 0, then no recalculation

is performed.

- **Adjust time** - adjust time for orders and trades to the exchange time.
- **Time zone** - information about the time zone where the exchange is located
- **Price shift** - a price shift from the last trade, which determines the boundaries of the maximum and minimum prices for the next session
- **Add additional volume** - add additional volume to the order book when registering orders with a large volume.
- **Trading session state** - checking the trading state.
- **Money** - check the money balance
- **Short** - the ability to open short positions.
- **Storage** - storage.

Instruments mapping

Instruments mapping

See the [Matching instruments-connections](#) section.

FIX/FAST connectivity

Connecting via FIX Protocol

[Hydra](#) can be used in server mode, which allows remote connection to [Hydra](#) to access data in the storage. Enabling [Hydra](#) server mode is described in the [Settings](#) section.

To connect via the [FIX protocol](#), you need to create and configure a Fix connection ([FIX Adapter Initialization](#)).

```
// Create a connector instance
private readonly Connector _connector = new Connector();

// Configure the adapter for market data via FIX protocol
var marketDataAdapter = new FixMessageAdapter(_connector.TransactionIdGenerator)
{
    Address = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 5002),
    SenderCompId = "hydra_user",
    TargetCompId = "StockSharpHydraMD",
    Login = "hydra_user",
    Password = "qwerty".To<SecureString>(),
};
_connector.Adapter.InnerAdapters.Add(marketDataAdapter);

// Configure the adapter for transaction data
var transactionDataAdapter = new FixMessageAdapter(_connector.TransactionIdGenerator)
{
    Address = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 5002),
    SenderCompId = "hydra_user",
    TargetCompId = "StockSharpHydraMD",
    Login = "hydra_user",
    Password = "qwerty".To<SecureString>(),
};
_connector.Adapter.InnerAdapters.Add(transactionDataAdapter);
```

Subscribe to events and set up data handlers:

```
// Successful connection event
_connector.Connected += () =>
{
    Console.WriteLine("Connection established");

    // Create a subscription to search for instruments
    var lookupSubscription = new Subscription(DataType.Securities);
    _connector.Subscribe(lookupSubscription);
};

// Connection lost event
_connector.Disconnected += () =>
{
    Console.WriteLine("Connection lost");
};

// Instrument received event
_connector.SecurityReceived += (subscription, security) =>
{
    Console.WriteLine($"Instrument received: {security.Code}, {security.Id}");
    BufferSecurity.Add(security);

    // If this is the target instrument, subscribe to its data
    if (security.Id == targetSecurityId)
    {
        // Order book subscription
        var depthSubscription = new Subscription(DataType.MarketDepth, security);
```

```

_connector.Subscribe(depthSubscription);

// Tick trades subscription
var tradesSubscription = new Subscription(DataType.Ticks, security);
_connector.Subscribe(tradesSubscription);

// Candles subscription
var candleSubscription = new Subscription(
    DataType.TimeFrame(TimeSpan.FromMinutes(5)),
    security)
{
    MarketData =
    {
        From = DateTime.Today.Subtract(TimeSpan.FromDays(30)),
        To = DateTime.Now
    }
};
_connector.Subscribe(candleSubscription);
};

// Tick trade received event
_connector.TickTradeReceived += (subscription, trade) =>
{
    Console.WriteLine($"Trade received: {trade.Security.Code}, {trade.Time}, {trade.Price},
{trade.Volume}");
};

// Order book changed event
_connector.OrderBookReceived += (subscription, depth) =>
{
    Console.WriteLine($"Order book received: {depth.SecurityId}, Best bid: {depth.BestBid()?.Price},
Best ask: {depth.BestAsk()?.Price}");
};

// Candle received event
_connector.CandleReceived += (subscription, candle) =>
{
    Console.WriteLine($"Candle received: {candle.SecurityId}, {candle.OpenTime}, O:{candle.OpenPrice},
H:{candle.HighPrice}, L:{candle.LowPrice}, C:{candle.ClosePrice}");
};

// Connection error event
_connector.ConnectionError += error =>
{
    Console.WriteLine($"Connection error: {error.Message}");
};

// General error event
_connector.Error += error =>
{
    Console.WriteLine($"Error: {error.Message}");
};

// Market data subscription error event
_connector.SubscriptionFailed += (subscription, error) =>
{
    Console.WriteLine($"Subscription error {subscription.DataType} for {subscription.SecurityId}:
{error}");
};

// Connect to the server
_connector.Connect();

```

Using Hydra Services

Hydra in server mode provides access to various types of data. Let's look at examples of getting historical data:

```

// Getting historical candles
private void RequestHistoricalCandles(Security security, DateTime from, DateTime to)
{
    // Create a subscription to historical candles
    var candleSubscription = new Subscription(
        DataType.TimeFrame(TimeSpan.FromMinutes(5)),
        security)
    {
        MarketData =
        {
            From = from,
            To = to
        }
    };

    // Subscribe to process received candles
    _connector.CandleReceived += OnCandleReceived;

    // Start the subscription
    _connector.Subscribe(candleSubscription);
}

private void OnCandleReceived(Subscription subscription, ICandleMessage candle)
{
    // Check that the candle belongs to our subscription
    if (subscription.DataType != DataType.TimeFrame(TimeSpan.FromMinutes(5)))
        return;

    Console.WriteLine($"Historical candle: {candle.OpenTime}, O: {candle.OpenPrice}, H:
{candle.HighPrice}, L: {candle.LowPrice}, C: {candle.ClosePrice}, V: {candle.TotalVolume}");

    // Process the received candles, for example, save to local storage
    // or use for analysis/visualization
}

```

Disconnecting from Hydra Server

```

// Proper connection closing
private void DisconnectFromServer()
{
    // Unsubscribe from all subscriptions
    foreach (var subscription in _connector.Subscriptions.ToArray())
    {
        _connector.UnSubscribe(subscription);
    }

    // Disconnect from the server
    _connector.Disconnect();
}

```

Hydra-client connectivity

Connecting a Hydra Client

In server mode, it's possible to connect another Hydra program, which will act as a client and download data to itself. Unlike [Connecting via FIX protocol](#), data is transmitted in the form of files in StockSharp format. This makes the source suitable for transferring a large volume of historical data.

A special source is used for connection:

[image: hydra tasks server]

Settings

[image: hydra tasks server]

- **Address** - the address of the Hydra server.
- **Login** - login (required if the server requires authorization).
- **Password** - password (required if the server requires authorization).
- **Time Offset** - A time offset in days from the current date, necessary to prevent downloading incomplete data for the current trading session.
- **Weekends** - whether to download data for weekends.

Main

- **Title** - the title of the task.
- **Working Hours** - setting the operation of the platform.
- **Interval of Operation** - operation interval.
- **Data Directory** - the directory with data where the final files in [S#](#) format will be saved.
- **Format** - the format of data: BIN/CSV.
- **Max. Errors** - the maximum number of errors, upon reaching which the task will be stopped. By default, 0 - the number of errors is ignored.
- **Dependency** - a task that must be completed before starting the current one.

Logging

- **Identifier** - identifier.
- **Logging Level** - the level of logging.

Instruments list

Instruments list

If you click the **Instruments** button on the General tab, the **Instruments** panel appears. The **Instruments** panel displays all available instruments, as well as the market data for them.

[image: hydra securitiesPanel 00]

At the bottom of the **Instruments** panel are buttons that allow you to:

- Add a new instrument, as described in the clause [Create instrument](#).
- Add a new index, as described in the clause [Index](#).
- Add a new continuous instrument, as described in the clause [Continuous futures](#).
- Change the instrument or group of instruments, as described in the clause [Editing instruments](#).
- Specify additional information about the instrument, as described in the clause [Extended instrument info](#).
- Match the instruments and connections, as described in clause [Matching instruments-connections](#).

Create instrument

Create instrument

See

[Create instrument](#)

Index

With [Hydra](#), you can create your own index.

On the **Common** tab select **Securities** so that the **All Securities** tab appears.

Before creating the **Index**, check which market data is available. Select the path where the data is stored and sequentially view the instruments that should participate in the calculation of the index. If there are gaps, download the necessary market data (for example, from Finam).

[image: HydraGluingCheckData]

As an example, we will consider the instrument ratio index AAPL@NYSE/GOOG@NYSE.

The first step is to create the **Index**. Click the **Create security=>Index** button on the **All Securities** tab, after which the following window will appear:

[image: hydra index sec]

To create the **Index** instrument, you must specify a name and add the mathematical formula of a combination of several instruments. Together with the standard mathematical operators, you can use the following functions:

- **abs(a)** - Returns the absolute value of a number.
- **acos(a)** - Returns the angle whose cosine is equal to the specified number.
- **asin(a)** - Returns the angle whose sine is equal to the specified number.
- **atan(a)** - Returns the angle whose tangent is equal to the specified number.
- **ceiling(a)** - Returns the smallest integer that is greater than or equal to a specified number.
- **cos(a)** - Returns the cosine of the specified angle.
- **exp(a)** - Returns the value of e raised to the specified degree.
- **floor(a)** - Returns the largest integer that is less than or equal to the specified number.
- **log(a)** - Returns the natural logarithm (with base e) of the specified number.
- **log10(a)** - Returns the logarithm with base 10 of the specified number.
- **max(a, b)** - Returns the larger of two decimal numbers.
- **min(a, b)** - Returns the smaller of two decimal numbers.
- **pow(a, b)** - Returns the specified number raised to the specified power.
- **sign(a)** - Returns an integer indicating the sign of the specified number.
- **sin(a)** - Returns the sine of the specified angle.
- **sqrt(a)** - Returns the square root of the specified number.
- **tan(a)** - Returns the tangent of the specified angle.
- **truncate(a)** - Calculates the integer part of the specified number.

Next, click the [Candles](#) button on the **Common** tab, select the created **Index** instrument and the data period, set **Composite Element** in the **Create From:** field, and then press the *[image: hydra find]* button.

[image: hydra index candle]

The generated data can be exported to Excel, XML or TXT formats. Export is done using the drop-down list.

[image: hydra export]

Continuous futures

The [Hydra](#) program allows the user to combine different types of market data from various contracts into a single continuous instrument.

To do this, on the **Common** tab select **Securities** so that the **All securities** tab appears. Before gluing the data, check which market data is available. Select the path where the data is located and sequentially view the instruments that you plan to glue. If there are gaps, download the missing market data (for example, from **Финам**).

[image: HydraGluingCheckData]

As an example, consider combining RTS index futures.

1. To create a continuous futures contract, click the **Create security => Continuous security** button on the **All securities** tab.*[image: Hydra Gluing Check Data 00]*

After that, the following window will appear:*[image: HydraGluingWindow]*

2. To create a continuous future, you need to specify a name and add contracts.

There are two ways to add contracts.

- Manually by clicking the *[image: hydra add]* button.*[image: HydraGluingCSCustom]*
- If you set the first two letters of the contract as a name, for example, RI, and click the **Auto** button, then all the instruments found in the database will be added.*[image: HydraGluingCSAuto]*

3. Select the required contracts and set their transition dates. *[image: Hydra GluingCSAuto 00]*

4. Next, assign the instrument identifier **RI_long9@FORTS** and click the **OK** button, after which a new instrument will be created.

5. Next, click the **Candles** button on the **Common** tab, select the resulting instrument and data period, set the **Composite element** value in the **Build from** field, and then click the *[image: hydra find]* button.

[image: HydraGluingTrades]

The generated data can be exported to Excel, XML, JSON or TXT formats. Export is performed using the drop-down list.

[image: hydra export]

Editing instruments

Editing instruments

To edit an instrument (for example, if it was created without all required data), double-click the instrument or click the *[image: hydra edit]* button to open the window where you can make the necessary changes:

Go to the **Securities** window.

[image: hydra security edit 00]

The editing window will open.

[image: hydra security edit]

If necessary, you can edit the instrument group. Select an instrument group and click the *[image: hydra edit]* button.

[image: hydra securities edit 00]

Then edit the group as needed.

[image: hydra securities edit]

If all selected instruments have the same value in a field, that value will be displayed. If the values differ, the field will be empty. For example, if two instruments have a Price Step of 1 but one has a lot size of 10 and the other 100, the Price Step field will show 1, and the Volume Step field will be empty.

Extended instrument info

Extended instrument info

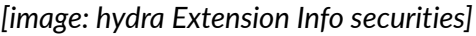
Sources of extended information are **CSV** files located in the `c:\Users\Users\Documents\StockSharp\Hydra\Extended info\` folder. They are automatically loaded when **Hydra** starts.

Extended information can include any additional details about an instrument (for example, country, city, board, etc.).

Each source of extended information (CSV file) contains a list of instruments and the available properties. For each source, the extended information is unique.

If the source does not contain extended information for an instrument, the corresponding columns in the instruments list will be empty.

To select the required extended information, you need:

1. In the **Securities** tab, click the **Extended information** button
2. A window will appear in which you need to select the path to the required CSV file

Below is an example of a **CSV** file of extended information opened in different editors: **MS Excel** and **Notepad**.





Matching instruments-connections

The same instrument in different trading systems can be called differently. It is possible to match the instrument and the connections via which this instrument will be traded, and specify how it is identified in the external trading system.

This allows you to organize the received data and simplify storage. Actually, all incoming data from various sources will be consolidated in one place and not by the source name, but by the instrument name.

This is also useful when trading the same instrument on different trading boards or via different connections (or brokers). Also, it allows getting data from one connection and making trades via another connection.

To match instruments and connections you need:

1. Go to the **Securities** tab and click the **Securities and Connections** button.*[image: Designer Security mapping 01 00]*
2. In the list of connections, select the required connection.*[image: Designer Security mapping 01 01]*
3. Fill in all the columns.
For example:
APPLE stock instrument.
 - Connection - **Interactive Brokers**. Click on the *[image: Designer Creation tool 00]* button, after which a new line will be added.
 - In the **Security** code and **Board code** columns, specify the instrument code and board code. In the columns **Security code in adapter** and **Board code in adapter** specify the instrument code and the board code as they are specified in the external trading system. Click **OK** *[image: Designer Security mapping 01 01]*
 - We repeat the steps for the **Transaq** and **CQG Continuum** connections in the same way.

Transaq	CQG Continuum
<i>[image: Designer Security mapping 01 02]</i>	<i>[image: Designer Security mapping 01 03]</i>

4. Now all downloaded data, in our case for APPLE shares, will be saved in one place.

Boards

Boards

On the **Common** tab, click the **Boards** button.

Here you can view basic information about trading boards and set their time zone and working schedule.

[image: hydra boards]

Tasks

Tasks

The tool is a repeating action at a specific time.

To view the list of tools, select the **Tools** item in the left pane. Work with tools is the same as with sources - the tool can be added, deleted, edited, enabled/disabled. To add the **Tool**, on the **Common** tab, select **Add** => **Tools**. A corresponding window will appear in which you can select the required **Tool**...

[image: hydra tasks]

Currently, there are four classes of tasks in [Hydra](#)

- [Import \(auto\)](#) - the task is to import data from CSV format.
- [Converter](#) - the task is to convert the exchange data to the ticks, candles or order books.
- [Export \(auto\)](#) - the task is to export candles to different formats.
- [Backup](#) - the task is to backup data to the cloud service.

Converter

The task converts exchange data. For example, from the Order Logs to ticks or from ticks to candles, etc.

[image: hydra tasks converter]

Converter

- **Converter** - converter.
- **From** - what data type will be converted.
- **Data format** - converted data format.
- **Start date** - from what date to start data conversion.
- **Time offset** - the time offset in days from the date the task was started. It is necessary to avoid converting data not for a full day. If real-time data conversion is configured, there is a possibility that the data will not be converted over a full day due to the update interval settings. To avoid this, a time offset is set.
- **Where** - the data directory where the converted data will be saved.

Order books

- **Interval** - order book generation interval.
- **Depth** - maximum depth of order book generation.
- **Order log** - how to build order books from the order log.
Each exchange has its own **Order Log** format, the [Hydra](#) program supports three formats:
 - **By default** - is used in most cases.
 - **ITCH** - is used for the ITCH protocol (exchanges: LSE and Nasdaq).
 - **Plaza** - used for the Moscow Exchange.

[image: hydra choose ITCH Plaza 00]

General

- **Header** - Converter.
- **Working hours** - setting up the board work schedule. *[image: hydra tasks backup desk]*
- **Interval of operation** - the interval of operation.
- **Data directory** - data directory, from where the data for conversion will be received.
- **Format** - the converted data format: BIN/CSV.
- **Max. errors** - the maximum number of errors, upon which the task will be stopped. By default, 0 - the number of errors is ignored.
- **Dependency** - a task that must be performed before running the current one.

Logging

- **Identifier** - the identifier.
- **Logging level** - the logging level.

Let's consider an example of data conversion.

1. Go to the **Converter** task. *[image: hydra tasks converter 00]*
2. Select the instrument and in the window that appears, set the data type that we should receive during conversion, as well as the data type from which we should convert. For example, you need to convert Ticks into candles with a Time Frame of 15 minutes.

[!TIP] IMPORTANT! The period of data to be received should correspond to the period of data available for conversion, otherwise the data will not be converted. In the settings, it is important to specify the correct format of data to be converted, it is necessary that the format corresponds to the format of data to be converted.
3. Specify the required directories. Time offset. Operation interval.
4. We start the conversion.*[image: hydra tasks converter 01]*

It can be seen that the data has been converted. [Let's review](#) the resulting data.

[image: hydra tasks converter 02]

This function is similar to [getting the required market data](#) from another data type.

Watch [video tutorial](#)

Export (auto)

Export (auto)

The task exports exchange data into various formats: Excel, xml, sql, bin, Json or txt.

[image: hydra tasks export]

Database

- **Connection** - connection to the database. Used in case of export via SQL.
- **Packet** - the size of transmitted data packet. By default, the size is 50 elements. Used in case of export via SQL.
- **Uniqueness** - check the data uniqueness in database. Affects performance. Enabled by default. Used in case of export via SQL.

| When using export via SQL, you need to set the connection string parameters

New connection string

[image: hydra tasks connstring]

- **Provider** - provider settings.
- **Server** - server address or path to database.
- **Database** - database name. Not used for SQLite.
- **Login** - login to access the database. Not used for anonymous access.
- **Password** - password for accessing the database. Not used for anonymous access.
- **Windows** - use the current Windows account to connect to the database.
- **Connection** - ready-made connection string.

| You can check the connection to database using the **Check** button.

General

- **Header** - Converter.
- **Working hours** - setting up the board work schedule. *[image: hydra tasks backup desk]*
- **Interval of operation** - the interval of operation.
- **Data directory** - data directory, from where the data for conversion will be received.
- **Format** - the converted data format: BIN/CSV.
- **Max. errors** - the maximum number of errors, upon which the task will be stopped. By default, 0 - the number of errors is ignored.
- **Dependency** - a task that must be performed before running the current one.

CSV

- **Templates** - Templates for each type of exported data.
- **Header** - the header in the first line. If an empty string is passed, the header will not be added to the file.
- **Name format** - the format for recording the exported file name.

Export (auto)

- **Type** - export type (format).
- **Start date** - from what date to start exporting data.
- **Time offset** - time offset in days.
- **Export directory** - directory where data will be exported.
- **Format** - data format.
- **Split** - split type.

Logging

- **Identifier** - the identifier.
- **Logging level** - the logging level.

Let's consider an example of automatic export:

1. Select security.
2. Set up the market data that needs to be exported.*[image: hydra tasks export 00]*
3. Set the export period. If the download of market data in real time is configured, then you can omit the end date of period. In this case, the data will be exported in real time, according to the work interval (data update). *[image: hydra tasks export 01]*
4. Setting up directories. Operation interval. Data type. Data format.
5. We start exporting.*[image: hydra tasks export 02]*

Let's view the exported data

[image: hydra tasks export 03]

Watch [video tutorial](#)

Import (auto)

Import (auto)

The task performs automatic import of exchange data from files in the specified directory according to the specified file mask.

For each selected market data type, the template is configured in the [Importing](#) tab.

[image: hydra tasks import]

At the bottom of the panel, you can select the instruments by which data will be imported, as well as the data type to import.

For each instrument, you can specify the following data import properties:

[image: hydra tasks proper import]

Import (auto)

Settings

- **Data type** - type of imported data.
- **Filename** - full path to the file.
- **Data directory** - data directory.
- **File mask** - file mask that is used when scanning the directory. For example, candles*.csv.
- **Subdirectories** - include subdirectories.
- **Column separator** - column separator. Tabulation is denoted by TAB.
- **Indent from the beginning** - the number of lines to skip from the beginning of file (if they contain meta information).
- **Time zone** - time zone.
- **Interval** - the frequency of data updates.
- **Extended information** - save extended imported fields in the extended information storage.
- **Duplicates** - whether duplicate instruments need to be updated if they already exist.
- **Ignore without ID** - ignore instruments without an identifier.

General

- **Header** - Converter.
- **Working hours** - setting up the board work schedule. *[image: hydra tasks backup desk]*
- **Interval of operation** - the interval of operation.
- **Data directory** - data directory, from where the data for conversion will be received.
- **Format** - the converted data format: BIN/CSV.
- **Max. errors** - the maximum number of errors, upon which the task will be stopped. By default, 0 - the number of errors is ignored.
- **Dependency** - a task that must be performed before running the current one.

Logging

- **Identifier** - the identifier.
- **Logging level** - the logging level.

Backup

Backup

See [Data backup](#)

Analytics

The **Analytics** feature provides users with a tool for market data analysis. This functionality enables in-depth analysis of price data, volumes and other market indicators through pluggable [scripts](#) in C#, F# and Python languages.

Key Features

- **Historical Data Analysis:** The Analytics function allows users to load, view and analyze historical trading data across different time periods.
- **Data Visualization:** Hydra offers various data visualization tools including charts and diagrams that help analyze trends and identify patterns.
- **Financial Indicator Calculations:** The program provides calculation of a wide range of financial indicators such as moving averages, RSI, MACD and others, enabling comprehensive technical analysis.

Watch the [video tutorial](#)

Running a Script

Running a Script

To create a new analytics script, select the **Analytics** tab in the data sources panel on the main window, and choose the desired template from the drop-down menu for a quick start:

[image: hydra_analytics_main_00]

The screenshot presents the main interface of this feature, consisting of several key components:

- **Navigation Tree:** Provides quick access to various analytics functions and created analytics scripts, such as intraday volume analysis, volume profile, charts, indicators, and other analysis tools.
- **Code Window:** The code window displays the source code of the selected analytical script. Users can directly edit the code to customize or create analytical calculations and strategies.
- **Parameter Panel:** On the right side, there is a parameter panel where you can set parameters for analytical scripts, including the selection of instruments, analysis period, data path, and other settings.
- **Error List:** At the bottom of the interface, there is a list of errors detected during the compilation or execution of the analytical script.

The analytics script is formatted as a class inherited from [IAalyticsScript](#).

When setting parameters:

- **Instrument** can be set from one to several instruments depending on the script's logic.

[image: hydra_analytics_main_01]

- The date range.
- The storage from which to retrieve data.
- The script's working time frame, if it uses one.

Clicking the **Start** button [image: hydra_analytics_compile] will open a new tab displaying the results of the script's execution.

Create own script

Creating a Script

Analytics allows creating your own scripts. As an example, let's review the **ChartDrawScript**, which demonstrates the capabilities of chart drawing:

```
namespace StockSharp.Algo.Analytics
{
    /// <summary>
    /// The analytic script, shows chart drawing possibilities.
    /// </summary>
    public class ChartDrawScript : IAnalyticsScript
    {
        Task IAnalyticsScript.Run(ILogReceiver logs, IAnalyticsPanel panel, SecurityId[] securities,
            DateTime from, DateTime to, IStorageRegistry storage, IMarketDataDrive drive, StorageFormats format,
            DataType dataType, CancellationToken cancellationToken)
        {
            if (securities.Length == 0)
            {
                logs.LogWarning("No instruments.");
                return Task.CompletedTask;
            }

            var lineChart = panel.CreateChart<DateTimeOffset, decimal>();
            var histogramChart = panel.CreateChart<DateTimeOffset, decimal>();

            foreach (var security in securities)
            {
                // stop calculation if user cancel script execution
                if (cancellationToken.IsCancellationRequested)
                    break;

                var candlesSeries = new Dictionary<DateTimeOffset, decimal>();
                var volsSeries = new Dictionary<DateTimeOffset, decimal>();

                // get candle storage
                var candleStorage = storage.GetCandleMessageStorage(security, dataType, drive, format);

                foreach (var candle in candleStorage.Load(from, to))
                {
                    // fill series
                    candlesSeries[candle.OpenTime] = candle.ClosePrice;
                    volsSeries[candle.OpenTime] = candle.TotalVolume;
                }

                // draw series on chart as line and histogram
                lineChart.Append($"{security} (close)", candlesSeries.Keys, candlesSeries.Values,
                    DrawStyles.DashedLine);
                histogramChart.Append($"{security} (vol)", volsSeries.Keys, volsSeries.Values,
                    DrawStyles.Histogram);
            }

            return Task.CompletedTask;
        }
    }
}
```

Overview

This script is designed to draw charts based on the price and volume data of financial instruments over a specific time period. It implements the [IAnalyticsScript](#) interface, which defines a contract for any analytical script that can be executed in the **Hydra** program.

IAalyticsScript Interface

The [IAalyticsScript](#) interface ensures that any implementing analytical script will have the [Run](#) method, which is necessary for performing the script's analytical operations.

Run Method

The [Run](#) method is the entry point of an analytical script, where actual data processing and analytical operations are performed.

Parameters:

- `logs`: Takes an instance of [ILogReceiver](#) for logging within the script.
- `panel`: Provides [IAalyticsPanel](#), which is a user interface element for drawing charts and displaying results.
- `securities`: An array of [SecurityId](#) identifying the financial instruments for analysis.
- `from`: The start date of the data range for analysis.
- `to`: The end date of the data range for analysis.
- `storage`: An instance of [IStorageRegistry](#) allowing access to the market data storage.
- `drive`: Represents [IMarketDataDrive](#) to specify the location of market data storage.
- `format`: A [StorageFormats](#) value indicating the market data format.
- `dataType`: [DataType](#) describing the requested market data type and its parameters (for example, the candle time-frame).
- `cancellationToken`: [CancellationToken](#) monitoring for cancellation requests.

Returns:

- [Task](#), representing the asynchronous operation of the analytical script.

Implementation Details

The `ChartDrawScript` class specifically processes market data for each provided security. It creates two types of charts: a line chart for closing prices and a histogram for volume data.

Main Processing Stages:

1. Check for the presence of instruments to process. If none are available, log a warning and complete the task.
2. Create a line chart and a histogram using the [IAalyticsPanel.CreateChart](#) method.
3. Iterate through each security and check for cancellation requests.
4. Get the candle storage using the `storage.GetCandleMessageStorage` method.
5. Load candle data within the specified date range.
6. Fill dictionaries with open time series data, corresponding closing prices, and total volumes.
7. Draw series data on charts using `lineChart.Append` and `histogramChart.Append` methods.

The script utilizes styles, such as [DrawStyles.DashedLine](#) for the line chart and [DrawStyles.Histogram](#) for the histogram, to visually distinguish different data presentations.

By implementing [IAalyticsScript](#), the `ChartDrawScript` class enables integrating an approach to

executing customizable analytical scripts, making it a versatile tool for traders and analysts using the StockSharp platform.

Execution Result

[image: hydra_analytics_chart]

Intraday Volume

The "Intraday Volume" script is a tool for analyzing the distribution of trading volume of securities by hours within a single trading session. Designed for use within the StockSharp platform, it targets traders and quantitative analysts aiming for an in-depth study of market behavior and trading strategy optimization.

[image: hydra_analytics_intraday_volume]

Functional Description

The script collects data on trading operations for a selected time period and presents it in a graph format, allowing users to visualize changes in trading volume by hour. This provides the ability to assess which hours of the day witness increased or decreased trading activity.

Practical Significance

- **For Trading:** Understanding peak and off-peak hours helps identify the most active market periods, influencing decisions on when to enter or exit positions.
- **For Quantitative Analysis:** Quant analysts can use intraday volume data to create mathematical models and algorithms predicting market behavior based on volume indicators.

Hourly Distribution

The distribution of trading volume by hour sheds light on market dynamics, highlighting the time intervals with the main trading activity. This may indicate changes in trends, support and resistance levels, as well as potential moments of increased liquidity or its shortage.

Data Application

The "Intraday Volume" script can be integrated into a broader market analysis system, providing data that can be used for:

- **Strategy Adaptation:** Adjusting trading algorithm parameters according to market activity levels.
- **Risk Assessment:** Calculating the probability of significant price movements depending on the time of day.

Using the "Intraday Volume" script within the StockSharp trading platform allows traders and analysts to base their decisions on specific data regarding market activity and adapt their strategies to match current trading conditions optimally.

Script Code on C#

```
namespace StockSharp.Algo.Analytics
{
    /// <summary>
    /// The analytic script, calculating distribution of the biggest volume by hours.
    /// </summary>
    public class TimeVolumeScript : IAnalyticsScript
    {
        Task IAnalyticsScript.Run(ILogReceiver logs, IAnalyticsPanel panel, SecurityId[] securities,
            DateTime from, DateTime to, IStorageRegistry storage, IMarketDataDrive drive, StorageFormats format,
            DataType dataType, CancellationToken cancellationToken)
        {
            if (securities.Length == 0)
            {
            }
        }
    }
}
```

```

{
    logs.LogWarning("No instruments.");
    return Task.CompletedTask;
}

// script can process only 1 instrument
var security = securities.First();

// get candle storage
var candleStorage = storage.GetCandleMessageStorage(security, dataType, drive, format);

// get available dates for the specified period
var dates = candleStorage.GetDates(from, to).ToArray();

if (dates.Length == 0)
{
    logs.LogWarning("no data");
    return Task.CompletedTask;
}

// grouping candles by opening time (time part only) with 1 hour truncating
var rows = candleStorage.Load(from, to)
    .GroupBy(c => c.OpenTime.TimeOfDay.Truncate(TimeSpan.FromHours(1)))
    .ToDictionary(g => g.Key, g => g.Sum(c => c.TotalVolume));

// put our calculations into grid
var grid = panel.CreateGrid("Time", "Volume");

foreach (var row in rows)
    grid.SetRow(row.Key, row.Value);

// sorting by volume column (descending)
grid.SetSort("Volume", false);

return Task.CompletedTask;
}
}
}

```

Script Code on Python

```

import clr

# Add .NET references
clr.AddReference("StockSharp.Algo.Analytics")
clr.AddReference("StockSharp.Messages")
clr.AddReference("Ecng.Drawing")

from Ecng.Drawing import DrawStyles
from System import TimeSpan
from System.Threading.Tasks import Task
from StockSharp.Algo.Analytics import IAnalyticsScript
from storage_extensions import *
from candle_extensions import *
from chart_extensions import *
from indicator_extensions import *

# The analytic script, calculating distribution of the biggest volume by hours.
class time_volume_script(IAnalyticsScript):
    def Run(
        self,
        logs,
        panel,
        securities,
        from_date,
        to_date,
        storage,

```

```

drive,
format,
data_type,
cancellation_token
):
# Check if there are no instruments
if not securities:
    logs.LogWarning("No instruments.")
    return Task.CompletedTask

# Script can process only 1 instrument
security = securities[0]

if data_type is None:
    logs.LogWarning(f"Unsupported data type {data_type}.")
    return Task.CompletedTask

message_type = data_type.MessageType

# Get candle storage
candle_storage = get_candle_storage(storage, security, data_type, drive, format)

# Get available dates for the specified period
dates = get_dates(candle_storage, from_date, to_date)

if len(dates) == 0:
    logs.LogWarning("no data")
    return Task.CompletedTask

# Grouping candles by opening time (hourly truncation) and summing their volumes
candles = load_range(candle_storage, message_type, from_date, to_date)
rows = {}
for candle in candles:
    time_of_day = candle.OpenTime.TimeOfDay
    truncated = TimeSpan.FromHours(int(time_of_day.TotalHours))
    rows[truncated] = rows.get(truncated, 0) + candle.TotalVolume

# Put our calculations into grid
grid = panel.CreateGrid("Time", "Volume")

for key, value in rows.items():
    grid.SetRow(key, value)

# Sorting by Volume column in descending order
grid.SetSort("Volume", False)

return Task.CompletedTask

```

Volume profile

Volume Profile

The "Volume Profile" script serves as a tool for analyzing the distribution of trading volume across price levels over a selected period. It enables traders and quantitative analysts to visualize and examine where the main trading activity was concentrated in terms of price levels.

[image: hydra_analytics_volume_profile]

Functionality Description

The script aggregates transaction data to form a profile that displays the volumes executed at various price levels. This information can be represented on a chart, illustrating the density of trades across different price points.

Practical Significance

Analyzing the volume profile helps in identifying key demand and supply zones and can be used for:

- Identifying support and resistance levels where the instrument finds significant interest from market participants.
- Assessing the strength of the current trend or its potential weakening, based on the change in volume distribution.
- Planning entry and exit points from the market, considering levels with maximum accumulated liquidity.

Application in Trading and Quantitative Analysis

- **Trading:** The volume profile can be utilized to develop strategies based on volume analysis, providing a clear view of where the main trading operations occur.
- **Quantitative Analysis:** Data on volume distribution can serve as input for quantitative models predicting the likelihood of price movements based on the volume accumulated at a level.

Script Implementation

The "Volume Profile" script performs the following steps:

1. **Data Collection:** The script aggregates transaction data for the specified period.
2. **Profile Formation:** Based on the collected data, the script forms a volume profile reflecting trading activity at each price level.
3. **Visualization:** The results of the script's operation are visualized as a chart or histogram, where each bar corresponds to a specific price level and its trading volume.

Utilizing the "Volume Profile" script within the StockSharp platform allows for comprehensive market analysis, building well-founded trading hypotheses, and improving the quality of trading decisions made.

Script Code on C#

```
namespace StockSharp.Algo.Analytics
{
    /// <summary>
    /// The analytic script, calculating distribution of the volume by price levels.
    /// </summary>
```

```

public class PriceVolumeScript : IAnalyticsScript
{
    Task IAnalyticsScript.Run(ILogReceiver logs, IAnalyticsPanel panel, SecurityId[] securities,
    DateTime from, DateTime to, IStorageRegistry storage, IMarketDataDrive drive, StorageFormats format,
    DataType dataType, CancellationToken cancellationToken)
    {
        if (securities.Length == 0)
        {
            logs.LogWarning("No instruments.");
            return Task.CompletedTask;
        }

        // script can process only 1 instrument
        var security = securities.First();

        // get candle storage
        var candleStorage = storage.GetCandleMessageStorage(security, dataType, drive, format);

        // get available dates for the specified period
        var dates = candleStorage.GetDates(from, to).ToArray();

        if (dates.Length == 0)
        {
            logs.LogWarning("no data");
            return Task.CompletedTask;
        }

        // grouping candles by middle price
        var rows = candleStorage.Load(from, to)
            .GroupBy(c => c.LowPrice + c.GetLength() / 2)
            .ToDictionary(g => g.Key, g => g.Sum(c => c.TotalVolume));

        // draw on chart
        panel.CreateChart<decimal, decimal>()
            .Append(security.ToStringId(), rows.Keys, rows.Values, DrawStyles.Histogram);

        return Task.CompletedTask;
    }
}

```

Script Code on Python

```

import clr

# Add .NET references
clr.AddReference("StockSharp.Messages")
clr.AddReference("StockSharp.Algo.Analytics")
clr.AddReference("Ecng.Drawing")

from Ecng.Drawing import DrawStyles
from System.Threading.Tasks import Task
from StockSharp.Algo.Analytics import IAnalyticsScript
from storage_extensions import *
from candle_extensions import *
from chart_extensions import *
from indicator_extensions import *

# The analytic script, calculating distribution of the volume by price levels.
class price_volume_script(IAnalyticsScript):
    def Run(
        self,
        logs,
        panel,
        securities,
        from_date,
        to_date,

```

```

storage,
drive,
format,
data_type,
cancellation_token
):
# Check if there are no instruments
if not securities:
    logs.LogWarning("No instruments.")
    return Task.CompletedTask

# Script can process only 1 instrument
security = securities[0]

if data_type is None:
    logs.LogWarning(f"Unsupported data type {data_type}.")
    return Task.CompletedTask

message_type = data_type.MessageType

# Get candle storage
candle_storage = get_candle_storage(storage, security, data_type, drive, format)

# Get available dates for the specified period
dates = get_dates(candle_storage, from_date, to_date)

if len(dates) == 0:
    logs.LogWarning("no data")
    return Task.CompletedTask

# Grouping candles by middle price and summing their volumes
candles = load_range(candle_storage, message_type, from_date, to_date)
rows_dict = {}
for candle in candles:
    # Calculate middle price of the candle
    key = candle.LowPrice + get_length(candle) / 2
    # Sum volumes for same price level
    rows_dict[key] = rows_dict.get(key, 0) + candle.TotalVolume

# Draw on chart
chart = create_chart(panel, float, float)
chart.Append(to_string_id(security), list(rows_dict.keys()), list(rows_dict.values()),
DrawStyles.Histogram)

return Task.CompletedTask

```


Indicators

The "Indicator" script is intended to demonstrate working with technical analysis indicators within the StockSharp platform. It enables users to load historical data, apply various indicators to it, and display the results on a chart. This approach aids in market trend analysis and making informed trading decisions.

[image: hydra_analytics_indicator]

Functional Capabilities

The script provides the following functionality:

- **Loading historical data:** Selecting financial instruments of interest and loading their historical data for a specified period.
- **Applying indicators:** Applying one or more technical analysis indicators to the loaded data.
- **Visualization:** Displaying data and analysis results using indicators on a chart, providing a clear view of market dynamics.

Examples of Indicators

The script can work with a wide range of indicators, including but not limited to:

- **Moving Averages (MA):** Represent the average price over a certain period, helping to identify trends.
- **Relative Strength Index (RSI):** Assesses the magnitude and speed of price changes, helping to identify overbought or oversold conditions.
- **Bollinger Bands (BB):** Show the range and volatility of prices, based on moving averages and standard deviations.

Application in Trading and Analysis

Using technical analysis indicators through this script allows for:

- **Identifying trends:** Detecting the market's direction of movement to plan entry and exit strategies.
- **Identifying reversal points:** Determining moments when the market trend may change its direction.
- **Analyzing volatility:** Assessing the level of price instability to adapt strategies to market conditions.

Implementation in the Script

To work with the script, the following steps need to be taken:

1. **Selecting an instrument and period:** Determining securities and the time frame for analysis.
2. **Applying indicators:** Choosing and setting parameters for the indicators to be applied to the data.
3. **Displaying results:** Visualizing historical data and indicators on a chart for analysis.

The "Indicator" script provides a powerful tool for in-depth analysis of financial markets, allowing traders and analysts to use these indicators to develop effective trading strategies.

Script Code on C#

```

namespace StockSharp.Algo.Analytics
{
    /// <summary>
    /// The analytic script, using indicator ROC.
    /// </summary>
    public class IndicatorScript : IAnalyticsScript
    {
        Task IAnalyticsScript.Run(ILogReceiver logs, IAnalyticsPanel panel, SecurityId[] securities,
            DateTime from, DateTime to, IStorageRegistry storage, IMarketDataDrive drive, StorageFormats format,
            DataType dataType, CancellationToken cancellationToken)
        {
            if (securities.Length == 0)
            {
                logs.LogWarning("No instruments.");
                return Task.CompletedTask;
            }

            // creating 2 panes for candles and indicator series
            var candleChart = panel.CreateChart<DateTimeOffset, decimal>();
            var indicatorChart = panel.CreateChart<DateTimeOffset, decimal>();

            foreach (var security in securities)
            {
                // stop calculation if user cancel script execution
                if (cancellationToken.IsCancellationRequested)
                    break;

                var candlesSeries = new Dictionary<DateTimeOffset, decimal>();
                var indicatorSeries = new Dictionary<DateTimeOffset, decimal>();

                // creating ROC
                var roc = new RateOfChange();

                // get candle storage
                var candleStorage = storage.GetCandleMessageStorage(security, dataType, drive, format);

                foreach (var candle in candleStorage.Load(from, to))
                {
                    // fill series
                    candlesSeries[candle.OpenTime] = candle.ClosePrice;
                    indicatorSeries[candle.OpenTime] = roc.Process(candle).ToDecimal();
                }

                // draw series on chart
                candleChart.Append($"{security} (close)", candlesSeries.Keys, candlesSeries.Values);
                indicatorChart.Append($"{security} (ROC)", indicatorSeries.Keys, indicatorSeries.Values);
            }

            return Task.CompletedTask;
        }
    }
}

```

Script Code on Python

```

import clr

# Add .NET references
clr.AddReference("StockSharp.Messages")
clr.AddReference("StockSharp.Algo.Analytics")
clr.AddReference("Ecng.Drawing")

from Ecng.Drawing import DrawStyles
from System.Threading.Tasks import Task
from StockSharp.Algo.Analytics import IAnalyticsScript
from StockSharp.Algo.Indicators import ROC
from storage_extensions import *

```

```

from candle_extensions import *
from chart_extensions import *
from indicator_extensions import *

# The analytic script, using indicator ROC.
class indicator_script(IAnalyticsScript):
    def Run(self, logs, panel, securities, from_date, to_date, storage, drive, format, data_type,
cancellation_token):
        if not securities:
            logs.LogWarning("No instruments.")
            return Task.CompletedTask

        # creating 2 panes for candles and indicator series
        candle_chart = create_chart(panel, datetime, float)
        indicator_chart = create_chart(panel, datetime, float)

        if data_type is None:
            logs.LogWarning(f"Unsupported data type {data_type}.")
            return Task.CompletedTask

        message_type = data_type.MessageType

        for security in securities:
            # stop calculation if user cancel script execution
            if cancellation_token.IsCancellationRequested:
                break

            candles_series = {}
            indicator_series = {}

            # creating ROC
            roc = ROC()

            # get candle storage
            candle_storage = get_candle_storage(storage, security, data_type, drive, format)

            for candle in load_range(candle_storage, message_type, from_date, to_date):
                # fill series
                candles_series[candle.OpenTime] = candle.ClosePrice
                indicator_series[candle.OpenTime] = to_decimal(process_candle(roc, candle))

            # draw series on chart
            candle_chart.Append(
                f"{security} (close)",
                list(candles_series.keys()),
                list(candles_series.values())
            )
            indicator_chart.Append(
                f"{security} (ROC)",
                list(indicator_series.keys()),
                list(indicator_series.values())
            )

        return Task.CompletedTask

```

3D Chart

3D Chart

The Chart3DScript script demonstrates the creation of a 3D chart to visualize the distribution of trading volumes by hour for various financial instruments. This visualization method allows for a clear representation of trading dynamics and identification of market activity peaks.

[image: hydra_analytics_chart3d]

Script Operation Description

The script analyzes candle data for the specified period, groups it by hour, and calculates the total trading volume for each hour. The results are presented in a 3D chart, where the axes represent:

- **X-Axis:** Financial instruments.
- **Y-Axis:** Trading session hours (from 0 to 23).
- **Z-Axis:** Trading volumes.

Usefulness of Using a 3D Chart

Market Activity Analysis

The 3D chart allows for the assessment of when the greatest activity occurs for multiple instruments simultaneously. This can be useful for identifying optimal trading windows or studying the impact of global events on the market.

Instrument Comparison

Thanks to the visualization of trading volumes by hour in three-dimensional space, traders can compare instruments against each other in terms of activity level and preferred trading times. This can assist in selecting the most liquid instruments at certain hours or in finding instruments with similar activity patterns for portfolio diversification.

Strategy Optimization

Analyzing the distribution of trading volumes can serve as a basis for optimizing trading strategies, allowing for adaptation to time frames with the highest market activity. This is particularly relevant for algorithmic and high-frequency trading.

Script Implementation

The script performs the following actions:

1. Checking for the presence of financial instruments for analysis.
2. Forming labels for the X (instruments) and Y (hours) axes.
3. Loading and grouping candle data.
4. Calculating the total trading volumes by hour and filling in the data for the Z-axis.
5. Drawing the 3D chart using the `pane1.Draw3D` method.

Script Code on C#

```

namespace StockSharp.Algo.Analytics
{
    /// <summary>
    /// The analytic script, calculating distribution of the biggest volume by hours
    /// and shows its in 3D chart.
    /// </summary>
    public class Chart3DScript : IAnalyticsScript
    {
        Task IAnalyticsScript.Run(ILogReceiver logs, IAnalyticsPanel panel, SecurityId[] securities,
            DateTime from, DateTime to, IStorageRegistry storage, IMarketDataDrive drive, StorageFormats format,
            DataType dataType, CancellationToken cancellationToken)
        {
            if (securities.Length == 0)
            {
                logs.LogWarning("No instruments.");
                return Task.CompletedTask;
            }

            var x = new List<string>();
            var y = new List<string>();

            // fill Y labels
            for (var h = 0; h < 24; h++)
                y.Add(h.ToString());

            var z = new double[securities.Length, y.Count];

            for (var i = 0; i < securities.Length; i++)
            {
                // stop calculation if user cancel script execution
                if (cancellationToken.IsCancellationRequested)
                    break;

                var security = securities[i];

                // fill X labels
                x.Add(security.ToStringId());

                // get candle storage
                var candleStorage = storage.GetCandleMessageStorage(security, dataType, drive, format);

                // get available dates for the specified period
                var dates = candleStorage.GetDates(from, to).ToArray();

                if (dates.Length == 0)
                {
                    logs.LogWarning("no data");
                    return Task.CompletedTask;
                }

                // grouping candles by opening time (time part only) with 1 hour truncating
                var byHours = candleStorage.Load(from, to)
                    .GroupBy(c => c.OpenTime.TimeOfDay.Truncate(TimeSpan.FromHours(1)))
                    .ToDictionary(g => g.Key.Hours, g => g.Sum(c => c.TotalVolume));

                // fill Z values
                foreach (var pair in byHours)
                    z[i, pair.Key] = (double)pair.Value;
            }

            panel.Draw3D(x, y, z, "Instruments", "Hours", "Volume");

            return Task.CompletedTask;
        }
    }
}

```

Script Code on Python

```

import clr

# Add .NET references
clr.AddReference("StockSharp.Messages")
clr.AddReference("StockSharp.Algo.Analytics")
clr.AddReference("Ecng.Drawing")

from Ecng.Drawing import DrawStyles
from System.Threading.Tasks import Task
from StockSharp.Algo.Analytics import IAnalyticsScript
from storage_extensions import *
from candle_extensions import *
from chart_extensions import *
from numpy_extensions import nx

# The analytic script, calculating distribution of the biggest volume by hours and shows its in 3D
chart.
class chart3d_script(IAnalyticsScript):
    def Run(
        self,
        logs,
        panel,
        securities,
        from_date,
        to_date,
        storage,
        drive,
        format,
        data_type,
        cancellation_token
    ):
        # Check if there are no instruments
        if not securities:
            logs.LogWarning("No instruments.")
            return Task.CompletedTask

        x = [] # X labels for instruments
        y = [] # Y labels for hours

        # Fill Y labels with hours 0 to 23
        for h in range(24):
            y.append(str(h))

        # Create a 2D array for Z values with dimensions: (number of securities) x (number of hours)
        z = [[0.0 for _ in range(len(y))] for _ in range(len(securities))]

        if data_type is None:
            logs.LogWarning(f"Unsupported data type {data_type}.")
            return Task.CompletedTask

        message_type = data_type.MessageType

        for i, security in enumerate(securities):
            # Stop calculation if user cancels script execution
            if cancellation_token.IsCancellationRequested:
                break

            # Fill X labels with security identifiers
            x.append(to_string_id(security))

            # Get candle storage for current security
            candle_storage = get_candle_storage(storage, security, data_type, drive, format)

            # Get available dates for the specified period
            dates = get_dates(candle_storage, from_date, to_date)

            if len(dates) == 0:
                logs.LogWarning("no data")
                return Task.CompletedTask

```

```

# Grouping candles by opening time (truncated to the nearest hour) and summing volumes
candles = load_range(candle_storage, message_type, from_date, to_date)
by_hours = {}
for candle in candles:
    hour = int(candle.OpenTime.TimeOfDay.TotalHours)
    by_hours[hour] = by_hours.get(hour, 0) + candle.TotalVolume

# Fill Z values for current security
for hour, volume in by_hours.items():
    if hour < len(y):
        z[i][hour] = float(volume)

# Draw the 3D chart using panel
panel.Draw3D(x, y, nx.to2darray(z), "Instruments", "Hours", "Volume")

return Task.CompletedTask

```

Normalization

Closing Price Normalization

The "Closing Price Normalization" script is designed to standardize the closing prices of financial instruments, enabling comparison and analysis of different assets on a unified scale. This is particularly useful when comparing instruments with varying costs and volatility.

[image: hydra_analytics_normalize]

Script Operation Description

The script adapts closing price data by scaling or transforming them according to the selected normalization method. The result is a set of standardized values that can be used for quantitative comparison and multi-instrument analysis.

Application of Normalization

- **Price Scale Unification:** Normalization helps bring data from various instruments to a single scale, simplifying visual comparison and analytical assessment.
- **Correlation Analysis:** Standardized data allows for the identification of correlational relationships between assets and the formation of diversified portfolios.
- **Index and Model Creation:** Normalized prices are used to create composite indices, pricing models, and other quantitative research.

Normalization Methodology

Normalization may include the following methods:

1. **Scaling:** Adjusting closing prices to a certain range of values, such as from 0 to 1.
2. **Z-score:** Transforming closing prices using the Z-score, which indicates how many standard deviations a value deviates from the mean.
3. **Logarithmization:** Applying logarithmic transformation to smooth out data dispersion and reduce the impact of extreme values.

Script Implementation

The normalization process typically includes the following steps:

1. **Normalization Selection:** Determining the normalization method based on the analysis goals and data characteristics.
2. **Data Processing:** Applying the chosen normalization method to the closing prices for each instrument.
3. **Results Analysis:** Using normalized data for subsequent analysis and comparison of instruments.

The "Closing Price Normalization" script is a crucial tool for preparing data for trading and quantitative analysis, allowing traders and analysts to more accurately compare and evaluate financial assets within various strategies and studies.

Script Code on C#

```
namespace StockSharp.Algo.Analytics
```



```

{
    /// <summary>
    /// The analytic script, normalize securities close prices and shows on same chart.
    /// </summary>
    public class NormalizePriceScript : IAnalyticsScript
    {
        Task IAnalyticsScript.Run(ILogReceiver logs, IAnalyticsPanel panel, SecurityId[] securities,
            DateTime from, DateTime to, IStorageRegistry storage, IMarketDataDrive drive, StorageFormats format,
            DataType dataType, CancellationToken cancellationToken)
        {
            if (securities.Length == 0)
            {
                logs.LogWarning("No instruments.");
                return Task.CompletedTask;
            }

            var chart = panel.CreateChart<DateTimeOffset, decimal>();

            foreach (var security in securities)
            {
                // stop calculation if user cancel script execution
                if (cancellationToken.IsCancellationRequested)
                    break;

                var series = new Dictionary<DateTimeOffset, decimal>();

                // get candle storage
                var candleStorage = storage.GetCandleMessageStorage(security, dataType, drive, format);

                decimal? firstClose = null;

                foreach (var candle in candleStorage.Load(from, to))
                {
                    firstClose ??= candle.ClosePrice;

                    // normalize close prices by dividing on first close
                    series[candle.OpenTime] = candle.ClosePrice / firstClose.Value;
                }

                // draw series on chart
                chart.Append(security.ToStringId(), series.Keys, series.Values);
            }

            return Task.CompletedTask;
        }
    }
}

```

Script Code on Python

```

import clr

# Add .NET references
clr.AddReference("StockSharp.Messages")
clr.AddReference("StockSharp.Algo.Analytics")
clr.AddReference("Ecng.Drawing")

from Ecng.Drawing import DrawStyles
from System.Threading.Tasks import Task
from StockSharp.Algo.Analytics import IAnalyticsScript
from storage_extensions import *
from candle_extensions import *
from chart_extensions import *
from indicator_extensions import *

# The analytic script, normalize securities close prices and shows on same chart.
class normalize_price_script(IAnalyticsScript):

```

```

def Run(self, logs, panel, securities, from_date, to_date, storage, drive, format, data_type,
cancellation_token):
    if not securities:
        logs.LogWarning("No instruments.")
        return Task.CompletedTask

    chart = create_chart(panel, datetime, float)

    if data_type is None:
        logs.LogWarning(f"Unsupported data type {data_type}.")
        return Task.CompletedTask

    message_type = data_type.MessageType

    for security in securities:
        # stop calculation if user cancel script execution
        if cancellation_token.IsCancellationRequested:
            break

        series = {}

        # get candle storage
        candle_storage = get_candle_storage(storage, security, data_type, drive, format)

        first_close = None

        for candle in load_range(candle_storage, message_type, from_date, to_date):
            if first_close is None:
                first_close = candle.ClosePrice

            # normalize close prices by dividing on first close
            series[candle.OpenTime] = candle.ClosePrice / first_close

        # draw series on chart
        chart.Append(to_string_id(security), list(series.keys()), list(series.values()))

    return Task.CompletedTask

```

Pearson Correlation

Pearson Correlation

The Pearson correlation is a statistical method used to measure the degree of linear relationship between two quantitative variables. Within financial analysis, this method is widely utilized to explore the relationships between different assets, such as stocks or currency pairs.

[image: hydra_analytics_pearson_correlation]

Method Description

The Pearson correlation coefficient ranges from -1 to 1, where:

- 1 indicates a perfect positive correlation,
- 0 suggests no linear relationship,
- -1 signifies a perfect negative correlation.

The value of the coefficient shows how closely two variables are linearly related.

Practical Application

- **Portfolio Management:** Assessing correlations between assets aids in constructing diversified portfolios to minimize risk.
- **Hedging Strategies:** Identifying assets with high negative correlation can be used to develop hedging strategies.
- **Market Trend Analysis:** Studying correlations between different markets and instruments provides insights into global economic interconnections.

Calculating Pearson Correlation

The Pearson correlation coefficient is calculated using the following formula:

$$[r = \frac{n(\sum xy) - (\sum x)(\sum y)}{\sqrt{[n\sum x^2 - (\sum x)^2][n\sum y^2 - (\sum y)^2]}}]$$

where:

- (n) is the number of observations,
- (x) and (y) are the values of the variables,
- ($\sum$) denotes summation.

Script Implementation

A script for calculating Pearson correlation should include the following steps:

1. **Data Collection:** Load time series for the two variables being analyzed.
2. **Preliminary Processing:** Align time series by dates and remove missing values.
3. **Calculation:** Apply the Pearson correlation formula to the processed data.
4. **Results Analysis:** Interpret the resulting correlation coefficient for decision-making.

Calculating Pearson correlation provides crucial information for market analysis and investment strategy optimization, allowing the assessment of the degree of mutual influence between financial assets.

Script Code on C#

```
namespace StockSharp.Algo.Analytics
{
    using MathNet.Numerics.Statistics;

    /// <summary>
    /// The analytic script, calculating Pearson correlation by specified securities.
    /// </summary>
    public class PearsonCorrelationScript : IAnalyticsScript
    {
        Task IAnalyticsScript.Run(ILogReceiver logs, IAnalyticsPanel panel, SecurityId[] securities,
            DateTime from, DateTime to, IStorageRegistry storage, IMarketDataDrive drive, StorageFormats format,
            DataType dataType, CancellationToken cancellationToken)
        {
            if (securities.Length == 0)
            {
                logs.LogWarning("No instruments.");
                return Task.CompletedTask;
            }

            var closes = new List<double[]>();

            foreach (var security in securities)
            {
                // stop calculation if user cancel script execution
                if (cancellationToken.IsCancellationRequested)
                    break;

                // get candle storage
                var candleStorage = storage.GetCandleMessageStorage(security, dataType, drive, format);

                // get closing prices
                var prices = candleStorage.Load(from, to).Select(c => (double)c.ClosePrice).ToArray();

                if (prices.Length == 0)
                {
                    logs.LogWarning("No data for {0}", security);
                    return Task.CompletedTask;
                }

                closes.Add(prices);
            }

            // all array must be same length, so truncate longer
            var min = closes.Select(arr => arr.Length).Min();

            for (var i = 0; i < closes.Count; i++)
            {
                var arr = closes[i];

                if (arr.Length > min)
                    closes[i] = arr.Take(min).ToArray();
            }

            // calculating correlation
            var matrix = Correlation.PearsonMatrix(closes);

            // displaying result into heatmap
            var ids = securities.Select(s => s.ToStringId());
            panel.DrawHeatmap(ids, ids, matrix.ToArray());

            return Task.CompletedTask;
        }
    }
}
```

Script Code on Python

```
import clr

# Add .NET references
clr.AddReference("StockSharp.Messages")
clr.AddReference("StockSharp.Algo.Analytics")
clr.AddReference("Ecng.Drawing")

from Ecng.Drawing import DrawStyles
from System.Threading.Tasks import Task
from StockSharp.Algo.Analytics import IAnalyticsScript
from storage_extensions import *
from candle_extensions import *
from chart_extensions import *
from indicator_extensions import *
from numpy_extensions import nx

clr.AddReference("NumpyDotNet")
from NumpyDotNet import np

# The analytic script, calculating Pearson correlation by specified securities.
class pearson_correlation_script(IAnalyticsScript):
    def Run(
        self,
        logs,
        panel,
        securities,
        from_date,
        to_date,
        storage,
        drive,
        format,
        data_type,
        cancellation_token
    ):
        if not securities:
            logs.LogWarning("No instruments.")
            return Task.CompletedTask

        closes = []

        if data_type is None:
            logs.LogWarning(f"Unsupported data type {data_type}.")
            return Task.CompletedTask

        message_type = data_type.MessageType

        for security in securities:
            # stop calculation if user cancel script execution
            if cancellation_token.IsCancellationRequested:
                break

            # get candle storage
            candle_storage = get_candle_storage(storage, security, data_type, drive, format)

            # get closing prices
            prices = [float(c.ClosePrice) for c in load_range(candle_storage, message_type, from_date,
to_date)]

            if len(prices) == 0:
                logs.LogWarning("No data for {0}", security)
                return Task.CompletedTask

            closes.append(prices)

        # all arrays must be the same length, so truncate longer ones
        min_length = min(len(arr) for arr in closes)
```

```
closes = [arr[:min_length] for arr in closes]

# convert list or array into 2D array
array2d = nx.to2darray(closes)

# calculating correlation using NumSharp
np_array = np.array(array2d)
matrix = np.corrcoef(np_array)

# displaying result into heatmap
ids = [to_string_id(s) for s in securities]
panel.DrawHeatmap(ids, ids, nx.toSystemArray(matrix))

return Task.CompletedTask
```

Largest Candles

Largest Candles

The "Largest Candles" script is designed to identify candles with the maximum volume and the greatest body length on the charts of selected financial instruments over a given period. This tool allows traders and analysts to identify significant market events and the reaction of market participants.

[image: hydra_analytics_big_candle]

Key Features

The script analyzes a set of specified instruments, searches among them for candles with the greatest volume and body length, and displays this data on two graphs:

- **Candle Body Length Chart:** Shows candles with the greatest difference between the opening and closing price.
- **Trading Volume Chart:** Demonstrates candles with the maximum trading volume for the existence time of the candle.

Workflow

1. **Selection of Instruments and Analysis Period:** Determines the list of instruments and the time interval for analysis.
2. **Data Analysis:** Involves loading and analyzing historical candle data to identify candles with the greatest indicators.
3. **Visualization of Results:** Found candles are displayed on charts in the analytics panel interface.

Application

- **Market Activity Analysis:** Helps to determine moments of the greatest trader activity and potential market reversals.
- **Key Levels Identification:** Candles with significant volume and body length often form around key support and resistance levels.
- **Strategic Planning:** Information about the largest candles can be used for planning entry and exit points from the market, considering potential volatility.

Script Code on C#

```
namespace StockSharp.Algo.Analytics
{
    /// <summary>
    /// The analytic script, shows biggest candle (by volume and by length) for specified securities.
    /// </summary>
    public class BiggestCandleScript : IAnalyticsScript
    {
        Task IAnalyticsScript.Run(ILogReceiver logs, IAnalyticsPanel panel, SecurityId[] securities,
            DateTime from, DateTime to, IStorageRegistry storage, IMarketDataDrive drive, StorageFormats format,
            DataType dataType, CancellationToken cancellationToken)
        {
            if (securities.Length == 0)
            {
                logs.LogWarning("No instruments.");
                return Task.CompletedTask;
            }
        }
    }
}
```

```

}

var priceChart = panel.CreateChart<DateTimeOffset, decimal, decimal>();
var volChart = panel.CreateChart<DateTimeOffset, decimal, decimal>();

var bigPriceCandles = new List<CandleMessage>();
var bigVolCandles = new List<CandleMessage>();

foreach (var security in securities)
{
    // stop calculation if user cancel script execution
    if (cancellationToken.IsCancellationRequested)
        break;

    // get candle storage
    var candleStorage = storage.GetCandleMessageStorage(security, dataType, drive, format);

    var allCandles = candleStorage.Load(from, to).ToArray();

    // first orders by volume desc will be our biggest candle
    var bigPriceCandle = allCandles.OrderByDescending(c => c.GetLength()).FirstOrDefault();
    var bigVolCandle = allCandles.OrderByDescending(c => c.TotalVolume).FirstOrDefault();

    if (bigPriceCandle != null)
        bigPriceCandles.Add(bigPriceCandle);

    if (bigVolCandle != null)
        bigVolCandles.Add(bigVolCandle);
}

// draw series on chart
priceChart.Append("prices", bigPriceCandles.Select(c => c.OpenTime), bigPriceCandles.Select(c =>
c.GetMiddlePrice(null)), bigPriceCandles.Select(c => c.GetLength()));
volChart.Append("prices", bigVolCandles.Select(c => c.OpenTime), bigPriceCandles.Select(c =>
c.GetMiddlePrice(null)), bigVolCandles.Select(c => c.TotalVolume));

return Task.CompletedTask;
}
}
}

```

Script Code on Python

```

import clr

# Add .NET references
clr.AddReference("StockSharp.Messages")
clr.AddReference("StockSharp.Algo.Analytics")
clr.AddReference("Ecng.Drawing")

from Ecng.Drawing import DrawStyles
from System.Threading.Tasks import Task
from StockSharp.Algo.Analytics import IAnalyticsScript
from storage_extensions import *
from candle_extensions import *
from chart_extensions import *
from indicator_extensions import *

# The analytic script, shows biggest candle (by volume and by length) for specified securities.
class biggest_candle_script(IAnalyticsScript):
    def Run(self, logs, panel, securities, from_date, to_date, storage, drive, format, data_type,
cancellation_token):
        if not securities:
            logs.LogWarning("No instruments.")
            return Task.CompletedTask

    price_chart = create_3d_chart(panel, datetime, float, float)

```



```

vol_chart = create_3d_chart(panel, datetime, float, float)

big_price_candles = []
big_vol_candles = []

if data_type is None:
    logs.LogWarning(f"Unsupported data type {data_type}.")
    return Task.CompletedTask

message_type = data_type.MessageType

for security in securities:
    # stop calculation if user cancel script execution
    if cancellation_token.IsCancellationRequested:
        break

    # get candle storage
    candle_storage = get_candle_storage(storage, security, data_type, drive, format)
    all_candles = load_range(candle_storage, message_type, from_date, to_date)

    if len(all_candles) > 0:
        # first orders by volume desc will be our biggest candle
        big_price_candle = max(all_candles, key=lambda c: get_length(c))
        big_vol_candle = max(all_candles, key=lambda c: c.TotalVolume)

        if big_price_candle is not None:
            big_price_candles.append(big_price_candle)

        if big_vol_candle is not None:
            big_vol_candles.append(big_vol_candle)

    # draw series on chart
    price_chart.Append(
        "prices",
        [c.OpenTime for c in big_price_candles],
        [get_middle_price(c) for c in big_price_candles],
        [get_length(c) for c in big_price_candles]
    )

    vol_chart.Append(
        "prices",
        [c.OpenTime for c in big_vol_candles],
        [get_middle_price(c) for c in big_price_candles],
        [c.TotalVolume for c in big_vol_candles]
    )

return Task.CompletedTask

```

Settings reset

Settings reset

Quick access panel of the **Settings => Settings** reset application.

Resets [Hydra](#) settings to the initial state.

Delete data

Delete data

Common tab, the **Delete data...** button.

[image: hydra data delete]

It allows you to delete the data for all or particular instruments for specific dates.

Network settings

Network settings

Quick access panel of the **Settings** => **Network** settings application.

[image: hydra proxy settings]

It allows you to configure the proxy settings.

Synchronization

Synchronization

Tab Import => Synchronize....

[image: hydra sync]

It allows you to recover the instruments from downloaded files.

Backup

Backup

The [S#](#) provides functionality for backup of downloaded data to the [Amazon Web Services \(AWS\)](#) cloud storage.

For backup purposes, it is required to create and configure an account at AWS, and then to create and configure the "Backup task".

Recommended content

[Setup](#)

[Hydra settings](#)

[Recovery](#)

Setup

Setup

The following describes how to create and configure an AWS account.

1. To create an AWS account, go to the [Create an AWS Account](#) page and click the **Create Account** button.*[image: Aws3 CreateAccount]*
2. Next, fill out the forms that the web service will offer.
3. Fill in your card details, this is done to verify your identity.*[image: Aws 3 Paytest]*
4. At one of the registration steps, you will be asked to enter a phone number and initiate a call to your phone using the **Call Me Now** button.*[image: Aws3 CallMeNow]*
You should answer the call and dial the code on the phone, which will be shown on the computer screen.
5. Next, you will be prompted to select a support plan. Upon completion of creating an account, you need to go to the management console.*[image: Aws3 console]*
6. The first step in setting up an account is to create a Bucket.*[image: Aws3 CreateBucket]*
Bucket is a container for storing objects in the cloud. For Bucket, you need to set a unique name, and also select a regional data center (Region) where the data is physically stored. Please note that further when configuring the backup task: 1) in the **Storage** field, you will need to enter the bucket name, 2) in the **Address** field, you need to use not the name, but the address of the regional data center, which can be found [here](#). Then continue the configuration.*[image: Aws3 CreateBucket][image: Aws 3 Create Bucket Name][image: Aws 3 Create Bucket Name property]*
7. Next, you need to set keys to programmatically access AWS services. To do this, go to the **Security Credentials** link in the AWS console.*[image: Aws3 SecurityCredentials]*
8. Expand the **Access Keys (Access Key ID and Secret Access Key)** header and create access keys using the *[image: Aws3 CreateNewAccessKey]* button.*[image: Aws3 SecurityCredentialsCreate]*

The created keys can be saved to a file using the **Download Key File** button.

Please note that when configuring a backup task, the **Access Key ID** must be used as a login and the **Secret Access Key** as a password.

Recommended content

[Creating and configuring a task](#)

Hydra settings

Hydra settings

The following describes how to create and configure a backup task.

1. To create a task, click on the **Add tasks...** button, in the window that opens, select the **Backup** item and click the **OK** button.*[image: hydra tasks backup add]*
2. Next, you need to configure the task.*[image: hydra tasks backup]*

Backup

- **Service** - the service address.
- **Address** - the region address. The address of the region specified in the bucket settings. See [Regions and Endpoints](#).
- **Storage** - the bucket name.
- **Login** - login. Access Key ID.
- **Password** - password. **Secret Access Key**.
- **Start date** - from what date to start the backup.
- **Time offset** - an offset in days from the current date.

General

- **Header** - Converter.
- **Working hours** - setting up the board work schedule. *[image: hydra tasks backup desk]*
- **Interval of operation** - the interval of operation.
- **Data directory** - data directory, from which the data for conversion will be received.
- **Format** - the converted data format: BIN/CSV.
- **Max. errors** - the maximum number of errors, upon which the task will be stopped. By default, 0 - the number of errors is ignored.
- **Dependency** - a task that must be performed before running the current one.

Logging

- **Identifier** - the identifier.
 - **Logging level** - the logging level.
3. After setting up the task, add the instruments that should be saved in the backup storage and click the **Start** button.

Recommended content

[Creating and configuring an account](#)

Recovery

You can restore the data from the storage in two ways:

- To download stored objects from the AWS console. To do this, right-click on the object and select **Download** in the context menu.*[image: Aws3 download]*
| [!TIP] This method allows you to download only one object (file) at a time.
- To use the free application [S3 Browser](#) for data downloading.

Create new source

Create new source

To add your own source, you need to create your own [IMessageAdapter](#). How to do this is described in [Creating your own connector](#). After creating, you need to add the resulting dll file to the folder with [Hydra](#).

Create new task

Create new task

[Hydra](#) is built with the ability to create custom extensions. To add your own extension, you need to copy the dll to the **Plugins** subfolder where [Hydra](#) is installed. Below is the process of creating a utility for backing up existing data (it is included as standard and is presented as training material).

Each utility has to implement the [IHydraTask](#) interface (or inherit from the [BaseHydraTask](#) or [ConnectorHydraTask`1](#) classes):

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.IO;
using System.Linq;
using System.Security;
using Amazon;
using Ecng.Backup;
using Ecng.Backup.Amazon;
using Ecng.Backup.Yandex;
using Ecng.Common;
using Ecng.ComponentModel;
using Ecng.Serialization;
using StockSharp.Algo;
using StockSharp.Algo.Storages;
using StockSharp.Hydra.Core;
using StockSharp.Localization;
using Ecng.Logging;
using StockSharp.Messages;
using DataType = StockSharp.Algo.DataType;
enum BackupServices
{
    AwsS3,
    AwsGlacier,
    Yandex,
}
[DisplayNameLoc(LocalizedStrings.BackupKey)]
[DescriptionLoc(LocalizedStrings.BackupDescriptionKey)]
[Doc("5a056352-64c7-41ea-87a8-2e112935e3b9.htm")]
[Icon("backup_logo.png")]
[MessageAdapterCategory(MessageAdapterCategories.Tool)]
class BackupTask : BaseHydraTask
{
    private const string _sourceName = LocalizedStrings.BackupKey;
    public BackupTask()
    {
        Offset = 1;
        StartFrom = new DateTime(2000, 1, 1);
        Interval = TimeSpan.FromDays(1);
        Service = BackupServices.AwsS3;
        Address = RegionEndpoint.USEast1.SystemName;
        ServiceRepo = "stocksharp";
        Login = string.Empty;
        Password = new SecureString();
    }
    [Display(
        ResourceType = typeof(LocalizedStrings),
        Name = LocalizedStrings.Str3427Key,
        Description = LocalizedStrings.Str3427Key + LocalizedStrings.Dot,
        GroupName = _sourceName,
        Order = 0)]
    public BackupServices Service { get; set; }
    [Display(
        ResourceType = typeof(LocalizedStrings),
```

```

    Name = LocalizedStrings.AddressKey,
    Description = LocalizedStrings.ServerAddressKey + LocalizedStrings.Dot,
    GroupName = _sourceName,
    Order = 1)]
public string Address { get; set; }
[Display(
    ResourceType = typeof(LocalizedStrings),
    Name = LocalizedStrings.Str1405Key,
    Description = LocalizedStrings.Str1405Key + LocalizedStrings.Dot,
    GroupName = _sourceName,
    Order = 2)]
public string ServiceRepo { get; set; }
[Display(
    ResourceType = typeof(LocalizedStrings),
    Name = LocalizedStrings.LoginKey,
    Description = LocalizedStrings.LoginKey + LocalizedStrings.Dot,
    GroupName = _sourceName,
    Order = 3)]
public string Login { get; set; }
[Display(
    ResourceType = typeof(LocalizedStrings),
    Name = LocalizedStrings.PasswordKey,
    Description = LocalizedStrings.PasswordKey + LocalizedStrings.Dot,
    GroupName = _sourceName,
    Order = 4)]
public SecureString Password { get; set; }
[Display(
    ResourceType = typeof(LocalizedStrings),
    Name = LocalizedStrings.Str2282Key,
    Description = LocalizedStrings.Str3779Key,
    GroupName = _sourceName,
    Order = 5)]
public DateTime StartFrom { get; set; }
[Display(
    ResourceType = typeof(LocalizedStrings),
    Name = LocalizedStrings.Str2284Key,
    Description = LocalizedStrings.Str3778Key,
    GroupName = _sourceName,
    Order = 6)]
public int Offset { get; set; }
public override void Save(SettingsStorage storage)
{
    base.Save(storage);
    storage.SetValue(nameof(Service), Service);
    storage.SetValue(nameof(Address), Address);
    storage.SetValue(nameof(ServiceRepo), ServiceRepo);
    storage.SetValue(nameof(Login), Login);
    storage.SetValue(nameof>Password), Password);
    storage.SetValue(nameof(StartFrom), StartFrom);
    storage.SetValue(nameof(Offset), Offset);
}
public override void Load(SettingsStorage storage)
{
    base.Load(storage);
    Service = storage.GetValue(nameof(Service), Service);
    Address = storage.GetValue(nameof(Address), Address);
    ServiceRepo = storage.GetValue(nameof(ServiceRepo), ServiceRepo);
    Login = storage.GetValue(nameof(Login), Login);
    Password = storage.GetValue(nameof>Password), Password);
    StartFrom = storage.GetValue(nameof(StartFrom), StartFrom);
    Offset = storage.GetValue(nameof(Offset), Offset);
}
public override IEnumerable<DataType> SupportedDataTypes => Enumerable.Empty<DataType>();

```

An important step in creating the utility is the implementation of the [BaseHydraTask.OnProcess](#), method, which implements the utility logic. In the case of implementing data backup, all the logic is inside [API](#):

```

[
protected override TimeSpan OnProcess()
{
    using (var service = CreateService())
    {
        var hasSecurities = false;
        this.AddInfoLog(LocalizedStrings.Str2306Params.Put(StartFrom));
        var startDate = StartFrom;
        var endDate = DateTime.Today - TimeSpan.FromDays(Offset);
        var allDates = startDate.Range(endDate, TimeSpan.FromDays(1)).ToArray();
        var pathEntry = ToEntry(new DirectoryInfo(Drive.Path));
        var workingSecurities = GetWorkingSecurities().ToArray();
        foreach (var date in allDates)
        {
            foreach (var security in workingSecurities)
            {
                hasSecurities = true;
                if (!CanProcess())
                    break;
                var dateEntry = new BackupEntry
                {
                    Name = LocalMarketDataDrive.GetDirName(date),
                    Parent = new BackupEntry
                    {
                        Parent = new BackupEntry
                        {
                            Name = security.Security.Id.Substring(0, 1),
                            Parent = pathEntry
                        },
                        Name = security.Security.Id,
                    }
                };
                var dataTypes = Drive.GetAvailableDataTypes(security.Security.ToSecurityId(), StorageFormat);
                foreach (var dataType in dataTypes)
                {
                    var storage = StorageRegistry.GetStorage(security.Security, dataType.MessageType,
dataType.Arg, Drive, StorageFormat);
                    var drive = storage.Drive;
                    var stream = drive.LoadStream(date);
                    if (stream == Stream.Null)
                        continue;
                    var entry = new BackupEntry
                    {
                        Name = LocalMarketDataDrive.GetFileName(dataType.MessageType, dataType.Arg,
StorageFormat), // + LocalMarketDataDrive.GetExtension(StorageFormats.Binary),
                        Parent = dateEntry,
                    };
                    var token = service.Upload(entry, stream, p => { });
                    if (token.IsCancellationRequested)
                    {
                        this.AddWarningLog(LocalizedStrings.Cancelling);
                        return TimeSpan.MaxValue;
                    }
                    this.AddInfoLog(LocalizedStrings.Str1580Params, GetPath(entry));
                }
            }
        }
        if (CanProcess())
        {
            StartFrom += TimeSpan.FromDays(1);
            this.SaveSettings();
        }
    }
    if (!hasSecurities)
    {
        this.AddWarningLog(LocalizedStrings.Str2292);
        return TimeSpan.MaxValue;
    }
    if (CanProcess())
        this.AddInfoLog(LocalizedStrings.Str2300);
}

```

```

        return base.OnProcess();
    }
}
private static string GetPath(BackupEntry entry)
{
    if (entry == null)
        return null;
    return GetPath(entry.Parent) + "/" + entry.Name;
}
private static BackupEntry ToEntry(DirectoryInfo di)
{
    // is a disk
    if (di.Parent == null)
        return null;
    return new BackupEntry
    {
        Name = di.Name,
        Parent = di.Parent != null ? ToEntry(di.Parent) : null
    };
}
public override bool CanTestConnect => true;
public override void TestConnect(Action<Exception> connectionChanged)
{
    if (connectionChanged == null)
        throw new ArgumentNullException(nameof(connectionChanged));
    try
    {
        using (var service = CreateService())
            service.Find(null, string.Empty);
        connectionChanged(null);
    }
    catch (Exception ex)
    {
        connectionChanged(ex);
    }
}
private IBackupService CreateService()
{
    switch (Service)
    {
        case BackupServices.AwsS3:
            return new AmazonS3Service(Address, ServiceRepo, Login, Password.To<string>());
        case BackupServices.AwsGlacier:
            return new AmazonGlacierService(Address, ServiceRepo, Login, Password.To<string>());
        case BackupServices.Yandex:
            return new YandexDiskService();
        default:
            throw new ArgumentOutOfRangeException();
    }
}
}
}

```

Hydra server

Hydra Server

Hydra Server is a special service that can broadcast data over the network for connecting external programs (for example, [Designer](#)).

Unlike [server mode](#), **Hydra Server** is a separate cross-platform program made as a console application and can be run on Windows or Linux servers.

Under Windows, the program can be registered as a Windows Service and launched at computer startup. For more details about [services](#).

Hydra Server uses the same settings as [Hydra](#). Therefore, for initial setup, it is necessary to run the [Hydra](#) program and then use the settings created by the **Hydra Server** program.

[image: Hydra server]

The program has a configuration file `appsettings.json`:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "Server": {
    "WebApiAddress": "api.stocksharp.com/v1/",
    "LogLevel": "Inherit",
    "AutoDownload": false,
    "CompanyPath": "",
    "AppDataPath": ""
  }
}
```

- **WebApiAddress** - the address of StockSharp WebAPI. Used for management via [Telegram](#).
- **LogLevel** - the logging level.
- **AutoDownload** - whether to enable automatic downloading of sources at startup.
- **CompanyPath** - if using the program as a Windows Service, you need to set the path like "C:\Users\%user_name%\Documents\StockSharp".
- **AppDataPath** - in case of moving the settings directory of [Hydra](#), a new path to the settings must be specified.

Telegram services

Telegram Services

For remote management of trading robots (e.g., when they're placed on virtual servers), integration with the popular messaging app Telegram has been implemented.

This integration works with both ready-made programs (such as [Designer](#)) and your own software developed using the [API](#).

Currently, two services are provided:

- [Alerts](#) - a service for receiving notifications from a trading robot.
- [Control Panel](#) - a service for controlling trading robots (starting, stopping) via a Telegram bot.

Authorization

Authorization

To activate [StockSharpBot](#), you need to launch it and proceed with the authorization so that the bot can recognize your login on the StockSharp website. For this, select the **/login** command from the menu:

[image: TelegramActivating]

The bot will ask you to visit your profile at <https://stocksharp.com/profile/> and copy the token from there.

[image: Profile]

Then, send it back to the bot in chat in the format **token: %your_token%**. If successful, the bot will address you by the name you used to register on the StockSharp website.

[image: TelegramActivated]

Alerts

Alerts

A service for sending messages from applications (such as [Designer](#) or your own custom program) to both private and public channels or groups in the Telegram messenger.

To configure:

1. Complete the [bot authorization process](#).
2. Create a channel or group (either private or public).
[image: Creating a Telegram Channel] [image: Choosing a Channel Type in Telegram]
3. Add the bot [StockSharpBot](#)
[image: Adding the Bot to Telegram]
4. Make it an administrator
[image: Setting the Bot as an Administrator in Telegram]
5. Necessary permissions for correct operation
[image: Permissions for the Telegram Bot]
6. Write the special word **activate** in the channel or group
[image: Activating the Channel in Telegram]
7. If successful, you will receive a response
[image: Channel Activation Confirmation in Telegram]

The channel you've created is now available for your strategies and trading robots:

- When using [Designer](#), click the channel list on the top panel:

[image: Designer Ribbon Channels]

In the window that appears, you will see lists of all the channels and groups where you've activated the bot:

[image: List of Channels in Telegram]

Pressing the Telegram icon button will send a test message. If it's received, it means everything has been set up correctly.

[image: Sending a Test Message from Designer]

In the free tariff, a line mentioning the StockSharp site is added. In paid tariffs, this line is removed.

If you have several output channels in Telegram and want to direct different strategies to separate channels, you can specify the channels for each strategy in the settings:

[image: Remote Settings in Designer]

- In other programs, settings are done similarly to [Designer](#). For example, in the [Hydra](#) program, you can configure error logging for market data download if [Hydra](#) is located on a server, and you need to promptly receive information about a non-working connection.
- In the case of [Shell](#) or [S#](#), you can view the code that integrates your strategies with the Telegram service.

Control panel

Control Panel

A service for managing trading strategies and robots via a Telegram bot.

For setup, go through the [bot authorization process](#) beforehand.

After that, the bot is ready for use. Next, for the bot to start seeing your strategies, you need to:

- When using the [Designer](#) program, enable Remote mode in the Cloud panel:

[image: DesignerRibbon.png]

All strategies that run in Live mode will automatically be transferred to the Telegram bot, and you will be able to control them from your phone.

In [StockSharpBot](#) select the /apps command to see a list of all programs:

[image: TelegramControlApps.png]

Once the desired program is selected, you can see the strategies and their controls:

[image: TelegramControlApp.png]

[image: TelegramControlStrategies.png]

[image: TelegramControlStrategy.png]

- When using [Shell](#), go to the Remote Manager panel and configure settings similar to [Designer](#).
- When using [Hydra](#), perform actions similar to [Designer](#). Integration with [Hydra](#) allows you to manage the downloading of market data and monitor quantitative statistics.

[image: TelegramHydra.png] *[image: TelegramHydraStat.png]*

- When using [S#](#), you can integrate using the code from [Shell](#). Thanks to [S#](#) being cross-platform, your robots can run on any operating system.

Runner

Runner is a special program for launching strategies written on the StockSharp platform. Features of the program include:

- Cross-platform - can be run on both Windows and Linux systems.
- Integrated with [Telegram](#), allowing remote control of trading activities.
- Consumes minimal computer resources due to the absence of a heavy graphical interface.
- Capable of launching strategies in the form of C# code, in the form of [Designer](#) schematics, and as ready-made DLL assemblies.
- Allows strategies to be run in live trading mode, as well as for testing on history.

Export from Designer

Export from Designer

Runner allows you to run strategies created in [Designer](#). This is the most convenient way to set up **Runner** as all configurations are done visually.

To export a strategy from [Designer](#):

- Select the desired strategy in the tree, right-click it, and choose the **Runner** menu item:
[image: Designer_Runner_1]
- In the window that appears, you need to select which types of connections need to be exported to **Runner**, as well as settings for managing the strategy via [Telegram](#):

[image: Designer_Runner_1]

The following files will be copied to the selected export directory:

- connector.json - a file containing connection settings
- params.json - a file containing strategy parameters
- start.bat - a bat file with a command line already written for quick **Runner** startup
- strategy.json - a file containing the strategy
- connector.json - a file containing connection settings
- telegram.json - a file containing settings for integration with [Telegram](#)

Command line

Command Line

Runner, being a console application, offers the ability to launch in different modes by specifying parameters on the command line. Launching the program without parameters will display a help message showing the available parameters:

[image: Runner_command_line_1]

Launching **Runner** for historical data testing:

```
b -s SmaStrategy.cs -h "C:\Storage" --hf 20200401 --ht 20200430 --sec SBER@TQBR -r json
```

Available parameters:

- -s - path to the strategy file (with cs, json, or dll extension).
- -t - (optional) if a dll file is selected, and the assembly contains more than one strategy class, it is necessary to specify the required type through this parameter.
- -h - path to the directory with historical data. Can be a network address in case of using server mode [server](#).
- --hl - (optional) login, used in server mode [server](#).
- --hp - (optional) password, used in server mode [server](#).
- --hf - start date for testing in YYYYMMDD format.
- --ht - end date for testing in YYYYMMDD format.
- -f - (optional) storage format (Binary or Csv).
- --sec - (optional) [instrument identifier](#).
- -r - (optional) format of the test result report (json, xml, csv).
- --tm - (optional) strategy timeout.
- --memory - (optional) maximum memory size (in megabytes).
- --cpu - (optional) processor mask.
- -l - (optional) logging level (Info, Debug, Error, Warning, Verbose).

Launching **Runner** for optimization:

```
o -s SmaStrategy.cs -h "C:\Storage" --hf 20200401 --ht 20200430 --sec SBER@TQBR -r json -p sma_optimization.json
```

All parameters from the historical testing mode, plus additional ones:

- -p - path to the parameters file.
- --ol - (optional) maximum number of iterations.
- --ob - (optional) number of strategies tested simultaneously.

Parameters file format:

```
[
```

```
{
  "Name": "SMA_80",
  "Value": "200,201"
},
{
  "Name": "SMA_30",
  "From": "40",
  "To": "50",
  "Step": "1"
},
{
  "Name": "Security",
  "Value": "SBER@TQBR,GAZP@TQBR"
}
]
```

Launching **Runner** for live trading:

```
1 -s SmaStrategy.cs -c connector.json --tg telegram.json
```

- -c - connection settings file.
- --tg - telegram integration settings file.

Connection setup

Connection Setup

Runner, in addition to [exporting settings through Designer](#), offers the ability to configure the program through its console interface. To do this, you need to run the program with the **setup** command:

```
stocksharp.studio.runner setup
```

A menu will appear:

[image: runner_setup_1]

By selecting the Connections item, the program will enter the connector setup mode:

[image: runner_setup_2]

Here you can either edit a previously saved connection or create a new one:

[image: runner_setup_3]

After selecting the required type of new connection, the program will move to the editing menu of its settings:

[image: runner_setup_4]

For [Binance](#), you need to enter its main settings:

[image: runner_setup_5]

[image: runner_setup_6]

[image: runner_setup_7]

To verify the correctness of the entered data, select **Check**:

[image: runner_setup_8]

The connection check will start:

[image: runner_setup_9]

In case of success, a message will be displayed:

[image: runner_setup_10]

After entering all settings and verifying them, you must press **Save**:

[image: runner_setup_11]

In the Data folder, a **connector.json** file will be created (if it was not created earlier), which will contain the saved settings.

To set up integration with [Telegram](#), select the menu item:

[image: runner_telegram_1]

And authenticate by a convenient method:

[image: runner_telegram_2]

For authentication by token, enter the token from <https://stocksharp.ru/profile/>:

[image: Profile]

In case of success, the program will display the available Telegram operation options:

[image: runner_telegram_3]

Integration with Visual Studio

Integration with Visual Studio

Runner can be used as a means of debugging strategies similarly to [Designer](#). This is convenient if the strategy is planned to be run only in **Runner**. Otherwise, it is more convenient to launch and work with the strategy inside the [Designer](#) program.

To set up the debugging process, the following steps need to be taken:

1. Right-click on the trading strategy project and select **Properties** from the context menu:

[image: runner_debug_00]

In the tab that appears, find the **Debug** item, select the **General** section and click **Open debug launch profiles UI**.

2. Next, in the window that opens, create a new debugging profile with the launch of an external program:

[image: runner_debug_01]

3. Enter the full path to **Runner**, and specify the command line parameters for launching. More about [Runner's command line](#).

[image: runner_debug_02]

Command line arguments for the example:

```
1 -s "${TargetPath}" -c "C:\StockSharp\Runner\Data\connection.json" --sec BTCUSDT_PERPETUAL@BNB --pf
Binance_-298049655_Futures
```

`$(TargetPath)` - is a special **Visual Studio** macro that is automatically replaced with the path to the compiled DLL with the strategy during debugging launch.

4. Close the project settings window, and start the project debugging (for example, through F5). The **Runner** program window will appear, displaying the trading connection process:

[image: runner_debug_03]

5. When setting breakpoints, the program execution will stop upon reaching them. For example, to debug the trading logic when a new candle appears:

[image: runner_debug_04]

API Documentation

Overview

StockSharp API, also known as S# API, is a comprehensive Software Development Kit (SDK) designed for creating trading applications similar to [Designer](#), [Terminal](#), and others. This SDK serves as the architectural foundation for numerous significant projects within the trading community.

Features

- **Strategy Scripting:** StockSharp API provides a robust scripting mechanism for writing and implementing trading strategies directly in [Designer](#). Users can develop, test, and deploy trading algorithms using C#, F# or Python.
- **Analytical Tools:** The API is integrated with Hydra, a platform for detailed [market data analysis](#). It offers extensive support for data manipulation and storage, enabling complex analytical operations.
- **Custom Application Development:** Besides scripting within existing applications, StockSharp API allows developers to create custom [independent trading solutions](#). This feature is crucial for users who require tailor-made functionalities not typically available in standard trading applications.
- **Connectors and Graphical Controls:** The API includes a wide range of [connectors](#) for real-time market data integration from various exchanges. Additionally, it supports the development of customizable [graphical user interfaces](#), making it versatile for creating professional trading platforms.

Architecture

StockSharp API is built with a focus on modularity and [extensibility](#). It enables developers to extend its capabilities through plugins and additional modules without altering the core system. This modular architecture makes it an ideal choice for developers looking to build scalable and maintainable trading applications.

Open Source

The core of the StockSharp API is developed under an open-source paradigm, allowing the community to contribute and ensuring transparency. The source code is available on GitHub, providing both novice and experienced developers with the opportunity to study, modify, and enhance the system according to their specific needs.

GitHub Repository

The official StockSharp API source code can be found at the GitHub repository:

[StockSharp GitHub Repository](#)

Setup

Setting Up the Environment

Requirements

To work with StockSharp you need:

- **.NET 10** (SDK and Runtime) — [download](#)
- **IDE** — Visual Studio 2022+, JetBrains Rider, or VS Code
- **NuGet** — package manager (built into Visual Studio and Rider)

Verify the SDK is installed:

```
dotnet --version
```

Creating a Project

Visual Studio 2022+

1. **File → New → Project**
2. Select **Console App** or **WPF Application** template
3. Set the target framework to **.NET 10**

JetBrains Rider

1. **File → New Solution**
2. Select **.NET / .NET Core → Console Application**
3. Set **Target Framework: net10.0**

Command Line (CLI)

```
# Console application
dotnet new console -n MyTradingApp --framework net10.0
cd MyTradingApp

# WPF application (Windows only)
dotnet new wpf -n MyTradingGui --framework net10.0-windows
```

NuGet Package Catalog

StockSharp is distributed via NuGet. Below is the complete package catalog organized by category.

Core

Package	Description
StockSharp.Messages	Base messages and contracts. Foundation of the entire framework
StockSharp.BusinessEntities	Trading entities: Security, Order, Trade, Portfolio,

	etc.
StockSharp.Algo	Core algorithmic trading, Connector, subscriptions, candles
StockSharp.Configuration	Configuration management, connection settings
StockSharp.Localization	Localization system (English by default)

Strategies and Indicators

Package	Description
StockSharp.Algo.Strategies	Strategy framework – base Strategy class, positions, PnL
StockSharp.Algo.Indicators	100+ technical indicators (SMA, EMA, RSI, MACD, Bollinger, etc.)

Testing

Package	Description
StockSharp.Algo.Testing	Backtesting on historical data, trade emulation

Storage and Data

Package	Description
StockSharp.Algo.Export	Market data export (CSV, Excel, JSON, etc.)
StockSharp.Algo.Import	Data import from external formats

Analytics and Computation

Package	Description
StockSharp.Algo.Analytics	Analytics scripts interfaces
StockSharp.Algo.Compilation	Runtime code compilation
StockSharp.Algo.Gpu	GPU-accelerated indicator calculations (CUDA via ILGPU)

GUI Components (Windows Only)

Package	Description
StockSharp.Xaml	WPF controls: instrument, portfolio, and order tables
StockSharp.Xaml.Charting	Candlestick charts, indicators, equity curves
StockSharp.Charting.Interfaces	Chart component interfaces
StockSharp.Alerts.Interfaces	Alert system interfaces
StockSharp.Diagram.Core	Visual strategy designer core

Connectors (Exchanges and Brokers)

Each connector is a separate NuGet package. Main connectors:

Package	Exchange/Broker
StockSharp.Binance	Binance
StockSharp.InteractiveBrokers	Interactive Brokers
StockSharp.Fix	FIX protocol (universal)
StockSharp.Connectors.Tinkoff	Tinkoff Invest
StockSharp.Connectors.Coinbase	Coinbase
StockSharp.Connectors.BitStamp	Bitstamp
StockSharp.Connectors.Bittrex	Bittrex
StockSharp.Finam	Finam

For the full list of connectors, see the [Connectors](#) section. Some connectors are only available through the [private NuGet server](#).

Localization

The base StockSharp.Localization package includes English. Additional languages are installed as separate packages:

Package	Language
StockSharp.Localization.ru	Russian
StockSharp.Localization.zh	Chinese
StockSharp.Localization.de	German
StockSharp.Localization.es	Spanish
StockSharp.Localization.ja	Japanese
StockSharp.Localization.ko	Korean
StockSharp.Localization.All	All languages (meta-package)

Also available: ar, bn, ca, cs, da, el, fa, fi, fr, he, hi, hu, it, jv, ms, my, nl, no, pa, pl, pt, ro, sk, sr, sv, ta, th, tl, tr, uk, uz, vi.

Installing Packages

Via CLI (Recommended)

```
# Core packages
dotnet add package StockSharp.Algo
dotnet add package StockSharp.Algo.Strategies

# Connector (example – Binance)
dotnet add package StockSharp.Binance

# Indicators
dotnet add package StockSharp.Algo.Indicators

# Backtesting
dotnet add package StockSharp.Algo.Testing
```

```
# Localization (Russian)
dotnet add package StockSharp.Localization.ru
```

Via Visual Studio

1. Right-click on the project → **Manage NuGet Packages...**
2. Search for StockSharp
3. Select the desired package → **Install**

All dependencies are installed automatically.

Via JetBrains Rider

1. Right-click on the project → **Manage NuGet Packages**
2. Search for StockSharp
3. Select the package → **Install**

Via Package Manager Console (Visual Studio)

```
Install-Package StockSharp.Algo
Install-Package StockSharp.Binance
Install-Package StockSharp.Algo.Strategies
```

Private NuGet Server

Some components (crypto connectors, etc.) are only available through the private NuGet server for registered users.

Method 1: Authentication via Token in URL

1. Register on the StockSharp website.
2. Copy the token from your [personal account](#).
3. Add the package source:

CLI:

```
dotnet nuget add source "https://nuget.stocksharp.com/{YOUR_TOKEN}/v3/index.json" --name
StockSharpPrivate
```

Visual Studio: open **Tools** → **Options** → **NuGet Package Manager** → **Package Sources** and add a new source with the URL `https://nuget.stocksharp.com/{YOUR_TOKEN}/v3/index.json`.

Rider: open **Settings** → **Build, Execution, Deployment** → **NuGet** → **Sources** and add the source.

Method 2: Authentication via Username and Password

1. Add a package source with the URL `https://nuget.stocksharp.com/x/v3/index.json`
2. When you try to use this source, a login prompt will appear
3. Enter your StockSharp account credentials (or x as the username and your token as the password)

CLI:

```
dotnet nuget add source "https://nuget.stocksharp.com/x/v3/index.json" --name StockSharpPrivate --username YOUR_LOGIN --password YOUR_PASSWORD --store-password-in-clear-text
```

To reset saved credentials on Windows, open **Control Panel → User Accounts → Credential Manager** and delete entries related to `nuget.stocksharp.com`.

Updating Packages

CLI

```
# Check for available updates
dotnet list package --outdated

# Update a specific package
dotnet add package StockSharp.Algo
```

Visual Studio

1. **Manage NuGet Packages...** → **Updates** tab
2. Select packages → **Update**

Rider

1. **Manage NuGet Packages** → **Upgrades** tab
2. Select packages → **Upgrade**

Troubleshooting

Version Mismatch

All StockSharp packages must be the same version. If you encounter type or method errors, ensure all StockSharp.* packages are updated to the same version.

Package Not Found on Public NuGet

Some connectors are only available through the [private server](#). Make sure the correct source is added.

GUI Issues on Non-Windows

GUI components (StockSharp.Xaml, StockSharp.Xaml.Charting) work only on Windows. On Linux/macOS, use console applications without GUI packages.

Package Restore Errors

```
# Clear NuGet cache
dotnet nuget locals all --clear

# Retry restore
dotnet restore
```

Project File Structure (.csproj)

Minimal (Console Trading Bot)


```

<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net10.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="StockSharp.Algo" Version="*" />
    <PackageReference Include="StockSharp.Binance" Version="*" />
  </ItemGroup>
</Project>

```

Extended (Strategy with Indicators and Testing)

```

<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net10.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <!-- Core -->
    <PackageReference Include="StockSharp.Algo" Version="*" />
    <PackageReference Include="StockSharp.Configuration" Version="*" />

    <!-- Strategies and indicators -->
    <PackageReference Include="StockSharp.Algo.Strategies" Version="*" />
    <PackageReference Include="StockSharp.Algo.Indicators" Version="*" />

    <!-- Connector -->
    <PackageReference Include="StockSharp.Binance" Version="*" />

    <!-- Backtesting -->
    <PackageReference Include="StockSharp.Algo.Testing" Version="*" />

    <!-- Localization -->
    <PackageReference Include="StockSharp.Localization.ru" Version="*" />
  </ItemGroup>
</Project>

```

WPF Application with Charts

```

<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net10.0-windows</TargetFramework>
    <UseWPF>true</UseWPF>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="StockSharp.Algo" Version="*" />
    <PackageReference Include="StockSharp.Algo.Strategies" Version="*" />
    <PackageReference Include="StockSharp.Xaml.Charting" Version="*" />
    <PackageReference Include="StockSharp.Binance" Version="*" />
  </ItemGroup>
</Project>

```

Verifying the Installation

Create a minimal application:

```

using StockSharp.Algo;
using StockSharp.BusinessEntities;

```

```
Console.WriteLine("StockSharp successfully configured!");  
  
var connector = new Connector();  
Console.WriteLine($"Connector created: {connector}");
```

```
dotnet run
```

Examples

Ready-made examples of StockSharp usage are available in the [Samples/](#) directory of the repository. They cover connecting to exchanges, subscribing to data, building candles, indicators, strategies, and testing.

AI-Assisted Development

Modern AI tools can significantly accelerate the development of trading strategies and connectors using StockSharp. Instead of writing code from scratch, you describe the task in natural language and the AI generates working code using the current API.

Why Use AI

- **Quick start** — create a working strategy prototype in minutes, not hours
- **Learn the API** — the AI will show you the correct StockSharp methods and patterns
- **Debugging** — describe an error in plain language, get a fix
- **Refactoring** — improve existing code while preserving logic

Suitable Tools

Tool	Type	Features
Claude Code	CLI agent	Works with the full project, executes commands, runs tests
OpenAI Codex	Cloud agent	Autonomous task execution in sandbox environment, GitHub integration
Cursor	IDE	Built-in AI assistant, auto-completion, project context awareness
GitHub Copilot	IDE plugin	Code completion in Visual Studio, VS Code, Rider
JetBrains AI	Built into Rider	Native integration with Rider and the debugger

General Principles

1. Provide Context

The more context the AI has, the more accurate the result. Specify:

- Which StockSharp packages are being used
- Which connector (exchange/broker)
- Strategy type (trend-following, arbitrage, scalping)
- Constraints (long only, maximum position size, etc.)

2. Use CLAUDE.md / .cursorrules

Create a project rules file in the repository root:

```
# Project Rules
```

- Using StockSharp 5.x API
- Target framework: .NET 10
- Strategies inherit from Strategy
- Connectors implement MessageAdapter
- All subscriptions via Connector.Subscribe()
- Logging via this.AddInfoLog() / this.AddErrorLog()

3. Iterative Approach

1. Describe the task in broad terms
2. Get the first version of the code
3. Refine requirements and ask for improvements
4. Review the code: compilation, logic, error handling
5. Test on historical data

4. Verify the Result

AI can use outdated methods or invent non-existent APIs. Always verify:

- Whether the code compiles
- Whether the used classes and methods exist
- Whether subscription and event signatures are correct
- Whether error handling is proper

Sections

- [Writing a Strategy with AI](#) — step-by-step guide to creating a trading strategy
- [Writing a Connector with AI](#) — step-by-step guide to creating an exchange connector

Writing Strategies

Writing a Strategy with AI

A step-by-step guide to creating a StockSharp trading strategy using AI tools.

Preparation

1. Install an AI Tool

Choose one of the available tools:

- **Claude Code** — `npm install -g @anthropic-ai/claude-code` (requires Node.js)
- **Cursor** — download from cursor.com
- **GitHub Copilot** — install the plugin for your IDE

2. Create a Project

```
dotnet new console -n MyStrategy --framework net10.0
cd MyStrategy
dotnet add package StockSharp.Algo
dotnet add package StockSharp.Algo.Strategies
dotnet add package StockSharp.Algo.Indicators
dotnet add package StockSharp.Algo.Testing
dotnet add package StockSharp.Binance
```

3. Prepare Context

Create a `CLAUDE.md` (or `.cursorrules`) file in the project root:

```
# Project Rules

- Framework: StockSharp 5.x, .NET 10
- Strategies inherit from Strategy class
- Subscribe to candles via Connector.Subscribe(subscription)
- Register orders via RegisterOrder(order)
- Logging: this.AddInfoLog(), this.AddWarningLog(), this.AddErrorLog()
- Indicators: create via new and call indicator.Process(candle)
- Always handle connector.Error and strategy errors
```

Step-by-Step Example: SMA Strategy

Step 1: Describe the Task to the AI

Example prompt:

```
Create a trading strategy using StockSharp that:
- Inherits from Strategy
- Uses two simple moving averages (SMA): fast (period 10) and slow (period 30)
- When the fast SMA crosses above the slow SMA – buy
- When the fast SMA crosses below the slow SMA – sell
- Position size: 1 lot
- Uses 5-minute candles
- Subscribes to candles in OnStarted()
- Processes candles via subscription rules
```

Step 2: Review the Generated Code

The AI will generate something like:

```
using StockSharp.Algo;
using StockSharp.Algo.Indicators;
using StockSharp.Algo.Strategies;
using StockSharp.BusinessEntities;
using StockSharp.Messages;

public class SmaCrossStrategy : Strategy
{
    private readonly StrategyParam<int> _fastPeriod;
    private readonly StrategyParam<int> _slowPeriod;
    private readonly StrategyParam<DataType> _candleType;

    private SimpleMovingAverage _fastSma;
    private SimpleMovingAverage _slowSma;

    private bool _prevFastAbove;
    private bool _isFirstCandle = true;

    public SmaCrossStrategy()
    {
        _fastPeriod = Param(nameof(FastPeriod), 10);
        _slowPeriod = Param(nameof(SlowPeriod), 30);
        _candleType = Param(nameof(CandleType), DataType.TimeFrame(TimeSpan.FromMinutes(5)));
    }

    public int FastPeriod
    {
        get => _fastPeriod.Value;
        set => _fastPeriod.Value = value;
    }

    public int SlowPeriod
    {
        get => _slowPeriod.Value;
        set => _slowPeriod.Value = value;
    }

    public DataType CandleType
    {
        get => _candleType.Value;
        set => _candleType.Value = value;
    }

    protected override void OnStarted(DateTimeOffset time)
    {
        base.OnStarted(time);

        _fastSma = new SimpleMovingAverage { Length = FastPeriod };
        _slowSma = new SimpleMovingAverage { Length = SlowPeriod };
        _isFirstCandle = true;

        var subscription = new Subscription(CandleType, Security);

        this
            .WhenCandlesFinished(subscription)
            .Do(ProcessCandle)
            .Apply(this);

        Subscribe(subscription);
    }

    private void ProcessCandle(ICandleMessage candle)
    {
        var fastValue = _fastSma.Process(candle);
```

```

var slowValue = _slowSma.Process(candle);

if (!_fastSma.IsFormed || !_slowSma.IsFormed)
    return;

var fastAbove = fastValue.GetValue<decimal>() > slowValue.GetValue<decimal>();

if (!_isFirstCandle)
{
    if (fastAbove && !_prevFastAbove)
    {
        // Fast SMA crossed above slow – buy
        if (Position <= 0)
            RegisterOrder(this.BuyAtMarket(Volume));
    }
    else if (!fastAbove && _prevFastAbove)
    {
        // Fast SMA crossed below slow – sell
        if (Position >= 0)
            RegisterOrder(this.SellAtMarket(Volume));
    }
}

_prevFastAbove = fastAbove;
_isFirstCandle = false;
}
}

```

Step 3: What to Check

Go through this checklist:

- **Inheritance:** class inherits from Strategy ✓
- **Parameters:** uses StrategyParam<T> for optimization ✓
- **Candle subscription:** via Subscribe(new Subscription(...)) ✓
- **Candle processing:** via WhenCandlesFinished rule ✓
- **IsFormed check:** indicators are checked for readiness ✓
- **Orders:** via RegisterOrder() with BuyAtMarket / SellAtMarket ✓
- **Position:** Position is checked before placing orders ✓

Step 4: Ask the AI to Add Backtesting

Add backtesting code for this strategy using historical data. Use HistoryEmulationConnector, load data from local storage, and output summary statistics (PnL, trade count, max drawdown).

Example Prompts

Bollinger Bands Strategy

Create a StockSharp strategy that trades using Bollinger Bands:

- Buy when price touches the lower band
- Sell when price touches the upper band
- Period 20, multiplier 2.0
- Stop-loss: 1% from entry price
- Take-profit: 2% from entry price
- Use StrategyParam for all parameters

Arbitrage Strategy

Create a pairs arbitrage strategy on StockSharp:

- Two instruments (specified via parameters)
- Calculate the spread between prices
- Enter when spread deviates by 2 standard deviations
- Exit when spread returns to the mean
- Volume neutralization (equal positions in monetary terms)

Order Book Scalping

Create a scalping strategy on StockSharp:

- Subscribe to order book (MarketDepth) via Subscribe
- Analyze bid/ask imbalance
- Enter on strong imbalance (> 3:1)
- Quick exit on take-profit (5 ticks)
- Stop-loss: 3 ticks
- Maximum 1 position at a time

Common AI Mistakes

1. Outdated Events

Wrong (old API):

```
connector.NewSecurities += securities => { ... };  
connector.CandleSeriesProcessing += (series, candle) => { ... };
```

Correct (current API):

```
// Use subscriptions  
var subscription = new Subscription(DataType.TimeFrame(TimeSpan.FromMinutes(5)), security);  
connector.Subscribe(subscription);
```

2. Creating Orders Without Helpers

Wrong:

```
var order = new Order  
{  
    Security = Security,  
    Portfolio = Portfolio,  
    Side = Sides.Buy,  
    Type = OrderTypes.Market,  
    Volume = 1,  
};
```

Correct (using strategy helpers):

```
RegisterOrder(this.BuyAtMarket(Volume));  
// or  
RegisterOrder(this.SellAtLimit(price, Volume));
```

3. Missing IsFormed Check

Wrong:


```
var value = _sma.Process(candle);  
// Using value immediately – may not be ready
```

Correct:

```
var value = _sma.Process(candle);  
if (!_sma.IsFormed)  
    return;
```

Tips

1. **Give the AI documentation** – point it to doc.stocksharp.com or copy code examples from Samples/
2. **Use CLAUDE.md** – a project rules file greatly reduces the number of mistakes
3. **Start simple** – create a basic strategy first, then add filters and risk management
4. **Test on history** – always run a backtest before live trading
5. **Clone the repository** – if the AI has access to StockSharp sources, it will use the API more accurately

Writing Connectors

Writing a Connector with AI

A step-by-step guide to creating an exchange connector for StockSharp using AI tools.

Preparation

1. Study the Exchange API

Before starting, prepare:

- REST/WebSocket API documentation for the exchange
- Test API keys (sandbox/testnet)
- List of supported data types (candles, order book, ticks, trades)
- List of supported order types (limit, market, stop)

2. Create a Project

```
dotnet new classlib -n StockSharp.MyExchange --framework net10.0
cd StockSharp.MyExchange
dotnet add package StockSharp.Messages
dotnet add package StockSharp.Algo
```

3. Prepare Context for the AI

Create a CLAUDE.md file:

```
# Project Rules – Exchange Connector

- Framework: StockSharp 5.x, .NET 10
- Connector is implemented as a MessageAdapter
- Inherit from AsyncMessageAdapter for async/await
- All HTTP requests via HttpClient with CancellationToken
- WebSocket subscriptions via native client or ClientWebSocket
- Type mapping: exchange types → StockSharp Messages
- Error handling: SendOutError() for connection errors
- All strings in localization resources (or at least const)
```

Connector Architecture

A connector in StockSharp is a MessageAdapter that:

1. Receives incoming messages (requests) from the core
2. Processes them (calls the exchange API)
3. Sends back response messages (results)

```
StockSharp Core → [Message] → MessageAdapter → [HTTP/WS] → Exchange
Exchange → [HTTP/WS] → MessageAdapter → [Message] → StockSharp Core
```

Step-by-Step Example

Step 1: Basic Adapter Structure

Prompt:

```
Create a basic MessageAdapter for a cryptocurrency exchange connector
called MyExchange using StockSharp:
- Inherit from AsyncMessageAdapter
- Implement connect/disconnect (ConnectMessage, DisconnectMessage)
- Add settings: ApiKey, Secret, demo mode
- Use HttpClient for REST API
- Base API URL: https://api.myexchange.com/v1
```

Step 2: Instrument Lookup

Prompt:

```
Add SecurityLookupMessage handling to the adapter:
- Request GET /api/v1/symbols returns a JSON list of instruments
- Each instrument has: symbol, baseAsset, quoteAsset,
  minQty, maxQty, tickSize, status
- Mapping: symbol → SecurityId, baseAsset/quoteAsset → name,
  tickSize → SecurityMessage.PriceStep
- Send SecurityMessage for each instrument
- Send SubscriptionFinishedMessage at the end
```

Step 3: Market Data

Prompt:

```
Add market data subscription to the adapter:

1. Candles (MarketDataTypes.CandleTimeFrame):
  - REST: GET /api/v1/klines?symbol={}&interval={}&limit=1000
  - WebSocket: subscribe to channel kline_{symbol}_{interval}
  - Interval mapping: 1m, 5m, 15m, 1h, 4h, 1d

2. Order book (MarketDataTypes.MarketDepth):
  - WebSocket: subscribe to channel depth_{symbol}
  - Parse bids/asks into QuoteChangeMessage

3. Ticks (MarketDataTypes.Trades):
  - WebSocket: subscribe to channel trades_{symbol}
  - Parse into ExecutionMessage with ExecutionTypes.Tick
```

Step 4: Trading Operations

Prompt:

```
Add trading operation support to the adapter:

1. Order registration (OrderRegisterMessage):
  - POST /api/v1/order with params: symbol, side, type, quantity, price
  - Return ExecutionMessage with ExecutionTypes.Transaction

2. Order cancellation (OrderCancelMessage):
  - DELETE /api/v1/order/{orderId}
  - Return ExecutionMessage with status OrderStates.Done

3. Portfolio retrieval (PortfolioLookupMessage):
  - GET /api/v1/account
  - Parse balances into PositionChangeMessage

4. WebSocket for order updates:
```

- Channel orders_{listenKey}
- Parse order status updates

Step 5: Code Review

After generating each step, ask the AI:

Review the generated adapter for StockSharp API compliance:

1. Are all message types handled?
2. Is CancellationToken used correctly?
3. Is HTTP error handling in place?
4. Are SubscriptionFinishedMessage sent after completion?
5. Does WebSocket reconnection work on disconnect?

Key Implementation Details

MessageAdapter — What to Handle

Incoming Message	Action	Response Message
ConnectMessage	Connect to API	ConnectMessage (response)
DisconnectMessage	Disconnect	DisconnectMessage (response)
SecurityLookupMessage	Request instruments	SecurityMessage × N
MarketDataMessage (subscribe)	Subscribe to data	SubscriptionResponseMessage
OrderRegisterMessage	Create order	ExecutionMessage
OrderCancelMessage	Cancel order	ExecutionMessage
PortfolioLookupMessage	Request portfolios	PortfolioMessage, PositionChangeMessage

Async Pattern

```
public class MyExchangeAdapter : AsyncMessageAdapter
{
    private HttpClient _httpClient;

    protected override ValueTask OnConnectAsync(ConnectMessage msg, CancellationToken token)
    {
        _httpClient = new HttpClient();
        _httpClient.BaseAddress = new Uri("https://api.myexchange.com/v1/");
        _httpClient.DefaultRequestHeaders.Add("X-API-KEY", Key.To<string>());

        SendOutMessage(new ConnectMessage());
        return default;
    }

    protected override ValueTask OnSecurityLookupAsync(SecurityLookupMessage msg, CancellationToken token)
    {
        // ... request instruments
    }

    // ... other methods
}
```

Request Signing

Most exchanges require HMAC signing for private requests:

```
private string SignRequest(string payload)
{
    using var hmac = new HMACSHA256(Encoding.UTF8.GetBytes(Secret.To<string>()));
    var hash = hmac.ComputeHash(Encoding.UTF8.GetBytes(payload));
    return Convert.ToHexString(hash).ToLowerInvariant();
}
```

Connector Review Checklist

Connection

- Markdig.Extensions.TaskLists.TaskList Connect/disconnect works correctly
- Markdig.Extensions.TaskLists.TaskList Authentication errors are handled
- Markdig.Extensions.TaskLists.TaskList Reconnection works on disconnect

Instruments

- Markdig.Extensions.TaskLists.TaskList Instrument list loads successfully
- Markdig.Extensions.TaskLists.TaskList Correctly mapped: SecurityId, PriceStep, VolumeStep
- Markdig.Extensions.TaskLists.TaskList SubscriptionFinishedMessage is sent

Market Data

- Markdig.Extensions.TaskLists.TaskList Candles: history loading + subscription to new ones
- Markdig.Extensions.TaskLists.TaskList Order book: correct depth, updates
- Markdig.Extensions.TaskLists.TaskList Ticks: correct time, volume, direction

Trading

- Markdig.Extensions.TaskLists.TaskList Limit orders: creation, cancellation
- Markdig.Extensions.TaskLists.TaskList Market orders: creation
- Markdig.Extensions.TaskLists.TaskList Order status updates
- Markdig.Extensions.TaskLists.TaskList Portfolio balance updates

General

- Markdig.Extensions.TaskLists.TaskList All CancellationToken are propagated
- Markdig.Extensions.TaskLists.TaskList Errors are logged via SendOutError()
- Markdig.Extensions.TaskLists.TaskList No resource leaks (WebSocket, HttpClient)
- Markdig.Extensions.TaskLists.TaskList Compiles without errors or warnings

Example Prompts

Adding a New Data Type

```
Add Level 1 data support (BestBid/BestAsk) to my connector:
- WebSocket channel: ticker_{symbol}
- Parse bid, ask, last, volume
```

- Send `Level1ChangeMessage` with fields:
`Level1Fields.BestBidPrice`, `Level1Fields.BestAskPrice`,
`Level1Fields.LastTradePrice`, `Level1Fields.Volume`

Handling Rate Limits

- Add rate limit handling to my connector:
- API returns headers `X-RateLimit-Remaining` and `X-RateLimit-Reset`
 - When limit is reached: wait until `Reset`, log a warning
 - Use `SemaphoreSlim` to limit concurrent requests

Tips

1. **Start read-only** — first implement connection, instruments, and market data. Add trading operations after verification
2. **Use sandbox** — test on the exchange's test environment
3. **Reference existing connectors** — give the AI code from an existing StockSharp connector as a reference
4. **Log everything** — detailed logs are invaluable when debugging a connector
5. **Handle edge cases** — reconnection, instrument changes, non-standard order types

Examples

S# Examples

Examples of using the StockSharp API library can be found in our GitHub repository at the following path: [StockSharp Samples](#).

These examples provide ready-made solutions and demonstrations of various capabilities of the StockSharp API library. You will find useful code examples that will help you quickly familiarize yourself with using our platform.

Documentation

The code provided in the documentation is taken from these examples. You can explore them in more detail by following the link above. Use these examples as a basis for creating your own solutions and for understanding how to work with various aspects of the StockSharp API library.

We recommend carefully studying the examples and documentation to make the most effective use of our platform's capabilities.

If you have any questions, you can always seek help in our [Telegram chat](#).

Connectors

Connectors

For working with exchanges and data sources in [S#](#), it is recommended to use the base class [Connector](#).

Let's look at working with [Connector](#). The source code of the example can be found in the Samples/01_Basic/01_ConnectAndDownloadInstruments project.

[image: multiconnection main]

Create an instance of the [Connector](#) class:

```
...
public Connector Connector;
...
public MainWindow()
{
    InitializeComponent();
    Connector = new Connector();
    InitConnector();
}
```

For configuring [Connector](#), the **API** has a special graphical interface that allows you to configure multiple connections simultaneously. How to use it is described in the section [Graphical Configuration](#).

```
...
private const string _connectorFile = "ConnectorFile.json";
...
private void Setting_Click(object sender, RoutedEventArgs e)
{
    if (Connector.Configure(this))
    {
        Connector.Save().Serialize(_connectorFile);
    }
}
```

[image: API GUI ConnectorWindow]

Similarly, you can add connections directly from code (without graphical windows) by using the extension method [TraderHelper.AddAdapter<TAdapter>\(StockSharp.Algo.Connector connector, System.Action<TAdapter> init \)](#):

```
...
// Add adapter for connecting to Binance
connector.AddAdapter<BinanceMessageAdapter>(a =>
{
    a.Key = "<Your API Key>";
    a.Secret = "<Your Secret Key>";
});

// Add RSS for news
connector.AddAdapter<RssMessageAdapter>(a =>
{
    a.Address = "https://news-source.com/feed";
    a.IsEnabled = true;
});
```

You can add an unlimited number of connections to a single [Connector](#) object. Therefore, you can connect to multiple exchanges and brokers simultaneously from the program.

In the *InitConnector* method, we set the required event handlers for [IConnector](#):


```

private void InitConnector()
{
    // Subscribe to successful connection event
    Connector.Connected += () =>
    {
        this.GuiAsync(() => ChangeConnectStatus(true));
    };

    // Subscribe to connection error event
    Connector.ConnectionError += error => this.GuiAsync(() =>
    {
        ChangeConnectStatus(false);
        MessageBox.Show(this, error.ToString(), LocalizedStrings.ErrorConnection);
    });

    // Subscribe to disconnection event
    Connector.Disconnected += () => this.GuiAsync(() => ChangeConnectStatus(false));

    // Subscribe to error event
    Connector.Error += error =>
        this.GuiAsync(() => MessageBox.Show(this, error.ToString(), LocalizedStrings.Str2955));

    // Subscribe to market data subscription failure event
    Connector.SubscriptionFailed += (subscription, error) =>
        this.GuiAsync(() => MessageBox.Show(this, error.ToString(),
            LocalizedStrings.Str2956Params.Put(subscription.DataType, subscription.SecurityId)));

    // Subscriptions for data reception

    // Instruments
    Connector.SecurityReceived += (sub, security) =>
        _securitiesWindow.SecurityPicker.Securities.Add(security);

    // Tick trades
    Connector.TickTradeReceived += (sub, trade) => _tradesWindow.TradeGrid.Trades.TryAdd(trade);

    // Orders
    Connector.OrderReceived += (sub, order) => _ordersWindow.OrderGrid.Orders.TryAdd(order);

    // Own trades
    Connector.OwnTradeReceived += (sub, trade) => _myTradesWindow.TradeGrid.Trades.TryAdd(trade);

    // Positions
    Connector.PositionReceived += (sub, position) =>
        _portfoliosWindow.PortfolioGrid.Positions.TryAdd(position);

    // Order registration failures
    Connector.OrderRegisterFailReceived += (sub, fail) =>
        _ordersWindow.OrderGrid.AddRegistrationFail(fail);

    // Order cancellation failures
    Connector.OrderCancelFailReceived += (sub, fail) => OrderFailed(fail);

    // Set market data provider
    _securitiesWindow.SecurityPicker.MarketDataProvider = Connector;

    try
    {
        if (File.Exists(_connectorFile))
        {
            var ctx = new ContinueOnExceptionContext();
            ctx.Error += ex => ex.LogError();
            using (new Scope<ContinueOnExceptionContext>(ctx))
            {
                Connector.Load(_connectorFile.Deserialize<SettingsStorage>());
            }
        }
    }
    catch
    {
    }
}

```

```
ConfigManager.RegisterService<IExchangeInfoProvider>(new InMemoryExchangeInfoProvider());

// Register adapter provider for graphical configuration
ConfigManager.RegisterService<IMessageAdapterProvider>(
    new FullInMemoryMessageAdapterProvider(Connector.Adapter.InnerAdapters));
}
```

How to save and load settings for [Connector](#) to a file can be found in the section [Saving and Loading Settings](#).

Information about creating your own [Connector](#) can be found in the section [Creating Your Own Connector](#).

Order placement is described in the sections [Orders](#), [Creating a New Order](#), [Creating a New Stop Order](#).

See Also

[Graphical Configuration](#)

Graphical configuration

Graphical configuration

See [Connection settings window](#).

Alpaca

Alpaca - financial broker.

Before you start writing trading robots for the current trading platform, it is recommended to read the links in the [Connectors](#).

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Alpaca

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the broker and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Alpaca

Graphical configuration Alpaca

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Alpaca]

- **Key** - Key.
- **Secret** - Secret.
- **Demo** - Sandbox mode.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Alpaca

Adapter initialization Alpaca

The code below demonstrates how to initialize the [AlpacaMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new AlpacaMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),

    // uncomment for sandbox mode
    //IsDemo = true,
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

AlphaVantage

Alpha Vantage - Market data provider of realtime and historical data on stocks, forex (FX), and digital/crypto currencies.

Before you start writing trading robots for the current trading platform, it is recommended to read the links in the [Connectors](#).

| Connector supports only market-data feed, transactions are not supported.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration AlphaVantage

Configuration AlphaVantage

To work with a connector, you have to specify the **Token**. **Token** are provided by the provider. To get the API access it is recommended to contact the provider.

Graphical configuration AlphaVantage

Graphical configuration AlphaVantage

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings AlphaVantage]

- **Token** - Token.
- **Heart beat** - Server check interval for track the connection alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization AlphaVantage

Adapter initialization AlphaVantage

The code below demonstrates how to initialize the [AlphaVantageMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();  
...  
var messageAdapter = new AlphaVantageMessageAdapter(Connector.TransactionIdGenerator)  
{  
    Token = "<Your Token>".To<SecureString>(),  
};  
Connector.Adapter.InnerAdapters.Add(messageAdapter);  
...
```

Recommended content

[Connection settings window](#)

BarChart

Barchart - Real-time market data provider for stock quotes, Forex, news, etc.

Before you start writing trading robots for the current trading platform, it is recommended to read the links in the [Connectors](#).

| Connector supports only market-data feed, transactions are not supported.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration BarChart

Configuration BarChart

To work with a connector, you have to specify the **Login** and **Password**. **Login** and **Password** are provided by the broker. To get the API access it is recommended to contact the broker.

Graphical configuration BarChart

Graphical configuration BarChart

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Barchart]

- **Login** - Login.
- **Password** - Password.
- **Heart beat** - Server check interval for track the connection alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization BarChart

Adapter initialization BarChart

The code below demonstrates how to initialize the [BarChartMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();  
...  
var messageAdapter = new BarChartMessageAdapter(Connector.TransactionIdGenerator)  
{  
    Token = "<Your Token>".To<SecureString>(),  
};  
Connector.Adapter.InnerAdapters.Add(messageAdapter);  
...
```

Recommended content

[Connection settings window](#)

Blackwood (Fusion)

Blackwood (Fusion)

Fusion (Blackwood) - platforms for trading stocks on exchanges NYSE, NASDAQ, AMEX, etc.

Before you start writing trading robots for the current trading platform, it is recommended to read the links in the [Connectors](#).

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Blackwood (Fusion)

Configuration Blackwood (Fusion)

To work with a connector, you have to specify the **Login** and **Password**. **Login** and **Password** are provided by the broker. To get the API access it is recommended to contact the broker.

The interaction mechanism is shown in this figure:

[image: blackwood trader]

Graphical configuration Blackwood (Fusion)

Graphical configuration Blackwood (Fusion)

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Fusion (Blackwood)]

- **Login** - Login.
- **Password** - Password.
- **Market data** - Market data server address.
- **History** - History data server address.
- **Transactions** - Server address of transactions execution.
- **Path to logs** - Path to directory, where BWOrders.Log file will be created.
- **Override** - Override dll file from resources. Turned on by default.
- **Heart beat** - Server check interval for track the connection alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Blackwood (Fusion)

Adapter initialization Blackwood (Fusion)

The code below demonstrates how to initialize the [BlackwoodMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var address = "<Address>".To<IPAddress>();
var messageAdapter = new BlackwoodMessageAdapter(Connector.TransactionIdGenerator)
{
    Login = "<Your Login>",
    Password = "<Your Password>".To<SecureString>(),
    ExecutionAddress = new IPEndPoint(address, BlackwoodAddresses.ExecutionPort),
    MarketDataAddress = new IPEndPoint(address, BlackwoodAddresses.MarketDataPort),
    HistoricalDataAddress = new IPEndPoint(address, BlackwoodAddresses.HistoricalDataPort)
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

CQG

CQG – trading platform for trading on the international exchanges CME, CBOT, NYMEX, ICE, EUREX.

Before you start writing trading robots for the current trading platform, it is recommended to read the links in the [Connectors](#).

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

[Adapter initialization CQG](#)

Configuration CQG

There are two ways to work with the **CQG** connector:

1. **CQG COM**, connection through the local **CQG Integrated Client**, which is installed directly on the computer from which the connection will be made. **CQG Integrated Client** can be downloaded [on the developer's site](#). To connect to the **CQG Integrated Client**, you have to specify the **Login** and **Password**.
2. **CQG Continuum**, connection directly to the server, to connect you have to specify the **Login**, **Password** and **server address**.

Login, **Password** and **server address** are provided by the broker. To get the API access it is recommended to contact the broker.

Graphical configuration CQG COM

Graphical configuration CQG COM

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings CQG COM]

- **Login** - Login.
- **Password** - Password.
- **Heart beat** - Server check interval for track the connection alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Graphical configuration CQG Continuum

Graphical configuration CQG Continuum

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings CQG Continuum]

- **Login** - Login.
- **Password** - Password.
- **Address** - Server address.
- **Software ID** - Unique software ID.
- **Heart beat** - Server check interval for track the connection alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization CQG

Adapter initialization CQG

The code below demonstrates how to initialize the [CqgComMessageAdapter](#) and [CqgContinuumMessageAdapter](#) and send it to [Connector](#).

1. CQG COM, connection by local CQG Integrated Client:

```
Connector Connector = new Connector();
...
var messageAdapter = new CqgComMessageAdapter(Connector.TransactionIdGenerator)
{
    UserName = "<Your Login>",
    Password = "<Your Password>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

2. CQG Continuum, connecting directly to the server:

```
Connector Connector = new Connector();
...
var messageAdapter = new CqgContinuumMessageAdapter(Connector.TransactionIdGenerator)
{
    UserName = "<Your Login>",
    Password = "<Your Password>".To<SecureString>(),
    Address = "<Address>".To<IPAddress>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

E*TRADE

E-Trade Financial Corporation (E*TRADE) - trading platform to trade financial assets including common stocks, preferred stocks, exchange-traded funds, futures contracts, options, mutual funds, and fixed income investments.

Before you start writing trading robots for the current trading platform, it is recommended to read the links in the [Connectors](#).

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

[Adapter initialization E*TRADE](#)

Configuration E*TRADE

Configuration E*TRADE

To work with a connector, you have to specify the **Login** and **Password**. **Login** and **Password** are provided by the broker. To get the API access, it is recommended to contact the broker.

The interaction mechanism is shown in this figure:

[image: ETrade]

E*TRADE uses the OAuth 1.0a authorization protocol, which requires a login and password over the browser on the E*TRADE site. The full authorization procedure sequence is shown in the following figure:

[image: etrade authorization]

A full authorization procedure should be performed only once a day (the E*TRADE server resets AccessTokens issued earlier at midnight by EST). If the full authorization procedure is already carried out on current day by EST, ETradeMessageAdapter automatically downloads AccessToken, stored in a subdirectory of E*TRADE algorithm.

Graphical configuration E*TRADE

Graphical configuration E*TRADE

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings E TRADE]

- **Key** - Key.
- **Secret** - Secret.
- **Demo** - Connect to demo trading instead of real trading server.
- **Verification code** - Verification code, received by user in browser, after confirming program's permission to work.
- **Heart beat** - Server check interval for tracking the connection alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization E*TRADE

Adapter initialization E*TRADE

The code below demonstrates how to initialize the [ETradeMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new ETradeMessageAdapter(Connector.TransactionIdGenerator)
{
    ConsumerSecret = "<Your Secret>".To<SecureString>(),
    ConsumerKey = "<Your Key>",
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

IEX

Investors Exchange (IEX) - Real-time market data provider for stock quotes, Forex, news, futures contracts, etc..

Before you start writing trading robots for the current trading platform, it is recommended to read the links in the [Connectors](#).

| Connector supports only market-data feed, transactions are not supported.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration IEX

To work with a connector, you have to specify the **Token**. **Token** is provided by the broker. To get the API access, it is recommended to contact the broker.

Graphical configuration IEX

Graphical configuration IEX

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings IEX]

- **Token** - Token.
- **Heart beat** - Server check interval for tracking the connection alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization IEX

Adapter initialization IEX

The code below demonstrates how to initialize the [IEXMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new IEXMessageAdapter(Connector.TransactionIdGenerator)
{
    Token = "<Your Token>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Interactive Brokers

Interactive Brokers

Interactive Brokers - trading platform to trade financial assets including stocks, options, futures, EFPs, futures options, forex, bonds, and funds.

Before you start writing trading robots for the current trading platform, it is recommended to read the links in the [Connectors](#).

Configuration Interactive Brokers

To work correctly with Interactive Brokers you should preset [TWS](#) trading terminal.

Trader Workstation terminal configuration

1. You must allow connections from other programs (such as the trading algorithm on [S#](#)). To do this, open the settings menu "File -> Global configuration...". Select "Configuration -> API -> Settings" in the new window:
[image: ib settings]
2. Turn on "Enable ActiveX and Socket Clients" mode.
3. Also it is recommended to add the address of the computer that will run the algorithm (the local address - 127.0.0.1). This eliminates the need to confirm the terminal connection permission each time you start the algorithm.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

[Adapter initialization Interactive Brokers](#)

Graphical configuration Interactive Brokers

Graphical configuration Interactive Brokers

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Interactive Brokers]

- **Address** - TWS address.
- **Identifier** - Unique ID. Used when several clients are connected to one terminal or gateway.
- **Real-time** - Should real-time or 'frozen' on broker server data be used.
- **Logging level** - Server messages logging level.
- **Market data fields** - Market data fields, which will be received with subscribed to Level1 messages.
- **Protocol** - SSL protocol to establish connection
- **Certificate** - SSL certificate.
- **Password** - SSL certificate password.
- **Check revocation** - Check certificate revocation.
- **Validate remote** - Validate remote certificates.
- **Host name** - The name of the server that shares SSL connection.
- **MaxVersion** - MaxVersion
- **Heart beat** - Server check interval for tracking the connection alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Interactive Brokers

Adapter initialization Interactive Brokers

The code below demonstrates how to initialize the [InteractiveBrokersMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new InteractiveBrokersMessageAdapter(Connector.TransactionIdGenerator)
{
    Address = "<Your Address>".To<EndPoint>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

IQFeed

DTN IQFeed - Real-time market data provider for stock quotes, Forex, news, futures contracts, etc..

Before you start writing trading robots for the current trading platform, it is recommended to read the links in the [Connectors](#).

Configuration IQFeed

The interaction mechanism is shown in this figure:

[image: IQFeed]

To work with the **IQFeed** connector, you need to install the **IQ Feed Client** router on the computer, which can be installed on both the local computer and the remote one. Data exchange between the client application and the **IQ Feed Client**, as well as between the **IQ Feed Client** and the servers, is performed via the TCP/IP protocol.

To download the **IQ Feed Client** from the [IQFeed](#) site, you must first authorize using the password and login received from **IQFeed**.

After installing **IQ Feed Client**, it is recommended to restart the computer.

After installing **IQ Feed Client**, **IQLink Launcher** must be started.

[image: iQFeedIQLinkLauncher]

In the **IQLink Launcher** window that opens, click **Start IQLink**.

[image: iQFeedIQConnectLogin]

In the opened **IQ Connect Login** window it is necessary to enter the **Login** and **Password** (or PIN) received from the **IQFeed** service, in this case the Login and Password are not the Login and Password from the **IQFeed** site. After filling in Login and Password, you have to click **Connect** to connect.

To receive data, the client application uses four connections through different ports:

1. Level1 (port 5009) is used to get real-time data on instruments (ticks, opening and closing prices, volatility, etc.) and news.
2. Level2 (port 9200) is used to get extended quotes for instruments, for each ECN you can get the best pair of quotes.
3. Lookup (port 9100) is used to search for instruments, retrieve historical data, get advanced information on news.
4. Admin (port 9300) is used to get general information about connection and changing settings.

The port numbers that are used by default for connection to the **IQ Feed Client** are specified in brackets. For client connections, the port numbers can be changed in the registry, for example, for Level1 on the following path: [HKEY_CURRENT_USER\SOFTWARE\DTN\IQFEED\Startup\Level1Port]. Port numbers for connecting to IQ servers can not be changed.

| Connector supports only market-data feed, transactions are not supported.

Recommended content

[Connectors](#)

[Graphical configuration](#)
[Save and load settings](#)
[Creating own connector](#)
[Orders management](#)
[Create new order](#)
[Create new stop order](#)
[Connection IQFeed](#)
[Adapter initialization IQFeed](#)

Connection IQFeed

The connection to the IQ servers is established using the IQConnect application, which can be installed on either the local or a remote computer. The communication between the client application and the IQConnect, as well as between IQConnect and servers is carried out by the TCP/IP protocol.

To get the data the client application uses four connections through various ports:

1. Level1 (port 5009) – used for real time data on instruments (ticks, the opening price, the closing price, volatility, etc.) and news.
2. Level2 (port 9200) – used to get extended quotes on instruments, the best couple of quotes can get for each ECN.
3. Lookup (port 9100) – used for instruments search, historical data getting, obtaining extended information on the news.
4. Admin (port 9300) – used to obtain general information about the connection and the settings change.

The numbers of ports in brackets are used by default to connect to the IQConnect. For client connections, you can change the port numbers in the registry, for example, for Level1 on the following path: [HKEY_CURRENT_USER\SOFTWARE\DTN\IQFEED\Startup\Level1Port]. The ports numbers to connect to IQ servers can not be changed.

Graphical configuration IQFeed

Graphical configuration IQFeed

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings IQFeed]

- **Level1 server** - Address for obtaining data on Level1.
- **Level2 server** - Address for obtaining data on Level2.
- **Lookup server** - Address for obtaining history data.
- **Admin server** - Address for obtaining service data.
- **Derivatives** - Address for obtaining derivative data.
- **Data for Level1** - All types of data for Level1, which have to be translated.
- **Data type** - Securities types, for which data must be received.
- **Load securities** - Should the whole set of securities be loaded from IQFeed website archive.
- **File with securities** - Path to file with IQFeed list of securities, downloaded from the website. If path is specified, then secondary download from website does not occur, and only the local copy gets parsed.
- **Version** - Version.
- **Heart beat** - Server check interval for tracking the connection alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization IQFeed

Adapter initialization IQFeed

The code below demonstrates how to initialize the [IQFeedMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new IQFeedMessageAdapter(Connector.TransactionIdGenerator)
{
    Level1Address = "127.0.0.1:5009".To<EndPoint>(),
    Level2Address = "127.0.0.1:9200".To<EndPoint>(),
    LookupAddress = "127.0.0.1:9100".To<EndPoint>(),
    AdminAddress = "127.0.0.1:9200".To<EndPoint>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

ITCH

ITCH

ITCH - Real-time market data provider for stock quotes, Forex, news, futures contracts, etc..

Before you start writing trading robots for the current trading platform, it is recommended to read the links in the [Connectors](#).

| Connector supports only market-data feed, transactions are not supported.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

[Adapter initialization ITCH](#)

Configuration ITCH

Configuration ITCH

To work with a connector, you have to specify the **Login** and **Password**. **Login** and **Password** are provided by the broker. To get the API access, it is recommended to contact the broker.

ITCH - direct data-feed protocol, which allows to connect via TCP and UDP protocols. **ITCH** provides the order log to track all changes to any order. This protocol is used by the NASDAQ (see [TotalView-ITCH](#)) and LSE (see [MITCH](#)).

Graphical configuration ITCH

Graphical configuration ITCH

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings ITCH]

- **Main** - The main UDP group.
- **Duplicate** - Duplicate UDP group.
- **Recovery** - Recovery server.
- **Replay** - Replay server.
- **Login** - Login.
- **Password** - Password.
- **Securities** - File with securities.
- **Delay** - Delay load instruments.
- **Active** - Only active instruments.
- **Group ID** - Group ID.
- **Time zone** - Information about the time zone where the exchange is located.
- **Heart beat** - Server check interval for tracking the connection alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization ITCH

Adapter initialization ITCH

The code below demonstrates how to initialize the [ItchMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new ItchMessageAdapter(Connector.TransactionIdGenerator)
{
    Login = "<Your Login>",
    Password = "<Your Password>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

OpenECry

OpenECry (GAIN Futures) – trading platform for trading on the international exchanges.

Before you start writing trading robots for the current trading platform, it is recommended to read the links in the [Connectors](#).

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

[Adapter initialization OpenECry](#)

[Several algorithms connection](#)

[Simultaneous terminal and algorithm operation](#)

Configuration OpenECry

Configuration OpenECry

The interaction mechanism is shown in this figure:

[image: OECTrader]

As can be seen from the figure, [OpenECryMessageAdapter](#) communicates with the OEC server through the [GainFutures API](#). Using the [GainFutures API](#) does not require a working OEC Trader terminal.

To work with a connector, you have to specify the **Login** and **Password**. **Login** and **Password** are provided by the broker. To get the API access, it is recommended to contact the broker.

Graphical configuration OpenECry

Graphical configuration OpenECry

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings OpenECry GainFutures]

- **Address** - Server address.
- **Login** - Login.
- **Password** - Password.
- **Work schedule** - Required mode of connection to the terminal.
- **Reconnection** - Use «native» reconnection mechanism.
- **Logging level** - Use OpenECry API logging.
- **Heart beat** - Server check interval for tracking the connection alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization OpenECry

Adapter initialization OpenECry

The code below demonstrates how to initialize the [OpenECryMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new OpenECryMessageAdapter(Connector.TransactionIdGenerator)
{
    Login = "<Your Login>",
    Password = "<Your Password>.To<SecureString>()",
    Address = "<Address>.To<EndPoint>()",
    EnableOECLogging = true,
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Several algorithms connection

Several algorithms connection

Depending on the particular user/application the OEC server may not support simultaneous connection of multiple applications. In this case, other connections can be interrupted. For circumvention of these limitations this [OpenECryTrader](#) implementation supports simultaneous operation of multiple applications through a single connection to the OEC server – [OECRemoting](#).

The following modes of [OpenECryRemoting](#) are supported:

- [None](#) - [OpenECryRemoting](#) disconnected. The application creates its own connection to the OEC server. The application can not serve as the [Primary](#) for other applications.
- [Primary](#) – the application creates its own connection to the OEC server.
- [Secondary](#) - it searches for local applications running in the [Primary](#) mode at the time of initialization. If such applications are found it uses their connection to the OEC server. Otherwise, the application enters [None](#) mode.

To explicitly set the [OECRemoting](#) mode you should specify the desired mode immediately after the [OpenECryTrader](#) object creation. For example, to set the [Secondary](#) mode:

```
Trader.RemotingRequired = OECRemoting.Secondary;
```

By default, the [OpenECryTrader](#) adapter operates in the [OpenECryRemoting.None](#) mode.

Simultaneous terminal and algorithm operation

Simultaneous terminal and algorithm operation

If necessary, you can configure the OEC Trader terminal to operate in [Primary](#) mode and then run the algorithm [S#](#) in [Primary](#) mode. In this case, the terminal and the algorithm will use the same connection to the OEC server. To configure it you should check the menu item File -> Allow Remoting in OEC Trader terminal:

[image: OECTraderSettings]

PolygonIO

PolygonIO - Market data provider of realtime and historical data on stocks, forex (FX), and digital/crypto currencies.

Before you start writing trading robots for the current trading platform, it is recommended to read the links in the [Connectors](#).

| Connector supports only market-data feed, transactions are not supported.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration PolygonIO

To work with a connector, you have to specify the **Token**. **Tokens** are provided by the provider. To get API access it is recommended to contact the provider.

Graphical configuration PolygonIO

Graphical configuration PolygonIO

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings PolygonIO]

- **Token** - Token.
- **Connection type** - Connect to History (REST) or Real-Time (WebSockets).

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization PolygonIO

Adapter initialization PolygonIO

The code below demonstrates how to initialize the [PolygonIOMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new PolygonIOMessageAdapter(Connector.TransactionIdGenerator)
{
    Token = "<Your Token>".To<SecureString>(),
    ConnectionType = PolygonIOConnectionTypes.History, // connection for REST data sources
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Quandl

Quandl - Real-time market data provider for stock quotes, Forex, news, futures contracts, etc..

Before you start writing trading robots for the current trading platform, it is recommended to read the links in the [Connectors](#).

| Connector supports only market-data feed, transactions are not supported.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Quandl

To work with a connector, you have to specify the **Token**. **Tokens** are provided by the broker. To get API access it is recommended to contact the broker.

Graphical configuration Quandl

Graphical configuration Quandl

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Quandl]

- **Token** - Token.
- **Database** - The database Quandl identifier.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Quandl

Adapter initialization Quandl

The code below demonstrates how to initialize the [QuandlMessageAdapter](#) and send it to [Connector](#).

```
var messageAdapter = new QuandlMessageAdapter(Connector.TransactionIdGenerator);
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

QuantFEED

QuantFEED - Real-time market data provider for stock quotes, Forex, news, futures contracts, etc..

Before you start writing trading robots for the current trading platform, it is recommended to read the links in the [Connectors](#).

| Connector supports only market-data feed, transactions are not supported.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

[Adapter initialization QuantFEED](#)

Configuration QuantFEED

To work with a connector, you have to specify the **Login** and **Password**. **Login** and **Password** are provided by the broker. To get API access it is recommended to contact the broker.

Graphical configuration QuantFEED

Graphical configuration QuantFEED

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings QuantFEED]

- **Address** - Server address.
- **Login** - Login.
- **Password** - Password.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization QuantFEED

Adapter initialization QuantFEED

The code below demonstrates how to initialize the [QuantFeedMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new QuantFeedMessageAdapter(Connector.TransactionIdGenerator)
{
    Login = "<Your Login>",
    Password = "<Your Password>".To<SecureString>(),
    Address = "<Address>".To<EndPoint>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Rithmic

Rithmic — trading platform for trading on the international exchanges.

Before you start writing trading robots for the current trading platform, it is recommended to read the links in the [Connectors](#).

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

[Adapter initialization Rithmic](#)

Configuration Rithmic

Configuration Rithmic

To work with a connector, you have to specify the **Login** and **Password**. **Login** and **Password** are provided by the broker. To get API access it is recommended to contact the broker.

Graphical configuration Rithmic

Graphical configuration Rithmic

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Rithmic]

- **Login** - Login.
- **Password** - Password.
- **Certificate** - Path to certificate file, necessary to connect to Rithmic system.
- **File log** - Path to log file.
- **Server type** - Server type.
- **Point (admin)** - Connection point for administrative functions (initialization/deinitialization).
- **Point (data)** - Connection point to market data.
- **Login (trans)** - Additional login. Used when transaction sending is carried out to a separate server.
- **Point (transactions)** - Connection point to the transactions execution system.
- **Password (trans)** - Additional password. Used when transaction sending is carried out to a separate server.
- **Point (positions)** - Connection point for access to portfolios and positions information.
- **Point (history)** - Connection point for access to history data.
- **Domain (address)** - Domain address.
- **Domain (name)** - Domain name.
- **Licenses** - Licenses server address.
- **Broker** - Broker address.
- **Log (address)** - Logger address.
- **User name (hist)** - Additional login. User id used for authentication with the history plant.
- **Password (hist)** - Additional password. Password used for authentication with the history plant
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Rithmic

Adapter initialization Rithmic

The code below demonstrates how to initialize the [RithmicMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new RithmicMessageAdapter(Connector.TransactionIdGenerator)
{
    UserName = "<Your Login>",
    Password = "<Your Password>".To<SecureString>(),
    CertFile = "<Path to certificate file>",
    Server = RithmicServers.Real,
    //Server = RithmicServers.Test,
    //Server = RithmicServers.Simulator,
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Sterling

Sterling

Sterling — trading platform for trading on the international exchanges.

Before you start writing trading robots for the current trading platform, it is recommended to read the links in the [Connectors](#).

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

[Adapter initialization Sterling](#)

Configuration Sterling

Configuration Sterling

To work with the **Sterling** connector, you need to use the local **Sterling Trader Pro** terminal, which is installed directly on the computer from which the connection will be made. To connect to **Sterling Trader Pro**, you have to specify the **Login** and **Password**.

[image: Sterling login]

Sterling Trader Pro, **Login**, **Password** and server address are provided by the broker. To get API access it is recommended to contact the broker.

For the Sterling API to work correctly, you need to run Visual Studio as an administrator.

Graphical configuration Sterling

Graphical configuration Sterling

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Sterling]

- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Sterling

Adapter initialization Sterling

The code below demonstrates how to initialize the [SterlingMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();  
...  
var messageAdapter = new SterlingMessageAdapter(Connector.TransactionIdGenerator);  
Connector.Adapter.InnerAdapters.Add(messageAdapter);  
...
```

Recommended content

[Connection settings window](#)

Tradier

Tradier — trading platform for trading on the international exchanges.

Before you start writing trading robots for the current trading platform, it is recommended to read the links in the [Connectors](#).

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

[Adapter initialization Rithmic](#)

Configuration Tradier

To work with a connector, you have to specify the **Token**. **Tokens** are provided by the broker. To get API access it is recommended to contact the broker.

Graphical configuration Tradier

Graphical configuration of Tradier

For all StockSharp products, graphical connection setup is performed in the [Connection settings window](#) screen form:

[image: API GUI Settings Tradier]

- **Token** - Authorization token.
- **Demo** - Demo mode.

OAuth Authorization:

1. You can directly insert the token into the "Token" field.
2. If you leave the token field empty and the "Demo" mode is not selected, OAuth authorization will be used.

OAuth authorization process:

1. When you click the "Check" button, a window will open:

[image: OAuth Start]

2. After clicking "Start", the user will be redirected to the Tradier website to log in:

[image: Tradier Login]

3. On the Tradier website, you need to allow the StockSharp application access to trading operations:

[image: Tradier Permissions]

4. After that, you will be redirected back to the StockSharp website, and the program will automatically log in.

See also

[Connectors](#)

[OAuth](#)

[Graphical configuration](#)

[Creating your own connector](#)

[Saving and loading settings](#)

Adapter initialization Tradier

Adapter initialization Tradier

The code below demonstrates how to initialize the [TradierMessageAdapter](#) and send it to [Connector](#).

```
var messageAdapter = new TradierMessageAdapter(Connector.TransactionIdGenerator);
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Xignite

Xignite - Real-time market data provider for stock quotes, Forex, news, futures contracts, etc..

Before you start writing trading robots for the current trading platform, it is recommended to read the links in the [Connectors](#).

| Connector supports only market-data feed, transactions are not supported.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Xignite

To work with a connector, you have to specify the **Token**. **Tokens** are provided by the broker. To get API access it is recommended to contact the broker.

Graphical configuration Xignite

Graphical configuration Xignite

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Xignite]

- **Token** - Token.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Xignite

Adapter initialization Xignite

The code below demonstrates how to initialize the [XigniteMessageAdapter](#) and send it to [Connector](#).

```
var messageAdapter = new XigniteMessageAdapter(Connector.TransactionIdGenerator);
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Yahoo

Yahoo - market data provider for stock quotes, Forex, news, futures contracts, etc.

Before you start writing trading robots for the current trading platform, it is recommended to read the links in the [Connectors](#).

| Connector supports only market-data feed, transactions are not supported.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Graphical configuration Yahoo

Graphical configuration Yahoo

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Yahoo]

- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Yahoo

Adapter initialization Yahoo

The code below demonstrates how to initialize the [YahooMessageAdapter](#) and send it to [Connector](#).

```
var messageAdapter = new YahooMessageAdapter(Connector.TransactionIdGenerator);
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

MetaTrader

[StockSharp](#) integrates with MT4 and MT5 terminals through special connectors. To install these connectors, use [Installer](#) (for more details, see [Installing and Removing Programs](#)).

Both connectors are used in the same way, so below is the description of the process for connecting to MT5:

Setting Up the MT Connector

1. Select the MT connector in [Installer](#) and start the installation process.
[image: MT Install 1]
2. [Installer](#) will ask which folder to install the connector in (it must be installed in the Experts folder).
[image: MT Install 2]
3. If multiple terminals are installed, you need to choose the one where you want to install the connector.
[image: MT Install 3]
4. After selecting the desired terminal, the path to the Experts folder will be displayed.
[image: MT Install 4]
[!TIP]
 - If the path cannot be automatically determined, you need to manually select it through directory search C:\Users\%your_user_name%\AppData\Roaming\MetaQuotes\Terminal\%many_letters_and_numbers%\MQL4\Experts\ (for MT5, the path will include MQL5).
5. Complete the installation and wait for it to finish. At the end of the installation, [Installer](#) will warn that you now need to configure the terminal. To do this, launch the MT terminal and connect to trading.
6. In the Tools->Options menu, select the **Experts Advisors** tab and make sure that permission for external DLL trading (**Allow DLL imports**) is enabled:*[image: MT 1]*
7. If the terminal was running during the connector installation (step 2), you need to refresh the list of experts by right-clicking on Experts and selecting **Refresh** from the menu:
[image: MT 2]
8. Select the S# expert, right-click and choose **Attach to a chart** from the menu:
[image: MT 3]
9. A settings window will appear where you can set the login-password (anonymous authorization is enabled by default), as well as the connection address (if connecting to multiple terminals at once, the addresses must contain unique ports).
10. A smiley icon should appear in the upper right corner of the chart (the first one encountered):
[image: MT 4]
Also, information about the successful script launch, the number of instruments, should appear in the expert's log window.
11. If the MT4 or MT5 license is not obtained, a line similar to the following will appear in the log:
[image: MT 5]
12. Connection to MT goes through the FIX protocol, using the [FIX protocol](#) connector. The program [Terminal](#) was used for demonstration. Below are settings for transactional connection and market data

connection (for MT5, the default port is 23001 instead of 23000):

*[image: MT 6]**[image: MT 7]*

Similar settings need to be made in [Designer](#), [Hydra](#), or any API programs.

Login and password are left empty in case of anonymous authorization (previous item). If connecting to MT with multiple robots, a unique login must be provided for different connections identification.

[!TIP]

- The script must be launched before connecting StockSharp to MetaTrader and kept running as long as this connection is needed.
- To see historical candles in StockSharp, they need to be downloaded from the MetaTrader server. How to do this, read in MetaTrader's documentation.

In case of a successful connection, the example should show a list of instruments and accounts:

[image: MT 8]

13. In case of errors, the connector logs are kept, which are available in the folder **Experts\StockSharp\Data\Log**:

[image: MT 9]

DXtrade

DXtrade - a platform for trading on the Forex markets.

Before starting to write trading robots for the current trading platform, it is recommended to familiarize yourself with the links provided in the [Connectors](#) section.

See also

[Connectors](#)

[Graphical Configuration](#)

[Save and Load Settings](#)

[Creating Your Own Connector](#)

[Orders](#)

[Create New Order](#)

[Create New Stop Order](#)

DXtrade Connector Settings

When working with the connector, you need to specify the **Key** and **Secret** to connect to the trading platform. **Key** and **Secret** are provided by the broker. To obtain API access, it is recommended to contact the broker.

Graphical configuration DXtrade

Graphical Configuration of DXtrade

For all [S\#](#) products, graphical configuration of the connection is performed in the [Connection Settings Window](#):

[image: API GUI Settings DXtrade]

- **Key** - Key.
- **Secret** - Secret.
- **Demo** - Connection to demo trading.

See also

[Connectors](#)

[Graphical Configuration](#)

[Creating Your Own Connector](#)

[Save and Load Settings](#)

Adapter initialization DXtrade

Initialization of DXtrade Adapter

The code below demonstrates how to initialize the [DXtradeMessageAdapter](#) and pass it to the [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new DXtradeMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

cTrader

cTrader - a platform for trading on the Forex markets.

Before starting to write trading robots for the current trading platform, it is recommended to familiarize yourself with the links provided in the [Connectors](#) section.

See also

[Connectors](#)

[Graphical Configuration](#)

[Save and Load Settings](#)

[Creating Your Own Connector](#)

[Orders](#)

[Create New Order](#)

[Create New Stop Order](#)

cTrader Connector Settings

To connect to cTrader, you need to choose the type of connection (demo or real trading). After connecting for the first time, the program will redirect to the website to confirm access to trading.

Graphical configuration cTrader

Graphical configuration of cTrader

For all StockSharp products, graphical connection setup is performed in the [Connection settings window](#) screen form:

[image: API GUI Settings cTrader]

- **Demo** - Connection to demo trading.

OAuth Authorization:

cTrader provides only OAuth method of authorization.

OAuth authorization process:

1. When you click the "Check" button, a window will open:

[image: OAuth Start]

2. After clicking "Start", the user will be redirected to the cTrader website to log in. On the cTrader website, you need to allow the StockSharp application access to trading operations:

[image: cTrader Login]

3. After that, you will be redirected back to the StockSharp website, and the program will automatically log in.

See also

[Connectors](#)

[OAuth](#)

[Graphical configuration](#)

[Creating your own connector](#)

[Saving and loading settings](#)

Adapter initialization cTrader

Initialization of cTrader Adapter

The code below demonstrates how to initialize the [cTraderMessageAdapter](#) and pass it to the [Connector](#).

```
Connector Connector = new Connector();  
...  
var messageAdapter = new cTraderMessageAdapter(Connector.TransactionIdGenerator)  
{  
    IsDemo = true, // Demo  
};  
Connector.Adapter.InnerAdapters.Add(messageAdapter);  
...
```

Recommended content

[Connection settings window](#)

DukasCopy

DukasCopy - the historical data provides a variety of financial instruments (e.g. Forex, Commodities and Indices).

Before you start writing trading robots for the current trading platform, it is recommended to read the links in the [Connectors](#).

| Connector supports only market-data feed; transactions are not supported.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Graphical configuration DukasCopy

Graphical configuration DukasCopy

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Dukascopy]

- **Heartbeat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization DukasCopy

Adapter initialization DukasCopy

The code below demonstrates how to initialize the [DukasCopyMessageAdapter](#) and send it to [Connector](#).

```
var messageAdapter = new DukasCopyMessageAdapter(Connector.TransactionIdGenerator);  
Connector.Adapter.InnerAdapters.Add(messageAdapter);  
...
```

Recommended content

[Connection settings window](#)

FXCM

FXCM - Forex trading platform.

Before you start writing trading robots for the current trading platform, it is recommended to read the links in the [Connectors](#).

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration FXCM

Configuration FXCM

To work with a connector, you have to specify the **Login** and **Password**. **Login** and **Password** are provided by the broker. To get the API access it is recommended to contact the broker.

Graphical configuration FXCM

Graphical configuration FXCM

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings FXCM]

- **Token** - Token.
- **Demo** - Connect to demo trading instead of real trading server.
- **Heartbeat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization FXCM

Adapter initialization FXCM

The code below demonstrates how to initialize the [FxcmlMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new FxcmlMessageAdapter(Connector.TransactionIdGenerator)
{
    Login = "<Your Login>",
    Password = "<Your Password>".To<SecureString>(),
    Address = "<Your Address>".To<Uri>(),
    IsDemo = true
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

LMAX

LMAX Global - LMAX Global is an FCA regulated Broker for FX. LMAX Global offers brokers and professional traders the ability to trade on the LMAX Exchange central limit order book, offering tighter spreads on streaming firm liquidity from top tier banks and proprietary trading firms.

Before you start writing trading robots for the current trading platform, it is recommended to read the links in the [Connectors](#).

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration LMAX

Configuration LMAX

The interaction mechanism is shown in this figure:

[image: LMaxTrader]

To work with a connector, you have to specify the **Login** and **Password**. **Login** and **Password** are provided by the broker. To get the API access it is recommended to contact the broker.

Graphical configuration LMAX

Graphical configuration LMAX

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings LMAX]

- **Login** - Login.
- **Password** - Password.
- **Demo** - Connect to demo trading instead of real trading server.
- **LMAX location** - LMAX exchange location.
- **Heartbeat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization LMAX

Adapter initialization LMAX

The code below demonstrates how to initialize the [LmaxMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new LmaxMessageAdapter(Connector.TransactionIdGenerator)
{
    Login = "<Your Login>",
    Password = "<Your Password>".To<SecureString>(),
    IsDemo = true
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Oanda

Oanda

Oanda - Forex trading platform.

Before you start writing trading robots for the current trading platform, it is recommended to read the links in the [Connectors](#).

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Oanda

Configuration Oanda

To work with a connector, you have to specify the **Login** and **Password**. **Login** and **Password** are provided by the broker. To get API access, it is recommended to contact the broker.

Graphical configuration Oanda

Graphical configuration Oanda

For all [S#](#) products, graphical configuration of the connection is performed in the [Connection settings window](#):

[image: API GUI Settings OANDA]

- **Server** - Server
- **Token** - Token
- **Compression** - Compression.
- **Transactions only** - Write log messages only for transaction stream.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Oanda

Adapter initialization Oanda

The code below demonstrates how to initialize the [OandaMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new OandaMessageAdapter(Connector.TransactionIdGenerator)
{
    Token = "<Your Token>".To<SecureString>(),
    IsDemo = true, // Demo
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Bibox

Bibox - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Bibox

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Bibox

Graphical configuration Bibox

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Bibox]

- **Key** - Key.
- **Secret** - Secret.
- **Password** - Administrative password.
- **Heart beat** - Server check interval to track that the connection is alive. By default it is 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Bibox

Adapter initialization Bibox

The code below demonstrates how to initialize the [BiboxMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new BiboxMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Binance

Binance

Binance - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Binance

Configuration Binance

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Binance

Graphical configuration Binance

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Binance]

- **Key** - Key.
- **Secret** - Secret.
- **Password** - Administrative password.
- **Heart beat** - Server check interval to track that the connection is alive. By default it is 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Binance

Adapter initialization Binance

The code below demonstrates how to initialize the [BinanceMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new BinanceMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Binance History

Binance History

Binance History - historical service from Binance exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

| Connector supports only market-data feed, transactions are not supported.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Binance History

Configuration Binance History

The connector does not need to be configured.

Graphical configuration Binance History

Graphical configuration Binance History

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Binance History]

- **Check dates** - Validate requested dates before sending download requests.

Adapter initialization Binance History

Adapter initialization Binance History

The code below demonstrates how to initialize the [BinanceHistoryMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new BinanceHistoryMessageAdapter(Connector.TransactionIdGenerator)
{
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Bybit History

Bybit History

Bybit History - historical service from Bybit exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

| Connector supports only market-data feed, transactions are not supported.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Bybit History

Configuration Bybit History

The connector does not need to be configured.

Graphical configuration Bybit History

Graphical configuration Bybit History

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Bybit History]

The window does not expose Bybit History-specific options.

Adapter initialization Bybit History

Adapter initialization Bybit History

The code below demonstrates how to initialize the [BybitHistoryMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();  
...  
var messageAdapter = new BybitHistoryMessageAdapter(Connector.TransactionIdGenerator)  
{  
};  
Connector.Adapter.InnerAdapters.Add(messageAdapter);  
...
```

Recommended content

[Connection settings window](#)

BingX

BingX

BingX is a cryptocurrency trading exchange.

Before starting to write trading robots for the current trading platform, it is recommended to familiarize yourself with the links provided in the [Connectors](#) section.

See also

[Connectors](#)

Configuration BingX

Configuration BingX

To work with the connector, generate an **API Key** and **Secret** in your exchange account and specify them in the connection settings.

Note that the **Key and Secret** are not your **Login and Password**. The key pair is generated separately in your account. Login and password are not required for the connector.

Graphical configuration BingX

Graphical Configuration of BingX

For all [S#](#) products, graphical configuration of the connection is performed in the [Connection Settings Window](#):

[image: API GUI Settings BingX]

- **Key** - Key.
- **Secret** - Secret.
- **Demo** - Demo mode.
- **Reconnection settings** - Parameters of the reconnection mechanism with the trading system ([Reconnection settings](#)).

Adapter initialization BingX

Adapter initialization BingX

The code below shows how to initialize [BingXMessageAdapter](#) and pass it to the [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new
BingXMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

See also

[Connection Settings Window](#)

Bitalong

Bitalong

Bitalong - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Bitalong

Configuration Bitalong

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Bitalong

Graphical configuration Bitalong

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Bitalong]

- **Key** - Key.
- **Secret** - Secret.
- **Heart beat** - Server check interval to track that the connection is alive. By default it is 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Bitalong

Adapter initialization Bitalong

The code below demonstrates how to initialize the [BitalongMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new BitalongMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Bitbank

Bitbank - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Bitbank

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Bitbank

Graphical configuration Bitbank

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Bitbank]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdrawal actions.
- **Heart beat** - Server check interval to track that the connection is alive. By default it is 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))
- **RequestWithdrawAccounts** - RequestWithdrawAccounts

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Bitbank

Adapter initialization Bitbank

The code below demonstrates how to initialize the [BitbankMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new BitbankMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Bitget

Bitget

Bitget - a cryptocurrency trading exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

See also

[Connectors](#)

[Graphical Configuration](#)

[Save and Load Settings](#)

[Creating Your Own Connector](#)

[Orders](#)

[Create New Order](#)

[Create New Stop Order](#)

Bitget Connector Settings

To work with the connector, you need to generate an **API Key** and **Secret** in the exchange's personal account and specify them when setting up the connection.

Note that the **API Key and Secret** are not **Login and Password**. **API Key and Secret** are generated separately in the personal account. Login and Password are not required for the connector and should not be specified anywhere.

Graphical configuration Bitget

Graphical Configuration of Bitget

For all [S#](#) products, graphical configuration of the connection is performed in the [Connection Settings Window](#):

[image: API GUI Settings Bitget]

- **Key** - Key.
- **Secret** - Secret.
- **Sections** - Trading sections.
- **Demo** - Connection to demo trading.

See also

[Connectors](#)

[Graphical Configuration](#)

[Creating Your Own Connector](#)

[Save and Load Settings](#)

Adapter initialization Bitget

Initialization of Bitget Adapter

The code below demonstrates how to initialize the [BitgetMessageAdapter](#) and pass it to the [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new BitgetMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Bitexbook

Bitexbook - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Bitexbook

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Bitexbook

Graphical configuration Bitexbook

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Bitexbook]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdrawal actions.
- **Heart beat** - Server check interval to track that the connection is alive. By default it is 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Bitexbook

Adapter initialization Bitexbook

The code below demonstrates how to initialize the [BitexbookMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new BitexbookMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Bitfinex

Bitfinex - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Bitfinex

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them in the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Bitfinex

Graphical configuration Bitfinex

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Bitfinex]

- **Key** - Key.
- **Secret** - Secret.
- **Cancel On Disconnect** - Cancel On Disconnect.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Bitfinex

Adapter initialization Bitfinex

The code below demonstrates how to initialize the [BitfinexMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new BitfinexMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Bithumb

Bibox - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Bithumb

Configuration Bithumb

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them in the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Bithumb

Graphical configuration Bithumb

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Bithumb]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdrawal actions.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))
- **IsPrime** - IsPrime

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Bithumb

Adapter initialization Bithumb

The code below demonstrates how to initialize the [BithumbMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new BithumbMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Bitmart

Bitmart - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Bitmart

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them in the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Bitmart

Graphical configuration Bitmart

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Bitmart]

- **Key** - Key.
- **Secret** - Secret.
- **Board** - Section to connect (Spot, Futures).
- **Passphrase** - Administrative password.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Bitmart

Adapter initialization Bitmart

The code below demonstrates how to initialize the [BitmartMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new BitmartMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

BitMax has been renamed to AscendEX. This documentation is preserved for historical reference.

BitMax

BitMax - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration BitMax

BitMax has been renamed to AscendEX. This documentation is preserved for historical reference.

Configuration BitMax

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them in the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration BitMax

BitMax has been renamed to AscendEX. This documentation is preserved for historical reference.

Graphical configuration BitMax

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings BitMax]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdrawal actions.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization BitMax

BitMax has been renamed to AscendEX. This documentation is preserved for historical reference.

Adapter initialization BitMax

The code below demonstrates how to initialize the [BitMaxMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new BitMaxMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

BitMEX

BitMEX - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration BitMEX

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them in the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration BitMEX

Graphical configuration BitMEX

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings BitMEX]

- **Key** - Key.
- **Secret** - Secret.
- **Demo** - Connect to demo trading instead of real trading server.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization BitMEX

Adapter initialization BitMEX

The code below demonstrates how to initialize the [BitmexMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new BitmexMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

BitStamp

BitStamp - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration BitStamp

Configuration BitStamp

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them in the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration BitStamp

Graphical configuration BitStamp

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Bitstamp]

- **Key** - Key.
- **Secret** - Secret.
- **Client** - Client ID.
- **Balance** - Balance check interval. Required in case of deposit and withdrawal actions.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))
- **AuthV2** - AuthV2

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization BitStamp

Adapter initialization BitStamp

The code below demonstrates how to initialize the [BitStampMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new BitStampMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Bittrex

Bittrex - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Bittrex

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Bittrex

Graphical configuration Bittrex

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Bittrex]

- **Key** - Key.
- **Secret** - Secret.
- **Heart beat** - Server check interval for tracking the connection alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Bittrex

Adapter initialization Bittrex

The code below demonstrates how to initialize the [BittrexMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();  
...  
var messageAdapter = new BittrexMessageAdapter(Connector.TransactionIdGenerator)  
{  
    Key = "<Your API Key>".To<SecureString>(),  
    Secret = "<Your API Secret>".To<SecureString>(),  
};  
Connector.Adapter.InnerAdapters.Add(messageAdapter);  
...
```

Recommended content

[Connection settings window](#)

BitZ

This exchange has permanently shut down (~2021 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

BitZ

BitZ - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration BitZ

This exchange has permanently shut down (~2021 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Configuration BitZ

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration BitZ

This exchange has permanently shut down (~2021 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Graphical configuration BitZ

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Bit]

- **Key** - Key.
- **Secret** - Secret.
- **Domain (address)** - Domain address.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Password** - Administrative password.
- **Heart beat** - Server check interval for track the connection alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization BitZ

This exchange has permanently shut down (~2021 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Adapter initialization BitZ

The code below demonstrates how to initialize the [BitZMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new BitZMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

ByBit

ByBit - a cryptocurrency trading exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

See also

[Connectors](#)

[Graphical Configuration](#)

[Save and Load Settings](#)

[Creating Your Own Connector](#)

[Orders](#)

[Create New Order](#)

[Create New Stop Order](#)

ByBit Connector Settings

To work with the connector, you need to generate an **API Key** and **Secret** in the exchange's personal account and specify them when setting up the connection.

Note that the **API Key and Secret** are not **Login and Password**. **API Key and Secret** are generated separately in the personal account. Login and Password are not required for the connector and should not be specified anywhere.

Graphical configuration ByBit

Graphical Configuration of ByBit

For all [S#](#) products, graphical configuration of the connection is performed in the [Connection Settings Window](#):

[image: API GUI Settings ByBit]

- **Key** - Key.
- **Secret** - Secret.
- **Sections** - Trading sections.
- **Demo** - Connection to demo trading.

See also

[Connectors](#)

[Graphical Configuration](#)

[Creating Your Own Connector](#)

[Save and Load Settings](#)

Adapter initialization ByBit

Initialization of ByBit Adapter

The code below demonstrates how to initialize the [ByBitMessageAdapter](#) and pass it to the [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new ByBitMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

BW

This exchange has permanently shut down (~2021 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

BW

BW - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration BW

This exchange has permanently shut down (~2021 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Configuration BW

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration BW

This exchange has permanently shut down (~2021 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Graphical configuration BW

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings BW]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Heart beat** - Server check interval for tracking the connection alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization BW

This exchange has permanently shut down (~2021 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Adapter initialization BW

The code below demonstrates how to initialize the [BWMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new BWMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

CEX.IO

CEX.IO - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration CEX.IO

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration CEX.IO

Graphical configuration CEX.IO

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings CEX.IO]

- **Key** - Key.
- **Secret** - Secret.
- **Heart beat** - Server check interval for track the connection alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization CEX.IO

Adapter initialization CEX.IO

The code below demonstrates how to initialize the [CexMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();  
...  
var messageAdapter = new CexMessageAdapter(Connector.TransactionIdGenerator)  
{  
    Key = "<Your API Key>".To<SecureString>(),  
    Secret = "<Your API Secret>".To<SecureString>(),  
};  
Connector.Adapter.InnerAdapters.Add(messageAdapter);  
...
```

Recommended content

[Connection settings window](#)

Coinbase

Coinbase

Coinbase - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Coinbase

Configuration Coinbase

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Coinbase

Graphical configuration Coinbase

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Coinbase]

- **Key** - Key.
- **Secret** - Secret.
- **Passphrase** - Passphrase.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Heart beat** - Server check interval for tracking the connection alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Coinbase

Adapter initialization Coinbase

The code below demonstrates how to initialize the [CoinbaseMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();  
...  
var messageAdapter = new CoinbaseMessageAdapter(Connector.TransactionIdGenerator)  
{  
    Key = "<Your API Key>".To<SecureString>(),  
    Secret = "<Your API Secret>".To<SecureString>(),  
};  
Connector.Adapter.InnerAdapters.Add(messageAdapter);  
...
```

Recommended content

[Connection settings window](#)

CoinBene

This exchange has permanently shut down (~2021 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

CoinBene

Bibox - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration CoinBene

This exchange has permanently shut down (~2021 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Configuration CoinBene

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration CoinBene

This exchange has permanently shut down (~2021 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Graphical configuration CoinBene

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings CoinBene]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Heart beat** - Server check interval for track the connection alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization CoinBene

This exchange has permanently shut down (~2021 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Adapter initialization CoinBene

The code below demonstrates how to initialize the [CoinBeneMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();  
...  
var messageAdapter = new CoinBeneMessageAdapter(Connector.TransactionIdGenerator)  
{  
    Key = "<Your API Key>".To<SecureString>(),  
    Secret = "<Your API Secret>".To<SecureString>(),  
};  
Connector.Adapter.InnerAdapters.Add(messageAdapter);  
...
```

Recommended content

[Connection settings window](#)

CoinCap

CoinCap - Market data provider of Real-time data on digital/crypto currencies.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

| Connector support only market-data feed, transaction are not supported.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration CoinCap

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration CoinCap

Graphical configuration CoinCap

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings CoinCap]

- **Heart beat** - Server check interval for track the connection alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization CoinCap

Adapter initialization CoinCap

The code below demonstrates how to initialize the [CoinCapMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new CoinCapMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Coincheck

Coincheck - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Coincheck

Configuration Coincheck

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Coincheck

Graphical configuration Coincheck

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Coincheck]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Heart beat** - Server check interval for track the connection alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Coincheck

Adapter initialization Coincheck

The code below demonstrates how to initialize the [CoincheckMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();  
...  
var messageAdapter = new CoincheckMessageAdapter(Connector.TransactionIdGenerator)  
{  
    Key = "<Your API Key>".To<SecureString>(),  
    Secret = "<Your API Secret>".To<SecureString>(),  
};  
Connector.Adapter.InnerAdapters.Add(messageAdapter);  
...
```

Recommended content

[Connection settings window](#)

CoinEx

CoinEx - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration CoinEx

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration CoinEx

Graphical configuration CoinEx

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings CoinEx]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Heart beat** - Server check interval for tracking the connection alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization CoinEx

Adapter initialization CoinEx

The code below demonstrates how to initialize the [CoinExMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();  
...  
var messageAdapter = new CoinExMessageAdapter(Connector.TransactionIdGenerator)  
{  
    Key = "<Your API Key>".To<SecureString>(),  
    Secret = "<Your API Secret>".To<SecureString>(),  
};  
Connector.Adapter.InnerAdapters.Add(messageAdapter);  
...
```

Recommended content

[Connection settings window](#)

CoinExchange

This exchange has permanently shut down (October 2019 – shut down). This connector is no longer operational. Documentation is preserved for historical reference.

CoinExchange

CoinExchange - Market data provider of Real-time data on crypto currencies.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Connector support only market-data feed, transaction are not supported.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration CoinExchange

This exchange has permanently shut down (October 2019 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Configuration CoinExchange

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration CoinExchange

This exchange has permanently shut down (October 2019 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Graphical configuration CoinExchange

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings CoinExchange]

- **Key** - Key.
- **Secret** - Secret.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization CoinExchange

This exchange has permanently shut down (October 2019 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Adapter initialization CoinExchange

The code below demonstrates how to initialize the [CoinExchangeMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new CoinExchangeMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Coinigy

Coinigy - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Coinigy

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Coinigy

Graphical configuration Coinigy

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Coinigy]

- **Key** - Key.
- **Secret** - Secret.
- **Password** - Administrative password.
- **Websocket** - Websocket id.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Coinigy

Adapter initialization Coinigy

The code below demonstrates how to initialize the [CoinigyMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new CoinigyMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

CoinHub

CoinHub - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration CoinHub

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration CoinHub

Graphical configuration CoinHub

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings CoinHub]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization CoinHub

Adapter initialization CoinHub

The code below demonstrates how to initialize the [CoinHubMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new CoinHubMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Cryptopia

This exchange has permanently shut down (May 2019 — hacked and liquidated). This connector is no longer operational. Documentation is preserved for historical reference.

Cryptopia

Cryptopia - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Cryptopia

This exchange has permanently shut down (May 2019 — hacked and liquidated). This connector is no longer operational. Documentation is preserved for historical reference.

Configuration Cryptopia

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Cryptopia

This exchange has permanently shut down (May 2019 — hacked and liquidated). This connector is no longer operational. Documentation is preserved for historical reference.

Graphical configuration Cryptopia

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Cryptopia]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Cryptopia

This exchange has permanently shut down (May 2019 — hacked and liquidated). This connector is no longer operational. Documentation is preserved for historical reference.

Adapter initialization Cryptopia

The code below demonstrates how to initialize the [CryptopiaMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new CryptopiaMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Deribit

Deribit - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Deribit

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Deribit

Graphical configuration Deribit

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Deribit]

- **Key** - Key.
- **Secret** - Secret.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Deribit

Adapter initialization Deribit

The code below demonstrates how to initialize the [DeribitMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new DeribitMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Digifinex

DigiFinex - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Digifinex

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Digifinex

Graphical configuration Digifinex

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings DigiFinex]

- **Key** - Key.
- **Secret** - Secret.
- **Domain (address)** - Domain address.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Digifinex

Adapter initialization Digifinex

The code below demonstrates how to initialize the [DigifinexMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new DigifinexMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

DigitexFutures

This exchange has permanently shut down (~2022 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

DigitexFutures

DigitexFutures - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration DigitexFutures

This exchange has permanently shut down (~2022 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Configuration DigitexFutures

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration DigitexFutures

This exchange has permanently shut down (~2022 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Graphical configuration DigitexFutures

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings DigitexFutures]

- **Key** - Key.
- **Secret** - Secret.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization DigitexFutures

This exchange has permanently shut down (~2022 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Adapter initialization DigitexFutures

The code below demonstrates how to initialize the [DigitexFuturesMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new DigitexFuturesMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

EXMO

EXMO - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration EXMO

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration EXMO

Graphical configuration EXMO

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings EXMO]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Heart beat** - Server check interval for tracking that the connection is alive. By default, equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization EXMO

Adapter initialization EXMO

The code below demonstrates how to initialize the [ExmoMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new ExmoMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

FatBTC

This exchange has permanently shut down (~2022 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

FatBTC

FatBTC - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration FatBTC

This exchange has permanently shut down (~2022 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Configuration FatBTC

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration FatBTC

This exchange has permanently shut down (~2022 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Graphical configuration FatBTC

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings FatBTC]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Domain (address)** - Domain address.
- **Heart beat** - Server check interval for tracking that the connection is alive. By default, equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization FatBTC

This exchange has permanently shut down (~2022 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Adapter initialization FatBTC

The code below demonstrates how to initialize the [FatBtcMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new FatBtcMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Gate.io

Gate.io - a cryptocurrency trading exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

See also

[Connectors](#)

[Graphical Configuration](#)

[Save and Load Settings](#)

[Creating Your Own Connector](#)

[Orders](#)

[Create New Order](#)

[Create New Stop Order](#)

Gate.io Connector Settings

To work with the connector, you need to generate an **API Key** and **Secret** in the exchange's personal account and specify them when setting up the connection.

Note that the **API Key and Secret** are not **Login and Password**. **API Key and Secret** are generated separately in the personal account. Login and Password are not required for the connector and should not be specified anywhere.

Graphical configuration Gate.io

Graphical Configuration of Gate.io

For all [S#](#) products, graphical configuration of the connection is performed in the [Connection Settings Window](#):

[image: API GUI Settings Gate.io]

- **Key** - Key.
- **Secret** - Secret.
- **Sections** - Trading sections.
- **Demo** - Connection to demo trading.

See also

[Connectors](#)

[Graphical Configuration](#)

[Creating Your Own Connector](#)

[Save and Load Settings](#)

Adapter initialization Gate.io

Initialization of Gate.io Adapter

The code below demonstrates how to initialize the [GateIOMessageAdapter](#) and pass it to the [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new GateIOMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

GDAX

GDAX has been renamed to Coinbase Pro, and later to Coinbase Advanced Trade. This documentation is preserved for historical reference.

GDAX

GDAX - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration GDAX

GDAX has been renamed to Coinbase Pro, and later to Coinbase Advanced Trade. This documentation is preserved for historical reference.

Configuration GDAX

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration GDAX

GDAX has been renamed to Coinbase Pro, and later to Coinbase Advanced Trade. This documentation is preserved for historical reference.

Graphical configuration GDAX

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings GDAX]

- **Key** - Key.
- **Secret** - Secret.
- **Passphrase** - Passphrase.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Heart beat** - Server check interval for tracking that the connection is alive. By default, equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization GDAX

GDAX has been renamed to Coinbase Pro, and later to Coinbase Advanced Trade. This documentation is preserved for historical reference.

Adapter initialization GDAX

The code below demonstrates how to initialize the [GdaxMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new GdaxMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Gopax

This exchange has permanently shut down (~2023 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Gopax

GOPAX - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Gopax

This exchange has permanently shut down (~2023 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Configuration Gopax

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Gopax

This exchange has permanently shut down (~2023 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Graphical configuration Gopax

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Gopax]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Domain (address)** - Domain address.
- **Heart beat** - Server check interval for tracking that the connection is alive. By default, equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Gopax

This exchange has permanently shut down (~2023 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Adapter initialization Gopax

The code below demonstrates how to initialize the [GopaxMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new GopaxMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

HitBTC

HitBTC - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration HitBTC

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration HitBTC

Graphical configuration HitBTC

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings HitBTC]

- **Key** - Key.
- **Secret** - Secret.
- **Heart beat** - Server check interval for tracking that the connection is alive. By default, equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization HitBTC

Adapter initialization HitBTC

The code below demonstrates how to initialize the [HitBtcMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new HitBtcMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Hotbit

This exchange has permanently shut down (May 2023 – shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Hotbit

Hotbit - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Hotbit

This exchange has permanently shut down (May 2023 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Configuration Hotbit

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Hotbit

This exchange has permanently shut down (May 2023 – shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Graphical configuration Hotbit

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Hotbit]

- **Key** - Key.
- **Secret** - Secret.
- **Heart beat** - Server check interval for tracking that the connection is alive. By default, equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Hotbit

This exchange has permanently shut down (May 2023 – shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Adapter initialization Hotbit

The code below demonstrates how to initialize the [HotbitMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new HotbitMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Huobi

Huobi

Huobi - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Huobi

Configuration Huobi

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Huobi

Graphical configuration Huobi

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Huobi]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **HADAX** - Use HADAX.
- **Heart beat** - Server check interval for tracking that the connection is alive. By default, equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Huobi

Adapter initialization Huobi

The code below demonstrates how to initialize the [HuobiMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();  
...  
var messageAdapter = new HuobiMessageAdapter(Connector.TransactionIdGenerator)  
{  
    Key = "<Your API Key>".To<SecureString>(),  
    Secret = "<Your API Secret>".To<SecureString>(),  
};  
Connector.Adapter.InnerAdapters.Add(messageAdapter);  
...
```

Recommended content

[Connection settings window](#)

Idax

This exchange has permanently shut down (November 2019 – shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Idax

IDAX - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Idax

This exchange has permanently shut down (November 2019 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Idax configuration

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Idax

This exchange has permanently shut down (November 2019 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Idax graphical configuration

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings IDAX]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Idax

This exchange has permanently shut down (November 2019 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Idax adapter initialization

The code below demonstrates how to initialize the [IdaxMessageAdapter](#) and pass it to the [Connector](#).

```
Connector Connector = new Connector();  
...  
var messageAdapter = new IdaxMessageAdapter(Connector.TransactionIdGenerator)  
{  
    Key = "<Your API Key>".To<SecureString>(),  
    Secret = "<Your API Secret>".To<SecureString>(),  
};  
Connector.Adapter.InnerAdapters.Add(messageAdapter);  
...
```

Recommended content

[Connection settings window](#)

Kraken

Kraken

Kraken - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Kraken

Kraken configuration

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Kraken

Kraken graphical configuration

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Kraken]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Kraken

Kraken adapter initialization

The code below demonstrates how to initialize the [KrakenMessageAdapter](#) and pass it to the [Connector](#).

```
Connector Connector = new Connector();  
...  
var messageAdapter = new KrakenMessageAdapter(Connector.TransactionIdGenerator)  
{  
    Key = "<Your API Key>".To<SecureString>(),  
    Secret = "<Your API Secret>".To<SecureString>(),  
};  
Connector.Adapter.InnerAdapters.Add(messageAdapter);  
...
```

Recommended content

[Connection settings window](#)

Kucoin

KuCoin - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Kucoin configuration

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Kucoin

Kucoin graphical configuration

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Kucoin]

- **Key** - Key.
- **Secret** - Secret.
- **Passphrase** - Passphrase.
- **Demo** - Connect to demo trading instead of real trading server.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Kucoin

Kucoin adapter initialization

The code below demonstrates how to initialize the [KucoinMessageAdapter](#) and pass it to the [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new KucoinMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Latoken

LATOKEN - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Latoken

Latoken configuration

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Latoken

Latoken graphical configuration

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings LATOKEN]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Latoken

Latoken adapter initialization

The code below demonstrates how to initialize the [LatokenMessageAdapter](#) and pass it to the [Connector](#).

```
Connector Connector = new Connector();  
...  
var messageAdapter = new LatokenMessageAdapter(Connector.TransactionIdGenerator)  
{  
    Key = "<Your API Key>".To<SecureString>(),  
    Secret = "<Your API Secret>".To<SecureString>(),  
};  
Connector.Adapter.InnerAdapters.Add(messageAdapter);  
...
```

Recommended content

[Connection settings window](#)

LBank

LBank - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

LBank configuration

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration LBank

LBank graphical configuration

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings LBank]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization LBank

LBank adapter initialization

The code below demonstrates how to initialize the [LBankMessageAdapter](#) and pass it to the [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new LBankMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Liqui

This exchange has permanently shut down (January 2019 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Liqui

Liqui - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Liqui

This exchange has permanently shut down (January 2019 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Liqui configuration

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Liqui

This exchange has permanently shut down (January 2019 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Liqui graphical configuration

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Liqui]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Liqui

This exchange has permanently shut down (January 2019 — shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Liqui adapter initialization

The code below demonstrates how to initialize the [LiquiMessageAdapter](#) and pass it to the [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new LiquiMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Livecoin

This exchange has permanently shut down (December 2020 — hacked and shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Livecoin

Livecoin - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Livecoin

This exchange has permanently shut down (December 2020 — hacked and shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Livecoin configuration

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them at the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Livecoin

This exchange has permanently shut down (December 2020 — hacked and shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Livecoin graphical configuration

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Livecoin]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Livecoin

This exchange has permanently shut down (December 2020 — hacked and shut down). This connector is no longer operational. Documentation is preserved for historical reference.

Livecoin adapter initialization

The code below demonstrates how to initialize the [LiveCoinMessageAdapter](#) and pass it to the [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new LiveCoinMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

MEXC

MEXC

MEXC is a cryptocurrency trading exchange.

Before starting to write trading robots for the current trading platform, it is recommended to familiarize yourself with the links provided in the [Connectors](#) section.

See also

[Connectors](#)

Configuration MEXC

Configuration MEXC

To work with the connector, generate an **API Key** and **Secret** in your exchange account and specify them in the connection settings.

Note that the **Key and Secret** are not your **Login and Password**. The key pair is generated separately in your account. Login and password are not required for the connector.

Graphical configuration MEXC

Graphical Configuration of MEXC

For all [S#](#) products, graphical configuration of the connection is performed in the [Connection Settings Window](#):

[image: API GUI Settings MEXC]

- **Key** - Key.
- **Secret** - Secret.
- **Demo** - Demo mode.
- **Reconnection settings** - Parameters of the reconnection mechanism with the trading system ([Reconnection settings](#)).

Adapter initialization MEXC

Adapter initialization MEXC

The code below shows how to initialize [MexcMessageAdapter](#) and pass it to the [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new
MexcMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

See also

[Connection Settings Window](#)

OKCoin

OKCoin - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration OKCoin

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them in the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration OKCoin

Graphical configuration OKCoin

For all [S#](#) products, graphical configuration of the connection is performed in the [Connection settings window](#):

[image: API GUI Settings OKCoin]

- **Key** - Key.
- **Secret** - Secret.
- **Password** - Administrative password.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization OKCoin

Adapter initialization OKCoin

The code below demonstrates how to initialize the [OkcoinMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();  
...  
var messageAdapter = new OkcoinMessageAdapter(Connector.TransactionIdGenerator)  
{  
    Key = "<Your API Key>".To<SecureString>(),  
    Secret = "<Your API Secret>".To<SecureString>(),  
};  
Connector.Adapter.InnerAdapters.Add(messageAdapter);  
...
```

Recommended content

[Connection settings window](#)

OKEx

OKEx - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration OKEx

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them in the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration OKEx

Graphical configuration OKEx

For all [S#](#) products, graphical configuration of the connection is performed in the [Connection settings window](#):

[image: API GUI Settings OKEx]

- **Key** - Key.
- **Secret** - Secret.
- **Passphrase** - Passphrase.
- **Futures** - Futures section
- **Spot** - Spot section
- **Swap** - Swap section.
- **Password** - Administrative password.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization OKEx

Adapter initialization OKEx

The code below demonstrates how to initialize the [OkexMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new OkexMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

OKEx History

OKEx History

OKEx History - historical service from OKEx exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

| Connector supports only market-data feed, transactions are not supported.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration OKEx History

Configuration OKEx History

The connector does not need to be configured.

Graphical configuration OKEx History

Graphical configuration OKEx History

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings OKEx History]

- **Check dates** - Validate requested dates before sending download requests.

Adapter initialization OKEx History

Adapter initialization OKEx History

The code below demonstrates how to initialize the [OkexHistoryMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();  
...  
var messageAdapter = new OkexHistoryMessageAdapter(Connector.TransactionIdGenerator)  
{  
};  
Connector.Adapter.InnerAdapters.Add(messageAdapter);  
...
```

Recommended content

[Connection settings window](#)

Poloniex

Poloniex - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Poloniex

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them in the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Poloniex

Graphical configuration Poloniex

For all [S#](#) products, graphical configuration of the connection is performed in the [Connection settings window](#):

[image: API GUI Settings Poloniex]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdrawal actions.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Poloniex

Adapter initialization Poloniex

The code below demonstrates how to initialize the [PoloniexMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();  
...  
var messageAdapter = new PoloniexMessageAdapter(Connector.TransactionIdGenerator)  
{  
    Key = "<Your API Key>".To<SecureString>(),  
    Secret = "<Your API Secret>".To<SecureString>(),  
};  
Connector.Adapter.InnerAdapters.Add(messageAdapter);  
...
```

Recommended content

[Connection settings window](#)

PrizmBit

PrizmBit - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration PrizmBit

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them in the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration PrizmBit

Graphical configuration PrizmBit

For all [S#](#) products, graphical configuration of the connection is performed in the [Connection settings window](#):

[image: API GUI Settings PrizmBit]

- **Key** - Key.
- **Secret** - Secret.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization PrizmBit

Adapter initialization PrizmBit

The code below demonstrates how to initialize the [PrizmBitMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new PrizmBitMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

QUOINEX has been renamed to Liquid, which shut down in 2022. This documentation is preserved for historical reference.

Quoinex

QuoineX - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Quoinex

QUOINEX has been renamed to Liquid, which shut down in 2022. This documentation is preserved for historical reference.

Configuration Quoinex

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them in the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Quoinex

QUOINEX has been renamed to Liquid, which shut down in 2022. This documentation is preserved for historical reference.

Graphical configuration Quoinex

For all [S#](#) products, graphical configuration of the connection is performed in the [Connection settings window](#):

[image: API GUI Settings QUOINEX]

- **Key** - Key.
- **Secret** - Secret.
- **User** - User
- **Balance** - Balance check interval. Required in case of deposit and withdrawal actions.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))
- **IsQryptos** - IsQryptos
- **PusherAccessKey** - PusherAccessKey

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Quoinex

QUOINEX has been renamed to Liquid, which shut down in 2022. This documentation is preserved for historical reference.

Adapter initialization Quoinex

The code below demonstrates how to initialize the [QuoinexMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new QuoinexMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

TradeOgre

TradeOgre - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration TradeOgre

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them in the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration TradeOgre

Graphical configuration TradeOgre

For all [S#](#) products, graphical configuration of the connection is performed in the [Connection settings window](#):

[image: API GUI Settings TradeOgre]

- **Key** - Key.
- **Secret** - Secret.
- **Password** - Administrative password.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization TradeOgre

Adapter initialization TradeOgre

The code below demonstrates how to initialize the [TradeOgreMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new TradeOgreMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Upbit

Upbit - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Upbit

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them in the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Upbit

Graphical configuration Upbit

For all [S#](#) products, graphical configuration of the connection is performed in the [Connection settings window](#):

[image: API GUI Settings Upbit]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdrawal actions.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Upbit

Adapter initialization Upbit

The code below demonstrates how to initialize the [UpbitMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();  
...  
var messageAdapter = new UpbitMessageAdapter(Connector.TransactionIdGenerator)  
{  
    Key = "<Your API Key>".To<SecureString>(),  
    Secret = "<Your API Secret>".To<SecureString>(),  
};  
Connector.Adapter.InnerAdapters.Add(messageAdapter);  
...
```

Recommended content

[Connection settings window](#)

WEX (BTC-e)

This exchange has permanently shut down (July 2017 — seized). This connector is no longer operational. Documentation is preserved for historical reference.

WEX (BTC-e)

BTC-E - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration WEX (BTC-e)

This exchange has permanently shut down (July 2017 — seized). This connector is no longer operational. Documentation is preserved for historical reference.

Configuration WEX (BTC-e)

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them in the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration WEX (BTC-e)

This exchange has permanently shut down (July 2017 — seized). This connector is no longer operational. Documentation is preserved for historical reference.

Graphical configuration WEX (BTC-e)

For all [S#](#) products, graphical configuration of the connection is performed in the [Connection settings window](#):

[image: API GUI Settings WEX (BTC)]

- **Key** - Key.
- **Secret** - Secret.
- **Domain (address)** - Domain address.
- **Balance** - Balance check interval. Required in case of deposit and withdrawal actions.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization WEX (BTC-e)

This exchange has permanently shut down (July 2017 — seized). This connector is no longer operational. Documentation is preserved for historical reference.

Adapter initialization WEX (BTC-e)

The code below demonstrates how to initialize the [BtceMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new BtceMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Yobit

This exchange has permanently shut down (~2023 — effectively defunct). This connector is no longer operational. Documentation is preserved for historical reference.

Yobit

YoBit - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Yobit

This exchange has permanently shut down (~2023 — effectively defunct). This connector is no longer operational. Documentation is preserved for historical reference.

Configuration Yobit

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them in the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Yobit

This exchange has permanently shut down (~2023 — effectively defunct). This connector is no longer operational. Documentation is preserved for historical reference.

Graphical configuration Yobit

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Yobit]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Heartbeat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Yobit

This exchange has permanently shut down (~2023 — effectively defunct). This connector is no longer operational. Documentation is preserved for historical reference.

Adapter initialization Yobit

The code below demonstrates how to initialize the [YobitMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new YobitMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Zaif

Zaif

Zaif - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Zaif

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them in the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration Zaif

Graphical configuration Zaif

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings Zaif]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Heartbeat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Zaif

Adapter initialization Zaif

The code below demonstrates how to initialize the [ZaifMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();  
...  
var messageAdapter = new ZaifMessageAdapter(Connector.TransactionIdGenerator)  
{  
    Key = "<Your API Key>".To<SecureString>(),  
    Secret = "<Your API Secret>".To<SecureString>(),  
};  
Connector.Adapter.InnerAdapters.Add(messageAdapter);  
...
```

Recommended content

[Connection settings window](#)

ZB

ZB

ZB - cryptocurrency exchange.

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration ZB

To work with a connector, it is required to generate the **Key (API Key)** and **Secret** in the personal account of the exchange and specify them in the connection settings.

It is worth paying attention that the **Key** and **Secret** are not the **Login** and **Password**. The **Key** and **Secret** are generated separately in your personal account. You do not need the **Login** and **Password** to work with the connector and you do not need to specify them anywhere.

Graphical configuration ZB

Graphical configuration ZB

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings ZBCOM]

- **Key** - Key.
- **Secret** - Secret.
- **Balance** - Balance check interval. Required in case of deposit and withdraw actions.
- **Password** - Administrative password.
- **Heartbeat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization ZB

Adapter initialization ZB

The code below demonstrates how to initialize the [ZBMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();  
...  
var messageAdapter = new ZBMessageAdapter(Connector.TransactionIdGenerator)  
{  
    Key = "<Your API Key>".To<SecureString>(),  
    Secret = "<Your API Secret>".To<SecureString>(),  
};  
Connector.Adapter.InnerAdapters.Add(messageAdapter);  
...
```

Recommended content

[Connection settings window](#)

Withdraw

To withdraw funds from the cryptocurrency exchange, it is necessary to create an order for withdrawal and register it in the connector as a regular order. For example, for the [Binance](#) exchange, the withdrawal request code will look like this:

```
Connector Connector = new Connector();
...
public void Withdraw()
{
    var order = new Order
    {
        Type = OrderTypes.Conditional,
        Withdraw = 1,
        Condition = new BinanceOrderCondition
        {
            IsWithdraw = true,
            WithdrawInfo = new WithdrawInfo
            {
                // fill in the necessary details
                Comment = "My profit",
                //PaymentId = "45467dyjyttR8WBiTJXptyuTx4wbSerGZ5t45", //Riple
                CryptoAddress = "16zK3M53JBGnjs9ajTBGBfkRqoHtm4E573",
            }
        },
        Security = new Security() { Code = "BTC" },
    };
    _connector.RegisterOrder(order);
}
...
```

Aster

Aster - cryptocurrency exchange (DEX).

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Aster

Configuration Aster

To work with the connector, generate **API Key** and **Secret** in the exchange account and specify them in connection settings.

Main settings:

- **Key** and **Secret**.
- **Section**: Spot or Derivatives.
- **Derivatives mode**: Legacy or V3 Agent.
- **Spot REST / Spot WS** endpoints.
- **Derivatives REST / Derivatives WS** endpoints.
- **Demo** mode.

Official API documentation:

- [Spot API overview](#)
- [Spot account and trading API](#)
- [Spot websocket market data](#)
- [Spot websocket account info](#)
- [Futures v3 general info](#)
- [Futures user data streams](#)
- [Aster Code endpoints](#)

| Aster derivatives have two protocol families. Select the correct **Derivatives mode** before enabling trading.

Graphical configuration Aster

Graphical configuration Aster

For all [S#](#) products, graphical configuration is performed in the [Connection settings window](#).

Main fields:

- **Key** - API key.
- **Secret** - API secret.
- **Section** - trading section (Spot or Derivatives).
- **Derivatives mode** - derivatives protocol family.
- **Demo** - demo environment mode.
- **Spot REST / Derivatives REST** - REST endpoints.
- **Spot WS / Derivatives WS** - websocket endpoints.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Aster

Adapter initialization Aster

The code below demonstrates how to initialize [AsterMessageAdapter](#) and pass it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new AsterMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
    Section = AsterSections.Derivatives,
    DerivativesProtocolMode = AsterDerivativesProtocolModes.Legacy,
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

edgeX

edgeX - cryptocurrency exchange (DEX).

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration edgeX

To work with the connector, generate **API Key** and **Secret** in the exchange account and specify them in connection settings.

Main settings:

- **Key** and **Secret**.
- **Clearing account** and **Passphrase**.
- **Section**: Spot or Derivatives.
- **Enable spot**: enables the spot section when API support is available.
- **Demo** mode.
- **Spot REST / Derivatives REST** endpoints.
- **Spot WS / Derivatives public WS / Derivatives private WS** endpoints.

Official API documentation:

- [Authentication](#)
- [Order API](#)
- [Account API](#)
- [Private websocket stream](#)
- [Funding API](#)
- [Meta-data API](#)
- [Quote API](#)

Derivatives is fully implemented. Spot should be enabled only when the target API environment supports it.

Graphical configuration edgeX

Graphical configuration edgeX

For all [S#](#) products, graphical configuration is performed in the [Connection settings window](#).

Main fields:

- **Key** - API key.
- **Secret** - API secret.
- **Clearing account** - clearing account value.
- **Passphrase** - passphrase value.
- **Section** - trading section (Spot or Derivatives).
- **Enable spot** - enables spot section.
- **Demo** - demo environment mode.
- **Spot REST / Derivatives REST** - REST endpoints.
- **Spot WS / Derivatives public WS / Derivatives private WS** - websocket endpoints.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization edgeX

Adapter initialization edgeX

The code below demonstrates how to initialize [EdgeXMessageAdapter](#) and pass it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new EdgeXMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
    ClearingAccount = "<Your Clearing Account>",
    Passphrase = "<Your Passphrase>".To<SecureString>(),
    Section = EdgeXSections.Derivatives,
    EnableSpotSection = false,
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Hyperliquid

Hyperliquid

Hyperliquid - cryptocurrency exchange (DEX).

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Hyperliquid

Configuration Hyperliquid

To work with the connector, specify wallet-level trading credentials.

Main settings:

- **Wallet address.**
- **Private key.**
- **Vault address** (optional).
- **Expires after** (optional).
- **Section:** Spot or Derivatives.
- **Testnet** mode.
- **Info endpoint / Exchange endpoint / WS endpoint.**
- **Market slippage** for market orders.

Official API documentation:

- [Hyperliquid API](#)
- [Info endpoint](#)
- [Exchange endpoint](#)
- [Websocket](#)

Graphical configuration Hyperliquid

Graphical configuration Hyperliquid

For all [S#](#) products, graphical configuration is performed in the [Connection settings window](#).

Main fields:

- **Wallet address** - wallet address (0x. . .).
- **Private key** - private key for signed actions.
- **Vault address** - optional vault or sub-account address.
- **Expires after** - optional signature expiration timestamp.
- **Section** - trading section (Spot or Derivatives).
- **Testnet** - testnet endpoints.
- **Info endpoint / Exchange endpoint / WS endpoint** - endpoint URLs.
- **Market slippage** - default slippage for market orders.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Hyperliquid

Adapter initialization Hyperliquid

The code below demonstrates how to initialize [HyperliquidMessageAdapter](#) and pass it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new HyperliquidMessageAdapter(Connector.TransactionIdGenerator)
{
    WalletAddress = "<Your Wallet Address>",
    PrivateKey = "<Your Private Key>".To<SecureString>(),
    Section = HyperliquidSections.Derivatives,
    IsTestnet = false,
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Ligther

Ligther (Lighter) - cryptocurrency exchange (DEX).

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Ligther

To work with the connector, generate **API Key** and **Secret** in the exchange account and specify them in connection settings.

Main settings:

- **Key** and **Secret**.
- **Account index** and **API key index**.
- **Section**: Spot or Derivatives.
- **Demo** mode.
- **Spot REST / Derivatives REST** endpoints.
- **Spot WS / Derivatives WS** endpoints.
- **WS read-only mode**.

Official API documentation:

- [Getting started](#)
- [Get API key and secret](#)
- [API rate limits](#)
- [Websocket channels](#)
- [Websocket API docs](#)

Graphical configuration Ligther

Graphical configuration Ligther

For all [S#](#) products, graphical configuration is performed in the [Connection settings window](#).

Main fields:

- **Key** - API key.
- **Secret** - API secret.
- **Account index** - account index.
- **API key index** - API key index.
- **Section** - trading section (Spot or Derivatives).
- **Demo** - demo environment mode.
- **Spot REST / Derivatives REST** - REST endpoints.
- **Spot WS / Derivatives WS** - websocket endpoints.
- **WS read-only mode** - websocket read-only mode flag.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Ligther

Adapter initialization Ligther

The code below demonstrates how to initialize [LigtherMessageAdapter](#) and pass it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new LigtherMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
    AccountIndex = 0,
    ApiKeyIndex = 0,
    Section = LigtherSections.Derivatives,
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Paradex

Paradex - cryptocurrency exchange (DEX).

Before you start writing trading robots for this trading platform, we recommend reading the links in the [Connectors](#) section.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

Configuration Paradex

Configuration Paradex

To work with the connector, specify your API credentials and Starknet authentication settings.

Main settings:

- **Key** and **Secret**.
- **Starknet account** and **Starknet key**.
- **Section**: Spot or Derivatives.
- **Enable spot**: enables the spot section when API support is available.
- **Demo** mode.
- **Spot REST / Derivatives REST** endpoints.
- **Spot WS / Derivatives WS** endpoints.
- **Auth path** (default: /v1/auth).

Official API documentation:

- [API URLs](#)
- [Authentication](#)
- [REST API](#)
- [Create a new order](#)
- [Websocket introduction](#)
- [Websocket channels](#)
- [Order book channel](#)

Paradex derivatives are fully supported. Enable Spot only when target API environment confirms spot support.

Graphical configuration Paradex

Graphical configuration Paradex

For all [S#](#) products, graphical configuration is performed in the [Connection settings window](#).

Main fields:

- **Key** - API key.
- **Secret** - API secret.
- **Starknet account** - Starknet account header value.
- **Starknet key** - Starknet private key material.
- **Section** - trading section (Spot or Derivatives).
- **Enable spot** - enables spot section.
- **Demo** - demo environment mode.
- **Spot REST / Derivatives REST** - REST endpoints.
- **Spot WS / Derivatives WS** - websocket endpoints.
- **Auth path** - auth endpoint path.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization Paradex

Adapter initialization Paradex

The code below demonstrates how to initialize [ParadexMessageAdapter](#) and pass it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new ParadexMessageAdapter(Connector.TransactionIdGenerator)
{
    Key = "<Your API Key>".To<SecureString>(),
    Secret = "<Your API Secret>".To<SecureString>(),
    StarknetAccount = "<Your Starknet Account>",
    StarknetPrivateKey = "<Your Starknet Private Key>".To<SecureString>(),
    Section = ParadexSections.Derivatives,
    AuthPath = "/v1/auth",
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

CSV

CSV - importing data from a text format.

Before you start writing trading robots for the current trading platform, it is recommended to review the links in the [Connectors](#) section.

| The connector supports only market-data feeds; transactions are not supported.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

[Adapter initialization Rithmic](#)

Graphical configuration CSV

CSV graphical configuration

For all [S#](#) products, graphical configuration of the connection is performed in the [Connection settings window](#):

[image: API GUI Settings CSV]

- **Settings** - Import settings
- **Reconnection settings** - Mechanism for tracking the connection with the trading system. ([Reconnection settings](#))
- **Heartbeat** - Server check interval to track that the connection is alive. By default, it equals 1 minute.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization CSV

CSV adapter initialization

The adapter is initialized using the [Connection settings window](#).

The adapter settings are described in the topic [Importing](#).

Recommended content

[Connection settings window](#)

FIX protocol

FIX protocol

[FIX](#) allows you to create trading robots that use standard international protocols for access to electronic markets - [FIX](#).

Recommended content

[Adapter initialization FIX](#)

[Fix Trading Community](#)

Configuration FIX

Configuration FIX

The connector uses a direct connection that requires a **VPN** or **encrypted channel**.

To work with a connector, you have to specify the **Login** and **Password**. **Login** and **Password** are provided by the broker. To get the API access it is recommended to contact the broker.

Graphical configuration FIX

Graphical configuration FIX

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings FIX]

- **Dialect** - Dialect FIX protocol.
- **Login** - Login.
- **Password** - Password.
- **Name** - Source name (to distinguish in log files).
- **Logging level** - Logging level.
- **MaxRestores** - MaxRestores
- **MaxRestoreError** - MaxRestoreError
- **MaxIncrementalWrongOrder** - MaxIncrementalWrongOrder
- **MaxRestoreCount** - MaxRestoreCount
- **MaxSuspended** - MaxSuspended
- **Settings file** - Settings file
- **Heart beat** - Server check interval to track that the connection is alive. By default, it equals 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization FIX

Adapter initialization FIX

The code below demonstrates how to initialize the [FixMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new FixMessageAdapter(Connector.TransactionIdGenerator)
{
    Login = "<Your Login>",
    Password = "<Your Password>".To<SecureString>(),
    Address = "<Address>".To<EndPoint>(),
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

FAST protocol

FAST protocol

[FAST](#) allows you to create trading robots that use standard international protocols for access to electronic markets - [FAST](#).

Recommended content

[FAST protocol](#)

Graphical configuration FAST

Graphical configuration FAST

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings FAST]

- **Dialect** - Dialect FAST protocol. Must be set as a first parameter. After selecting, the next parameter (**FAST settings**) will contain the specified dialect IP addresses.
- **FAST settings** - Dialect's (selected in prev param) IP addresses and ports. You can use the following option for quick settings load.
- **Settings file** - Quick option to load dialect settings from an exchange config file.
- **Login** - Login (used for TCP recovery services).
- **Password** - Password (used for TCP recovery services).

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization FAST

Adapter initialization FAST

The code below demonstrates how to initialize the [FastMessageAdapter](#) and send it to [Connector](#).

```
Connector Connector = new Connector();
...
var messageAdapter = new FastMessageAdapter(Connector.TransactionIdGenerator)
{
    // choose required dialect
    Dialect = typeof(StockSharp.Fix.Dialects.Bovespa.BovespaFastDialect),
};
// load all dialect settings from an exchange config file
messageAdapter.DialectSettings.LoadSettingsFromFile(configFile);
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Dump mode

Dump mode

The FAST connector is available in dump mode. In this case, the data does not come from a real network connection, but from the accumulated files of the app [UDP Dumper](#).

In order to enable dump mode, it is necessary to transfer files through the dump method [IFastDialect.Dump\(System.Collections.Generic.IDictionary<Ecng.Net.MulticastSourceAddress, System.Collections.Generic.IEnumerable<System.IO.Stream>> dumpFiles \)](#) before connecting:

```
// ... connector initialization
var fastAdapter = (FastMessageAdapter)connector.Adapters.InnerAdapters.First();
IEnumerable<string> dumpFiles = Directory.GetFiles(dumpDir, "*.bin");
var dict = dumpFiles.Select(f =>
{
    var name = Path.GetFileNameWithoutExtension(f);
    var parts = name.Split('_').Skip(1).ToArray();
    var groupAddr = parts[0];
    var port = parts[1];
    var sourceAddr = parts[2];
    if (sourceAddr.IsNullOrEmpty())
        sourceAddr = null;
    return Tuple.Create(new MulticastSourceAddress
    {
        GroupAddress = groupAddr.To<IPAddress>(),
        Port = port.To<int>(),
        SourceAddress = sourceAddr.To<IPAddress>(),
    }, f);
}).GroupBy(t => t.Item1).ToDictionary(g => g.Key, g => (IEnumerable<Stream>)g.Select(p =>
File.OpenRead(p.Item2)).ToArray());

fastAdapter.DialectSettings.Dump(dict);
// ...
connector.Connect();
```

After that, the work with the connector proceeds as usual, as if it was receiving data from the network.

FIX Server

FIX Server

[FixServer](#) allows you to create your own server based on the FIX/FAST protocols. It uses FIX 4.4 for transactions and FAST for market data. The scheme of interaction is shown in the following figure:

[image: FixServer]

RSS

RSS

RSS — connection to RSS feeds.

Before you start writing trading robots for the current trading platform, it is recommended to review the links in the [Connectors](#) section.

| The connector supports only market-data feeds; transactions are not supported.

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Save and load settings](#)

[Creating own connector](#)

[Orders management](#)

[Create new order](#)

[Create new stop order](#)

[Adapter initialization Rithmic](#)

Graphical configuration RSS

Graphical configuration RSS

For all [S#](#) products, graphical configuration of the connection is performed on the [Connection settings window](#):

[image: API GUI Settings RSS]

- **Address** - RSS feed address.
- **Dates format** - Dates format. Required if the RSS stream format is different from ddd, dd MMM yyyy HH:mm:ss zzzz.
- **Heart beat** - Server check interval to track that the connection is alive. By default equal to 1 minute.
- **Reconnection settings** - Mechanism for tracking connections with the trading system settings. ([Reconnection settings](#))

Recommended content

[Connectors](#)

[Graphical configuration](#)

[Creating own connector](#)

[Save and load settings](#)

Adapter initialization RSS

Adapter initialization RSS

The code below demonstrates how to initialize the [RssMessageAdapter](#) and send it to [Connector](#).

```
var messageAdapter = new RssMessageAdapter(Connector.TransactionIdGenerator)
{
    Address = new Uri("http://energy.rss"),
    CustomDateFormat = "ddd, dd MMM yyyy HH:mm:ss zzzz"
};
Connector.Adapter.InnerAdapters.Add(messageAdapter);
...
```

Recommended content

[Connection settings window](#)

Creating own connector

Creating Your Own Connector

The messaging mechanism is an internal logical layer of the [StockSharp](#) architecture, which provides interaction between various platform elements using a standard protocol.

There are two main classes:

- [Message](#) - a message carrying information.
- [AsyncMessageAdapter](#) - a message adapter (=converter).

A **message** acts as an agent that transmits information. Messages have their own type [MessageTypes](#). Each message type corresponds to a specific class. In turn, all message classes inherit from the abstract class [Message](#), which endows descendants with properties such as message type [Message.Type](#) and [Message.LocalTime](#) - the local time of message creation/receipt.

Messages can be *incoming* and *outgoing*:

- *Incoming* messages - messages that are sent to an external system. Usually, these are commands generated by the program, for example, the [ConnectMessage](#) message - a command requesting a connection to the server.
- *Outgoing* messages - messages coming from an external system. These are messages that transmit information about market data, transactions, portfolios, connection events, etc. For example, the [QuoteChangeMessage](#) message transmits information about changes in the order book.

The **message adapter** plays the role of an intermediary between the trading system and the program. For each type of connector, there is a separate adapter class that inherits from the abstract class [AsyncMessageAdapter](#).

The adapter performs two main functions:

1. Converts incoming messages into commands of a specific trading system.
2. Converts information received from the trading system (connection, market data, transactions, etc.) into outgoing messages.

Below is a description of the process of creating your own adapter for [Coinbase](#) (all connectors with source code are available in the [StockSharp repository](#) and are provided as a tutorial).

Example of Creating a Coinbase Message Adapter

1. Creating an Adapter Class

First, we create the **CoinbaseMessageAdapter** message adapter class, inherited from the abstract class [AsyncMessageAdapter](#).

```
public partial class CoinbaseMessageAdapter : AsyncMessageAdapter
{
    private Authenticator _authenticator;
    private HttpClient _restClient;
    private SocketClient _socketClient;

    // Other adapter fields and properties
}
```

2. Adapter Constructor

In the adapter constructor, you need to perform the following actions:

1. Pass the transaction ID generator that will be used to create message IDs.
2. Specify the supported message types using the methods:
 - [AddMarketDataSupport](#) - support for messages to subscribe to market data.
 - [AddTransactionalSupport](#) - support for transactional messages.
3. Specify the specific types of market data supported by the adapter using the [AddSupportedMarketDataType](#) method.

```
public CoinbaseMessageAdapter(IdGenerator transactionIdGenerator)
: base(transactionIdGenerator)
{
    HeartbeatInterval = TimeSpan.FromSeconds(5);

    // Add support for market data and transactions
    this.AddMarketDataSupport();
    this.AddTransactionalSupport();

    // Remove unsupported message types
    this.RemoveSupportedMessage(MessageTypes.Portfolio);
    this.RemoveSupportedMessage(MessageTypes.OrderGroupCancel);

    // Add supported market data types
    this.AddSupportedMarketDataType(DataType.Ticks);
    this.AddSupportedMarketDataType(DataType.MarketDepth);
    this.AddSupportedMarketDataType(DataType.Level1);
    this.AddSupportedMarketDataType(DataType.CandleTimeFrame);
}
```

3. Connecting and Disconnecting the Adapter

To connect the adapter to the trading system, the [AsyncMessageAdapter.ConnectAsync](#) method is called. It is passed the incoming [ConnectMessage](#) message. If the connection is successful, the adapter sends an outgoing [ConnectMessage](#) message.

```
public override async ValueTask ConnectAsync(ConnectMessage connectMsg, CancellationToken
cancellationToken)
{
    // Check the presence of keys for transactional mode
    if (this.IsTransactional())
    {
        if (Key.IsEmpty())
            throw new InvalidOperationException(LocalizedStrings.KeyNotSpecified);

        if (Secret.IsEmpty())
            throw new InvalidOperationException(LocalizedStrings.SecretNotSpecified);
    }

    // Initialize the authenticator
    _authenticator = new(this.IsTransactional(), Key, Secret, Passphrase);

    // Check that clients are not yet created
    if (_restClient != null)
        throw new InvalidOperationException(LocalizedStrings.NotDisconnectPrevTime);

    if (_socketClient != null)
        throw new InvalidOperationException(LocalizedStrings.NotDisconnectPrevTime);

    // Create REST client
    _restClient = new(_authenticator) { Parent = this };

    // Create and configure WebSocket client
```



```

_socketClient = new(_authenticator, ReConnectionSettings.ReAttemptCount) { Parent = this };
SubscribePusherClient();

// Connect WebSocket client
await _socketClient.Connect(cancellationToken);

// Send successful connection message
SendOutMessage(new ConnectMessage());
}

```

To disconnect the adapter from the trading system, the [AsyncMessageAdapter.DisconnectAsync](#) method is called. If the disconnection is successful, the adapter sends an outgoing [DisconnectMessage](#) message.

```

public override ValueTask DisconnectAsync(DisconnectMessage disconnectMsg, CancellationToken
cancellationToken)
{
    // Check that clients are created
    if (_restClient == null)
        throw new InvalidOperationException(LocalizedStrings.ConnectionNotOk);

    if (_socketClient == null)
        throw new InvalidOperationException(LocalizedStrings.ConnectionNotOk);

    // Free REST client resources
    _restClient.Dispose();
    _restClient = null;

    // Disconnect WebSocket client
    _socketClient.Disconnect();

    // Send disconnection message
    SendOutDisconnectMessage(true);
    return default;
}

```

In addition, the adapter provides the [AsyncMessageAdapter.ResetAsync](#) method to reset the state, which closes the connection and returns the adapter to its initial state.

```

public override ValueTask ResetAsync(ResetMessage resetMsg, CancellationToken cancellationToken)
{
    // Free REST client resources
    if (_restClient != null)
    {
        try
        {
            _restClient.Dispose();
        }
        catch (Exception ex)
        {
            SendOutError(ex);
        }
    }

    _restClient = null;

    // Disconnect and clear WebSocket client
    if (_socketClient != null)
    {
        try
        {
            UnsubscribePusherClient();
            _socketClient.Disconnect();
        }
        catch (Exception ex)
        {
            SendOutError(ex);
        }
    }
}

```

```

    }

    _socketClient = null;
}

// Free authenticator resources
if (_authenticator != null)
{
    try
    {
        _authenticator.Dispose();
    }
    catch (Exception ex)
    {
        SendOutError(ex);
    }

    _authenticator = null;
}

// Clear additional data
_candlesTransIds.Clear();

// Send reset message
SendOutMessage(new ResetMessage());
return default;
}

```

After Finishing Development

Once the connector is implemented, there are two options for using it:

1. Publish it to the [StockSharp Store](#) as either a paid or free product. In this case, users install the connector automatically through the [Installer](#).
2. For personal use, copy the built connector *.dll* file to the folder of your application (or any StockSharp product). At startup, the application scans the current directory for adapters using the following criteria:
 - Only files with the *.dll* extension whose names begin with StockSharp. are considered.
 - Each remaining file is checked to ensure it is a valid .NET assembly.
 - The assembly is loaded, and all types implementing *IMessageAdapter* are collected.
 - Any errors encountered during scanning or loading are written to the log and do not stop the search. If loading fails, open the application's log window or log file to view the error details.

This document describes the general principles of the adapter's operation, its creation, and management of the connection with the trading system. The following documents will be devoted to the implementation of the adapter's functionality:

- [Instrument Lookup](#)
- [Working with Market Data](#)
- [Requesting the Current State of the Portfolio and Orders](#)
- [Working with Trading Operations](#)
- [Storing Settings](#)
- [Extended Order Conditions](#)

Instrument Lookup

When creating your own adapter for working with an exchange, you need to implement the instrument lookup method. This method is called when sending a [SecurityLookupMessage](#) message and returns information about instruments through [SecurityMessage](#) messages.

Implementing the SecurityLookupAsync Method

The **SecurityLookupAsync** method usually performs the following actions:

1. Retrieves the list of supported instrument types from the incoming message.
2. Requests the list of instruments from the exchange via API.
3. For each received instrument, creates a [SecurityMessage](#) message, filling it with instrument data.
4. Checks if the instrument matches the search criteria.
5. Sends the created [SecurityMessage](#) message via the **SendOutMessageAsync** method.
6. After processing all instruments, sends a message about the completion of the search.

Below is an example of the implementation of the **SecurityLookupAsync** method based on the adapter for the Coinbase exchange. When creating your own adapter, you need to adapt this code to the API of the exchange being used.

```
public override async ValueTask SecurityLookupAsync(SecurityLookupMessage lookupMsg,
CancellationToken cancellationToken)
{
    // Get the list of instrument types to find
    var secTypes = lookupMsg.GetSecurityTypes();

    // Determine the maximum number of instruments to search for
    var left = lookupMsg.Count ?? long.MaxValue;

    // Iterate over the instrument types supported by the exchange
    foreach (var type in new[] { "SPOT", "FUTURE" })
    {
        // Request the list of instruments from the exchange
        var products = await _restClient.GetProducts(type, cancellationToken);

        foreach (var product in products)
        {
            // Create the instrument identifier
            var secId = product.ProductId.ToStockSharp();

            // Create a message with instrument information
            var secMsg = new SecurityMessage
            {
                SecurityType = product.ProductType.ToSecurityType(),
                SecurityId = secId,
                Name = product.DisplayName,
                PriceStep = product.QuoteIncrement?.ToDecimal(),
                VolumeStep = product.BaseIncrement?.ToDecimal(),
                MinVolume = product.BaseMinSize?.ToDecimal(),
                MaxVolume = product.BaseMaxSize?.ToDecimal(),
                ExpiryDate = product.FutureProductDetails?.ContractExpiry,
                Multiplier = product.FutureProductDetails?.ContractSize?.ToDecimal(),

                // It is necessary to fill in the subscription identifier
                // so that the external code can understand which subscription the data was received for
            }
        }
    }
}
```

```

    OriginalTransactionId = lookupMsg.TransactionId,
}
.TryFillUnderlyingId(product.BaseCurrencyId.ToUpperInvariant());

// Check if the instrument matches the search criteria
if (!secMsg.IsMatch(lookupMsg, secTypes))
    continue;

// Send a message with instrument information
await SendOutMessageAsync(secMsg, cancellationToken);

// Decrease the counter of remaining instruments
if (--left <= 0)
    break;
}

if (left <= 0)
    break;
}

// Send a message about the completion of the search
await SendSubscriptionResultAsync(lookupMsg, cancellationToken);
}

```

This method allows you to retrieve information about the instruments available on the exchange, including their main characteristics such as instrument type, minimum volume, price step, etc.

Market data

Market Data

When creating your own adapter for working with an exchange, you need to implement methods for subscribing to various types of market data. These methods are called when a [MarketDataMessage](#) message is received and provide for receiving and processing data from the exchange.

Schematically, the algorithm for processing a subscription or unsubscription request looks like this:

1. Sends a confirmation of receiving the subscription request using the [SendSubscriptionReplyAsync](#) method.
2. Checks whether the request is a subscription or unsubscription using the [MarketDataMessage.IsSubscribe](#) property.
3. In case of a subscription, sets up a subscription to receive real-time data via WebSocket or another mechanism (specific to each exchange).
4. In case of an unsubscription, cancels the corresponding subscription (specific to each exchange).
5. Sends a message about the subscription result using the [SendSubscriptionResultAsync](#) or [SendSubscriptionFinishedAsync](#) methods, depending on the subscription type and the operation result.

Candle Data

When implementing a subscription to candle data in your own adapter, it is important to consider the specifics of how a particular exchange works with this type of data. In the case of Coinbase, the following methods and properties were overridden:

Supported Timeframes

The `TimeFrames` property defines the list of timeframes supported by the adapter for candles. This allows StockSharp to know which timeframes can be requested through this adapter.

```
protected override IEnumerable<TimeSpan> TimeFrames { get; } = Extensions.TimeFrames.Keys.ToArray();
```

Support for Candle Updates

The `IsSupportCandlesUpdates` method determines whether the adapter supports real-time candle updates for a specific subscription request. In the case of Coinbase, only updates for 5-minute candles are supported.

```
private static readonly DataType _tf5min = DataType.TimeFrame(TimeSpan.FromMinutes(5));

public override bool IsSupportCandlesUpdates(MarketDataMessage subscription)
{
    // Coinbase only supports 5-minute candles for updating via websockets
    // Therefore, other timeframes will be built from ticks (automatically by the StockSharp core)
    return subscription.DataType2 == _tf5min;
}
```

Overriding these methods and properties allows the adapter to correctly handle requests for subscribing to candle data, taking into account the specifics of the Coinbase API. For example, if a timeframe other than 5 minutes is requested, StockSharp will know that it needs to use tick data to build candles of other timeframes.

Subscribing to Candle Data

To subscribe to candle data, the **OnTFCandlesSubscriptionAsync** method is implemented. This method, like the method for subscribing to tick data, can request historical data, as well as set up a subscription to receive new candles in real time.

```
protected override async ValueTask OnTFCandlesSubscriptionAsync(MarketDataMessage mdMsg,
Cancellation token cancellationToken)
{
    // Send confirmation of receiving the subscription request
    await SendSubscriptionReplyAsync(mdMsg.TransactionId, cancellationToken);

    var symbol = mdMsg.SecurityId.ToSymbol();

    if (mdMsg.IsSubscribe)
    {
        var tf = mdMsg.GetTimeFrame();

        // If historical data is requested
        if (mdMsg.From is not null)
        {
            var from = (long)mdMsg.From.Value.ToUnix();
            var to = (long)(mdMsg.To ?? DateTimeOffset.UtcNow).ToUnix();
            var left = mdMsg.Count ?? long.MaxValue;
            var step = (long)tf.Multiply(200).TotalSeconds;
            var granularity = mdMsg.GetTimeFrame().ToNative();

            while (from < to)
            {
                // Request historical candles
                var candles = await _restClient.GetCandles(symbol, from, from + step, granularity,
cancellationToken);
                var needBreak = true;
                var last = from;

                foreach (var candle in candles.OrderBy(t => t.Time))
                {
                    cancellationToken.ThrowIfCancellationRequested();

                    var time = (long)candle.Time.ToUnix();

                    if (time < from)
                        continue;

                    if (time > to)
                    {
                        needBreak = true;
                        break;
                    }

                    // Send information about each historical candle
                    await SendOutMessageAsync(new TimeFrameCandleMessage
                    {
                        OpenPrice = (decimal)candle.Open,
                        ClosePrice = (decimal)candle.Close,
                        HighPrice = (decimal)candle.High,
                        LowPrice = (decimal)candle.Low,
                        TotalVolume = (decimal)candle.Volume,
                        OpenTime = candle.Time,
                        State = CandleStates.Finished,

                        // In case of identifying data by subscription, filling instrument information is not required
                        OriginalTransactionId = mdMsg.TransactionId,
                    }, cancellationToken);

                    if (--left <= 0)
                    {
                        needBreak = true;
                        break;
                    }
                }
            }
        }
    }
}
```

```

        last = time;
        needBreak = false;
    }

    if (needBreak)
        break;

    from = last;
}
}

if (!mdMsg.IsHistoryOnly() && mdMsg.DataType2 == _tf5min)
{
    // Subscribe to receive new candles in real time
    _candlesTransIds[symbol] = mdMsg.TransactionId;
    await _socketClient.SubscribeCandles(symbol, cancellationToken);

    // Notify that the subscription has transitioned to online status
    await SendSubscriptionResultAsync(mdMsg, cancellationToken);
}
else
{
    // Send a response that the subscription is finished (not online)
    await SendSubscriptionFinishedAsync(mdMsg.TransactionId, cancellationToken);
}
}
else
{
    // Unsubscribe from receiving candles
    _candlesTransIds.Remove(symbol);
    await _socketClient.UnSubscribeCandles(symbol, cancellationToken);
}
}

```

Processing Candle Data

To process candle data received from the exchange in real time, a method with code like in the **SessionOnCandleReceived** method is usually implemented. This method converts the received data into a **TimeFrameCandleMessage** message and sends it using the **SendOutMessageAsync** method.

```

private async ValueTask SessionOnCandleReceived(Ohlc candle, CancellationToken cancellationToken)
{
    // Check if there is an active subscription to candles for this instrument
    if (!_candlesTransIds.TryGetValue(candle.Symbol, out var transId))
        return;

    // Create and send a message about a new candle
    await SendOutMessageAsync(new TimeFrameCandleMessage
    {
        OpenPrice = (decimal)candle.Open,
        ClosePrice = (decimal)candle.Close,
        HighPrice = (decimal)candle.High,
        LowPrice = (decimal)candle.Low,
        TotalVolume = (decimal)candle.Volume,
        OpenTime = candle.Time,
        State = CandleStates.Active, // The candle is considered active as it may still change

        // In case of identifying data by subscription, filling instrument information is not required
        OriginalTransactionId = transId,
    }, cancellationToken);
}

```

Level 1 (Best Bid and Ask Prices, Last Price)

Subscribing to Level 1 Data

To subscribe to Level 1 changes, the **OnLevel1SubscriptionAsync** method is implemented. This method usually performs the following actions:

```
protected override async ValueTask OnLevel1SubscriptionAsync(MarketDataMessage mdMsg,
Cancellation token cancellationToken)
{
    // Send confirmation of receiving the subscription request
    // This informs the system that the request has been received and is being processed
    await SendSubscriptionReplyAsync(mdMsg.TransactionId, cancellationToken);

    // Convert the instrument identifier to a symbol understood by the exchange
    var symbol = mdMsg.SecurityId.ToSymbol();

    if (mdMsg.IsSubscribe)
    {
        // If this is a subscription request
        // Subscribe to receive Level 1 data via WebSocket
        await _socketClient.SubscribeTicker(symbol, cancellationToken);

        // Send a message about successful subscription
        // This informs the system that the subscription is set up and data will be received
        await SendSubscriptionResultAsync(mdMsg, cancellationToken);
    }
    else
    {
        // If this is an unsubscription request
        // Cancel the subscription to receive Level 1 data
        await _socketClient.UnSubscribeTicker(symbol, cancellationToken);
    }
}
```

Processing Level 1 Data

To process Level 1 data received from the exchange in real time, a method with code like in the **SessionOnTickerChanged** example is usually implemented. This method converts the received data into a [Level1ChangeMessage](#) message and sends it using the **SendOutMessageAsync** method.

```
private async ValueTask SessionOnTickerChanged(Ticker ticker, Cancellation token cancellationToken)
{
    // Create a message with Level 1 data changes
    await SendOutMessageAsync(new Level1ChangeMessage
    {
        // Specify the instrument identifier
        SecurityId = ticker.Product.ToStockSharp(),
        // Set the time of receiving data
        ServerTime = CurrentTime.ConvertToUtc(),
    })
    // Add various Level 1 fields if they are present in the data from the exchange
    .TryAdd(Level1Fields.LastTradeId, ticker.LastTradeId)
    .TryAdd(Level1Fields.LastTradePrice, ticker.LastTradePrice?.ToDecimal())
    .TryAdd(Level1Fields.LastTradeVolume, ticker.LastTradePrice?.ToDecimal())
    .TryAdd(Level1Fields.LastTradeOrigin, ticker.LastTradeSide?.ToSide())
    .TryAdd(Level1Fields.HighPrice, ticker.High?.ToDecimal())
    .TryAdd(Level1Fields.LowPrice, ticker.Low?.ToDecimal())
    .TryAdd(Level1Fields.Volume, ticker.Volume?.ToDecimal())
    .TryAdd(Level1Fields.Change, ticker.Change?.ToDecimal())
    .TryAdd(Level1Fields.BestBidPrice, ticker.Bid?.ToDecimal())
    .TryAdd(Level1Fields.BestAskPrice, ticker.Ask?.ToDecimal())
    .TryAdd(Level1Fields.BestBidVolume, ticker.BidSize?.ToDecimal())
    .TryAdd(Level1Fields.BestAskVolume, ticker.AskSize?.ToDecimal())
    , cancellationToken);
}
```

Order Book

Support for Incremental Order Book Updates

When implementing order book functionality in your own adapter, it is important to consider whether the exchange supports incremental order book updates. For this, the `IsSupportOrderBookIncrements` property was overridden in the Coinbase adapter:

```
public override bool IsSupportOrderBookIncrements => true;
```

The `IsSupportOrderBookIncrements` property indicates whether the adapter supports incremental order book updates. Setting this property to true means that the exchange can send partial order book updates rather than a full snapshot with each change.

Overriding this property allows StockSharp to optimize the processing of order book data. If the property is set to true, the system will expect and correctly handle incremental updates.

Subscribing to Order Book Data

To subscribe to order book changes, the `OnMarketDepthSubscriptionAsync` method is implemented. This method performs actions similar to the `OnLevel1SubscriptionAsync` method, but for order book data.

```
protected override async ValueTask OnMarketDepthSubscriptionAsync(MarketDataMessage mdMsg,
    CancellationToken cancellationToken)
{
    // Send confirmation of receiving the subscription request
    await SendSubscriptionReplyAsync(mdMsg.TransactionId, cancellationToken);

    // Convert the instrument identifier to a symbol understood by the exchange
    var symbol = mdMsg.SecurityId.ToSymbol();

    if (mdMsg.IsSubscribe)
    {
        // If this is a subscription request
        // Subscribe to receive order book data via WebSocket
        await _socketClient.SubscribeOrderBook(symbol, cancellationToken);

        // Send a message about successful subscription
        await SendSubscriptionResultAsync(mdMsg, cancellationToken);
    }
    else
    {
        // If this is an unsubscription request
        // Cancel the subscription to receive order book data
        await _socketClient.UnSubscribeOrderBook(symbol, cancellationToken);
    }
}
```

Processing Order Book Data

To process order book data received from the exchange in real time, a method with code like in the `SessionOnOrderBookReceived` method is usually implemented. This method converts the received data into a [QuoteChangeMessage](#) message and sends it using the `SendOutMessageAsync` method.

```
private async ValueTask SessionOnOrderBookReceived(string type, string symbol,
    IEnumerable<OrderBookChange> changes, CancellationToken cancellationToken)
{
    var bids = new List<QuoteChange>();
    var asks = new List<QuoteChange>();

    // Distribute changes by bids and asks
    foreach (var change in changes)
    {
        var side = change.Side.ToSide();
```

```

var quotes = side == Sides.Buy ? bids : asks;
quotes.Add(new((decimal)change.Price, (decimal)change.Size));
}

// Create and send a message with changes in the order book
await SendOutMessageAsync(new QuoteChangeMessage
{
    SecurityId = symbol.ToStockSharp(),
    Bids = bids.ToArray(),
    Asks = asks.ToArray(),
    ServerTime = CurrentTime.ConvertToUtc(),

    // Determine if this is a full order book snapshot or an incremental update.
    // If the exchange always sends only full order books and does not support incrementality,
    // then setting this property is not required at all
    State = type == "snapshot" ? QuoteChangeStates.SnapshotComplete : QuoteChangeStates.Increment,
}, cancellationToken);
}

```

Tick Data (Trades)

Subscribing to Tick Data

To subscribe to tick data, the **OnTicksSubscriptionAsync** method is implemented. This method, in addition to actions similar to the previous subscription methods, can also request historical data if specified in the request.

```

protected override async ValueTask OnTicksSubscriptionAsync(MarketDataMessage mdMsg,
Cancellation token cancellationToken)
{
    // Send confirmation of receiving the subscription request
    await SendSubscriptionReplyAsync(mdMsg.TransactionId, cancellationToken);

    var symbol = mdMsg.SecurityId.ToSymbol();

    if (mdMsg.IsSubscribe)
    {
        // If historical data is requested
        if (mdMsg.From is not null)
        {
            var from = (long)mdMsg.From.Value.ToUnix(false);
            var to = (long)(mdMsg.To ?? DateTimeOffset.UtcNow).ToUnix(false);
            var left = mdMsg.Count ?? long.MaxValue;

            while (from < to)
            {
                // Request historical trades
                var trades = await _restClient.GetTrades(symbol, from, to, cancellationToken);
                var needBreak = true;
                var last = from;

                foreach (var trade in trades.OrderBy(t => t.Time))
                {
                    cancellationToken.ThrowIfCancellationRequested();

                    var time = (long)trade.Time.ToUnix();

                    if (time < from)
                        continue;

                    if (time > to)
                    {
                        needBreak = true;
                        break;
                    }
                }
            }
        }
    }
}

```

```

// Send information about each historical trade
await SendOutMessageAsync(new ExecutionMessage
{
    // Set that the message carries information about a tick trade
    // (not a transaction like an order or own trade)
    DataTypeEx = DataType.Ticks,

    TradeId = trade.TradeId,
    TradePrice = trade.Price?.ToDecimal(),
    TradeVolume = trade.Size?.ToDecimal(),
    ServerTime = trade.Time,
    OriginSide = trade.Side.ToSide(),

    // For history, always set the subscription identifier,
    // so that the external code can understand which subscription the data was received for.
    // In case of identifying data by subscription, filling instrument information is not required
    OriginalTransactionId = mdMsg.TransactionId,
}, cancellationTokens);

if (--left <= 0)
{
    needBreak = true;
    break;
}

last = time;
needBreak = false;
}

if (needBreak)
    break;

from = last;
}
}

if (!mdMsg.IsHistoryOnly())
{
    // Subscribe to receive new trades in real time
    await _socketClient.SubscribeTrades(symbol, cancellationTokens);
}

// Send a message about successful subscription
await SendSubscriptionResultAsync(mdMsg, cancellationTokens);
}
else
{
    // Unsubscribe from receiving trades in real time
    await _socketClient.UnSubscribeTrades(symbol, cancellationTokens);
}
}
}

```

Processing Tick Data

To process tick data received from the exchange in real time, a method with code like in the **SessionOnTradeReceived** method is usually implemented. This method converts the received data into an [ExecutionMessage](#) message with the [DataType.Ticks](#) type and sends it using the `SendOutMessageAsync` method.

```

private async ValueTask SessionOnTradeReceived(Trade trade, CancellationToken cancellationTokens)
{
    // Create and send a message about a new trade
    await SendOutMessageAsync(new ExecutionMessage
    {
        // Set that the message carries information about a tick trade
        // (not a transaction like an order or own trade)

```

```

        DataTypeEx = DataType.Ticks,

        SecurityId = trade.ProductId.ToStockSharp(),
        TradeId = trade.TradeId,
        TradePrice = (decimal)trade.Price,
        TradeVolume = (decimal)trade.Size,
        ServerTime = trade.Time,
        OriginSide = trade.Side.ToSide(),
    }, cancellationToken);
}

```

Subscribing to Order Log

The order log is detailed information about all changes in the order book, including the addition, modification, and deletion of orders. This data is specific and not provided by all data sources. For example, Coinbase does not support providing an order log.

To implement a subscription to an order log in an adapter, the **OnOrderLogSubscriptionAsync** method is used. This method is called when a [MarketDataMessage](#) message with the [DataType.OrderLog](#) data type is received.

Below is an example implementation of this method taken from the [BitStamp](#) connector, which supports order log:

```

protected override async ValueTask OnOrderLogSubscriptionAsync(MarketDataMessage mdMsg,
    CancellationToken cancellationToken)
{
    // Send confirmation of receiving the subscription request
    await SendSubscriptionReplyAsync(mdMsg.TransactionId, cancellationToken);

    // Convert the instrument identifier to a currency pair
    var symbol = mdMsg.SecurityId.ToCurrency();

    if (mdMsg.IsSubscribe)
    {
        if (!mdMsg.IsHistoryOnly())
        {
            // Subscribe to receive order log in real time
            await _pusherClient.SubscribeOrderLog(symbol, cancellationToken);
        }

        // Send a message about successful subscription
        await SendSubscriptionResultAsync(mdMsg, cancellationToken);
    }
    else
    {
        // Unsubscribe from receiving order log
        await _pusherClient.UnSubscribeOrderLog(symbol, cancellationToken);
    }
}

```

When processing order log data received from the exchange, a separate method is usually used, which converts the received data into [ExecutionMessage](#) messages with the [ExecutionTypes.OrderLog](#) type:

```

private async ValueTask SessionOnNewOrderLog(string symbol, OrderStates state, Order order,
    CancellationToken cancellationToken)
{
    // Create and send a message with information about a new entry in the order log
    await SendOutMessageAsync(new ExecutionMessage
    {
        DataTypeEx = DataType.OrderLog,
        SecurityId = symbol.ToStockSharp(),
        ServerTime = order.Time,
        OrderVolume = (decimal)order.Amount,
        OrderPrice = (decimal)order.Price,
        OrderId = order.Id,
    }, cancellationToken);
}

```

```
Side = order.Type.ToSide(),
OrderState = state,
}, cancellationToken);
}
```

It is important not to forget to add support for this data type in the adapter constructor:

```
this.AddSupportedMarketDataType(DataType.OrderLog);
```

Specifics of Processing Historical and Live Data

When implementing requests for historical data and processing live data in your own adapter, it is important to consider the following points:

Historical Data

When sending historical data in response to a request:

1. Setting [OriginalTransactionId](#) is mandatory. This allows the system to associate the received data with the original request.
2. Setting [SecurityId](#) or [TimeFrameCandleMessage.TimeFrame](#) (in the case of candles) is not required, but also not prohibited. The StockSharp core will automatically fill these fields with the necessary values from the original request.

Live Data

When processing live data, for example, received via WebSocket:

1. Setting [OriginalTransactionId](#) is optional. If the transaction ID is not set, the system will distribute the data to all active subscriptions for the corresponding instrument and data type.
2. Setting [SecurityId](#) and other specific fields (for example, [TimeFrameCandleMessage.TimeFrame](#) for candles) is mandatory, as this information is necessary for the correct routing of data in the system.

Positions and orders info

Information about Portfolios and Orders

When creating your own adapter for working with an exchange, it is necessary to implement methods for requesting the current state of the portfolio and orders. These methods are called when receiving [PortfolioLookupMessage](#) and [OrderStatusMessage](#) messages respectively.

Requesting Portfolio State

To request the portfolio state, the **PortfolioLookupAsync** method is implemented. This method usually performs the following actions:

1. Sends a confirmation of receiving the request using [SendSubscriptionReplyAsync](#).
2. Checks whether the request is a subscription or unsubscription using the [IsSubscribe](#) property.
3. In case of a subscription:
 - Sends a [PortfolioMessage](#) message with information about the portfolio.
 - Requests the current account balances from the exchange.
 - For each account, creates and sends a [PositionChangeMessage](#) message with information about the position.
4. Sends a message about the subscription result using [SendSubscriptionResultAsync](#).

```
public override async ValueTask PortfolioLookupAsync(PortfolioLookupMessage lookupMsg,
CancellationToken cancellationToken)
{
    var transId = lookupMsg.TransactionId;

    // Send confirmation of receiving the request
    await SendSubscriptionReplyAsync(transId, cancellationToken);

    if (!lookupMsg.IsSubscribe)
        return;

    // Send a message with information about the portfolio
    await SendOutMessageAsync(new PortfolioMessage
    {
        PortfolioName = PortfolioName,
        BoardCode = BoardCodes.Coinbase,
        OriginalTransactionId = transId,
    }, cancellationToken);

    // Request current account balances
    var accounts = await _restClient.GetAccounts(cancellationToken);

    foreach (var account in accounts)
    {
        // For each account, create and send a message with information about the position
        await SendOutMessageAsync(new PositionChangeMessage
        {
            PortfolioName = PortfolioName,
            SecurityId = new SecurityId
            {
                SecurityCode = account.Currency,
                BoardCode = BoardCodes.Coinbase,
            },
            ServerTime = CurrentTime.ConvertToUtc(),
        }, cancellationToken);
    }
}
```

```

        .TryAdd(PositionChangeTypes.CurrentValue, (decimal)account.Available, true)
        .TryAdd(PositionChangeTypes.BlockedValue, (decimal)account.Hold, true), cancellationToken);
    }

    // Send a message about successful completion of the subscription
    await SendSubscriptionResultAsync(lookupMsg, cancellationToken);
}

```

Requesting Orders State

To request the orders state, the **OrderStatusAsync** method is implemented. This method usually performs the following actions:

1. Sends a confirmation of receiving the request using [SendSubscriptionReplyAsync](#).
2. Checks whether the request is a subscription or unsubscription using the [OrderStatusMessage.IsSubscribe](#) property.
3. In case of a subscription:
 - Requests the list of current orders from the exchange.
 - For each order, creates and sends an [ExecutionMessage](#) message with information about the order.
 - If necessary, sets up a subscription to receive order updates in real time.
4. Sends a message about the subscription result using [SendSubscriptionResultAsync](#).

```

public override async ValueTask OrderStatusAsync(OrderStatusMessage statusMsg, CancellationToken
cancellationToken)
{
    // Send confirmation of receiving the request
    await SendSubscriptionReplyAsync(statusMsg.TransactionId, cancellationToken);

    if (!statusMsg.IsSubscribe)
        return;

    // Request the list of current orders
    var orders = await _restClient.GetOrders(cancellationToken);

    foreach (var order in orders)
        await ProcessOrder(order, statusMsg.TransactionId, cancellationToken);

    if (!statusMsg.IsHistoryOnly())
    {
        // Set up a subscription to receive order updates in real time
        await _socketClient.SubscribeOrders(cancellationToken);
    }

    // Send a message about successful completion of the subscription
    await SendSubscriptionResultAsync(statusMsg, cancellationToken);
}

private async ValueTask ProcessOrder(Order order, long originTransId, CancellationToken
cancellationToken)
{
    if (!long.TryParse(order.ClientOrderId, out var transId))
        return;

    var state = order.Status.ToOrderState();

    // Create and send a message with information about the order
    await SendOutMessageAsync(new ExecutionMessage
    {
        ServerTime = originTransId == 0 ? CurrentTime.ConvertToUtc() : order.CreationTime,
        DataTypeEx = DataType.Transactions,
    });
}

```

```

SecurityId = order.Product.ToStockSharp(),
TransactionId = originTransId == 0 ? 0 : transId,
OriginalTransactionId = originTransId,
OrderState = state,
Error = state == OrderStates.Failed ? new InvalidOperationException() : null,
OrderType = order.Type.ToOrderType(),
Side = order.Side.ToSide(),
OrderStringId = order.Id,
OrderPrice = order.Price?.ToDecimal() ?? 0,
OrderVolume = order.Size?.ToDecimal(),
TimeInForce = order.TimeInForce.ToTimeInForce(),
Balance = (decimal?)order.LeavesQuantity,
HasOrderInfo = true,
}, cancellationToken);
}

```

Processing Real-Time Updates

To process real-time order state updates, a separate method is usually implemented, which is called when receiving corresponding events from the WebSocket client:

```

private async ValueTask SessionOnOrderReceived(Order order, Cancellation_token cancellationToken)
{
    // Process the received order update
    // OriginTransId = 0, since this is a real-time update, not a response to a specific request
    await ProcessOrder(order, 0, cancellationToken);
}

```


Trading Operations

When creating your own adapter for working with an exchange, it is necessary to implement methods for performing trading operations, such as registering, replacing, and canceling orders. These methods are called when receiving the corresponding messages from the StockSharp core.

Order Registration

To register a new order, the **RegisterOrderAsync** method is implemented. This method is called when receiving the [OrderRegisterMessage](#) message.

The main steps when registering an order:

1. Checking the order type and additional conditions.
2. Converting order parameters to a format understood by the exchange.
3. Sending a request to register the order through the exchange API.
4. Processing the response from the exchange and sending the corresponding [ExecutionMessage](#) message.

```
public override async ValueTask RegisterOrderAsync(OrderRegisterMessage regMsg, CancellationToken
cancellationTokn)
{
    var condition = (CoinbaseOrderCondition)regMsg.Condition;

    switch (regMsg.OrderType)
    {
        case null:
        case OrderTypes.Limit:
        case OrderTypes.Market:
            break;
        case OrderTypes.Conditional:
        {
            // Handling conditional orders, for example, withdrawal of funds
            if (!condition.IsWithdraw)
                break;

            var withdrawId = await _restClient.Withdraw(regMsg.SecurityId.SecurityCode, regMsg.Volume,
condition.WithdrawInfo, cancellationTokn);

            await SendOutMessageAsync(new ExecutionMessage
            {
                DataTypeEx = DataType.Transactions,
                OrderStringId = withdrawId,
                ServerTime = CurrentTime.ConvertToUtc(),
                OriginalTransactionId = regMsg.TransactionId,
                OrderState = OrderStates.Done,
                HasOrderInfo = true,
            }, cancellationTokn);

            await PortfolioLookupAsync(null, cancellationTokn);
            return;
        }
        default:
            throw new NotSupportedException(LocalizedStrings.OrderUnsupportedType.Put(regMsg.OrderType,
regMsg.TransactionId));
    }

    // Determining the order type (market or limit)
```

```

var isMarket = regMsg.OrderType == OrderTypes.Market;
var price = isMarket ? (decimal?)null : regMsg.Price;

// Sending the order to the exchange
var result = await _restClient.RegisterOrder(
    regMsg.TransactionId.To<string>(), regMsg.SecurityId.ToSymbol(),
    regMsg.OrderType.ToNative(), regMsg.Side.ToNative(), price,
    condition?.StopPrice, regMsg.Volume, regMsg.TimeInForce,
    regMsg.TillDate.EnsureToday(), regMsg.Leverage, cancellationToken);

var orderState = result.Status.ToOrderState();

// Processing the order registration result
if (orderState == OrderStates.Failed)
{
    await SendOutMessageAsync(new ExecutionMessage
    {
        DataTypeEx = DataType.Transactions,
        ServerTime = result.CreationTime,
        OriginalTransactionId = regMsg.TransactionId,
        OrderState = OrderStates.Failed,
        Error = new InvalidOperationException(),
        HasOrderInfo = true,
    }, cancellationToken);
}
}

```

Order Replacement

To replace an existing order, the **ReplaceOrderAsync** method is implemented. This method is called when receiving the [OrderReplaceMessage](#) message.

The main steps when replacing an order:

1. Checking the possibility of replacing the order on the exchange.
2. Sending a request to replace the order through the exchange API.
3. Processing the response from the exchange and sending the corresponding [ExecutionMessage](#) message.

```

public override async ValueTask ReplaceOrderAsync(OrderReplaceMessage replaceMsg, CancellationToken
cancellationToken)
{
    // Sending a request to replace the order
    await _restClient.EditOrder(
        replaceMsg.OldOrderId.To<string>(),
        replaceMsg.Price,
        replaceMsg.Volume,
        cancellationToken);

    // Note: Processing the order replacement result usually occurs
    // in a separate method that is called when receiving an update from the exchange
}

```

Specifics of Order Replacement

When implementing the order replacement method, it is important to consider the specifics of the exchange protocol. For this, StockSharp provides the [MessageAdapter.IsReplaceCommandEditCurrent](#) property.

If the exchange protocol assumes changing an order while retaining its old identifier, it is necessary to override this property and return true. This indicates to StockSharp that when replacing an order, it is not necessary to expect a new identifier from the exchange.

```
public override bool IsReplaceCommandEditCurrent => true;
```

If, when changing an order, the old one is canceled and a new one is registered with a new exchange identifier, then this property does not need to be overridden. By default, it returns `false`, which corresponds to the behavior of most exchanges.

Order Cancellation

To cancel an existing order, the **CancelOrderAsync** method is implemented. This method is called when receiving the [OrderCancelMessage](#) message.

The main steps when canceling an order:

1. Checking the presence of the order identifier.
2. Sending a request to cancel the order through the exchange API.
3. Processing the response from the exchange and sending the corresponding [ExecutionMessage](#) message.

```
public override async ValueTask CancelOrderAsync(OrderCancelMessage cancelMsg, CancellationTokens cancellationTokens)
{
    // Checking the presence of the order identifier
    if (cancelMsg.OrderStringId.IsNullOrEmpty())
        throw new
InvalidOperationException(LocalizedStrings.OrderNoExchangeId.Put(cancelMsg.OriginalTransactionId));

    // Sending a request to cancel the order
    await _restClient.CancelOrder(cancelMsg.OrderStringId, cancellationTokens);

    // Note: Processing the order cancellation result usually occurs
    // in a separate method that is called when receiving an update from the exchange
}
```

Mass Order Cancellation

Some exchanges support the mass order cancellation function, which allows canceling multiple or all active orders with a single request. This can be useful for quickly closing positions or clearing the order book under certain market conditions.

To implement mass order cancellation in the adapter, the **CancelOrderGroupAsync** method is usually used. This method is called when receiving the [OrderGroupCancelMessage](#) message.

It is worth noting that not all exchanges support this function. For example, Coinbase does not provide an API for mass order cancellation. In such cases, it may be necessary to implement sequential cancellation of individual orders.

Below is an example of the implementation of the mass order cancellation method, taken from the [BitStamp](#) connector, which supports this function:

```
public override async ValueTask CancelOrderGroupAsync(OrderGroupCancelMessage cancelMsg,
CancellationTokens cancellationTokens)
{
    await _httpClient.CancelAllOrders(cancellationTokens);
}
```

It is important not to forget to remove the deletion of support for this command type from the adapter constructor:

```
//this.RemoveSupportedMessage(MessageTypes.OrderGroupCancel);
```

Tracking Order State

In the case of Coinbase, as well as some other modern exchanges, order state updates are broadcast through a WebSocket connection. This means that after performing trading operations (registration, replacement, or cancellation of an order), there is no need to immediately request the new order state through the REST API. Instead, the adapter will receive updates automatically through the established WebSocket connection.

Processing these updates occurs in a method similar to `SessionOnOrderReceived`, which was discussed in the section on [requesting the current state of the portfolio and orders](#). This method is called every time the exchange sends an update about the order state, regardless of whether this update was triggered by user actions or changes on the exchange itself.

This approach allows for more efficient tracking of order states, reduces the load on the exchange API, and ensures receiving updates in real time. However, when implementing your own adapter, it is necessary to carefully study the API documentation of the exchange being used in order to properly configure and handle these WebSocket updates.

Error Handling

When performing trading operations, it is important to correctly handle possible errors and exceptions. In case of an error, it is necessary to send an `ExecutionMessage` message with the `Error` property set.

Implementation Specifics

When implementing methods for working with trading operations, it is necessary to take into account the specifics of a particular exchange:

- Supported order types (market, limit, stop orders, etc.).
- Format of order identifiers.
- Specifics of the exchange API for working with orders.
- Possible restrictions on the frequency of sending requests.

Settings storage

Storing Settings

When creating your own adapter, it is important to provide the ability to save and load settings. For this, StockSharp uses the Save and Load methods, which work with the `SettingsStorage` object.

To save settings, you need to override the Save method:

```
public override void Save(SettingsStorage storage)
{
    base.Save(storage);

    storage.SetValue(nameof(Key), Key);
    storage.SetValue(nameof(Secret), Secret);
    storage.SetValue(nameof(Passphrase), Passphrase);
}
```

To load settings, you need to override the Load method:

```
public override void Load(SettingsStorage storage)
{
    base.Load(storage);

    Key = storage.GetValue<SecureString>(nameof(Key));
    Secret = storage.GetValue<SecureString>(nameof(Secret));
    Passphrase = storage.GetValue<SecureString>(nameof(Passphrase));
}
```

Basic Interfaces

An adapter can implement various basic interfaces that allow standardizing some settings. Although the implementation of these interfaces is not mandatory, it can simplify working with the adapter, as the properties of the basic interfaces are displayed in the basic mode in the connection editing window.

Examples of basic interfaces:

1. [IKeySecretAdapter](#) - for adapters that require a key and secret for authentication.
2. [ILoginPasswordAdapter](#) - for adapters that use a login and password.
3. [ITokenAdapter](#) - for adapters that use an authentication token.
4. [IPassphraseAdapter](#) - for adapters that require a password or passphrase.
5. [IDemoAdapter](#) - for adapters that support demo mode.
6. [IAddressAdapterMarkdig.Syntax.Inlines.HtmlInline](#) - for adapters that require specifying a server address.
7. [ISenderTargetAdapter](#) - for adapters that use sender and recipient identifiers.

When implementing these interfaces, you need to add the corresponding properties:

```
/// <inheritdoc />
[Display(
    ResourceType = typeof(LocalizedStrings),
    Name = LocalizedStrings.KeyKey,
    Description = LocalizedStrings.KeyKey + LocalizedStrings.Dot,
    GroupName = LocalizedStrings.ConnectionKey,
    Order = 0)]
```

```

public SecureString Key { get; set; }

/// <inheritdoc />
[Display(
    ResourceType = typeof(LocalizedStrings),
    Name = LocalizedStrings.SecretKey,
    Description = LocalizedStrings.SecretDescKey,
    GroupName = LocalizedStrings.ConnectionKey,
    Order = 1)]
public SecureString Secret { get; set; }

/// <summary>
/// Passphrase.
/// </summary>
[Display(
    ResourceType = typeof(LocalizedStrings),
    Name = LocalizedStrings.PassphraseKey,
    Description = LocalizedStrings.PassphraseKey + LocalizedStrings.Dot,
    GroupName = LocalizedStrings.ConnectionKey,
    Order = 2)]
public SecureString Passphrase { get; set; }

```

Attributes for Display Configuration

To improve the user experience when working with adapter settings, it is recommended to use attributes from the `System.ComponentModel.DataAnnotations` namespace. These attributes allow you to set the display name, description, group, and display order for each property.

```

[MediaIcon("Coinbase_logo.svg")]
[Doc("topics/api/connectors/crypto_exchanges/coinbase.html")]
[Display(
    ResourceType = typeof(LocalizedStrings),
    Name = LocalizedStrings.CoinbaseKey,
    Description = LocalizedStrings.CryptoConnectorKey,
    GroupName = LocalizedStrings.CryptocurrencyKey)]
[MessageAdapterCategory(MessageAdapterCategories.Crypto | MessageAdapterCategories.RealTime |
MessageAdapterCategories.OrderLog |
MessageAdapterCategories.Free | MessageAdapterCategories.Level1 |
MessageAdapterCategories.Transactions)]
public partial class CoinbaseMessageAdapter : MessageAdapter, IKeySecretAdapter, IPassphraseAdapter

```

Correct implementation of settings storage and the use of basic interfaces allows creating a more user-friendly and understandable adapter that is easily integrated into the StockSharp ecosystem.

Extended order condition

Extended Order Conditions

When working with some exchanges or trading systems, standard fields for registering an order may not be enough. For example, when it is required:

1. When registering [stop orders](#).
2. When it is necessary to specify additional properties to set custom order rules.

StockSharp provides a flexible system for working with such extended order conditions.

Base Class OrderCondition

[OrderCondition](#) is an abstract base class for all order conditions. It provides the basic functionality:

- Parameters dictionary for storing additional order parameters
- Clone() method for creating a copy of the condition
- Overridden ToString() method for convenient output of information about the condition

```
public class MyOrderCondition : OrderCondition
{
    public decimal? SpecialPrice
    {
        get => (decimal?)Parameters[nameof(SpecialPrice)];
        set => Parameters[nameof(SpecialPrice)] = value;
    }
}
```

Specialized Interfaces

StockSharp defines several interfaces for specific types of order conditions:

- [ITakeProfitOrderCondition](#) - for orders with Take-Profit condition
- [IStopLossOrderCondition](#) - for orders with Stop-Loss condition
- [IWithdrawOrderCondition](#) - for withdrawal orders
- [IRepoOrderCondition](#) - for REPO orders
- [INtmOrderCondition](#) - for orders of the negotiated deals mode (NDM)

BaseWithdrawOrderCondition

[BaseWithdrawOrderCondition](#) is a base class for order conditions that support funds withdrawal. It implements the [IWithdrawOrderCondition](#) interface and contains fields for the withdrawal transaction.

```
public class MyWithdrawCondition : BaseWithdrawOrderCondition
{
    public string DestinationAddress
    {
        get => (string)Parameters[nameof(DestinationAddress)];
        set => Parameters[nameof(DestinationAddress)] = value;
    }
}
```

CoinbaseOrderCondition

The `CoinbaseOrderCondition` class is inherited from `BaseWithdrawOrderCondition`, since Coinbase supports programmatic asset withdrawal. In addition, it implements the `IStopLossOrderCondition` interface, which allows it to be used for stop-loss orders.

```
[Serializable]
[DataContract]
[Display(ResourceType = typeof(LocalizedStrings), Name = LocalizedStrings.CoinbaseKey)]
public class CoinbaseOrderCondition : BaseWithdrawOrderCondition, IStopLossOrderCondition
{
    /// <summary>
    /// Initializes a new instance of the <see cref="CoinbaseOrderCondition"/>.
    /// </summary>
    public CoinbaseOrderCondition()
    {
    }

    /// <summary>
    /// Activation price, when reached an order will be placed.
    /// </summary>
    [DataMember]
    [Display(
        ResourceType = typeof(LocalizedStrings),
        Name = LocalizedStrings.StopPriceKey,
        Description = LocalizedStrings.StopPriceDescKey,
        GroupName = LocalizedStrings.StopLossKey,
        Order = 0)]
    public decimal? StopPrice
    {
        get => (decimal?)Parameters.TryGetValue(nameof(StopPrice));
        set => Parameters[nameof(StopPrice)] = value;
    }

    decimal? IStopLossOrderCondition.ClosePositionPrice { get; set; }

    decimal? IStopLossOrderCondition.ActivationPrice
    {
        get => StopPrice;
        set => StopPrice = value;
    }

    bool IStopLossOrderCondition.IsTrailing
    {
        get => false;
        set { }
    }
}
```

Usage in the Adapter

When developing your own adapter, you can create your own class of order conditions by inheriting it from `OrderCondition` or one of its descendants and implementing the necessary interfaces. This will allow you to add support for parameters specific to your exchange.

To specify the type of order condition supported by the adapter, the [OrderConditionAttribute](#) attribute is used.

```
[OrderCondition(typeof(CoinbaseOrderCondition))]
public partial class CoinbaseMessageAdapter
```

Such an approach provides flexibility when working with various exchanges and their unique requirements for order parameters while maintaining uniformity within the StockSharp architecture.

Message list

Message List

A message carries information about market data, transactions, or a command. Below is a list of the main messages used in connectors.

Message	Description
Message	An abstract message class from which other message classes inherit. Contains information about the time, type MessageTypes , and the adapter that created the message.
BoardMessage	Contains information about the electronic trading board.
CandleMessage	An abstract message class that contains general information about a candle. This class is inherited by message classes for specific candle types: TimeFrameCandleMessage , TickCandleMessage , VolumeCandleMessage , RangeCandleMessage , PnFCandleMessage , and RenkoCandleMessage .
ConnectMessage	Used as a command to establish a connection, as well as for an incoming message about a successful connection or a connection error.
DisconnectMessage	Used as a command to break a connection, as well as for an incoming message about a connection break.
ErrorMessage	An error message.
ExecutionMessage	Description see below.
Level1ChangeMessage	The message contains information about the value of a Level1 field of a certain type. The available field types are defined in the Level1Fields enumeration.
NewsMessage	The message contains information about news.
OrderCancelMessage	The message contains information for canceling an order. There is also a message OrderGroupCancelMessage , which contains information for canceling a group of orders by a filter.
OrderRegisterMessage	The message contains information for registering an order.
OrderStatusMessage	The message requests information about currently registered orders and trades.
OrderReplaceMessage	The message contains information for replacing an order.
PortfolioMessage	Contains information about a portfolio.
PortfolioLookupMessage	Requests information about portfolios by a

	specified criterion. The result of the request is returned using a PortfolioMessage .
PositionChangeMessage	Contains information about a position. Contains information about a change in a certain position property. The property type is defined in the PositionChangeTypes enumeration.
QuoteChangeMessage	Contains information about order book quotes.
ResetMessage	Resetting the state.
SecurityMessage	Contains information about an instrument.
SecurityLookupMessage	Requests a list of instruments by a specified criterion. The result of the request will be returned using a SecurityMessage .
BoardStateMessage	Contains information about a change in the trading session state. The available values of the session state are defined in the SessionStates enumeration.
TimeMessage	Contains information about the current time.
DataTypeLookupMessage	Requests a list of supported data types. The result of the request will be returned using a DataTypeInfoMessage .
DataTypeInfoMessage	Contains information about a supported data type.
SubscriptionResponseMessage	A response to a subscription or unsubscription request. In case of an error executing the request, the error description is filled in SubscriptionResponseMessage.Error .
SubscriptionOnlineMessage	A message about the transition of a subscription to an online state.
SubscriptionFinishedMessage	A message about the end of a subscription in case of receiving all necessary data.

The [ExecutionMessage](#) is a universal message that allows transmitting various exchange information related to orders and trades: tick trades, order logs, own orders, and trades.

The type of information in the message is determined by the value of the [ExecutionMessage.ExecutionType](#) property:

- [ExecutionTypes.Tick](#) - tick trade.
- [ExecutionTypes.Transaction](#) - transaction (information about own trade or order).
- [ExecutionTypes.OrderLog](#) - order log.

If the [ExecutionTypes.Transaction](#) type is used, then it refers to own orders or trades. In this case, if the message contains information about an order, the [ExecutionMessage.HasOrderInfo](#) property = true, if there is information about a trade, the [ExecutionMessage.HasTradeInfo](#) property = true. Note that an *own trade* contains information about both the trade itself and the order associated with that trade. Therefore, in this case, the above properties have a value of true. These properties allow differentiating messages with own trades and orders.

Best practices

Best Practices

When developing a connector for a number of exchanges within the StockSharp platform, it is recommended to follow an established approach that involves dividing functionality into several key components:

1. [Authentication](#) - Handles API key management and request signature generation.
2. [REST Client](#) - Facilitates interaction with the exchange's REST API.
3. [WebSocket Client](#) - Manages real-time data through WebSocket connections.
4. [Type Conversion](#) - Provides methods for converting between StockSharp data types and exchange-specific formats.

This division allows for a more modular and maintainable code structure, facilitates testing, and enables the reuse of components in other projects.

When implementing each of these components, it's important to consider the specifics of the particular exchange, but the general structure remains similar for most exchanges.

Authentication

Authentication

The authentication component plays a key role in ensuring secure interaction with a number of exchange APIs. It is responsible for storing API keys, generating signatures for requests, and other security aspects.

Main Functions

1. Storing API keys (public key, secret key, passphrase).
2. Generating signatures for requests according to the requirements of a specific exchange.
3. Adding necessary authentication headers to HTTP requests.

Implementation Example

Below is an example of the Authenticator class for working with authentication:

```
class Authenticator : Disposable
{
    private readonly HashAlgorithm _hasher;

    public Authenticator(bool canSign, SecureString key, SecureString secret, SecureString passphrase)
    {
        CanSign = canSign;
        Key = key;
        Secret = secret;
        Passphrase = passphrase;

        // Create a hashing algorithm based on the secret key
        _hasher = secret.IsNullOrEmpty() ? null : new HMACSHA256(secret.UnSecure().Base64());
    }

    protected override void DisposeManaged()
    {
        // Dispose of the hashing algorithm resources
        _hasher?.Dispose();
        base.DisposeManaged();
    }

    // Flag indicating whether the authenticator can create signatures
    public bool CanSign { get; }

    // API public key
    public SecureString Key { get; }

    // API secret key
    public SecureString Secret { get; }

    // Passphrase (if required by the exchange)
    public SecureString Passphrase { get; }

    // Method for creating a request signature
    public string MakeSign(string url, Method method, string parameters, out string timestamp)
    {
        // Generate a timestamp
        timestamp = DateTime.UtcNow.ToUnix().ToString("F0");

        // Create a signature based on the timestamp, method, URL, and parameters
        return _hasher
            .ComputeHash((timestamp + method.ToString().ToUpperInvariant() + url + parameters).UTF8())
            .Base64();
    }
}
```

```
}  
}
```

Recommendations

- Use `SecureString` for storing sensitive data, such as keys and passphrases.
- Implement the `IDisposable` interface for proper resource disposal, especially if cryptographic primitives are used.
- Ensure that signature generation methods correspond to the latest version of the exchange's API documentation.

With proper implementation of the authentication component, you will ensure secure interaction with the exchange and simplify the process of authorizing requests in other parts of the connector.

REST client

REST Client

When developing a connector for a number of exchanges, an important component is the HTTP client, which provides interaction with the exchange's REST API. In StockSharp, for this purpose, a `HttpClient` class is often created when developing a connector, built on the basis of the RestSharp library.

RestSharp Features

[RestSharp](#) is a popular .NET library for working with REST APIs. To simplify working with RestSharp within the framework of StockSharp, extension methods have been developed that are available in the [Ecng repository](#).

HttpClient Structure

`HttpClient` usually includes the following key elements:

1. **Constructor**
 - Initializes `RestClient`
 - Configures the base URL and headers
2. **Methods for different types of requests**
 - For example, `GetProducts`, `GetCandles`, `GetTrades`
3. **Methods for working with trading operations**
 - For example, `GetOrders`, `RegisterOrder`, `CancelOrder`
4. **Helper methods**
 - For forming requests
 - For processing responses

```
class HttpClient : BaseLogReceiver
{
    private readonly RestClient _restClient;
    private readonly Authenticator _authenticator;

    // The constructor initializes RestClient and configures the base URL
    public HttpClient(Authenticator authenticator)
    {
        _authenticator = authenticator ?? throw new ArgumentNullException(nameof(authenticator));

        var options = new RestClientOptions
        {
            BaseUrl = new Uri("https://api.example.com"),
            UserAgent = "YourAppName/1.0"
        };

        _restClient = new RestClient(options);
    }

    // Method for getting a list of products (instruments) from the exchange
    public async Task<IEnumerable<Product>> GetProducts(string type, CancellationToken
cancellationToken)
    {
        var request = new RestRequest("products", Method.Get)
```

```

        .AddParameter("type", type);

// Using the ExecuteAsync extension method to execute the request
var response = await _restClient.ExecuteAsync<List<Product>>(request, cancellationTokens);
return response.Data;
}

// Method for getting historical candles
public async Task<IEnumerable<Candle>> GetCandles(string symbol, long start, long end, string
granularity, CancellationToken cancellationToken)
{
    var request = new RestRequest($"products/{symbol}/candles", Method.Get)
        .AddParameter("start", start)
        .AddParameter("end", end)
        .AddParameter("granularity", granularity);

    var response = await _restClient.ExecuteAsync<List<Candle>>(request, cancellationToken);
    return response.Data;
}

// Method for registering a new order
public async Task<Order> RegisterOrder(string clientId, string symbol, string type, string
side, decimal? price, decimal volume, CancellationToken cancellationToken)
{
    var request = new RestRequest("orders", Method.Post)
        .AddJsonBody(new
        {
            client_order_id = clientId,
            symbol,
            type,
            side,
            price,
            volume
        });

    // Applying authentication before executing the request
    var response = await _restClient.ExecuteAsync<Order>(ApplyAuth(request), cancellationToken);
    return response.Data;
}

// Method for canceling an existing order
public async Task<bool> CancelOrder(string orderId, CancellationToken cancellationToken)
{
    var request = new RestRequest($"orders/{orderId}", Method.Delete);

    var response = await _restClient.ExecuteAsync(ApplyAuth(request), cancellationToken);
    return response.IsSuccessful;
}

// Helper method for applying authentication to a request
private RestRequest ApplyAuth(RestRequest request)
{
    {
        _authenticator.ApplyAuthentication(request);
        return request;
    }
}
}

```

Implementation Recommendations

- Use asynchronous methods for efficient network operation.
- Implement error handling and retries for unstable connections.
- Take into account the API rate limits of the specific exchange.
- Use RestSharp extension methods to simplify working with requests and responses.

Remember that the specific implementation may differ depending on the requirements and specifics of the

API of a particular exchange.

WebSocket Client

When developing a connector for a number of exchanges, an important component is the WebSocket client, which provides real-time data retrieval. In StockSharp, for this purpose, a `SocketClient` class is often created when developing a connector, built on the basis of `WebSocketClient`.

WebSocketClient Features

`WebSocketClient` is a system class developed for automatic reconnection in case of a WebSocket connection loss. Its source code is available in the [Ecng repository](#).

SocketClient Structure

`SocketClient` usually includes the following key elements:

- 1. Constructor**
 - Initializes the base `WebSocketClient`
 - Sets up event handlers
- 2. Connection and disconnection methods**
 - `Connect` / `ConnectAsync`
 - `Disconnect`
- 3. Subscription methods for different types of data**
 - For example, `SubscribeTrades`, `SubscribeOrderBook`
- 4. Unsubscription methods for data**
 - Corresponding unsubscription methods for each type of subscription
- 5. Event handlers**
 - For handling different types of incoming messages
- 6. Helper methods**
 - For forming subscription/unsubscription messages
 - For processing received data

```
class SocketClient : BaseLogReceiver
{
    private readonly WebSocketClient _client;
    private readonly Authenticator _authenticator;

    // Events for different types of data
    public event Action<Heartbeat> HeartbeatReceived;
    public event Action<Ticker> TickerReceived;
    public event Action<Trade> TradeReceived;
    public event Action<string, string, IEnumerable<OrderBookChange>> OrderBookReceived;
    public event Action<Order> OrderReceived;
    public event Action<Exception> Error;
    public event Action Connected;
    public event Action<bool> Disconnected;
```

```

public SocketClient(Authenticator authenticator, int reconnectAttempts)
{
    _authenticator = authenticator;
    _client = new WebSocketClient(/* parameters */);
    _client.ReconnectAttempts = reconnectAttempts;
}

public ValueTask Connect(Cancellation_token cancellation_token)
{
    // Connection logic
}

public void Disconnect()
{
    // Disconnection logic
}

public ValueTask SubscribeTicker(string symbol, Cancellation_token cancellation_token)
{
    // Ticker subscription logic
}

public ValueTask UnSubscribeTicker(string symbol, Cancellation_token cancellation_token)
{
    // Ticker unsubscription logic
}

// Similar methods for other types of subscriptions (trades, order book, etc.)

private void OnProcess(dynamic obj)
{
    // Processing incoming messages
}

// Helper methods
}

```

Implementation Recommendations

- Adapt the SocketClient structure to the specific exchange, taking into account the specifics of its API.
- Use asynchronous methods for efficient work with WebSocket.
- Implement handling of different types of messages from the exchange.
- Ensure proper error handling and reconnection upon connection loss.

Remember that the specific implementation may differ depending on the requirements and specifics of the API of a particular exchange.

Type conversion

Type Conversion

The type conversion component plays an important role in ensuring compatibility between the data types used in StockSharp and the formats specific to a particular exchange.

Main Functions

1. Converting StockSharp types (e.g., [Sides](#), [OrderTypes](#), [TimelnForce](#)) to string representations used by the exchange.
2. Reverse conversion of data received from the exchange to StockSharp types.
3. Converting instrument identifiers between StockSharp and exchange formats.
4. Converting time formats and timeframes.

Implementation Example

Below is an example of a class with extension methods for type conversion:

```
static class Extensions
{
    // Converting StockSharp order side to exchange string representation
    public static string ToNative(this Sides side)
    {
        return side switch
        {
            Sides.Buy => "buy",
            Sides.Sell => "sell",
            _ => throw new ArgumentOutOfRangeException(nameof(side), side, LocalizedStrings.InvalidValue),
        };
    }

    // Converting exchange order side string representation to StockSharp type
    public static Sides ToSide(this string side)
    => side?.ToLowerInvariant() switch
    {
        "buy" or "bid" => Sides.Buy,
        "sell" or "ask" or "offer" => Sides.Sell,
        _ => throw new ArgumentOutOfRangeException(nameof(side), side, LocalizedStrings.InvalidValue),
    };

    // Converting StockSharp order type to exchange string representation
    public static string ToNative(this OrderTypes? type)
    {
        return type switch
        {
            null => null,
            OrderTypes.Limit => "limit",
            OrderTypes.Market => "market",
            OrderTypes.Conditional => "stop",
            _ => throw new ArgumentOutOfRangeException(nameof(type), type, LocalizedStrings.InvalidValue),
        };
    }

    // Converting exchange order type string representation to StockSharp type
    public static OrderTypes ToOrderType(this string type)
    => type?.ToLowerInvariant() switch
    {
        "limit" => OrderTypes.Limit,
        "market" => OrderTypes.Market,
```

```

    "stop" or "stop limit" => OrderTypes.Conditional,
    _ => throw new ArgumentOutOfRangeException(nameof(type), type, LocalizedStrings.InvalidValue),
};

// Other conversion methods...

// Dictionary for mapping StockSharp timeframes to exchange string representations
public static readonly PairSet<TimeSpan, string> TimeFrames = new()
{
    { TimeSpan.FromMinutes(1), "ONE_MINUTE" },
    { TimeSpan.FromMinutes(5), "FIVE_MINUTE" },
    // Other timeframes...
};

// Converting StockSharp timeframe to exchange string representation
public static string ToNative(this TimeSpan timeframe)
    => TimeFrames.TryGetValue(timeframe) ?? throw new ArgumentOutOfRangeException(nameof(timeframe),
timeframe, LocalizedStrings.InvalidValue);

// Converting exchange timeframe string representation to TimeSpan
public static TimeSpan ToTimeFrame(this string name)
    => TimeFrames.TryGetKey2(name) ?? throw new ArgumentOutOfRangeException(nameof(name), name,
LocalizedStrings.InvalidValue);
}

```

Recommendations

- Use extension methods for convenient usage of conversion functions.
- Handle all possible enumeration values, including null and unknown values.
- Use switch expressions (C# 8.0+) for cleaner and more readable code.
- Add checks for invalid values and throw exceptions with clear error messages.
- Consider using dictionaries for mapping values, especially for complex or frequently changing mappings (e.g., for timeframes).

Proper implementation of type conversion significantly simplifies working with data in other parts of the connector and reduces the likelihood of errors related to format mismatches.

Save and load settings

Save and load settings

[Connector.Save](#) and [Connector.Load](#) methods, respectively, are used to save and load [Connector](#) settings.

To save and load settings from an external file, you can use the serialization and deserialization, respectively, implemented in [S#](#).

```
...
private readonly Connector _connector = new Connector();
private const string _connectorFile = "ConnectorFile.json";
...
public void Load()
{
    if (File.Exists(_connectorFile))
    {
        //Download connector settings from an existing configuration file
        _connector.Load(new JsonSerializer<SettingsStorage>().Deserialize(_connectorFile));
    }
}
...
public void Save()
{
    //Save the connector settings to the configuration file
    new JsonSerializer<SettingsStorage>().Serialize(_connector.Save(), _connectorFile);
}
...
```

Recommended content

[Creating own connector](#)

Reconnection settings

Reconnection settings

All connectors provide the ability to configure reconnection in case of disconnection. In the [Connection settings window](#) graphic element, it looks like this:

[image: API GUI ReconnectionSettings]

Reconnection properties

- **Interval** - The interval at which connection attempts will occur.
- **Initially** - The number of attempts to establish the initial connection if it was not established (timeout, network failure, etc.).
- **Reconnection** - The number of attempts to reconnect if the connection was disconnected during operation.
- **Timeout** - Timeout for successful connection/disconnection.
- **Operating mode** - The operating mode during which it is necessary to make connections.

Code reconnection settings

The reconnection mechanism is configured through the [ReConnectionSettings](#) property and allows you to monitor the following error scenarios:

- Unable to establish a connection (no communication, incorrect username/password, etc.). The [ReConnectionSettings.AttemptCount](#) property sets the number of attempts to establish a connection. By default, it is 0, which means that the mode is disabled. -1 means an infinite number of attempts.
- The connection was broken during operation. The [ReConnectionSettings.ReAttemptCount](#) property sets the number of attempts to reconnect the connection. By default, it is 100. -1 means an infinite number of attempts. 0 - mode is disabled.
- When connecting or disconnecting a connection, the corresponding [IConnector.Connected](#) or [IConnector.Disconnected](#) events may not be received for a long time. For such situations, you can use the [ReConnectionSettings.TimeOutInterval](#) property to set the maximum acceptable timeout for a successful event. If, after this time, the desired event does not occur, the [IConnector.ConnectionError](#) event is raised with a timeout error.

1. When creating a gateway, you need to initialize the settings of the reconnection mechanism through the [ReConnectionSettings](#) property:

```
// initialize the reconnection mechanism (it will automatically connect
// every 10 seconds if the gateway loses connection with the server)
Connector.Adapter.ReConnectionSettings.Interval = TimeSpan.FromSeconds(10);
// reconnection will work only during the operation of the RTS exchange
// (to disable reconnection when there is no trading normally, for example, at night)
Connector.Adapter.ReConnectionSettings.WorkingTime = Exchange.Rts.WorkingTime;
```

2. To check how the connection control mechanism works, you can turn off the Internet connection:
3. Below is the program log, which shows that the application is initially in a connected state, and after turning off the Internet connection, the application tries to reconnect. After restoring the Internet connection, the application connection is restored:

[image: API ReconnectionLog]

4. Since several connections can be used in [Connector](#), by default events related to reconnection, such as

`ConnectionRestored` are not triggered, and connection adapters try to reconnect themselves. For the event to start being raised, you need to set the value of the adapter's `BasketMessageAdapter.SuppressReconnectingErrors` property to **false**.

```
Connector.Adapter.SuppressReconnectingErrors = false;  
Connector.ConnectionError += error => this.Sync(() => MessageBox.Show(this, "Connection lost"));  
Connector.ConnectionRestored += adapter => this.Sync(() => MessageBox.Show(this, "Connection  
restored"));
```

[image: sampleconnectionerror][image: sampleconnectionrestore]

OAuth Authorization

To use OAuth authorization in connectors in your program, you need to register several services:

```
// Connecting the password storage for access to StockSharp WebAPI.
ConfigManager.RegisterService<ICredentialsProvider>(new DefaultCredentialsProvider());
//ConfigManager.RegisterService<ICredentialsProvider>(new TokenCredentialsProvider("%token%"));

// Connecting the service provider for access to StockSharp WebAPI
ConfigManager.RegisterService<IApiServiceProvider>(new ApiServiceProvider());

// OAuth authorization service that will be used by connectors
ConfigManager.RegisterService<IOAuthProvider>(new OAuthProvider());
//ConfigManager.RegisterService<IOAuthProvider>(new WebApiOAuthProvider());
```

- [DefaultCredentialsProvider](#) and [TokenCredentialsProvider](#) implement the [ICredentialsProvider](#) interface.
- [ApiServiceProvider](#) implements the [IApiServiceProvider](#) interface.
- [OAuthProvider](#) implements the [IOAuthProvider](#) interface.

There are two implementation options for [ICredentialsProvider](#):

1. [DefaultCredentialsProvider](#) - loads StockSharp account data from a local file. Prior authorization is required. For example, through the Installer.
2. [TokenCredentialsProvider](#) - passes the token directly from the code. No secret file is required on the machine. The token is obtained from <https://stocksharp.ru/profile/>:
[image: Profile]

There are two implementation options for [IOAuthProvider](#):

1. [WebApiOAuthProvider](#) - for console applications where no authorization window display is required.
2. [OAuthProvider](#) - for WPF applications where an authorization window needs to be displayed:
[image: OAuth Start]

Async Order Operations

The [Connector](#) class provides asynchronous versions of all order operations. Async methods avoid blocking the calling thread and support cancellation via `CancellationToken`.

Methods

RegisterOrderAsync

Asynchronous registration of a new order:

```
public async ValueTask RegisterOrderAsync(Order order, CancellationToken cancellationToken = default)
```

The method validates the order (checks the volume, automatically determines the order type -- limit or market), initializes the transaction, and sends the registration command to the adapter. In case of an error, a registration error event is generated.

The synchronous counterpart [RegisterOrder](#) internally calls `RegisterOrderAsync`.

CancelOrderAsync

Asynchronous cancellation of an existing order:

```
public async ValueTask CancelOrderAsync(Order order, CancellationToken cancellationToken = default)
```

The method creates a new transaction identifier for the cancel operation and sends the order withdrawal command to the adapter. The order must have been previously registered.

Synchronous counterpart: [CancelOrder](#).

EditOrderAsync

Asynchronous editing of an active order (changing price and/or volume without cancellation):

```
public async ValueTask EditOrderAsync(Order order, Order changes, CancellationToken cancellationToken = default)
```

Parameters:

- `order` -- the original order to edit.
- `changes` -- an [Order](#) object with the new field values (price, volume, etc.).

Before calling, it is recommended to check editing support:

```
if (connector.IsOrderEditable(order) == true)
{
    var changes = order.CreateOrder();
    changes.Price = newPrice;
    await connector.EditOrderAsync(order, changes);
}
```

Synchronous counterpart: [EditOrder](#).

ReRegisterOrderAsync

Asynchronous re-registration of an order (canceling the old one and registering a new one in a single operation):

```
public async ValueTask ReRegisterOrderAsync(Order oldOrder, Order newOrder, CancellationToken cancellationToken = default)
```

Used when the exchange does not support order editing but supports atomic replacement. Support can be checked via `IsOrderReplaceable`:

```
if (connector.IsOrderReplaceable(order) == true)
{
    var newOrder = order.CreateOrder();
    newOrder.Price = newPrice;
    await connector.ReRegisterOrderAsync(order, newOrder);
}
```

When to Use Async Methods

Use async methods when:

- Code executes in an async context (e.g., in ASP.NET handlers, cross-platform applications).
- You need to support operation cancellation via `CancellationToken`.
- You need to avoid blocking the UI thread.

Use synchronous methods when:

- Code runs within a strategy (`Strategy`), which internally manages threads.
- A simple script or console application where async is not required.

Synchronous methods (`RegisterOrder`, `CancelOrder`, `EditOrder`) internally call their async counterparts via `AsyncHelper.Run`, so they are fully equivalent in functionality.

Example

```
private readonly Connector _connector = new();

public async Task PlaceAndManageOrderAsync(Security security, Portfolio portfolio, CancellationToken cancellationToken)
{
    // Create an order
    var order = new Order
    {
        Security = security,
        Portfolio = portfolio,
        Direction = Sides.Buy,
        Volume = 1,
        Price = security.BestBid?.Price ?? 100m,
        Type = OrderTypes.Limit,
    };

    // Async registration
    await _connector.RegisterOrderAsync(order, cancellationToken);

    // ... wait for market condition changes ...

    // Async price edit (if supported)
    if (_connector.IsOrderEditable(order) == true)
    {

```

```
        var changes = order.CreateOrder();  
        changes.Price = order.Price - 0.01m;  
        await _connector.EditOrderAsync(order, changes, cancellationTokens);  
    }  
  
    // Async cancellation  
    await _connector.CancelOrderAsync(order, cancellationTokens);  
}
```

See Also

[Orders](#)

IFileSystem and Paths

IFileSystem and Paths.FileSystem

Overview

IFileSystem is a file system abstraction used throughout StockSharp. Instead of directly accessing `System.IO.File` and `System.IO.Directory`, platform components work through this interface. This provides:

- **Testability** -- the ability to substitute the file system in unit tests
- **Portability** -- a unified interface for different environments (local disk, cloud storage, memory)
- **Consistency** -- all components use the same approach for working with files

The IFileSystem interface is defined in the `Ecng.Common` package and provides methods for working with files and directories: creation, reading, writing, deletion, existence checking, and file enumeration.

Paths.FileSystem

The Paths class (namespace `StockSharp.Configuration`) provides the static `FileSystem` property -- the default IFileSystem implementation:

```
public static class Paths
{
    // Standard file system (LocalFileSystem)
    public static readonly IFileSystem FileSystem = Messages.Extensions.DefaultFileSystem;
}
```

`Paths.FileSystem` references `LocalFileSystem.Instance` -- a singleton that works with the local disk via standard `System.IO` facilities.

Main IFileSystem Methods

The IFileSystem interface includes methods for typical operations:

Method	Description
<code>FileExists(path)</code>	Check if a file exists
<code>DirectoryExists(path)</code>	Check if a directory exists
<code>CreateDirectory(path)</code>	Create a directory
<code>OpenRead(path)</code>	Open a file for reading (Stream)
<code>OpenWrite(path)</code>	Open a file for writing (Stream)
<code>MoveFile(src, dst)</code>	Move a file
<code>DeleteFile(path)</code>	Delete a file
<code>EnumerateFiles(path, mask)</code>	Enumerate files in a directory
<code>WriteAllTextAsync(path, text)</code>	Write text to a file asynchronously

Where It Is Used

Virtually all StockSharp components that work with the file system accept `IFileSystem` in their constructor. Below are the most common cases.

LocalMarketDataDrive

Local disk market data storage:

```
using StockSharp.Algo.Storages;
using StockSharp.Configuration;

// Recommended way -- explicitly passing IFileSystem
var drive = new LocalMarketDataDrive(Paths.FileSystem, @"C:\MarketData");

// Deprecated way (uses Paths.FileSystem internally)
// var drive = new LocalMarketDataDrive(@"C:\MarketData"); // [Obsolete]
```

CsvEntityRegistry

Entity registry (exchanges, securities, portfolios) in CSV format:

```
using StockSharp.Algo.Storages.Csv;
using StockSharp.Configuration;

var executor = new ChannelExecutor();
var registry = new CsvEntityRegistry(Paths.FileSystem, @"C:\Data", executor);
```

SnapshotRegistry

Market data snapshot registry:

```
using StockSharp.Algo.Storages;
using StockSharp.Configuration;

var snapshots = new SnapshotRegistry(Paths.FileSystem, Paths.SnapshotsDir);
```

Serialization and Deserialization

Extension methods in `Paths` for working with JSON configurations:

```
using StockSharp.Configuration;

var fs = Paths.FileSystem;

// Serialize an object to a file
settings.Serialize(fs, @"C:\config.json");

// Deserialize an object from a file
var loaded = @"C:\config.json".Deserialize<SettingsStorage>(fs);

// Async deserialization
var data = await @"C:\data.json".DeserializeAsync<MyData>(fs, cancellationToken);

// Check if a configuration file exists
if (@"C:\config.json".IsConfigExists(fs))
{
    // ...
}
```

CandlePatternFileStorage

Candle pattern storage:

```
using StockSharp.Algo.Candles.Patterns;
using StockSharp.Configuration;

var patternStorage = new CandlePatternFileStorage(
    Paths.FileSystem,
    Paths.CandlePatternsFile,
    executor
);
```

NativeIdStorage, SecurityMappingStorage, and Others

Most mapping and identifier storages also accept `IFileSystem`:

```
// NativeIdStorage
var nativeIdStorage = new NativeIdStorage(Paths.FileSystem, Paths.SecurityNativeIdDir, executor);

// SecurityMappingStorage
var mappingStorage = new SecurityMappingStorage(Paths.FileSystem, Paths.SecurityMappingDir,
executor);

// ExtendedInfoStorage
var extInfoStorage = new ExtendedInfoStorage(Paths.FileSystem, Paths.SecurityExtendedInfo,
executor);
```

Typical Usage Pattern

In StockSharp applications, it is recommended to keep a reference to `IFileSystem` and pass it to all components:

```
using StockSharp.Algo;
using StockSharp.Algo.Storages;
using StockSharp.Configuration;

// Get the file system
var fs = Paths.FileSystem;

// Create data storage
var drive = new LocalMarketDataDrive(fs, Paths.StorageDir);

// Create a connector and configure storage
var connector = new Connector();
connector.Adapter.StorageSettings.Drive = drive;

// Load settings from a file
var configFile = Path.Combine(Paths.AppDataPath, "connector_config.json");

if (configFile.IsConfigExists(fs))
{
    var config = configFile.Deserialize<SettingsStorage>(fs);
    connector.Load(config);
}
```

Deprecated Overloads

Many classes retain constructors without `IFileSystem` for backward compatibility, but they are marked with the `[Obsolete]` attribute. These constructors use `Paths.FileSystem` internally:

```
// Deprecated way
[Obsolete("Use IFileSystem overload.")]
public LocalMarketDataDrive(string path)
    : this(Paths.FileSystem, path) { }
```

```
// Recommended way  
public LocalMarketDataDrive(IFileSystem fileSystem, string path) { }
```

It is recommended to always use overloads with explicit `IFileSystem` passing, as deprecated constructors will be removed in future versions.

Subscription Lifecycle

Subscription Lifecycle

Subscriptions in StockSharp go through specific lifecycle stages. The [ISubscriptionProvider](#) interface provides events for tracking each stage.

Lifecycle Events

SubscriptionStarted

```
event Action<Subscription> SubscriptionStarted;
```

Called when a subscription has been successfully started -- the adapter has accepted the request and begun transmitting data. For historical subscriptions, this means data loading has begun. For live subscriptions -- that the server has accepted the request.

SubscriptionOnline

```
event Action<Subscription> SubscriptionOnline;
```

Called when the subscription has transitioned to real-time mode. For live subscriptions, this means that the historical catch-up (if any) has completed and data is now arriving in real time. This is an important signal for strategies to understand that indicators are "warmed up" and trading can begin.

SubscriptionStopped

```
event Action<Subscription, Exception> SubscriptionStopped;
```

Called when the subscription has ended. The *Exception* parameter contains the reason for stopping:

- *null* -- normal completion (user unsubscribed or historical data finished).
- An exception object -- an error (connection loss, server-side error, etc.).

SubscriptionFailed

```
event Action<Subscription, Exception, bool> SubscriptionFailed;
```

Called on a subscription error. The third *bool* parameter indicates whether this was a subscribe (*true*) or unsubscribe (*false*) operation.

Event Order

Typical sequence for a live subscription:

1. Call `Subscribe(subscription)`
2. `SubscriptionStarted` -- subscription accepted
3. Data arriving (candles, order books, trades, etc.)
4. `SubscriptionOnline` -- transition to real-time mode
5. Continued data arrival in real time

6. Call `UnSubscribe(subscription)` or connection loss

7. `SubscriptionStopped` -- subscription ended

For a historical subscription (with a specified date range):

1. Call `Subscribe(subscription)`

2. `SubscriptionStarted` -- subscription accepted

3. Historical data arriving

4. `SubscriptionStopped` with `null` -- all data received

SubscriptionsOnConnect

The [Connector.SubscriptionsOnConnect](#) property defines the set of subscriptions that are automatically sent on connection:

```
ISet<Subscription> SubscriptionsOnConnect { get; }
```

By default, subscriptions for security lookups, portfolio lookups, and order lookups are included:

```
SubscriptionsOnConnect.Add(SecurityLookup);  
SubscriptionsOnConnect.Add(PortfolioLookup);  
SubscriptionsOnConnect.Add(OrderLookup);
```

You can add your own subscriptions that will be automatically started on each connection:

```
// Add automatic Level1 data subscription  
var l1Sub = new Subscription(DataType.Level1, security);  
connector.SubscriptionsOnConnect.Add(l1Sub);  
  
// Remove automatic order lookup on connection  
connector.SubscriptionsOnConnect.Remove(connector.OrderLookup);
```

Per-Adapter Connection Events

When working with multiple connections (multiple adapters), events indicating which specific adapter connected or disconnected are useful:

ConnectedEx

```
event Action<IMessageAdapter> ConnectedEx;
```

Called on successful connection of a specific adapter. The parameter is the adapter that initiated the event.

DisconnectedEx

```
event Action<IMessageAdapter> DisconnectedEx;
```

Called on disconnection of a specific adapter.

ConnectionErrorEx

```
event Action<IMessageAdapter, Exception> ConnectionErrorEx;
```

Called on a connection error for a specific adapter.

Aggregated events `Connected`, `Disconnected`, and `ConnectionError` are also available, which fire without specifying a particular adapter.

Example

```
private readonly Connector _connector = new();

public void SetupSubscriptionTracking()
{
    // Track subscription lifecycle
    _connector.SubscriptionStarted += subscription =>
    {
        Console.WriteLine($"Subscription started: {subscription.DataType}, " +
            $"Security: {subscription.SecurityId}");
    };

    _connector.SubscriptionOnline += subscription =>
    {
        Console.WriteLine($"Subscription online: {subscription.DataType}");
    };

    _connector.SubscriptionStopped += (subscription, error) =>
    {
        if (error == null)
            Console.WriteLine($"Subscription completed: {subscription.DataType}");
        else
            Console.WriteLine($"Subscription interrupted: {subscription.DataType}, " +
                $"Error: {error.Message}");
    };

    // Track individual adapter connections
    _connector.ConnectedEx += adapter =>
    {
        Console.WriteLine($"Adapter connected: {adapter.Name}");
    };

    _connector.DisconnectedEx += adapter =>
    {
        Console.WriteLine($"Adapter disconnected: {adapter.Name}");
    };

    _connector.ConnectionErrorEx += (adapter, error) =>
    {
        Console.WriteLine($"Adapter connection error {adapter.Name}: {error.Message}");
    };

    // Connect
    _connector.Connect();

    // After connection -- create a subscription
    _connector.Connected += () =>
    {
        var subscription = new Subscription(DataType.Ticks, security);
        _connector.Subscribe(subscription);
    };
}
```

See Also

[Connection](#)

Subscriptions

Subscriptions

StockSharp API offers a data acquisition model based on subscriptions. This is a universal mechanism for receiving both market data and transaction information. This approach has significant advantages:

- **Subscription isolation** — each subscription works independently, allowing any number of subscriptions with different parameters to be run in parallel (with or without history request).
- **State tracking** — subscriptions have specific states that allow you to control whether historical data is currently flowing or the subscription has switched to real-time mode.
- **Universality** — the code for working with subscriptions is the same regardless of the types of data requested, making development more efficient.

To work with subscriptions, you need to use the [Subscription](#) class. Let's look at examples of using subscriptions to obtain various types of data.

Example of Candle Subscription

```
// Create a subscription for 5-minute candles
var subscription = new Subscription(DataType.TimeFrame(TimeSpan.FromMinutes(5)), security)
{
    // Configure subscription parameters via the MarketData property
    MarketData =
    {
        // Request data for the last 30 days
        From = DateTimeOffset.Now.Subtract(TimeSpan.FromDays(30)),
        // null means the subscription will switch to real-time mode after receiving history
        To = null
    }
};

// Processing received candles
_connector.CandleReceived += (sub, candle) =>
{
    if (sub != subscription)
        return;

    // Process the candle
    Console.WriteLine($"Candle: {candle.OpenTime} - O:{candle.OpenPrice} H:{candle.HighPrice} L:
{candle.LowPrice} C:{candle.ClosePrice} V:{candle.TotalVolume}");
};

// Handling the subscription's transition to online mode
_connector.SubscriptionOnline += (sub) =>
{
    if (sub != subscription)
        return;

    Console.WriteLine("Subscription switched to real-time mode");
};

// Handling subscription errors
_connector.SubscriptionFailed += (sub, error, isSubscribe) =>
{
    if (sub != subscription)
        return;

    Console.WriteLine($"Subscription error: {error}");
};
```

```
// Starting the subscription
_connector.Subscribe(subscription);
```

Example of Order Book Subscription

```
// Create a subscription to the order book for the selected instrument
var depthSubscription = new Subscription(DataType.MarketDepth, security);

// Processing received order books
_connector.OrderBookReceived += (sub, depth) =>
{
    if (sub != depthSubscription)
        return;

    // Process the order book
    Console.WriteLine($"Order book: {depth.SecurityId}, Time: {depth.ServerTime}");
    Console.WriteLine($"Bids: {depth.Bids.Count}, Asks: {depth.Asks.Count}");
};

// Starting the subscription
_connector.Subscribe(depthSubscription);
```

Example of Tick Trades Subscription

```
// Create a subscription to tick trades for the selected instrument
var tickSubscription = new Subscription(DataType.Ticks, security);

// Processing received ticks
_connector.TickTradeReceived += (sub, tick) =>
{
    if (sub != tickSubscription)
        return;

    // Process the tick
    Console.WriteLine($"Tick: {tick.SecurityId}, Time: {tick.ServerTime}, Price: {tick.Price}, Volume: {tick.Volume}");
};

// Starting the subscription
_connector.Subscribe(tickSubscription);
```

Example of Subscription with Candle Building Mode Configuration

```
// Subscription to 5-minute candles that will be built from ticks
var candleSubscription = new Subscription(DataType.TimeFrame(TimeSpan.FromMinutes(5)), security)
{
    MarketData =
    {
        // Specify the building mode and data source
        BuildMode = MarketDataBuildModes.Build,
        BuildFrom = DataType.Ticks,
        // You can also enable volume profile building
        IsCalcVolumeProfile = true,
    }
};

_connector.Subscribe(candleSubscription);
```

Example of Level1 Subscription (Basic Instrument Information)

```
// Creating a subscription for basic instrument information
var level1Subscription = new Subscription(DataType.Level1, security);

// Processing received Level1 data
_connector.Level1Received += (sub, level1) =>
{
    if (sub != level1Subscription)
        return;

    Console.WriteLine($"Level1: {level1.SecurityId}, Time: {level1.ServerTime}");

    // Output Level1 field values
    foreach (var pair in level1.Changes)
    {
        Console.WriteLine($"Field: {pair.Key}, Value: {pair.Value}");
    }
};

// Starting the subscription
_connector.Subscribe(level1Subscription);
```

Unsubscribing from Data

To stop receiving data, use the `UnSubscribe` method:

```
// Unsubscribe from a specific subscription
_connector.UnSubscribe(subscription);

// Or you can unsubscribe from all subscriptions
foreach (var sub in _connector.Subscriptions)
{
    _connector.UnSubscribe(sub);
}
```

Subscription States

Subscriptions can be in the following states:

- [SubscriptionStates.Stopped](#) — the subscription is inactive (stopped or not started).
- [SubscriptionStates.Active](#) — the subscription is active and may transmit historical data until switching to real-time mode or completion.
- [SubscriptionStates.Error](#) — the subscription is inactive and in an error state.
- [SubscriptionStates.Finished](#) — the subscription has completed its work (all data received).
- [SubscriptionStates.Online](#) — the subscription has switched to real-time mode and only transmits current data.

Getting historical data

Getting Historical Data

StockSharp API provides convenient mechanisms for obtaining historical data, which can be used both for testing trading strategies and for building [indicators](#).

Getting Historical Data via Connector

Setting Up the Connection

To get historical data, you first need to configure a connection to the trading system:

```
// Create a Connector instance
var connector = new Connector();

// Add an adapter for connecting to Binance
var messageAdapter = new BinanceMessageAdapter(connector.TransactionIdGenerator)
{
    Key = "<Your API Key>",
    Secret = "<Your Secret Key>",
};
connector.Adapter.InnerAdapters.Add(messageAdapter);

// Connect
connector.Connect();
```

The connection can also be configured using the graphical interface, as described in the [Connection Settings Window](#) section.

Subscribing to Historical Candles

To receive historical candles, you need to create a subscription and specify the parameters of the requested data:

```
// Create a subscription for 5-minute candles for the selected instrument
var subscription = new Subscription(
    DataType.TimeFrame(TimeSpan.FromMinutes(5)),
    security)
{
    MarketData =
    {
        // Specify the period for which to get historical data
        From = DateTime.Now.Subtract(TimeSpan.FromDays(30)),
        To = DateTime.Now,
        // Set the flag to receive only completed candles
        IsFinishedOnly = true
    }
};

// Subscribe to the candle received event
connector.CandleReceived += OnCandleReceived;

// Start the subscription
connector.Subscribe(subscription);

// Event handler for receiving candles
private void OnCandleReceived(Subscription subscription, ICandleMessage candle)
{
    // Check that the candle belongs to our subscription
    if (subscription != _subscription)
```

```

    return;

    // Process the received candle
    Console.WriteLine($"Candle received: {candle.OpenTime}, O:{candle.OpenPrice}, H:{candle.HighPrice},
L:{candle.LowPrice}, C:{candle.ClosePrice}, V:{candle.TotalVolume}");

    // For display on the chart, you can use:
    // Chart.Draw(_candleElement, candle);
}

```

Using Candles for Charts

The received candles can be displayed on a chart using StockSharp's built-in graphical components:

```

// Create and configure chart elements
var chart = new Chart();
var area = new ChartArea();
var candleElement = new ChartCandleElement();

// Add area and element to the chart
chart.AddArea(area);
chart.AddElement(area, candleElement, subscription);

// In the CandleReceived event handler, draw candles
private void OnCandleReceived(Subscription subscription, ICandleMessage candle)
{
    // Check that the candle belongs to our subscription
    if (subscription != _subscription)
        return;

    // If you need to display only completed candles
    if (candle.State == CandleStates.Finished)
    {
        var chartData = new ChartDrawData();
        chartData.Group(candle.OpenTime).Add(candleElement, candle);
        chart.Draw(chartData);
    }
}

```

Getting Other Types of Historical Data

Similarly, you can get other types of historical data:

Getting Historical Ticks

```

var tickSubscription = new Subscription(DataType.Ticks, security)
{
    MarketData =
    {
        From = DateTime.Now.Subtract(TimeSpan.FromDays(1)),
        To = DateTime.Now
    }
};

connector.TickTradeReceived += (subscription, tick) =>
{
    if (subscription == tickSubscription)
        Console.WriteLine($"Tick: {tick.ServerTime}, Price: {tick.Price}, Volume: {tick.Volume}");
};

connector.Subscribe(tickSubscription);

```

Getting Historical Order Books

```

var depthSubscription = new Subscription(DataType.MarketDepth, security)
{
    MarketData =
    {
        From = DateTime.Now.Subtract(TimeSpan.FromHours(1)),
        To = DateTime.Now
    }
};

connector.OrderBookReceived += (subscription, depth) =>
{
    if (subscription == depthSubscription)
        Console.WriteLine($"Order book: {depth.ServerTime}, Best bid: {depth.GetBestBid()?.Price}, Best
ask: {depth.GetBestAsk()?.Price}");
};

connector.Subscribe(depthSubscription);

```

See Also

- [Candles](#)
- [Subscriptions](#)
- [Indicators](#)

Getting news data

Getting News Data

StockSharp API allows you to receive news data from various sources. News can be an important source of information when making trading decisions or for market analysis.

Note that not all data sources provide news. Some cryptocurrency exchanges, including Binance, do not have a built-in news feed through their API. In such cases, it is recommended to use specialized news sources or RSS feeds.

Subscribing to News Data

To start receiving news, you need to create a subscription to news data and then handle news reception events:

```
// Create a subscription to news
var newsSubscription = new Subscription(DataType.News);

// Subscribe to the news received event
_connector.NewsReceived += OnNewsReceived;

// Start the subscription
_connector.Subscribe(newsSubscription);

// Event handler for receiving news
private void OnNewsReceived(Subscription subscription, News news)
{
    if (subscription != newsSubscription)
        return;

    // Process the received news
    Console.WriteLine($"News: {news.Id}");
    Console.WriteLine($"Headline: {news.Headline}");
    Console.WriteLine($"Source: {news.Source}");
    Console.WriteLine($"Time: {news.ServerTime}");
    Console.WriteLine($"URL: {news.Url}");

    // If there is news text
    if (!string.IsNullOrEmpty(news.Story))
        Console.WriteLine($"Story: {news.Story}");

    // If the news is related to specific instruments
    if (news.SecurityId != null)
        Console.WriteLine($"Instrument: {news.SecurityId}");
}
```

Filtering News

When subscribing to news, you can specify filtering parameters to receive only the news you are interested in:

```
// Create a subscription to news with filtering
var filteredNewsSubscription = new Subscription(DataType.News)
{
    MarketData =
    {
        // Specify the period for which to get news
        From = DateTime.Now.Subtract(TimeSpan.FromHours(24)),

        // You can specify a specific news source
    }
}
```

```
// For example, we use an RSS source
NewsSource = "CryptoNews"
}
};

_connector.Subscribe(filteredNewsSubscription);
```

Displaying News in the User Interface

StockSharp provides a special visual component [NewsPanel](#) for displaying news:

```
// Create and configure a news panel
var newsPanel = new NewsPanel();

// Subscribe to the news received event and add news to the panel
_connector.NewsReceived += (subscription, news) =>
{
    // To update UI elements
    // you need to use the GuiAsync or GuiSync method
    this.GuiAsync(() => newsPanel.NewsGrid.News.Add(news));
};
```

In XAML code:

```
<sx:NewsPanel x:Name="NewsPanel" Grid.Row="1" />
```

Historical News

To get historical news for a specific period, you can use the same subscription mechanism with a specified time range:

```
// Create a subscription to historical news
var historicalNewsSubscription = new Subscription(DataType.News)
{
    MarketData =
    {
        // Specify the period for which to get news
        From = DateTime.Now.Subtract(TimeSpan.FromDays(7)),
        To = DateTime.Now
    }
};

_connector.Subscribe(historicalNewsSubscription);
```

Connecting to RSS for News

If you are working with connectors that do not provide news feeds (for example, Binance), you can add an additional news source via RSS:

```
// Create a Connector instance
var connector = new Connector();

// Add the main adapter for connecting to Binance
var binanceAdapter = new BinanceMessageAdapter(connector.TransactionIdGenerator)
{
    Key = "<Your API Key>",
    Secret = "<Your Secret Key>",
};
connector.Adapter.InnerAdapters.Add(binanceAdapter);
```

```
// Add an adapter for receiving news via RSS
var rssAdapter = new RssMessageAdapter(connector.TransactionIdGenerator)
{
    Address = "https://news-source.com/feed",
    IsEnabled = true
};
connector.Adapter.InnerAdapters.Add(rssAdapter);

// Subscribe to the news received event
connector.NewsReceived += OnNewsReceived;

// Connect
connector.Connect();
```

Notes

- Not all connectors support receiving news. For example, Binance does not provide a news feed through the API.
- For cryptocurrency market news, it is recommended to use specialized RSS sources.
- For news related to specific instruments, additional subscription configuration may be required.
- When working with a graphical interface, remember to update UI elements in the user interface thread using the `GuiAsync` or `GuiSync` methods.

See Also

- [Subscriptions](#)
- [Graphical Components](#)

Orders management

Orders Management

[S#](#) provides a wide range of functionality for efficient management of trading orders at all stages of their lifecycle. This section covers key aspects of working with orders in trading applications.

Main Features

- **Order creation** - formation of various types of trading orders (market, limit, stop orders, etc.)
- **Status tracking** - receiving up-to-date information about the current status of orders
- **Order management** - cancellation, modification, and replacement of existing orders
- **Event handling** - responding to events of registration, execution, and cancellation of orders
- **Bulk operations** - efficient work with groups of orders

Order Lifecycle

Each order in [S#](#) goes through certain lifecycle stages:

1. **Creation** - forming an [Order](#) object with necessary parameters
2. **Registration** - sending the order to the trading system
3. **Execution** - partial or complete execution of the order, formation of trades
4. **Completion** - complete execution, cancellation, or rejection of the order

The API provides detailed information about the state of the order at each stage, which allows building complex trading algorithms with precise execution control.

Integration with Trading Strategies

The order management mechanism is closely integrated with components for developing trading strategies [Strategy](#), which allows:

- Encapsulating order management logic within the strategy
- Automatically tracking and processing events of order registration and execution
- Using a unified approach to order management both in real trading and during testing

See also

[Create a New Order](#)

[Create a New Stop Order](#)

[Order States](#)

[Order Cancellation](#)

[Bulk Order Cancellation](#)

[Order Replacement](#)

[Transaction Number](#)

Create new order

Create new order

To create a new order, you should create an [Order](#) object that contains information about the order and register it on the exchange. Further, if you want to work with the order (for example, to cancel it or change it), then this [Order](#) object should be used. To register orders on the exchange, use the [Connector.RegisterOrder\(StockSharp.BusinessEntities.Order order\)](#) method, which sends the order to the server.

The example below shows the creation of an order and its registration on the exchange:

```
var order = new Order
{
    Type = OrderTypes.Limit,
    Portfolio = Portfolio.SelectedPortfolio,
    Volume = Volume.Text.To<decimal>(),
    Price = Price.Text.To<decimal>(),
    Security = Security,
    Direction = Sides.Buy,
};
_connector.RegisterOrder(order);
```

Recommended content

[Order cancel](#)

Create new stop order

Create new stop order

To create a new stop order, you need to create an [Order](#) object that contains information about the order and register it on the exchange.

Unlike a regular order, for a stop order, you need to specify the [Order.Type](#) property as [OrderTypes.Conditional](#) and set the [Order.Condition](#) property with the necessary order conditions.

Further, if you need to work with the order (for example, cancel it or change it), then this [Order](#) object shall be used. To register orders on the exchange, the [Connector.RegisterOrder](#)([StockSharp.BusinessEntities.Order](#) order) method is provided, which sends an order to the server.

```
Connector Connector = new Connector();
...
private void StopOrder_Click(object sender, RoutedEventArgs e)
{
    var order = new Order
    {
        Security = SecurityEditor.SelectedSecurity,
        Portfolio = PortfolioEditor.SelectedPortfolio,
        Price = decimal.Parse(TextBoxPrice.Text),
        Volume = decimal.Parse(TextBoxVolumePrice.Text),
        Direction = Sides.Buy,
        Type = OrderTypes.Conditional,
        Condition = new FixOrderCondition()
        {
            Type = FixOrderConditionTypes.StopLimit,
            StopLimitPrice = decimal.Parse(TextBoxStopLimitPrice.Text),
        }
    };
    Connector.RegisterOrder(order);
}
...
```

Each connection has its own implementation of the [OrderCondition](#) class, since each connection has its own unique features. For example, for [Kucoin](#) it is [KucoinOrderCondition](#), etc.

Recommended content

[Orders states](#)

Orders states

Orders states

StockSharp API provides the ability to receive information about orders through the built-in subscription mechanism. As with market data, transaction information uses a unified approach based on [Subscription](#).

Order-related Events

[Connector](#) provides the following events for processing order information:

Event	Description
OrderReceived	Event for receiving order information
OrderRegisterFailReceived	Event for order registration failure
OrderCancelFailReceived	Event for order cancellation failure
OrderEditFailReceived	Event for order modification failure
OwnTradeReceived	Event for receiving information about own trades

OrderStates enum

During its lifetime, an order goes through the following states:

[image: OrderStates]

- [OrderStates.None](#) - the order has been created in the trading algorithm but has not yet been sent for registration.
- [OrderStates.Pending](#) - the order has been sent for registration ([RegisterOrder](#)). The system is waiting for confirmation of its acceptance from the exchange. If the acceptance is successful, the [OrderReceived](#) event will be triggered, and the order will be transferred to the [OrderStates.Active](#) state. The [Order.Id](#) and [Order.ServerTime](#) properties will also be initialized. If the order is rejected, the [OrderRegisterFailReceived](#) event will be triggered with an error description, and the order will be transferred to the [OrderStates.Failed](#) state.
- [OrderStates.Active](#) - the order is active on the exchange. Such an order will remain active until its entire volume [Order.Volume](#) is executed, or it is forcibly canceled through `[CancelOrder](xref:StockSharp.BusinessEntities.ITransactionProvider.CancelOrder(StockSharp.BusinessEntities.Order))`. If the order is partially executed, the [OwnTradeReceived](#) events about new trades for the placed order are triggered, as well as the [OrderReceived](#) event, which passes a notification about the change in the order balance [Order.Balance](#). The latter event will also be triggered in case of order cancellation.
- [OrderStates.Done](#) - the order is no longer active on the exchange (it was fully executed or canceled).
- [OrderStates.Failed](#) - the order was not accepted by the exchange (or an intermediate system, such as the server part of the trading platform) for some reason.

Automatic Subscriptions

By default, [Connector](#) automatically creates subscriptions for transaction information when connecting ([SubscriptionsOnConnect](#)). This includes subscriptions to:

- Order information
- Trade information

- Position information
- Basic instrument lookup

Example of handling an order reception event:

```
private void InitConnector()
{
    // Subscribe to order reception event
    Connector.OrderReceived += OnOrderReceived;

    // Subscribe to own trade reception event
    Connector.OwnTradeReceived += OnOwnTradeReceived;

    // Subscribe to order registration failure event
    Connector.OrderRegisterFailReceived += OnOrderRegisterFailed;
}

private void OnOrderReceived(Subscription subscription, Order order)
{
    // Process the received order
    _ordersWindow.OrderGrid.Orders.TryAdd(order);

    // Important! Check if the order belongs to the current subscription
    // to avoid duplicate processing
    if (subscription == _myOrdersSubscription)
    {
        // Additional processing for the specific subscription
        Console.WriteLine($"Order: {order.TransactionId}, State: {order.State}");
    }
}
```

Manual Creation of Order Subscriptions

In some cases, you may need to explicitly request information about orders. For this, you can create separate subscriptions:

```
// Create a subscription for orders of a specific portfolio
var ordersSubscription = new Subscription(DataType.Transactions, portfolio)
{
    TransactionId = Connector.TransactionIdGenerator.GetNextId(),
};

// Handler for receiving orders
Connector.OrderReceived += (subscription, order) =>
{
    if (subscription == ordersSubscription)
    {
        Console.WriteLine($"Order: {order.TransactionId}, State: {order.State}, Portfolio: {order.Portfolio.Name}");
    }
};

// Start the subscription
Connector.Subscribe(ordersSubscription);
```

Checking Order Status

Extension methods are used to determine the current state of an order:

```
// Check order status
Order order = ...; // received order

// Is the order canceled
```



```

bool isCanceled = order.IsCanceled();

// Is the order fully executed
bool isMatched = order.IsMatched();

// Is the order partially executed
bool isPartiallyMatched = order.IsMatchedPartially();

// Is at least part of the order executed
bool isNotEmpty = order.IsMatchedEmpty();

// Get the executed volume
decimal matchedVolume = order.GetMatchedVolume();

```

Advanced Approach: Working with Multiple Subscriptions

In complex scenarios, you may need to work with multiple order subscriptions simultaneously. In this case, it's important to properly handle events to avoid duplication:

```

private Subscription _portfolio1OrdersSubscription;
private Subscription _portfolio2OrdersSubscription;

private void RequestOrdersForDifferentPortfolios()
{
    // Subscription for orders of the first portfolio
    _portfolio1OrdersSubscription = new Subscription(DataType.Transactions, _portfolio1);

    // Subscription for orders of the second portfolio
    _portfolio2OrdersSubscription = new Subscription(DataType.Transactions, _portfolio2);

    // Common handler for receiving orders
    Connector.OrderReceived += OnMultipleSubscriptionOrderReceived;

    // Start subscriptions
    Connector.Subscribe(_portfolio1OrdersSubscription);
    Connector.Subscribe(_portfolio2OrdersSubscription);
}

private void OnMultipleSubscriptionOrderReceived(Subscription subscription, Order order)
{
    // Determine which subscription the order belongs to
    if (subscription == _portfolio1OrdersSubscription)
    {
        // Process orders of the first portfolio
    }
    else if (subscription == _portfolio2OrdersSubscription)
    {
        // Process orders of the second portfolio
    }
}

```

Such an advanced approach with multiple subscriptions to orders should be used only in exceptional cases when the standard subscription mechanism is insufficient.

Asynchronous Nature of Transactions

Transaction sending (registration, replacement, or cancellation of orders) is performed asynchronously. This allows the trading program not to wait for confirmation from the exchange, but to continue working, which speeds up the reaction to changes in the market situation.

To track the status of an order, you need to subscribe to the corresponding events:

- [OrderReceived](#) for receiving order status updates
- [OrderRegisterFailReceived](#) for handling registration errors

See Also

- [Subscriptions](#)
- [Order States](#)
- [Creating a New Order](#)
- [Canceling Orders](#)

Order cancel

Order cancel

Order cancellation is needed if the market situation changes in disfavour of the issued order. To cancel orders, use the [ITransactionProvider.CancelOrder\(StockSharp.BusinessEntities.Order order\)](#) method in [S#](#).

```
// registeredOrder - successfully registered order.  
_connector.CancelOrder(registeredOrder);
```

Recommended content

[Orders mass cancel](#)

[Orders replacement](#)

Orders mass cancel

Orders mass cancel

To cancel several orders, use the `Connector.CancelOrders(System.Nullable<System.Boolean> isStopOrder, StockSharp.BusinessEntities.Portfolio portfolio, System.Nullable<StockSharp.Messages.Sides> direction, StockSharp.BusinessEntities.ExchangeBoard board, StockSharp.BusinessEntities.Security security, System.Nullable<StockSharp.Messages.SecurityTypes> securityType, System.Nullable<System.Int64> transactionId)` method, which cancels the active orders according to the parameters mask passed.

Orders mass cancel examples

To cancel all ordinary orders (`OrderTypes.Limit`) for the specified portfolio and instrument:

```
_connector.CancelOrders(false, MainWindow.Instance.Portfolio, null, null, security);
```

To cancel all orders for the specified instrument:

```
_connector.CancelOrders(null, null, null, null, security);
```

To cancel all long stop orders:

```
_connector.CancelOrders(true, null, Sides.Buy, null, null);
```

Orders replacement

Orders replacement

Replacing orders when creating trading algorithms is a more advanced method than canceling them and registering again. To replace the order you should call the

[Connector.ReRegisterOrder](#)([StockSharp.BusinessEntities.Order](#) oldOrder, [StockSharp.BusinessEntities.Order](#) newOrder) method.

As a result of order replacement a new [Order](#) object is created, which contains the old order information plus the changed part. Subsequently, if you want to work with the changed order (for example, to cancel it or change it again), you must use this new [Order](#) object.

The following example shows how "to move" the order at the best price:

```
if (registeredOrder.Security.BestBid != null && registeredOrder.Security.BestAsk != null)
{
    // registeredOrder - successfully registered order.
    var newOrder = registeredOrder.Clone();
    // changing the price to be the best on order book
    newOrder.Price = (registeredOrder.Direction == Sides.Buy ? registeredOrder.Security.BestBid :
        registeredOrder.Security.BestAsk).Price;
    // sending request the replace our order with new price
    _connector.ReRegisterOrder(registeredOrder, newOrder);
}
```

Recommended content

[Orders cancel](#)

Transaction number

Transaction number

When working with orders the main identifier is `Order.TransactionId`, not `Order.Id`. This is done because the `Order.Id` generated by the exchange. Because of this, immediately after the execution the `Connector.RegisterOrder(StockSharp.BusinessEntities.Order order)` method, for some time the `Order.Id` may not be initialized. Therefore, at once after the transaction sending, the trading program generates `Order.TransactionId`.

`Order.TransactionId` automatically generated by the `IdGenerator` class. This is an abstract class that has two standard implementations:

- `IncrementalIdGenerator` - installed by default. It increases the number by 1. The initial value set through the `IncrementalIdGenerator.Current`, property, and by default, the value is equal to the number of milliseconds since the start of the day.
- `MillisecondIdGenerator`. It generates the transaction number, which equals the number of milliseconds from the time the generator was created.

Instruments

In StockSharp, financial instruments are represented by the [Security](#) class, which is a fundamental element for working with trading data. This section covers the main aspects of working with financial instruments within the platform.

Base Security Class

[Security](#) represents a financial instrument traded on an exchange. An instrument can be a stock, futures contract, option, currency pair, cryptocurrency, and other assets. The class contains all the necessary information for identifying and trading the instrument:

- **Identification information** - code, ISIN, name, instrument class
- **Trading parameters** - price step, lot size, minimum volume
- **Market data** - current values of prices, volumes, order books, etc.
- **Calculated values** - parameters for derivatives, risk calculation, etc.

Types of Instruments

StockSharp supports working with all major types of financial instruments:

- **Stocks** - equity securities
- **Bonds** - debt securities
- **Futures** - derivative contracts on an underlying asset
- **Options** - contracts giving the right (but not the obligation) to buy or sell an underlying asset
- **Currency pairs** - instruments for trading on the forex market
- **Cryptocurrencies** - digital assets for trading on crypto exchanges
- **ETFs** - exchange-traded funds
- **Indices** - calculated indicators of the state of a market or sector

Instrument Baskets

In addition to regular instruments, StockSharp implements special classes for working with groups of instruments:

- [IndexSecurity](#) - an instrument representing an index based on underlying instruments
- [WeightedIndexSecurity](#) - an index with weight coefficients for each instrument
- [ContinuousSecurity](#) - a continuous instrument for working with a series of futures contracts

These classes allow you to create composite instruments and work with them in the same way as with regular instruments, receiving aggregated market data, calculating statistics, and executing trading operations.

Working with Instrument Information

StockSharp provides powerful tools for working with financial instrument information:

- **Instrument search** - by various criteria (code, name, class)
- **Filtering** - selection of instruments according to specified parameters
- **Storage** - saving instrument information to local or remote storage
- **Retrieving exchange information** - loading detailed information from the exchange

Instrument Identification

Each instrument in StockSharp has a unique identifier [SecurityId](#), which is used to unambiguously identify the instrument in the system. The identifier includes:

- **SecurityCode** - exchange code of the instrument
- **BoardCode** - trading venue code
- **Bloomberg/Reuters/ISIN** and other codes - alternative identification methods

Special Features

- **Continuous futures** - automatic "splicing" of historical data for a series of futures contracts
- **Composite instruments** - creation of virtual instruments based on several real ones
- **Special identifier *@ALL** - for working with all instruments of a certain class

See also

[Instrument Identifier](#)

[Identifier *@ALL](#)

[Continuous Futures](#)

[Index](#)

[Instrument Search](#)

Instrument search

Instrument Search

Most connectors to US stock exchanges (for example, [Interactive Brokers](#), [PolygonIO](#), and others) do not transfer all available instruments to the client after establishing a connection through the [IConnector.Connect](#) method. This is due to the large number of instruments traded on American exchanges and is done to reduce the load on broker servers and data sources.

Instrument Search Basics

For searching instruments in S#, a subscription mechanism is used, similar to receiving market data. This approach allows using uniform code for all types of data, including instruments.

Creating a Subscription for Instrument Search

To search for instruments, you need to create an instance of the [Subscription](#) class based on the [SecurityLookupMessage](#) message, which contains filtering parameters:

```
// Create a filter object for search
var lookupMessage = new SecurityLookupMessage
{
    // Set search criteria
    SecurityId = new SecurityId
    {
        // Search by instrument code (you can use a mask like "AAPL*")
        SecurityCode = "AAPL",
        // Optionally, you can specify the board code
        BoardCode = "NASDAQ"
    },
    // You can specify the instrument type
    SecurityType = SecurityTypes.Stock,
    // Set transaction ID
    TransactionId = Connector.TransactionIdGenerator.GetNextId()
};

// Create a subscription for instrument search
var subscription = new Subscription(lookupMessage);
```

Possible Filtering Parameters

The [SecurityLookupMessage](#) message allows setting the following search criteria:

- **SecurityId** — instrument identifier, containing:
 - **SecurityCode** — code or mask of the instrument code (for example, "AAPL" or "MS*")
 - **BoardCode** — exchange board code (for example, [ExchangeBoard.Nasdaq](#))
- **SecurityType** — instrument type ([SecurityTypes.Stock](#), [SecurityTypes.Future](#), etc.)
- **SecurityTypes** — array of instrument types for advanced search
- **Currency** — trading currency of the instrument
- **ExpiryDate** — expiration date (for derivatives)
- **Strike** — strike price (for options)
- **OptionType** — option type (for options)
- **Name** — instrument name or part of it

- **Class** — instrument class

Processing Search Results

After creating the subscription, you need to subscribe to events for receiving instruments and send the request:

```
// Handler for instrument receiving event
private void OnSecurityReceived(Subscription subscription, Security security)
{
    if (subscription.SubscriptionMessage is not SecurityLookupMessage)
        return;

    Console.WriteLine($"Found instrument: {security.Id} - {security.Name}, Type: {security.Type}");

    // Here you can add the instrument to a collection or perform other actions
    Securities.Add(security);
}

// Handler for search completion event
private void OnSubscriptionFinished(Subscription subscription)
{
    if (subscription.SubscriptionMessage is not SecurityLookupMessage)
        return;

    Console.WriteLine($"Search completed. Instruments found: {Securities.Count}");
}

// Subscription error handler
private void OnSubscriptionFailed(Subscription subscription, Exception error, bool isSubscribe)
{
    if (subscription.SubscriptionMessage is not SecurityLookupMessage)
        return;

    Console.WriteLine($"Instrument search error: {error.Message}");
}

// Subscribe to events
Connector.SecurityReceived += OnSecurityReceived;
Connector.SubscriptionFinished += OnSubscriptionFinished;
Connector.SubscriptionFailed += OnSubscriptionFailed;

// Send the instrument search request
Connector.Subscribe(subscription);
```

Complete Example of Instrument Search

Below is a complete example of a method for searching instruments:

```
public void FindSecurities(string searchCode, SecurityTypes? securityType = null)
{
    // Create an object for instrument search
    var lookupMessage = new SecurityLookupMessage
    {
        SecurityId = new SecurityId
        {
            SecurityCode = searchCode,
            // If you need to search on a specific board
            // BoardCode = ExchangeBoard.Nyse.Code,
        },
        SecurityType = securityType,
        TransactionId = Connector.TransactionIdGenerator.GetNextId()
    };

    // Create a subscription
    var subscription = new Subscription(lookupMessage);
```

```

// Clear the collection for search results
_searchResults.Clear();

// Temporary collection for accumulating results
var foundSecurities = new List<Security>();

// Subscription for receiving instruments
void OnSecurityReceived(Subscription sub, Security security)
{
    if (sub != subscription)
        return;

    // Add the found instrument to the collection
    foundSecurities.Add(security);
    Console.WriteLine($"Found: {security.Id}, {security.Name}");
}

// Subscription for search completion
void OnSubscriptionFinished(Subscription sub)
{
    if (sub != subscription)
        return;

    // Copy results to the main collection
    _searchResults.AddRange(foundSecurities);

    Console.WriteLine($"Search completed. Instruments found: {foundSecurities.Count}");

    // Unsubscribe from events
    Connector.SecurityReceived -= OnSecurityReceived;
    Connector.SubscriptionFinished -= OnSubscriptionFinished;
    Connector.SubscriptionFailed -= OnSubscriptionFailed;
}

// Handling subscription errors
void OnSubscriptionFailed(Subscription sub, Exception error, bool isSubscribe)
{
    if (sub != subscription)
        return;

    Console.WriteLine($"Instrument search error: {error.Message}");

    // Unsubscribe from events
    Connector.SecurityReceived -= OnSecurityReceived;
    Connector.SubscriptionFinished -= OnSubscriptionFinished;
    Connector.SubscriptionFailed -= OnSubscriptionFailed;
}

// Subscribe to events
Connector.SecurityReceived += OnSecurityReceived;
Connector.SubscriptionFinished += OnSubscriptionFinished;
Connector.SubscriptionFailed += OnSubscriptionFailed;

// Send the search request
Connector.Subscribe(subscription);
}

```

Example of Usage in a WPF Application

In a graphical application, instrument search is often called from a button click handler:

```

private void FindButton_Click(object sender, RoutedEventArgs e)
{
    // Get search criteria from the text field
    var searchText = SearchTextBox.Text;
}

```

```

if (string.IsNullOrEmpty(searchText))
{
    MessageBox.Show("Enter a search criterion");
    return;
}

// Create and send a search subscription
var lookupMessage = new SecurityLookupMessage
{
    SecurityId = new SecurityId { SecurityCode = searchText },
    // If a type is selected in the interface
    SecurityType = SecurityTypeComboBox.SelectedItem as SecurityTypes?
};

var subscription = new Subscription(lookupMessage);

// Here you can show a loading indicator
LoadingIndicator.Visibility = Visibility.Visible;

// Send the request
Connector.Subscribe(subscription);
}

```

Using SecurityLookupWindow

StockSharp also provides a ready-made dialog for instrument search — [SecurityLookupWindow](#):

```

private void ShowSecurityLookupWindow_Click(object sender, RoutedEventArgs e)
{
    var lookupWindow = new SecurityLookupWindow
    {
        // Specify the ability to search for all instruments
        // (if the connector supports this function)
        ShowAllOption = Connector.Adapter.IsSupportSecuritiesLookupAll(),

        // Set initial search criteria
        CriteriaMessage = new SecurityLookupMessage
        {
            SecurityId = new SecurityId { SecurityCode = "AAPL" },
            SecurityType = SecurityTypes.Stock
        }
    };

    // Show the window as a modal dialog
    if (lookupWindow.ShowDialog(this))
    {
        // If the user confirmed the selection, send the request
        Connector.Subscribe(new Subscription(lookupWindow.CriteriaMessage));
    }
}

```

Conclusion

The subscription mechanism in StockSharp provides a unified way to obtain data, including instrument search. This allows using the same approach for working with different connectors and data types, which significantly simplifies the development of trading applications.

Instrument identifier

Instrument identifier

The instruments in the [S#](#) from different sources have the [Security.Id](#) uniform identifier. This is done so that the trading algorithm code does not depend on the type of connection ([OpenECry](#), [Rithmic](#), [Interactive Brokers](#) etc.). For the instrument identifier the following syntax is used - **[instrument code]@[board code]**. For example, for the Apple inc shares identifier will be **AAPL@NASDAQ**. For the derivatives market instruments board will be **NYSE** (or other board name where **AAPL** futures trade). For example, for the June futures on the ES index the identifier will be **ESM5@NYSE**.

The [Hydra](#) application for market data download enumerates folders with a history based on the same mechanism.

Identifiers generation algorithm overriding

1. To start the instrument identifiers generation on the own algorithm, you must create the descendant of the [SecurityIdGenerator](#) class, and override the [SecurityIdGenerator.GenerateId\(System.String secCode, System.String boardCode \)](#) method:

```
class CustomSecurityIdGenerator : SecurityIdGenerator
{
    public override string GenerateId(string secCode, ExchangeBoard board)
    {
        // will be generate in CODE--BOARD form
        return secCode + "--" + board.Code;
    }
}
```

2. Then, the created generator must be passed to the connector:

```
connector.SecurityIdGenerator = new CustomSecurityIdGenerator();
```

Identifier *@ALL

Identifier *@ALL

If the instrument is traded on several boards, then as the board code you can use the "ALL". For example, the instrument with the AAPL@ALL identifier means that it collects data from all boards for ticker AAPL (AAPL@BATS, AAPL@ICE, etc.) and these data also have a common order book.

The board code is used to send the order in a specific trading system. If you specify "ALL", then the broker or the exchange will determine the order route. In this case, the broker or the exchange independently decides on which board it is better to match an order. On the contrary, if you specify a particular board, such as BATS, then the order will be sent only to BATS.

Continuous futures

Continuous futures

[ContinuousSecurity](#) - continuous instrument (typically futures), containing the instruments affected by expiration (expiry of period of activity).

For example, two futures of the ES index - **ESM5** and **ESU5**. When the **ESM5** expired it automatically switches to the next instrument - **ESU5**.

[ContinuousSecurity](#) can be traded in the same way as [Security](#). Prior to the **RIM5** expiration an algo will be carried with this instrument. After the expiration an algo will be carried with **RIU5**, etc.

Creating ContinuousSecurity

1. To declare the compound instruments that will be included in the [ContinuousSecurity](#) and in the [ContinuousSecurity](#) itself:

```
private Security _esm5;  
private Security _esu5;  
private ContinuousSecurity _es;
```

2. To create the [ContinuousSecurity](#):

```
_es = new ContinuousSecurity { ExchangeBoard = ExchangeBoard.Nyse, Id = "ES" };
```

3. To add the compound instruments to it, specify the date and time of expiration for each added instrument:

```
_es.ExpirationJumps.Add(_esm5, new DateTime(2015, 6, 15, 18, 45, 00));  
_es.ExpirationJumps.Add(_esu5, new DateTime(2015, 9, 15, 18, 45, 00));
```

Index

Index

[IndexSecurity](#) - the index built from instruments. For example, to set a spread in arbitrage or pair trading. It has the following implementations.

1. [ExpressionIndexSecurity](#) - is the index built from a combination of several instruments through the [ExpressionIndexSecurity.Formula](#) mathematical formula.
2. [WeightedIndexSecurity](#) - is the index built from instruments using weighting factors [WeightedIndexSecurity.Weights](#).

Creating ExpressionIndexSecurity

1. To declare the compound instruments that will be included in [ExpressionIndexSecurity](#) and in the [ExpressionIndexSecurity](#) itself:

```
ConfigManager.RegisterService<ICompilerService>(new RoslynCompilerService());  
// Or  
//ConfigManager.RegisterService<ICompilerService>(new  
Fw40CompilerService(Directory.GetCurrentDirectory(), Directory.GetCurrentDirectory()));  
private Security _instr1;  
private Security _instr2;  
private ExpressionIndexSecurity _indexInstr;  
private const string _secCode1 = "GZM5";  
private const string _secCode2 = "LKM5";
```

2. To create the [ExpressionIndexSecurity](#):

```
_indexInstr = new ExpressionIndexSecurity  
{  
    Id = "IndexInstr@NYSE",  
    Expression = "ESM5@NYSE/APM5@NYSE",  
    Board = ExchangeBoard.Nyse,  
};
```

Creating WeightedIndexSecurity

1. To declare the compound instruments that will be included in [WeightedIndexSecurity](#) and in the [WeightedIndexSecurity](#) itself:

```
private Security _instr1;  
private Security _instr2;  
private WeightedIndexSecurity _indexInstr;  
private const string _secCode1 = "GZM5";  
private const string _secCode2 = "LKM5";
```

2. To create the [WeightedIndexSecurity](#):

```
_indexInstr = new WeightedIndexSecurity() { ExchangeBoard = ExchangeBoard.Nyse, Id = "IndexInstr"  
};
```

3. To add the compound instruments to it:

```
_indexInstr.Weights.Add(_instr1, 1);  
_indexInstr.Weights.Add(_instr2, -1);
```


Volume Continuous Security

Volume Continuous Futures (VolumeContinuousSecurity)

Overview

The `VolumeContinuousSecurity` class represents a continuous futures contract where the transition (rollover) between contracts occurs based on trading volume or open interest. This differs from `ExpirationContinuousSecurity`, where switching is performed according to predefined expiration dates.

Both classes inherit from `ContinuousSecurity`, which in turn inherits from `BasketSecurity`.

Difference from ExpirationContinuousSecurity

Characteristic	ExpirationContinuousSecurity	VolumeContinuousSecurity
Rollover condition	Expiration date (fixed)	Volume or open interest threshold
Configuration	SecurityId -> DateTime dictionary	SecurityId list + VolumeLevel
Predictability	Switching by schedule	Switching by market conditions
Basket code	CE	CV

`ExpirationContinuousSecurity` requires manually specifying transition dates for each contract. `VolumeContinuousSecurity` automatically switches to the next contract when its trading volume (or open interest) exceeds the specified threshold.

Main Properties

```
public class VolumeContinuousSecurity : ContinuousSecurity
{
    // List of inner securities (contracts), ordered by rollover sequence
    public SynchronizedList<SecurityId> InnerSecurities { get; }

    // Use open interest instead of volume for rollover determination
    public bool IsOpenInterest { get; set; }

    // Volume threshold at which switching to the next contract occurs
    public Unit VolumeLevel { get; set; }
}
```

The `VolumeLevel` property has the `Unit` type, which allows specifying both absolute and percentage values.

Usage Example

```
using StockSharp.Algo;
using StockSharp.Messages;

// Create a volume-based continuous futures
var continuous = new VolumeContinuousSecurity
{
    Id = "RTS-CONT@FORTS",
```

```

    Board = ExchangeBoard.Forts,
};

// Add contracts in rollover order
continuous.InnerSecurities.AddRange(new[]
{
    "RTS-3.26@FORTS".ToSecurityId(),
    "RTS-6.26@FORTS".ToSecurityId(),
    "RTS-9.26@FORTS".ToSecurityId(),
});

// Set volume threshold for switching
continuous.VolumeLevel = new Unit(10000);

// Or use open interest
continuous.IsOpenInterest = true;
continuous.VolumeLevel = new Unit(50000);

```

Example with ExpirationContinuousSecurity for Comparison

```

using StockSharp.Algo;
using StockSharp.Messages;

// Expiration-based continuous futures
var expContinuous = new ExpirationContinuousSecurity
{
    Id = "RTS-CONT-EXP@FORTS",
    Board = ExchangeBoard.Forts,
};

// Specify exact transition dates for each contract
expContinuous.ExpirationJumps.Add(
    "RTS-3.26@FORTS".ToSecurityId(),
    new DateTime(2026, 3, 15)
);
expContinuous.ExpirationJumps.Add(
    "RTS-6.26@FORTS".ToSecurityId(),
    new DateTime(2026, 6, 15)
);

```

When to Use

VolumeContinuousSecurity is suitable for situations where:

- Exact rollover dates are not known in advance
- Switching based on liquidity (trading volume or open interest) is required
- A more adaptive transition that responds to market conditions is needed

ExpirationContinuousSecurity is preferred when expiration dates are known in advance and deterministic rollover is required.

Market-data storage

Market-data storage

[S#](#) provides functionality for data storage and loading. Data storage may be necessary in a few cases:

1. During working of trading strategy, when necessary to save the state (orders, positions, settings, etc.);
2. Market data saving from the trading terminal for algorithm testing in real time;
3. Data accumulation for analysis and data mining.

[S#](#) makes clear the saving mechanism due to high-level access and the technical details hiding (for more details see [API](#)). And also versatile, with possibility to expand the types of supported storages.

Recommended content

[Working with the API](#)

[Working with Remote Storage](#)

Working with the API

Working with the API

Preparation

For working with historical data in the examples, a NuGet package with historical data samples is used. It can be installed from the [NuGet Gallery](#). This package provides a set of data that can be used to demonstrate working with the storage.

All codes are available in the [StockSharp repository](#).

Creating a Storage Registry

To work with market data storage in StockSharp, the [StorageRegistry](#) class is used. When creating an object of this class, you can set the path to the default storage via the [StorageRegistry.DefaultDrive](#) property or specify a specific folder for working with historical data using the [LocalMarketDataDrive](#).

```
// Creating StorageRegistry with default path
var storageRegistry = new StorageRegistry();
```

```
// Creating StorageRegistry with the path to data from the NuGet package
var pathHistory = Paths.HistoryDataPath; // path to data from the NuGet package
var localDrive = new LocalMarketDataDrive(pathHistory);
var storageRegistry = new StorageRegistry()
{
    DefaultDrive = localDrive,
};
```

Retrieving Data

Through the [StorageRegistry](#), you can access various types of market data for the desired time range. The methods used for this are:

- [StorageRegistry.GetTimeFrameCandleMessageStorage](#) for candles
- [StorageRegistry.GetTickMessageStorage](#) for ticks
- [StorageRegistry.GetQuoteMessageStorage](#) for order books

Each of these methods returns the corresponding storage, from which data can be loaded using the `LoadAsync` method, specifying the start and end dates.

```
// Retrieving candles
var securityId = "SBER@TQBR".ToSecurityId();
var candleStorage = storageRegistry.GetTimeFrameCandleMessageStorage(securityId,
    TimeSpan.FromMinutes(1), StorageFormats.Binary);
var candles = candleStorage.LoadAsync(new DateTime(2020, 4, 1), new DateTime(2020, 4, 2));

await foreach (var candle in candles)
{
    Console.WriteLine(candle);
}
```

```
// Retrieving ticks
var tradeStorage = storageRegistry.GetTickMessageStorage(securityId, StorageFormats.Binary);
var trades = tradeStorage.LoadAsync(new DateTime(2020, 4, 1), new DateTime(2020, 4, 2));
```

```
await foreach (var trade in trades)
{
    Console.WriteLine(trade);
}
```

```
// Retrieving order books
var marketDepthStorage = storageRegistry.GetQuoteMessageStorage(securityId, StorageFormats.Binary);
var marketDepths = marketDepthStorage.LoadAsync(new DateTime(2020, 4, 1), new DateTime(2020, 4, 2));

await foreach (var marketDepth in marketDepths)
{
    Console.WriteLine(marketDepth);
}
```

Saving Data

To save new data to the existing storage, use the `SaveAsync` method of the corresponding storage. This allows you to supplement historical data with new values.

```
// Saving new candles
var newCandles = new List<CandleMessage>
{
    // New CandleMessage objects are created here
};
await candleStorage.SaveAsync(newCandles);
```

```
// Saving new ticks
var newTrades = new List<ExecutionMessage>
{
    // New ExecutionMessage objects for ticks are created here
};
await tradeStorage.SaveAsync(newTrades);
```

```
// Saving new order books
var newMarketDepths = new List<QuoteChangeMessage>
{
    // New QuoteChangeMessage objects for order books are created here
};
await marketDepthStorage.SaveAsync(newMarketDepths);
```

Deleting Data

To delete data for a specific period, use the `DeleteAsync` method of the corresponding storage. Be careful when deleting data from the sample package.

```
// Deleting candles for the specified period
await candleStorage.DeleteAsync(new DateTime(2020, 4, 1), new DateTime(2020, 4, 2));
```

```
// Deleting ticks for the specified period
await tradeStorage.DeleteAsync(new DateTime(2020, 4, 1), new DateTime(2020, 4, 2));
```

```
// Deleting order books for the specified period
await marketDepthStorage.DeleteAsync(new DateTime(2020, 4, 1), new DateTime(2020, 4, 2));
```

These operations allow you to effectively manage historical data, whether loaded through [Hydra](#), provided in the NuGet package, or created during the operation of your application.

Working with Remote Storage

Working with Remote Storage

Introduction

In addition to local storage, the API provides the capability to work with remote market data storage. This is especially useful when using Hydra in [server mode](#) or when connecting to a [Hydra server](#).

Connecting to Remote Storage

To work with remote storage, use the [RemoteMarketDataDrive](#) class.

```
// Creating RemoteMarketDataDrive
var remoteDrive = new RemoteMarketDataDrive(RemoteMarketDataDrive.DefaultAddress, new
FixMessageAdapter(new IncrementalIdGenerator()))
{
    Credentials = { Email = "hydra_user", Password = "hydra_user".To<SecureString>() }
};

// This code creates an instance of RemoteMarketDataDrive to connect to remote storage.
// It uses the default address and FixMessageAdapter for communication.
// Credentials are set for authentication.
```

Loading Instrument Information

Before loading market data, you need to get information about available instruments.

```
// Loading instrument information
var exchangeInfoProvider = new InMemoryExchangeInfoProvider();
remoteDrive.LookupSecurities(Extensions.LookupAllCriteriaMessage, registry.Securities,
s => securityStorage.Save(s.ToSecurity(exchangeInfoProvider), false), () => false,
(c, t) => Console.WriteLine($"Downloaded [{c}]/[{t}]"));

var securities = securityStorage.LookupAll();

// This code loads information about all available instruments from the remote storage.
// The loaded instruments are saved in local storage and output to the console.
```

Loading Market Data

After obtaining instrument information, you can proceed to load market data.

```
// Loading market data
foreach (var dataType in remoteDrive.GetAvailableDataTypes(secId, format))
{
    var localStorage = storageRegistry.GetStorage(secId, dataType.MessageType, dataType.Arg,
localDrive, format);
    var remoteStorage = remoteDrive.GetStorageDrive(secId, dataType, format);

    // ... (data loading code)
}

// This loop iterates through all available data types for the specified instrument.
// For each data type, a local storage is created and remote storage is accessed.
```

Saving Data Locally

Loaded data can be saved locally for further use.

```
// Saving data locally
foreach (var dateTime in dates)
{
    using (var stream = remoteStorage.LoadStream(dateTime))
    {
        if (stream == Stream.Null)
            continue;

        localStorage.Drive.SaveStream(dateTime, stream);
    }

    // ... (data output code)
}

// This code loads data for each date from the remote storage and saves it to local storage.
```

Using Data for Testing

Loaded and locally saved data can be used for testing trading strategies with the [HistoryEmulationConnector](#).

```
// Using HistoryEmulationConnector
var connector = new HistoryEmulationConnector(secProvider, new[] { pf }, new StorageRegistry
{ DefaultDrive = remoteDrive });
```

Obtaining Available Date Ranges

```
// Working with various data types
foreach (var dataType in remoteDrive.GetAvailableDataTypes(secId, format))
{
    // ... (data processing code)

    // Error handling and logging
    Console.WriteLine($"Remote {dataType}: {remoteStorage.Dates.FirstOrDefault()}-
{remoteStorage.Dates.LastOrDefault()}");
    Console.WriteLine($"{{dataType}}={{dateTime}}");
}
```

Conclusion

The API's remote market data storage functionality provides flexible capabilities for obtaining and using historical data. This allows for efficient testing of trading strategies and market analysis using extensive datasets available through the [Hydra server](#).

Storage Formats

StockSharp supports two market data storage formats defined by the [StorageFormats](#) enumeration: **Binary** and **CSV**. Each format has its own advantages and is suitable for different use cases.

Binary Format

The binary format is the primary high-performance data storage format in StockSharp. The implementation is located in the `BinaryMarketDataSerializer` class within `Algo/Storages/Binary/`.

Features

- **Compactness** -- data is serialized in binary form with compression, ensuring minimal file sizes.
- **Performance** -- reading and writing are significantly faster compared to text formats.
- **Metadata** -- the `BinaryMetaInfo` class stores auxiliary information: first and last prices, fractional values, timestamps. This allows quickly retrieving general data information without fully reading the file.
- **Compression-ready architecture** -- the format is designed from the ground up for efficient streaming compression.

Binary files have the `.bin` extension.

Supported Data Types

The binary format supports serialization of all major market data types: candles, ticks (trades), order books (Level2), Level1 data, orders, and own trades. Each type has a specialized serializer optimized for the structure of the specific data.

CSV Format

The CSV (Comma-Separated Values) text format is implemented in the `CsvMarketDataSerializer` class located in `Algo/Storages/Csv/`.

Features

- **Readability** -- files can be opened in any text editor or in Excel for visual data analysis.
- **Editability** -- data can be manually corrected when necessary.
- **Metadata** -- the `CsvMetaInfo` class provides metadata storage with encoding support. It contains the `IncrementalOnly` property (incremental data support) and `LastId` (last record identifier).
- **Larger file size** -- text representation takes significantly more disk space.
- **Slower processing** -- parsing text data requires additional computational resources.

CSV files have the `.csv` extension.

When to Use Which Format

Scenario	Recommended Format
Production use	Binary
Large data volumes	Binary

Performance-critical	Binary
Debugging and diagnostics	CSV
Visual data analysis	CSV
Integration with external tools	CSV
Manual data correction	CSV

File Organization on Disk

Data on disk is organized according to the following path structure:

```
{root_folder}/{first_letter}/{instrument_identifiler}/{yyyy_MM_dd}/{file_name}.{extension}
```

Where the extension is .bin for binary format or .csv for text format. This hierarchical organization ensures fast data lookup by instrument and date.

For example, 5-minute candles for the SBER@TQBR instrument on April 1, 2024 in binary format would be located at a path like:

```
Storage/S/SBER@TQBR/2024_04_01/candles_5m.bin
```

Converting Between Formats

StockSharp allows loading data in one format and saving it in another. This can be useful, for example, for exporting binary data to CSV for analysis in external tools:

```
// Load from binary storage
var binaryStorage = storageRegistry.GetTimeFrameCandleMessageStorage(
    securityId, TimeSpan.FromMinutes(5), StorageFormats.Binary);
var candles = await binaryStorage.LoadAsync(from, to).ToArrayAsync();

// Save to CSV storage
var csvStorage = storageRegistry.GetTimeFrameCandleMessageStorage(
    securityId, TimeSpan.FromMinutes(5), StorageFormats.Csv);
await csvStorage.SaveAsync(candles);
```

Code Example

The format is selected when creating a storage via [StorageRegistry](#):

```
var storageRegistry = new StorageRegistry();

// Create candle storage in binary format
var binaryStorage = storageRegistry.GetTimeFrameCandleMessageStorage(
    securityId,
    TimeSpan.FromMinutes(5),
    StorageFormats.Binary);

// Create candle storage in CSV format
var csvStorage = storageRegistry.GetTimeFrameCandleMessageStorage(
    securityId,
    TimeSpan.FromMinutes(5),
    StorageFormats.Csv);

// Load data
var from = new DateTime(2024, 1, 1);
var to = new DateTime(2024, 1, 31);
var candles = await binaryStorage.LoadAsync(from, to).ToArrayAsync();
```

The storage format can also be specified when working with tick data and order books:

```
// Ticks in binary format
var tickStorage = storageRegistry.GetTickMessageStorage(
    securityId, StorageFormats.Binary);

// Order books in CSV format
var depthStorage = storageRegistry.GetQuoteMessageStorage(
    securityId, StorageFormats.Csv);
```

See Also

- [Working with the API](#)
- [Storage Drives](#)

Storage Drives

Storage Drives

Storage drives in StockSharp are responsible for the physical placement of market data -- on a local disk or on a remote server. The base interface [IMarketDataDrive](#) defines the common contract for all implementations.

IMarketDataDrive -- Base Interface

The [IMarketDataDrive](#) interface provides the following key capabilities:

- **Path** -- path to the data storage.
- **GetAvailableSecuritiesAsync()** -- retrieve a list of all available instruments in the storage.
- **GetAvailableDataTypesAsync()** -- retrieve a list of data types available for a specific instrument.
- **GetStorageDrive()** -- obtain a storage drive for a specific instrument and data type.
- **VerifyAsync()** -- verify the integrity of the storage.
- **LookupSecuritiesAsync()** -- search for instruments by specified criteria.

LocalMarketDataDrive -- Local File Storage

The [LocalMarketDataDrive](#) class is the primary drive implementation that stores data on the local disk in the file system.

Key Features

- **File system** -- data is organized in a hierarchical directory structure by instruments and dates.
- **Index system** -- the internal `Index` class provides fast data access without scanning the file system. Index files are stored in the format `{instrument_path}{file_name}Dates2.bin`.
- **Thread safety** -- data access is protected by locking mechanisms for correct operation in multi-threaded applications.
- **Index building** -- the `BuildIndexAsync()` method allows rebuilding indexes for improved performance after bulk data operations.

Usage Example

```
// Create a local drive with a specified path
var localDrive = new LocalMarketDataDrive(Path.Combine(
    Directory.GetCurrentDirectory(), "Storage"));

// Use with the storage registry
var storageRegistry = new StorageRegistry
{
    DefaultDrive = localDrive,
};

// Retrieve the list of available instruments
await foreach (var secId in localDrive.GetAvailableSecuritiesAsync())
{
    Console.WriteLine(secId);
}
```

RemoteMarketDataDrive -- Remote Storage

The [RemoteMarketDataDrive](#) class allows connecting to a remote Hydra server for accessing market data over the network.

Connection Settings

- **Address** -- remote server address. Default is 127.0.0.1:5002.
- **Credentials** -- authentication credentials (Email and Password).
- **TargetCompld** -- target component identifier, default is "StockSharpHydraMD".
- **SecurityBatchSize** -- batch size when loading instruments, default is 1000.
- **Timeout** -- connection timeout, default is 2 minutes.

Usage Example

```
// Create a remote drive
var remoteDrive = new RemoteMarketDataDrive
{
    Address = "192.168.1.100:5002".To<EndPoint>(),
    Credentials = { Email = "user", Password = "pass".Secure() }
};

// Retrieve available data types for an instrument
var secId = "SBER@TQBR".ToSecurityId();
await foreach (var dataType in remoteDrive.GetAvailableDataTypesAsync(secId, StorageFormats.Binary))
{
    Console.WriteLine(dataType);
}
```

For more details on working with remote storage, see the [Working with Remote Storage](#) section.

DriveCache -- Drive Management

The [DriveCache](#) class manages a collection of storage drives and provides caching for reuse.

Key Methods and Properties

- **GetDrive(path)** -- get an existing drive by path or create a new one.
- **DeleteDrive(drive)** -- remove a drive from the cache.
- **TryDefaultDrive** -- the first available [LocalMarketDataDrive](#).
- **NewDriveCreated** -- event for new drive creation.
- **DriveDeleted** -- event for drive deletion.
- **Changed** -- event for drive collection changes.

The class implements the `IPersistable` interface, which allows saving and loading drive configurations.

Usage Example

```
// Create a cache with a default local drive
var defaultDrive = new LocalMarketDataDrive(Path.Combine(
    Directory.GetCurrentDirectory(), "Storage"));
var driveCache = new DriveCache(defaultDrive);

// Get or create a drive by path
```

```
var anotherDrive = driveCache.GetDrive(@"D:\MarketData");

// Subscribe to events
driveCache.NewDriveCreated += drive =>
    Console.WriteLine($"Drive created: {drive.Path}");
```

See Also

- [Working with the API](#)
- [Working with Remote Storage](#)
- [Storage Formats](#)

Snapshot System

Snapshot System

Snapshots in StockSharp represent a mechanism for saving the latest actual state of market data. Instead of scanning the entire history, snapshots allow instantly obtaining the current Level1 value, order book, position, or transaction.

Purpose of Snapshots

When working with streaming data, it is often necessary to know the latest state of an instrument -- current price, order book, open position. Without snapshots, obtaining this information would require loading and processing the entire history. The snapshot system solves this problem by saving the latest state of each object and providing access to it in minimal time.

ISnapshotStorage -- Snapshot Storage Interface

The [ISnapshotStorage](#) interface defines the base contract for working with snapshots. The typed version `ISnapshotStorage<TKey, TMessage>` provides the following methods:

- **Update(message)** -- save or update a snapshot. If a snapshot for the given key already exists, it will be updated.
- **Get(key)** -- retrieve a snapshot by key (e.g., by instrument identifier).
- **GetAll(from, to)** -- retrieve all snapshots for the specified date range.
- **Clear(key)** -- delete the snapshot for a specific key.
- **ClearAll()** -- delete all snapshots.

ISnapshotSerializer -- Snapshot Serialization

The [ISnapshotSerializer](#) interface is responsible for converting snapshots to binary representation and back:

- **DataType** -- snapshot data type information.
- **Version** -- serialization format version.
- **Serialize(version, message)** -- serialize a message to a byte array.
- **Deserialize(version, buffer)** -- deserialize a byte array back to a message.
- **GetKey(message)** -- extract the key from a message.
- **Update(message, changes)** -- apply incremental changes to an existing snapshot.

SnapshotRegistry -- Snapshot Registry

The [SnapshotRegistry](#) class is the central component for snapshot management. It implements the `ISnapshotRegistry` interface and coordinates the operation of all snapshot storages.

Key Features

- **Storage management** -- provides access to snapshot storages for various data types.
- **Periodic writing** -- changes are flushed to disk every 10 seconds, ensuring a balance between performance and reliability.
- **Thread safety** -- all operations are safe for use from multiple threads.

File Organization

Snapshot files are stored at the following path:

```
{path}/{yyyy_MM_dd}/{serializer_name}.bin
```

Built-in Serializers

StockSharp includes four serializers for the main market data types:

Serializer	Message Type	Purpose
Level1BinarySnapshotSerializer	Level1ChangeMessage	Level1 data (prices, volumes, spreads)
QuotesBinarySnapshotSerializer	QuoteChangeMessage	Order book
PositionBinarySnapshotSerializer	PositionChangeMessage	Positions
TransactionBinarySnapshotSerializer	ExecutionMessage	Transactions (orders and trades)

Each serializer supports format versioning, ensuring backward compatibility when updating StockSharp.

Code Example

Creating a Snapshot Registry

```
var snapshotRegistry = new SnapshotRegistry(Path.Combine(
    Directory.GetCurrentDirectory(), "Snapshots"));
```

Working with Level1 Snapshots

```
// Get the Level1 snapshot storage
var level1Snapshots = snapshotRegistry.GetSnapshotStorage(
    DataType.Level1);

// Save a snapshot
var level1Msg = new Level1ChangeMessage
{
    SecurityId = "SBER@TQBR".ToSecurityId(),
    ServerTime = DateTimeOffset.Now,
};
level1Msg.TryAdd(Level1Fields.LastTradePrice, 260.5m);
level1Msg.TryAdd(Level1Fields.BestBidPrice, 260.4m);
level1Msg.TryAdd(Level1Fields.BestAskPrice, 260.6m);

level1Snapshots.Update(level1Msg);
```

Retrieving a Snapshot

```
// Get the latest snapshot for an instrument
var secId = "SBER@TQBR".ToSecurityId();
var snapshot = level1Snapshots.Get(secId);

if (snapshot != null)
{
    Console.WriteLine($"Last price: {snapshot.Changes[Level1Fields.LastTradePrice]}");
}
```

See Also

- [Working with the API](#)
- [Storage Formats](#)
- [Storage Drives](#)

Candles

Candles

[S#](#) supports the following types of candles:

- [TimeFrameCandleMessage](#) - a candle based on a time interval, timeframe. You can set both popular intervals (minutes, hours, daily) and customized ones. For example, 21 seconds, 4.5 minutes, etc.
- [RangeCandleMessage](#) - a price range candle. A new candle is created when a trade appears with a price that exceeds the acceptable limits. The acceptable limit is formed each time based on the price of the first trade.
- [VolumeCandleMessage](#) - a candle is formed until the total volume of trades exceeds a specified limit. If a new trade exceeds the allowable volume, it is included in a new candle.
- [TickCandleMessage](#) - the same as [VolumeCandleMessage](#), but the number of trades is used as a limitation instead of volume.
- [PnFCandleMessage](#) - a point-and-figure chart candle (X-O chart).
- [RenkoCandleMessage](#) - Renko candle.

How to work with candles is shown in the example located in the *Samples/02_Candles/01_Realtime* folder.

The following images show [TimeFrameCandleMessage](#) and [RangeCandleMessage](#) charts:

[image: sample timeframecandles]

[image: sample rangecandles]

Starting Data Retrieval

1. To get candles, create a subscription using the [Subscription](#) class:

```
// Create a subscription to 5-minute candles
var subscription = new Subscription(
    DataType.TimeFrame(TimeSpan.FromMinutes(5)), // Data type with timeframe specification
    security) // Instrument
{
    // Configure additional parameters through the MarketData property
    MarketData =
    {
        // Period for which we request historical data (last 30 days)
        From = DateTime.Today.Subtract(TimeSpan.FromDays(30)),
        To = DateTime.Now
    }
};
```

2. To receive candles, subscribe to the [Connector.CandleReceived](#) event, which signals the appearance of a new value for processing:

```
// Subscribe to the candle reception event
_connector.CandleReceived += OnCandleReceived;

// Candle reception event handler
private void OnCandleReceived(Subscription subscription, ICandleMessage candle)
{
    // Here subscription is the subscription object we created
    // candle - the received candle

    // Check if the candle belongs to our subscription
```

```

if (subscription == _candleSubscription)
{
    // Draw the candle on the chart
    Chart.Draw(_candleElement, candle);
}
}

```

The [Chart](#) graphical component is used to display candles.

3. Next, start the subscription through the [Connector.Subscribe](#) method:

```

// Start the subscription
_connector.Subscribe(subscription);

```

After this, the [Connector.CandleReceived](#) event will begin to be called.

4. The [Connector.CandleReceived](#) event is called not only when a new candle appears but also when the current one changes.

If you need to display only **"complete"** candles, you need to check the [ICandleMessage.State](#) property of the received candle:

```

private void OnCandleReceived(Subscription subscription, ICandleMessage candle)
{
    // Check if the candle belongs to our subscription
    if (subscription != _candleSubscription)
        return;

    // Check if the candle is completed
    if (candle.State == CandleStates.Finished)
    {
        // Create data for drawing
        var chartData = new ChartDrawData();
        chartData.Group(candle.OpenTime).Add(_candleElement, candle);

        // Draw the candle on the chart
        this.GuiAsync(() => Chart.Draw(chartData));
    }
}

```

5. Additional parameters can be configured for the subscription:

- **Candle building mode** - determines whether ready-made data will be requested or built from another data type:

```

// Request only ready-made data
subscription.MarketData.BuildMode = MarketDataBuildModes.Load;

// Only build from another data type
subscription.MarketData.BuildMode = MarketDataBuildModes.Build;

// Request ready-made data, and if not available - build
subscription.MarketData.BuildMode = MarketDataBuildModes.LoadAndBuild;

```

- **Source for building candles** - indicates from which data type to build candles if they are not directly available:

```

// Building candles from tick trades
subscription.MarketData.BuildFrom = DataType.Ticks;

// Building candles from order books
subscription.MarketData.BuildFrom = DataType.MarketDepth;

```

```
// Building candles from Level1
subscription.MarketData.BuildFrom = DataType.Level1;
```

- **Field for building candles** - must be specified for certain data types:

```
// Building candles from the best bid price in Level1
subscription.MarketData.BuildField = Level1Fields.BestBidPrice;

// Building candles from the best ask price in Level1
subscription.MarketData.BuildField = Level1Fields.BestAskPrice;

// Building candles from the middle of the spread in the order book
subscription.MarketData.BuildField = Level1Fields.SpreadMiddle;
```

- **Volume profile** - calculation of volume profile for candles:

```
// Enable volume profile calculation
subscription.MarketData.IsCalcVolumeProfile = true;
```

Examples of Subscriptions to Different Candle Types

Candles with Standard Timeframe

```
// 5-minute candles
var timeFrameSubscription = new Subscription(
    DataType.TimeFrame(TimeSpan.FromMinutes(5)),
    security);
_connector.Subscribe(timeFrameSubscription);
```

Loading Only Historical Candles

```
// Loading only historical candles without transitioning to real-time
var historicalSubscription = new Subscription(
    DataType.TimeFrame(TimeSpan.FromMinutes(5)),
    security)
{
    MarketData =
    {
        From = DateTime.Today.Subtract(TimeSpan.FromDays(30)),
        To = DateTime.Today, // Specify end date
        BuildMode = MarketDataBuildModes.Load // Only load ready-made data
    }
};
_connector.Subscribe(historicalSubscription);
```

Building Non-Standard Timeframe Candles from Ticks

```
// Candles with a 21-second timeframe, built from ticks
var customTimeFrameSubscription = new Subscription(
    DataType.TimeFrame(TimeSpan.FromSeconds(21)),
    security)
{
    MarketData =
    {
        BuildMode = MarketDataBuildModes.Build,
        BuildFrom = DataType.Ticks
    }
};
_connector.Subscribe(customTimeFrameSubscription);
```

Building Candles from Order Book Data

```
// Candles built from the middle of the spread in the order book
var depthBasedSubscription = new Subscription(
    DataType.TimeFrame(TimeSpan.FromMinutes(1)),
    security)
{
    MarketData =
    {
        BuildMode = MarketDataBuildModes.Build,
        BuildFrom = DataType.MarketDepth,
        BuildField = Level1Fields.SpreadMiddle
    }
};
_connector.Subscribe(depthBasedSubscription);
```

Candles with Volume Profile

```
// 5-minute candles with volume profile calculation
var volumeProfileSubscription = new Subscription(
    DataType.TimeFrame(TimeSpan.FromMinutes(5)),
    security)
{
    MarketData =
    {
        BuildMode = MarketDataBuildModes.LoadAndBuild,
        BuildFrom = DataType.Ticks,
        IsCalcVolumeProfile = true
    }
};
_connector.Subscribe(volumeProfileSubscription);
```

Volume Candles

```
// Volume candles (each candle contains 1000 contracts in volume)
var volumeCandleSubscription = new Subscription(
    DataType.Volume(1000m), // Specify candle type and volume
    security)
{
    MarketData =
    {
        BuildMode = MarketDataBuildModes.Build,
        BuildFrom = DataType.Ticks
    }
};
_connector.Subscribe(volumeCandleSubscription);
```

Tick Count Candles

```
// Tick count candles (each candle contains 1000 trades)
var tickCandleSubscription = new Subscription(
    DataType.Tick(1000), // Specify candle type and number of trades
    security)
{
    MarketData =
    {
        BuildMode = MarketDataBuildModes.Build,
        BuildFrom = DataType.Ticks
    }
};
_connector.Subscribe(tickCandleSubscription);
```

Price Range Candles

```
// Price range candles with a range of 0.1 units
var rangeCandleSubscription = new Subscription(
    DataType.Range(0.1m), // Specify candle type and price range
    security)
{
    MarketData =
    {
        BuildMode = MarketDataBuildModes.Build,
        BuildFrom = DataType.Ticks
    }
};
_connector.Subscribe(rangeCandleSubscription);
```

Renko Candles

```
// Renko candles with a step of 0.1
var renkoCandleSubscription = new Subscription(
    DataType.Renko(0.1m), // Specify candle type and block size
    security)
{
    MarketData =
    {
        BuildMode = MarketDataBuildModes.Build,
        BuildFrom = DataType.Ticks
    }
};
_connector.Subscribe(renkoCandleSubscription);
```

Point and Figure Candles (P&F)

```
// Point and Figure candles
var pnfCandleSubscription = new Subscription(
    DataType.PnF(new PnfArg { BoxSize = 0.1m, ReversalAmount = 1 }), // Specify P&F parameters
    security)
{
    MarketData =
    {
        BuildMode = MarketDataBuildModes.Build,
        BuildFrom = DataType.Ticks
    }
};
_connector.Subscribe(pnfCandleSubscription);
```

Next Steps

[Chart](#)

[Custom Candle Type](#)

Chart

For graphical display of candles, you can use the special [Chart](#) component (see [Components for building charts](#)), which renders candles as follows:

[image: sample candleschart]

Basic Approach to Displaying Candles

There are two approaches to displaying candles on a chart. The first approach is manual drawing of candles when receiving data:

```
// CandlesChart - StockSharp.Xaml.Chart
private ChartArea _areaComb;
private ChartCandleElement _candleElement;

// Chart initialization
private void InitializeChart()
{
    // Create chart area
    _areaComb = new ChartArea();
    _chart.Areas.Add(_areaComb);

    // Create chart element representing candles
    _candleElement = new ChartCandleElement() { FullTitle = "Candles" };
    _areaComb.Elements.Add(_candleElement);

    // Subscribe to candle reception event
    _connector.CandleReceived += OnCandleReceived;
}

// Create subscription to 5-minute candles
private void SubscribeToCandles()
{
    var subscription = new Subscription(
        DataType.TimeFrame(TimeSpan.FromMinutes(5)),
        _security)
    {
        MarketData =
        {
            // Request historical data for 5 days
            From = DateTime.Today.Subtract(TimeSpan.FromDays(5)),
            To = DateTime.Now
        }
    };

    // Start subscription
    _connector.Subscribe(subscription);
}

// Handler for candle reception event
private void OnCandleReceived(Subscription subscription, ICandleMessage candle)
{
    // Check if the candle is completed
    if (candle.State == CandleStates.Finished)
    {
        // Create data for drawing
        var chartData = new ChartDrawData();
        chartData.Group(candle.OpenTime).Add(_candleElement, candle);

        // Draw on chart in UI thread
        thisGuiAsync(() => _chart.Draw(chartData));
    }
}
```

```
}  
}
```

Automatic Binding of Subscription to Chart Element

The second approach is to use automatic binding of subscription to chart element. This allows automatically displaying the received data:

```
// Chart initialization with automatic binding  
private void InitializeChartWithAutoBinding()  
{  
    // Create chart area  
    var area = new ChartArea();  
    _chart.Areas.Add(area);  
  
    // Create element for displaying candles  
    var candleElement = new ChartCandleElement();  
  
    // Create subscription to candles  
    var subscription = new Subscription(  
        DataType.TimeFrame(TimeSpan.FromMinutes(5)),  
        _security)  
    {  
        MarketData =  
        {  
            From = DateTime.Today.Subtract(TimeSpan.FromDays(5)),  
            To = DateTime.Now  
        }  
    };  
  
    // Bind element to subscription  
    _chart.AddElement(area, candleElement, subscription);  
  
    // Start subscription  
    _connector.Subscribe(subscription);  
}
```

Working with Indicators

To display indicators on the chart together with candles, elements of type [ChartIndicatorElement](#) are used:

```
// Adding indicator to chart  
private void AddIndicatorToChart()  
{  
    // Create element for indicator  
    var smaElement = new ChartIndicatorElement  
    {  
        Title = "SMA (14)",  
        Color = Colors.Red  
    };  
  
    // Add element to the same area as candles  
    _areaComb.Elements.Add(smaElement);  
  
    // Create indicator  
    var sma = new SimpleMovingAverage { Length = 14 };  
  
    // Subscribe to candle reception event for indicator calculation  
    _connector.CandleReceived += (subscription, candle) =>  
    {  
        // Calculate indicator value  
        var indicatorValue = sma.Process(candle);  
  
        // Draw value on chart  
    }  
}
```

```

var chartData = new ChartDrawData();
chartData.Group(candle.OpenTime).Add(smaElement, indicatorValue);

this.GuiAsync(() => _chart.Draw(chartData));
};
}

```

Displaying Multiple Indicators on Different Areas

Indicators can be placed on separate chart areas:

```

// Adding indicators to different areas
private void AddIndicatorsToSeparateAreas()
{
    // Main area for candles
    var candleArea = new ChartArea();
    _chart.Areas.Add(candleArea);

    // Element for candles
    var candleElement = new ChartCandleElement();
    candleArea.Elements.Add(candleElement);

    // Element for SMA on the same area
    var smaElement = new ChartIndicatorElement { Title = "SMA (14)" };
    candleArea.Elements.Add(smaElement);

    // Separate area for RSI
    var rsiArea = new ChartArea();
    _chart.Areas.Add(rsiArea);

    // Element for RSI
    var rsiElement = new ChartIndicatorElement { Title = "RSI (14)" };
    rsiArea.Elements.Add(rsiElement);

    // Create indicators
    var sma = new SimpleMovingAverage { Length = 14 };
    var rsi = new RelativeStrengthIndex { Length = 14 };

    // Subscription to candles
    var subscription = new Subscription(
        DataType.TimeFrame(TimeSpan.FromMinutes(5)),
        _security);

    // Bind candle element to subscription
    _chart.AddElement(candleArea, candleElement, subscription);

    // Start subscription and process indicators
    _connector.Subscribe(subscription);

    _connector.CandleReceived += (sub, candle) =>
    {
        if (sub != subscription || candle.State != CandleStates.Finished)
            return;

        // Calculate indicator values
        var smaValue = sma.Process(candle);
        var rsiValue = rsi.Process(candle);

        // Draw values on chart
        var chartData = new ChartDrawData();
        chartData
            .Group(candle.OpenTime)
            .Add(smaElement, smaValue)
            .Add(rsiElement, rsiValue);

        this.GuiAsync(() => _chart.Draw(chartData));
    };
}

```



```
}
```

Displaying Orders and Trades on Chart

Special elements are used to display orders and trades on the chart:

```
// Adding elements for displaying orders and trades
private void AddOrdersAndTradesToChart()
{
    // Create elements for displaying orders and trades
    var orderElement = new ChartOrderElement();
    var tradeElement = new ChartTradeElement();

    // Add elements to chart area
    _areaComb.Elements.Add(orderElement);
    _areaComb.Elements.Add(tradeElement);

    // Subscribe to order and trade reception events
    _connector.OrderReceived += (subscription, order) =>
    {
        if (order.Security != _security)
            return;

        // Draw order on chart
        var chartData = new ChartDrawData();
        chartData.Group(order.Time).Add(orderElement, order);

        this.GuiAsync(() => _chart.Draw(chartData));
    };

    _connector.OwnTradeReceived += (subscription, trade) =>
    {
        if (trade.Order.Security != _security)
            return;

        // Draw trade on chart
        var chartData = new ChartDrawData();
        chartData.Group(trade.Time).Add(tradeElement, trade);

        this.GuiAsync(() => _chart.Draw(chartData));
    };
}
```

Configuring Chart Appearance

Various aspects of chart appearance can be configured:

```
// Configuring chart appearance
private void ConfigureChartAppearance()
{
    // Configuring chart area
    _areaComb.Height = 300;
    _areaComb.BackgroundMajorGridColor = Colors.Gray;
    _areaComb.BackgroundMinorGridColor = Colors.LightGray;

    // Configuring candle element
    _candleElement.DrawStyle = ChartCandleDrawStyles.CandleStick;
    _candleElement.UpBrush = Brushes.Green;
    _candleElement.DownBrush = Brushes.Red;
    _candleElement.StrokeThickness = 1;

    // Configuring entire chart
    _chart.IsAutoRange = true; // Automatic scaling
    _chart.IsManualVerticalValues = false; // Automatic calculation of vertical values
    _chart.BidEnabled = false; // Disable display of best bid price
}
```

```

_chart.AskEnabled = false;           // Disable display of best ask price
}

```

Chart Zooming and Scrolling

Managing chart zoom and scroll:

```

// Configuring zooming and scrolling
private void ConfigureChartZoomAndScroll()
{
    // Setting initial and final dates for display
    _chart.SetXRange(DateTime.Today.AddDays(-10), DateTime.Today);

    // Setting Y-axis range
    _chart.SetYRange(100, 150);

    // Buttons for zoom control
    zoomInButton.Click += (s, e) => _chart.ZoomIn();
    zoomOutButton.Click += (s, e) => _chart.ZoomOut();

    // Buttons for scrolling
    scrollLeftButton.Click += (s, e) => _chart.ScrollLeft();
    scrollRightButton.Click += (s, e) => _chart.ScrollRight();

    // Reset zoom to automatic
    resetZoomButton.Click += (s, e) => _chart.IsAutoRange = true;
}

```

Exporting Chart to Image

To save chart to a file:

```

// Exporting chart to image
private void ExportChartToImage()
{
    // Create object for saving image
    var saveFileDialog = new SaveFileDialog
    {
        Filter = "PNG Image|*.png|JPEG Image|*.jpg|BMP Image|*.bmp",
        Title = "Save Chart Image"
    };

    if (saveFileDialog.ShowDialog() == true)
    {
        // Create image from chart
        var rtb = new RenderTargetBitmap(
            (int)_chart.ActualWidth,
            (int)_chart.ActualHeight,
            96, 96,
            PixelFormats.Pbgra32);

        rtb.Render(_chart);

        // Save image in selected format
        BitmapEncoder encoder;

        switch (Path.GetExtension(saveFileDialog.FileName).ToLower())
        {
            case ".jpg":
                encoder = new JpegBitmapEncoder();
                break;
            case ".bmp":
                encoder = new BmpBitmapEncoder();
                break;
            default:

```

```

        encoder = new PngBitmapEncoder();
        break;
    }

    encoder.Frames.Add(BitmapFrame.Create(rtb));

    using (var fileStream = new FileStream(saveFileDialog.FileName, FileMode.Create))
    {
        encoder.Save(fileStream);
    }
}
}

```

Clearing the Chart

To clear data on the chart:

```

// Clearing chart or its elements
private void ClearChart()
{
    // Clear entire chart
    _chart.Reset();

    // Clear specific area
    _areaComb.Reset();

    // Clear specific element
    _candleElement.Reset();
}

```

An example of displaying candles on a chart is provided in the [Candles](#) section.

See also

[Components for building charts](#)

Custom type of candle

Custom Type of Candle

[S#](#) allows you to extend candle building capabilities by providing the ability to work with custom candle types. This is useful in cases where you need to work with candles that are not currently supported by [S#](#). Below is the process of creating your own candle type using the example of Delta-candles (candles formed based on the difference between buy and sell volumes).

Implementing Delta-candles

1. First, you need to create your own candle message type. The type must inherit from the [CandleMessage](#) class:

```
/// <summary>
/// Candle formed based on the delta of buy and sell volumes.
/// </summary>
public class DeltaCandleMessage : CandleMessage
{
    // We get the message type identifier from the helper
    // to use the same value in RegisterCandleType

    /// <summary>
    /// Initialize a new instance of <see cref="DeltaCandleMessage"/>.
    /// </summary>
    public DeltaCandleMessage()
        : base(DeltaCandleHelper.DeltaCandleType)
    {
    }

    /// <summary>
    /// Delta threshold value for candle formation.
    /// </summary>
    public decimal DeltaThreshold { get; set; }

    /// <summary>
    /// Current delta value.
    /// </summary>
    public decimal CurrentDelta { get; set; }

    /// <summary>
    /// Create a copy of <see cref="DeltaCandleMessage"/>.
    /// </summary>
    /// <returns>Copy.</returns>
    public override Message Clone()
    {
        return CopyTo(new DeltaCandleMessage
        {
            DeltaThreshold = DeltaThreshold,
            CurrentDelta = CurrentDelta
        });
    }

    /// <summary>
    /// Candle parameter.
    /// </summary>
    public override object Arg
    {
        get => DeltaThreshold;
        set => DeltaThreshold = (decimal)value;
    }

    /// <summary>
    /// Type of candle argument.

```

```

    /// </summary>
    public override Type ArgType => typeof(decimal);
}

```

2. Then you need to create your own data type in the [DataType](#) class:

```

public static class DeltaCandleHelper
{
    /// <summary>
    /// Define a unique MessageType for delta-candles.
    /// </summary>
    public const MessageType DeltaCandleType = (MessageType)10001;

    /// <summary>
    /// <see cref="DeltaCandleMessage"/> data type.
    /// </summary>
    public static readonly DataType CandleDelta =
        DataType.Create(typeof(DeltaCandleMessage)).Immutable();

    /// <summary>
    /// Create a data type for delta-candles.
    /// </summary>
    /// <param name="threshold">Delta threshold value.</param>
    /// <returns>Data type.</returns>
    public static DataType Delta(this decimal threshold)
    {
        return DataType.Create(typeof(DeltaCandleMessage), threshold);
    }

    /// <summary>
    /// Register the delta-candle type in the system.
    /// </summary>
    public static void RegisterDeltaCandleType()
    {
        // Register new candle type in StockSharp
        Extensions.RegisterCandleType<decimal>(
            typeof(DeltaCandleMessage), // Candle message type
            DeltaCandleType,             // Message type
            "delta",                      // File name for storage
            str => str.To<decimal>(),     // Converter from string to parameter
            arg => arg.ToString(),        // Converter from parameter to string
            a => a > 0,                   // Parameter validator
            false                         // Whether such candles can be obtained from the
source (not only built)
        );
    }
}

```

3. Next, you need to create a candle builder for the new type. To do this, create an implementation of [CandleBuilder<TCandleMessage>](#):

```

/// <summary>
/// Candle builder for <see cref="DeltaCandleMessage"/> type.
/// </summary>
public class DeltaCandleBuilder : CandleBuilder<DeltaCandleMessage>
{
    /// <summary>
    /// Initializes a new instance of <see cref="DeltaCandleBuilder"/>.
    /// </summary>
    /// <param name="exchangeInfoProvider">Exchange information provider.</param>
    public DeltaCandleBuilder(IEExchangeInfoProvider exchangeInfoProvider)
        : base(exchangeInfoProvider)
    {
    }

    /// <inheritdoc />
    protected override DeltaCandleMessage CreateCandle(ICandleBuilderSubscription subscription,

```

```

ICandleBuilderValueTransform transform)
{
    var time = transform.Time;

    return FirstInitCandle(subscription, new DeltaCandleMessage
    {
        DeltaThreshold = subscription.Message.GetArg<decimal>(),
        OpenTime = time,
        CloseTime = time,
        HighTime = time,
        LowTime = time,
        CurrentDelta = 0
    }, transform);
}

/// <inheritdoc />
protected override bool IsCandleFinishedBeforeChange(ICandleBuilderSubscription subscription,
DeltaCandleMessage candle, ICandleBuilderValueTransform transform)
{
    // The candle closes when the absolute value of delta exceeds the threshold
    return Math.Abs(candle.CurrentDelta) >= candle.DeltaThreshold;
}

/// <inheritdoc />
protected override void UpdateCandle(ICandleBuilderSubscription subscription,
DeltaCandleMessage candle, ICandleBuilderValueTransform transform)
{
    base.UpdateCandle(subscription, candle, transform);

    // Update delta based on the trade side
    if (transform.Side == Sides.Buy)
        candle.CurrentDelta += transform.Volume ?? 0;
    else if (transform.Side == Sides.Sell)
        candle.CurrentDelta -= transform.Volume ?? 0;
}
}

```

4. Then you need to register the candle builder in [CandleBuilderProvider](#):

```

private Connector _connector;
...
// Register delta-candle type in the system
DeltaCandleHelper.RegisterDeltaCandleType();

// Register the delta-candle builder
_connector.Adapter.CandleBuilderProvider.Register(new
DeltaCandleBuilder(_connector.ExchangeInfoProvider));

```

5. Create a subscription for the DeltaCandleMessage type candles and request data from it:

```

// Delta threshold value
decimal deltaThreshold = 1000m;

// Create a subscription for delta-candles
var subscription = new Subscription(
    // Use our extension method to create a data type
    deltaThreshold.Delta(),
    security)
{
    MarketData =
    {
        // Specify that candles will be built from ticks
        BuildMode = MarketDataBuildModes.Build,
        BuildFrom = DataType.Ticks
    }
};

```

```

// Subscribe to the candle received event
_connector.CandleReceived += (sub, candle) =>
{
    if (sub != subscription)
        return;

    var deltaCandle = (DeltaCandleMessage)candle;

    // Process delta-candle
    Console.WriteLine($"Delta-candle {candle.OpenTime}: O:{candle.OpenPrice} H:{candle.HighPrice}
" +
        $"L:{candle.LowPrice} C:{candle.ClosePrice} V:{candle.TotalVolume} Delta:
{deltaCandle.CurrentDelta}");
};

// Subscribe to the online mode transition
_connector.SubscriptionOnline += sub =>
{
    if (sub == subscription)
        Console.WriteLine("Delta-candle subscription has transitioned to online mode");
};

// Start the subscription
_connector.Subscribe(subscription);

```

Using Delta-candles in Trading Strategies

Example of a simple strategy using delta-candles:

```

public class DeltaCandleStrategy : Strategy
{
    private readonly StrategyParam<decimal> _deltaThreshold;
    private readonly StrategyParam<decimal> _volume;
    private readonly StrategyParam<decimal> _signalDelta;

    private IChart _chart;
    private IChartCandleElement _chartCandleElement;
    private IChartIndicatorElement _deltaIndicatorElement;

    public decimal DeltaThreshold
    {
        get => _deltaThreshold.Value;
        set => _deltaThreshold.Value = value;
    }

    public decimal Volume
    {
        get => _volume.Value;
        set => _volume.Value = value;
    }

    public decimal SignalDelta
    {
        get => _signalDelta.Value;
        set => _signalDelta.Value = value;
    }

    public DeltaCandleStrategy()
    {
        // Strategy parameters
        _deltaThreshold = Param(nameof(DeltaThreshold), 1000m)
            .SetDisplay("Delta Threshold Value", "Volume delta value for candle formation", "Main Settings")
            .SetGreaterThanZero()
            .SetCanOptimize(true)
            .SetOptimize(500m, 2000m, 100m);

        _volume = Param(nameof(Volume), 1m)

```

```

        .SetDisplay("Order Volume", "Volume for trading operations", "Main Settings")
        .SetGreaterThanZero();

        _signalDelta = Param(nameof(SignalDelta), 500m)
        .SetDisplay("Minimum Delta for Signal", "Minimum delta value for signal generation", "Main
Settings")
        .SetGreaterThanZero()
        .SetCanOptimize(true);

        Name = "DeltaCandleStrategy";
    }

protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    // Chart initialization, if available
    _chart = GetChart();
    if (_chart != null)
    {
        var area = _chart.AddArea();
        _chartCandleElement = area.AddCandles();
        _deltaIndicatorElement = area.AddIndicator();
        _deltaIndicatorElement.DrawStyle = DrawStyles.Histogram;
        _deltaIndicatorElement.Color = System.Drawing.Color.Purple;
    }

    // Create a subscription for delta-candles
    var subscription = new Subscription(DeltaThreshold.Delta(), Security)
    {
        MarketData =
        {
            BuildMode = MarketDataBuildModes.Build,
            BuildFrom = DataType.Ticks
        }
    };

    // Create a rule for processing delta-candles
    this
        .WhenCandleReceived(subscription)
        .Do(ProcessDeltaCandle)
        .Apply(this);

    // Start the subscription
    Subscribe(subscription);
}

private void ProcessDeltaCandle(ICandleMessage candle)
{
    // Draw on the chart, if available
    if (_chart != null)
    {
        var deltaCandle = (DeltaCandleMessage)candle;

        var data = _chart.CreateData();
        data.Group(candle.OpenTime)
            .Add(_chartCandleElement, candle)
            .Add(_deltaIndicatorElement, deltaCandle.CurrentDelta);

        _chart.Draw(data);
    }

    // Process only finished candles
    if (candle.State != CandleStates.Finished)
        return;

    var deltaCandle = (DeltaCandleMessage)candle;

    // Check if delta is sufficient for a signal

```



```

    if (Math.Abs(deltaCandle.CurrentDelta) < SignalDelta)
    {
        this.AddInfoLog($"Delta {deltaCandle.CurrentDelta} is less than the threshold value
{SignalDelta}. No signal is generated.");
        return;
    }

    // Operation direction depends on the delta sign
    var direction = deltaCandle.CurrentDelta > 0 ? Sides.Buy : Sides.Sell;

    this.AddInfoLog($"Delta-candle completed. Delta: {deltaCandle.CurrentDelta}. Direction:
{direction}");

    // Use the candle's close price to determine the price
    var price = deltaCandle.ClosePrice;
    var volume = Volume;

    // If we already have a position in the opposite direction,
    // increase the volume to close the existing position
    if ((Position < 0 && direction == Sides.Buy) ||
        (Position > 0 && direction == Sides.Sell))
    {
        volume = Math.Max(volume, Math.Abs(Position) + volume);
    }

    // Register an order
    RegisterOrder(this.CreateOrder(direction, price, volume));
}
}

```

Important Points When Creating Custom Candle Types

1. **Uniqueness of MessageTypes** — make sure that the MessageTypes identifier you choose does not conflict with existing types in StockSharp. It is recommended to use values greater than 10000 for custom types.
2. **Registration of the Candle Type** — registration through Extensions.RegisterCandleType is necessary for proper integration with StockSharp graphical controls and data storage. Without registration, the candle type will only work in code but will not be available in the user interface.
3. **Candle Parameter** — implement the ArgType property that returns the type of the candle argument. This is used for correct display of parameters in the graphical interface.
4. **File System** — the fileName parameter in the RegisterCandleType method is used to save candles in the file system when you use StockSharp data storage.
5. **Parameter Validation** — the parameter validation method is used in StockSharp to check the correctness of values before creating a subscription.

Thus, we have created a completely custom candle type that properly integrates with the entire StockSharp ecosystem (including the user interface and data storage) and can be used to build trading strategies based on volume delta analysis.

Merging candles, history + real-time

Gluing Candles: History + Real-Time

To combine historical candles with real-time data, you need to initialize the appropriate storages: storage for trading objects [CsvEntityRegistry](#), storage for market data [StorageRegistry](#), and snapshot storage registry [SnapshotRegistry](#).

Let's look at an example from the Samples/Candles/CombineHistoryRealtime project:

Setting Up Storages and Connector

```
public partial class MainWindow
{
    private readonly Connector _connector;
    private const string _connectorFile = "ConnectorFile.json";

    // Path to historical data
    private readonly string _pathHistory = Paths.HistoryDataPath;

    private Subscription _subscription;
    private ChartCandleElement _candleElement;

    public MainWindow()
    {
        InitializeComponent();

        // Initialize storages
        var entityRegistry = new CsvEntityRegistry(_pathHistory);
        var storageRegistry = new StorageRegistry
        {
            DefaultDrive = new LocalMarketDataDrive(_pathHistory)
        };

        // Create connector with configured storages
        _connector = new Connector(
            entityRegistry.Securities,
            entityRegistry.PositionStorage,
            new InMemoryExchangeInfoProvider(),
            storageRegistry,
            new SnapshotRegistry("SnapshotRegistry"));

        // Register message adapter provider
        ConfigManager.RegisterService<IMessageAdapterProvider>(
            new InMemoryMessageAdapterProvider(_connector.Adapter.InnerAdapters));

        // Load connector settings if file exists
        if (File.Exists(_connectorFile))
        {
            _connector.Load(_connectorFile.Deserialize<SettingsStorage>());
        }

        // Set default candle data type (5-minute)
        CandleDataTypeEdit.DataType = TimeSpan.FromMinutes(5).TimeFrame();
    }
}
```

Connection Setup

```
// Method for configuring connection parameters
private void Setting_Click(object sender, RoutedEventArgs e)
{
}
```

```

// Call connector configuration window
if (_connector.Configure(this))
{
    // Save settings to file
    _connector.Save().Serialize(_connectorFile);
}
}

// Method for connecting to trading system
private void Connect_Click(object sender, RoutedEventArgs e)
{
    // Set connector as data source for instrument selection
    SecurityPicker.SecurityProvider = _connector;

    // Subscribe to candle reception event
    _connector.CandleReceived += Connector_CandleReceived;

    // Connect
    _connector.Connect();
}

```

Processing Candles and Displaying on Chart

```

// Handler for candle reception event
private void Connector_CandleReceived(Subscription subscription, ICandleMessage candle)
{
    // Draw candle on chart
    Chart.Draw(_candleElement, candle);
}

```

Creating Candle Subscription

```

// Method called when an instrument is selected
private void SecurityPicker_SecuritySelected(Security security)
{
    // Check if instrument is selected
    if (security == null)
        return;

    // Unsubscribe from previous subscription if it exists
    if (_subscription != null)
        _connector.UnSubscribe(_subscription);

    // Create new subscription for selected instrument
    _subscription = new(CandleDataTypeEdit.DataType, security)
    {
        MarketData =
        {
            // Request historical data for last 720 days
            From = DateTime.Today.AddDays(-720),

            // Mode: load historical data and build in real-time
            BuildMode = MarketDataBuildModes.LoadAndBuild,
        }
    };

    // Configure chart
    Chart.ClearAreas();

    // Create chart area and element for displaying candles
    var area = new ChartArea();
    _candleElement = new ChartCandleElement();

    // Add area and element to chart
    Chart.AddArea(area);
}

```

```

// Link chart element with subscription for automatic drawing
Chart.AddElement(area, _candleElement, _subscription);

// Start subscription
_connector.Subscribe(_subscription);
}

```

Complete MainWindow Class Example

```

namespace StockSharp.Samples.Candles.CombineHistoryRealtime;

using System;
using System.Windows;
using System.IO;

using Ecng.Common;
using Ecng.Serialization;
using Ecng.Configuration;

using StockSharp.Configuration;
using StockSharp.Algo;
using StockSharp.Algo.Storages;
using StockSharp.Algo.Storages.Csv;
using StockSharp.BusinessEntities;
using StockSharp.Xaml;
using StockSharp.Messages;
using StockSharp.Xaml.Charting;
using StockSharp.Charting;

/// <summary>
/// Interaction logic for MainWindow.xaml
/// </summary>
public partial class MainWindow
{
    private readonly Connector _connector;
    private const string _connectorFile = "ConnectorFile.json";

    private readonly string _pathHistory = Paths.HistoryDataPath;

    private Subscription _subscription;
    private ChartCandleElement _candleElement;

    public MainWindow()
    {
        InitializeComponent();
        var entityRegistry = new CsvEntityRegistry(_pathHistory);
        var storageRegistry = new StorageRegistry
        {
            DefaultDrive = new LocalMarketDataDrive(_pathHistory)
        };
        _connector = new Connector(
            entityRegistry.Securities,
            entityRegistry.PositionStorage,
            new InMemoryExchangeInfoProvider(),
            storageRegistry,
            new SnapshotRegistry("SnapshotRegistry"));

        // registering all connectors
        ConfigManager.RegisterService<IMessageAdapterProvider>(
            new InMemoryMessageAdapterProvider(_connector.Adapter.InnerAdapters));

        if (File.Exists(_connectorFile))
        {
            _connector.Load(_connectorFile.Deserialize<SettingsStorage>());
        }
    }
}

```

```

    CandleDataTypeEdit.DataType = TimeSpan.FromMinutes(5).TimeFrame();
}

private void Setting_Click(object sender, RoutedEventArgs e)
{
    if (_connector.Configure(this))
    {
        _connector.Save().Serialize(_connectorFile);
    }
}

private void Connect_Click(object sender, RoutedEventArgs e)
{
    SecurityPicker.SecurityProvider = _connector;
    _connector.CandleReceived += Connector_CandleReceived;
    _connector.Connect();
}

private void Connector_CandleReceived(Subscription subscription, ICandleMessage candle)
{
    Chart.Draw(_candleElement, candle);
}

private void SecurityPicker_SecuritySelected(Security security)
{
    if (security == null) return;
    if (_subscription != null) _connector.UnSubscribe(_subscription);

    _subscription = new(CandleDataTypeEdit.DataType, security)
    {
        MarketData =
        {
            From = DateTime.Today.AddDays(-720),
            BuildMode = MarketDataBuildModes.LoadAndBuild,
        }
    };

    //-----Chart-----
    Chart.ClearAreas();

    var area = new ChartArea();
    _candleElement = new ChartCandleElement();

    Chart.AddArea(area);
    Chart.AddElement(area, _candleElement, _subscription);

    _connector.Subscribe(_subscription);
}
}

```

Example Features

1. Creating Storages:

- [CsvEntityRegistry](#) is used for storing entities
- [StorageRegistry](#) is configured with path to storage
- [SnapshotRegistry](#) is created for working with snapshots

2. Creating Subscription:

- The class [Subscription](#) is used
- The From parameter in MarketData specifies the initial date for loading history
- BuildMode = MarketDataBuildModes.LoadAndBuild is set for automatic combination of history and real-time

3. Chart Display:

- Chart.AddElement method is used to link chart element with subscription
- The chart is automatically updated when new candles are received

4. Event Handling:

- Subscription to CandleReceived event for processing received candles
- Unsubscribing from previous subscription when selected instrument changes

Extended Capabilities

This example can be extended by adding the following functions:

Tracking Transition to Real-Time Mode

```
// Subscription to the event of transition to real-time mode
_connector.SubscriptionOnline += OnSubscriptionOnline;

// Event handler
private void OnSubscriptionOnline(Subscription subscription)
{
    if (subscription == _subscription)
    {
        this.GuiAsync(() => StatusLabel.Content = "Online mode");
    }
}
```

Configuring History Loading Period

```
// Setting history loading period
private void SetHistoryPeriod(int days)
{
    if (_subscription != null)
    {
        _connector.UnSubscribe(_subscription);

        _subscription.MarketData.From = DateTime.Today.AddDays(-days);

        _connector.Subscribe(_subscription);
    }
}
```

Saving Received Data

```
// Method for saving received data
private void SaveReceivedData()
{
    if (_connector.StorageAdapter != null)
    {
        // Force save cached data to disk
        _connector.StorageAdapter.Flush();
    }
}
```

Additional Candle Processing

```
// Extended candle processing with information output
private void ExtendedCandleProcessing(Subscription subscription, ICandleMessage candle)
{
    // Draw candle on chart
    Chart.Draw(_candleElement, candle);
}
```

```
// Output information about candle to logs
this.GuiAsync(() =>
{
    var status = subscription.State == SubscriptionStates.Online ? "Real-time" : "History";
    LogControl.LogMessage($"{status}: {candle.OpenTime} - O:{candle.OpenPrice} H:{candle.HighPrice} L:
{candle.LowPrice} C:{candle.ClosePrice}");
});
}
```

Candle creation by the instruments basket

Candle Creation by the Instruments Basket

To create candles for [ContinuousSecurity](#), [WeightedIndexSecurity](#), or [ExpressionIndexSecurity](#), the same subscription mechanism is used as for regular [Security](#) instruments.

Below is an example of creating 1-minute candles for the GZM5 - LKM5 spread:

```
private Connector _connector;
private Security _instr1;
private Security _instr2;
private WeightedIndexSecurity _indexInstr;
private Subscription _indexSubscription;
private const string _secCode1 = "GZM5";
private const string _secCode2 = "LKM5";
readonly TimeSpan _timeFrame = TimeSpan.FromMinutes(1);
private ChartArea _area;
private ChartCandleElement _candleElement;

// Connection setup and connector configuration
private void ConfigureConnector()
{
    if (_connector.Configure(this))
    {
        _connector.Save().Serialize(_connectorFile);
    }
}

// Chart setup
private void SetupChart()
{
    _area = new ChartArea();
    _chart.Areas.Add(_area);
    _candleElement = new ChartCandleElement();
    _area.Elements.Add(_candleElement);

    // Subscribe to candle reception event
    _connector.CandleReceived += OnCandleReceived;
}

// Service registration
private void RegisterServices()
{
    ConfigManager.RegisterService<ISecurityProvider>(_connector);
    ConfigManager.RegisterService<ICompilerService>(new RoslynCompilerService());
}

// Creating index instrument and subscribing to candles
private void CreateIndexAndSubscribe()
{
    // Create index instrument (spread)
    _indexInstr = new WeightedIndexSecurity()
    {
        Board = ExchangeBoard.Nyse,
        Id = "IndexInstr"
    };

    // Add instruments with weights (1 and -1 for spread)
    _indexInstr.Weights.Add(_instr1, 1);
    _indexInstr.Weights.Add(_instr2, -1);

    // Create subscription to index instrument candles
    _indexSubscription = new Subscription(
        DataType.TimeFrame(_timeFrame), // 1-minute candles
```



```

_indexInstr) // Our index instrument
{
    MarketData =
    {
        // Configure subscription to build candles from ticks
        BuildMode = MarketDataBuildModes.Build,
        BuildFrom = DataType.Ticks,

        // Request historical data for 30 days
        From = DateTime.Today.Subtract(TimeSpan.FromDays(30)),
        To = DateTime.Now
    }
};

// Add element to chart and bind it to subscription
_chart.AddElement(_area, _candleElement, _indexSubscription);

// Start subscription
_connector.Subscribe(_indexSubscription);
}

// Handler for candle reception event
private void OnCandleReceived(Subscription subscription, ICandleMessage candle)
{
    // Check if the candle belongs to our subscription
    if (subscription != _indexSubscription)
        return;

    // If needed, limit processing to only completed candles
    if (candle.State != CandleStates.Finished)
        return;

    // Draw the candle on the chart
    var chartData = new ChartDrawData();
    chartData.Group(candle.OpenTime).Add(_candleElement, candle);

    this.GuiAsync(() => _chart.Draw(chartData));
}

// Unsubscribe when closing the application
private void Unsubscribe()
{
    if (_indexSubscription != null)
    {
        _connector.CandleReceived -= OnCandleReceived;
        _connector.UnSubscribe(_indexSubscription);
        _indexSubscription = null;
    }
}

```

Other Use Cases for Index Subscriptions

Creating a Subscription to Index Candles from Component Candles

```

// Create subscription to build index candles from component candles
var indexFromCandlesSubscription = new Subscription(
    DataType.TimeFrame(TimeSpan.FromMinutes(5)),
    _indexInstr)
{
    MarketData =
    {
        // Configure subscription to build from component candles
        BuildMode = MarketDataBuildModes.Build,
        BuildFrom = DataType.TimeFrame(TimeSpan.FromMinutes(5)),
        From = DateTime.Today.Subtract(TimeSpan.FromDays(30)),
        To = DateTime.Now
    }
}

```

```

    }
};

// Start subscription
_connector.Subscribe(indexFromCandlesSubscription);

```

Creating a Subscription to Index Candles from Order Books

```

// Create subscription to build index candles from order books
var indexFromDepthSubscription = new Subscription(
    DataType.TimeFrame(TimeSpan.FromMinutes(1)),
    _indexInstr)
{
    MarketData =
    {
        // Configure subscription to build from order books
        BuildMode = MarketDataBuildModes.Build,
        BuildFrom = DataType.MarketDepth,
        BuildField = Level1Fields.SpreadMiddle, // Use middle of spread
        From = DateTime.Today.Subtract(TimeSpan.FromDays(7)),
        To = DateTime.Now
    }
};

// Start subscription
_connector.Subscribe(indexFromDepthSubscription);

```

Working with Volatility Index

```

// Create volatility index based on expression
var volatilityIndex = new ExpressionIndexSecurity
{
    Board = ExchangeBoard.Nyse,
    Id = "VOLX",
    Expression = "StdDev({0}, 20) / SMA({0}, 20) * 100", // Formula for calculating volatility
};

// Add main instrument to index
volatilityIndex.InnerSecurityIds.Add(_instr1.ToSecurityId());

// Create subscription to volatility index candles
var volatilitySubscription = new Subscription(
    DataType.TimeFrame(TimeSpan.FromMinutes(5)),
    volatilityIndex)
{
    MarketData =
    {
        BuildMode = MarketDataBuildModes.Build,
        BuildFrom = DataType.Ticks,
        From = DateTime.Today.Subtract(TimeSpan.FromDays(30)),
        To = DateTime.Now
    }
};

// Start subscription
_connector.Subscribe(volatilitySubscription);

```

See also

[Continuous Futures](#)

[Index](#)

Compression of Tick Data and Spreads into Candles

Compression of Tick Data and Spreads into Candles

Introduction

The API provides powerful tools for compressing tick data and spreads (best bid/ask prices) into candles. This functionality is especially useful for analyzing historical data or building custom indicators.

The main extension methods for data compression are located in the `CandleHelper` class. The full source code for this class is [available on GitHub](#).

It is recommended to review this file for a complete understanding of all available methods and their parameters.

Compression Methods

Tick Data Compression into Candles

```
// Example usage of ToCandles for ticks
var tickStorage = storageRegistry.GetTickMessageStorage(securityId, Drive, StorageFormat);
var trades = tickStorage.LoadAsync(from, to);
var candles = trades.ToCandles(mdMsg, candleBuilderProvider: candleBuilderProvider);

// This code loads tick data from storage and converts it into candles.
// mdMsg - the message with parameters of the created candles (type, time frame, etc.).
// candleBuilderProvider - the provider that supplies a specific candle builder implementation.
```

Spread Data Compression into Candles

```
// Example usage of ToCandles for spread data
var depthStorage = storageRegistry.GetQuoteMessageStorage(securityId, Drive, StorageFormat);
var depths = depthStorage.LoadAsync(from, to);
var candles = depths.ToCandles(mdMsg, Level1Fields.SpreadMiddle, candleBuilderProvider:
candleBuilderProvider);

// Here we load spread data and convert it into candles.
// Level1Fields.SpreadMiddle indicates using the spread middle price for building candles.
// You can also use Level1Fields.BestBid or Level1Fields.BestAsk for the best bid or ask prices,
respectively.
```

Compression Parameters

When compressing data, the following parameters can be specified:

- `series`: The candle series defining the type and parameters of the created candles.
- `type`: The type of data for forming candles (e.g., best bid, best ask, or spread middle).
- `candleBuilderProvider`: The provider for the candle builder (optional parameter).

Usage Example

```
private IEnumerable<CandleMessage> InternalGetCandles(SecurityId securityId, DateTime? from,
DateTime? to)
{
    // ... (initialization code)
```

```

switch (type)
{
    case BuildTypes.Ticks:
        return StorageRegistry
            .GetTickMessageStorage(securityId, Drive, StorageFormat)
            .LoadAsync(from, to)
            .ToCandles(mdMsg, candleBuilderProvider: candleBuilderProvider);

    case BuildTypes.OrderLog:
        // ... (code for building candles from order log)

    case BuildTypes Depths:
        return StorageRegistry
            .GetQuoteMessageStorage(securityId, Drive, StorageFormat)
            .LoadAsync(from, to)
            .ToCandles(mdMsg, Convert(extraType), candleBuilderProvider: candleBuilderProvider);

    // ... (other cases)
}
}

// This method demonstrates various ways to build candles depending on the type of source data.
// It supports building from ticks, order log, spreads, and other sources.

```

Additional Features

Building Candles from Various Sources

The API allows building candles not only from ticks and spreads but also from other data sources:

```

// Example of building candles from various sources
switch (type)
{
    case BuildTypes.Ticks:
        // ... (code for ticks)

    case BuildTypes.OrderLog:
        // ... (code for order log)

    case BuildTypes Depths:
        // ... (code for spreads)

    case BuildTypes.Level1:
        // ... (code for Level1)

    case BuildTypes.SmallerTimeFrame:
        return candleBuilderProvider
            .GetCandleMessageBuildableStorage(StorageRegistry, securityId, mdMsg.GetTimeFrame(), Drive,
            StorageFormat)
            .LoadAsync(from, to);

    // ... (other cases)
}

// This code shows how to build candles from different data sources: ticks, order log, spreads,
// Level1 data, and even from smaller time frame candles.

```

Conclusion

The data compression methods in the API provide flexible tools for working with market data. They allow for the efficient conversion of tick data and spread data into candles of various types and time intervals, which is particularly useful for market analysis and developing trading strategies.

Indicators

Indicators

[S#](#) provides more than 140 standard technical analysis indicators. This allows you to use ready-made indicators rather than creating them from scratch. You can also create your own indicators based on existing ones, as shown in the [Custom Indicator](#) section. All base classes for working with indicators, as well as the indicators themselves, are located in the [StockSharp.Algo.Indicators](#) namespace.

Integrating Indicators into a Trading Algorithm

1. First, you need to create an indicator. An indicator is created like a regular .NET object:

```
var longSma = new SimpleMovingAverage { Length = 80 };
var shortSma = new SimpleMovingAverage { Length = 30 };

// It's recommended to add indicators to the strategy collection
Indicators.Add(longSma);
Indicators.Add(shortSma);
```

2. Next, you need to process market data for the indicators. The most efficient approach is to use the result returned by the [Process](#) method:

```
private void ProcessCandle(ICandleMessage candle)
{
    // Process the candle with indicators and immediately save the results
    var longValue = longSma.Process(candle);
    var shortValue = shortSma.Process(candle);

    // Use the results for trading decisions
    if (shortValue.GetValue<decimal>() > longValue.GetValue<decimal>())
    {
        // Buy signal
        BuyAtMarket();
    }
}
```

An indicator accepts [IIndicatorValue](#) as input. Some indicators operate with a simple number, such as [SimpleMovingAverage](#). Others require a complete candle, such as [MedianPrice](#). Therefore, input values need to be cast either to [DecimalIndicatorValue](#) or to [CandleIndicatorValue](#). The indicator's resulting value follows the same rules as the input value.

3. Both the resulting and input values of the indicator have the [IIndicatorValue.IsFinal](#) property, which indicates that the value is final and the indicator will not change at this point in time. For example, the [SimpleMovingAverage](#) indicator is formed based on the candle's closing price, but at the current moment, the final closing price is unknown and changing. In this case, the resulting value of [IIndicatorValue.IsFinal](#) will be false. If you pass a completed candle to the indicator, both the input and resulting values of [IIndicatorValue.IsFinal](#) will be true.
4. **Recommended approach:** directly use the values obtained from calling the [Process](#) method, instead of subsequently calling [GetCurrentValue](#):

```
// Example of a strategy with two moving averages
private void ProcessCandle(ICandleMessage candle)
{
    // Process the candle with indicators and immediately save the results
    var longValue = _longSma.Process(candle);
    var shortValue = _shortSma.Process(candle);

    // Draw on the chart
```

```

    DrawCandlesAndIndicators(candle, longValue, shortValue);

    if (!IsFormedAndOnlineAndAllowTrading())
        return;

    // Use the obtained values for comparison
    var isShortLessCurrent = shortValue.GetValue<decimal>() < longValue.GetValue<decimal>();
    var isShortLessPrev = _shortSma.GetValue(1) < _longSma.GetValue(1);

    // Check if a crossover occurred
    if (isShortLessCurrent == isShortLessPrev)
        return;

    var volume = Volume + Math.Abs(Position);

    // Trading actions based on the signal
    if (isShortLessCurrent)
        SellMarket(volume);
    else
        BuyMarket(volume);
}

```

This approach has the following advantages:

- It aligns with the streaming data processing model (receive → process → use the result)
- It's more efficient as it avoids repeatedly accessing the container of accumulated values
- It eliminates potential synchronization issues between calling `Process` and subsequent `GetCurrentValue` calls

5. Not recommended approach (less efficient):

```

// Suboptimal approach
foreach (var candle in candles)
{
    // Process the candle but ignore the returned value
    _longSma.Process(candle);
    _shortSma.Process(candle);
}

// Later try to get values via GetCurrentValue()
var isShortLessThenLong = _shortSma.GetCurrentValue() < _longSma.GetCurrentValue();

```

With this approach, there's an additional access to the container of historical indicator values, which introduces delays and disrupts the streaming model of data processing.

6. All indicators have the [BaseIndicator.IsFormed](#) property, which indicates whether the indicator is ready for use. For example, the [SimpleMovingAverage](#) indicator has a period, and until the indicator processes a number of candles equal to the indicator's period, the indicator will be considered not ready for use. And the [BaseIndicator.IsFormed](#) property will be false.

Example of a Complete Moving Average Strategy

Below is an example of a strategy that correctly uses indicators, processing candles and utilizing the results of the `Process` method:

```

public class SmaStrategy : Strategy
{
    private readonly StrategyParam<DataType> _series;
    private readonly StrategyParam<int> _longSmaLength;
    private readonly StrategyParam<int> _shortSmaLength;

    private SimpleMovingAverage _longSma;
    private SimpleMovingAverage _shortSma;
}

```

```

private IChartIndicatorElement _longSmaIndicatorElement;
private IChartIndicatorElement _shortSmaIndicatorElement;
private IChartCandleElement _chartCandleElement;
private IChartTradeElement _tradesElem;
private IChart _chart;

public SmaStrategy()
{
    base.Name = "SMA strategy";

    // Initialize strategy parameters
    _longSmaLength = Param(nameof(LongSmaLength), 80);
    _shortSmaLength = Param(nameof(ShortSmaLength), 30);
    _series = Param(nameof(Series), DataType.TimeFrame(TimeSpan.FromMinutes(15)));
}

protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    // Create indicators
    _shortSma = new SimpleMovingAverage { Length = _shortSmaLength.Value };
    _longSma = new SimpleMovingAverage { Length = _longSmaLength.Value };

    // Add indicators to the strategy collection
    Indicators.Add(_shortSma);
    Indicators.Add(_longSma);

    // Initialize chart
    _chart = GetChart();
    if (_chart != null)
    {
        InitChart();
    }

    // Subscribe to candles
    var subscription = new Subscription(_series.Value, Security);

    Connector
        .WhenCandlesFinished(subscription)
        .Do(ProcessCandle)
        .Apply(this);

    Connector.Subscribe(subscription);
}

private void ProcessCandle(ICandleMessage candle)
{
    // Process the candle with indicators and save the results
    var longValue = _longSma.Process(candle);
    var shortValue = _shortSma.Process(candle);

    // Draw on the chart
    DrawCandlesAndIndicators(candle, longValue, shortValue);

    // Check conditions for trading
    if (!IsFormedAndOnlineAndAllowTrading())
        return;

    // Compare current and previous indicator values
    var isShortLessCurrent = shortValue.GetValue<decimal>() < longValue.GetValue<decimal>();
    var isShortLessPrev = _shortSma.GetValue(1) < _longSma.GetValue(1);

    // Check for crossover
    if (isShortLessCurrent == isShortLessPrev)
        return;

    var volume = Volume + Math.Abs(Position);

```

```

// Trading actions based on the signal
if (isShortLessCurrent)
    SellMarket(volume);
else
    BuyMarket(volume);
}

private void DrawCandlesAndIndicators(ICandleMessage candle, IIndicatorValue longSma,
IIndicatorValue shortSma)
{
    if (_chart == null) return;
    var data = _chart.CreateData();
    data.Group(candle.OpenTime)
        .Add(_chartCandleElement, candle)
        .Add(_longSmaIndicatorElement, longSma)
        .Add(_shortSmaIndicatorElement, shortSma);
    _chart.Draw(data);
}

// Other chart initialization methods omitted for brevity
}

```

This example demonstrates the correct approach to working with indicators in the StockSharp streaming model.

Indicator adding to the chart

Adding an Indicator to the Chart

The following example demonstrates how to add an indicator for drawing on a chart:

```
private readonly Connector _connector = new Connector();
private Security _security;
private Subscription _candleSubscription;
private SimpleMovingAverage _sma;
readonly TimeSpan _timeFrame = TimeSpan.FromMinutes(1);
private ChartArea _area;
private ChartCandleElement _candlesElem;
private ChartIndicatorElement _longMaElem;

// Initializing chart and indicator
private void InitializeChart()
{
    // _chart - StockSharp.Xaml.Charting.Chart
    // Creating a chart area
    _area = new ChartArea();
    _chart.Areas.Add(_area);

    // Creating a chart element representing candles
    _candlesElem = new ChartCandleElement();
    _area.Elements.Add(_candlesElem);

    // Creating a chart element representing the indicator
    _longMaElem = new ChartIndicatorElement
    {
        Title = "Long"
    };
    _area.Elements.Add(_longMaElem);

    // Creating an indicator
    _sma = new SimpleMovingAverage() { Length = 80 };

    // Subscribing to the candle receiving event
    _connector.CandleReceived += OnCandleReceived;
}

// Method for subscribing to candles
private void SubscribeToCandles()
{
    // Creating a subscription to candles with the specified timeframe
    _candleSubscription = new Subscription(
        DataType.TimeFrame(_timeFrame),
        _security)
    {
        MarketData =
        {
            // Requesting historical data for 30 days
            From = DateTime.Today.Subtract(TimeSpan.FromDays(30)),
            To = DateTime.Now,
            // Receiving only finished candles
            IsFinishedOnly = true
        }
    };
};

// Starting the subscription
_connector.Subscribe(_candleSubscription);
}

// Handler for the candle receiving event
private void OnCandleReceived(Subscription subscription, ICandleMessage candle)
```

```

{
    // Checking if the candle belongs to our subscription
    if (subscription != _candleSubscription)
        return;

    // Checking the candle state
    if (candle.State != CandleStates.Finished)
        return;

    // Processing the candle with the indicator
    var longValue = _sma.Process(candle);

    // Creating data for drawing
    var data = new ChartDrawData();
    data
        .Group(candle.OpenTime)
        .Add(_candlesElem, candle)
        .Add(_longMaElem, longValue);

    // Drawing on the chart in the UI thread
    this.GuiAsync(() => _chart.Draw(data));
}

// Method for unsubscribing when closing the window
private void UnsubscribeFromCandles()
{
    if (_candleSubscription != null)
    {
        _connector.CandleReceived -= OnCandleReceived;
        _connector.UnSubscribe(_candleSubscription);
        _candleSubscription = null;
    }
}

```

[image: indicators chart]

Example of working with multiple indicators

```

private readonly Connector _connector = new Connector();
private Security _security;
private Subscription _candleSubscription;
private SimpleMovingAverage _shortSma;
private SimpleMovingAverage _longSma;
private ChartArea _mainArea;
private ChartArea _indicatorArea;
private ChartCandleElement _candlesElem;
private ChartIndicatorElement _shortSmaElem;
private ChartIndicatorElement _longSmaElem;
private RelativeStrengthIndex _rsi;
private ChartIndicatorElement _rsiElem;

// Initializing chart and indicators
private void InitializeChartWithMultipleIndicators()
{
    // Creating the main area for candles and moving averages
    _mainArea = new ChartArea();
    _chart.Areas.Add(_mainArea);

    // Creating an area for RSI
    _indicatorArea = new ChartArea();
    _chart.Areas.Add(_indicatorArea);

    // Creating chart elements
    _candlesElem = new ChartCandleElement();
    _shortSmaElem = new ChartIndicatorElement { Title = "SMA (short)" };
    _longSmaElem = new ChartIndicatorElement { Title = "SMA (long)" };
    _rsiElem = new ChartIndicatorElement { Title = "RSI" };
}

```

```

// Setting element colors
_shortSmaElem.Color = Colors.Red;
_longSmaElem.Color = Colors.Blue;
_rsiElem.Color = Colors.Green;

// Adding elements to their respective areas
_mainArea.Elements.Add(_candlesElem);
_mainArea.Elements.Add(_shortSmaElem);
_mainArea.Elements.Add(_longSmaElem);
_indicatorArea.Elements.Add(_rsiElem);

// Creating indicators
_shortSma = new SimpleMovingAverage { Length = 9 };
_longSma = new SimpleMovingAverage { Length = 20 };
_rsi = new RelativeStrengthIndex { Length = 14 };

// Subscribing to the candle receiving event
_connector.CandleReceived += OnCandleReceivedMultipleIndicators;

// Creating a subscription to candles
_candleSubscription = new Subscription(
    DataType.TimeFrame(TimeSpan.FromMinutes(5)),
    _security)
{
    MarketData =
    {
        From = DateTime.Today.Subtract(TimeSpan.FromDays(30)),
        To = DateTime.Now,
        IsFinishedOnly = true
    }
};

// Starting the subscription
_connector.Subscribe(_candleSubscription);
}

// Handler for the candle receiving event for multiple indicators
private void OnCandleReceivedMultipleIndicators(Subscription subscription, ICandleMessage candle)
{
    // Checking if the candle belongs to our subscription
    if (subscription != _candleSubscription)
        return;

    if (candle.State != CandleStates.Finished)
        return;

    // Processing the candle with indicators
    var shortSmaValue = _shortSma.Process(candle);
    var longSmaValue = _longSma.Process(candle);
    var rsiValue = _rsi.Process(candle);

    // Creating data for drawing
    var data = new ChartDrawData();
    data
        .Group(candle.OpenTime)
        .Add(_candlesElem, candle)
        .Add(_shortSmaElem, shortSmaValue)
        .Add(_longSmaElem, longSmaValue)
        .Add(_rsiElem, rsiValue);

    // Drawing on the chart in the UI thread
    this.GuiAsync(() => _chart.Draw(data));
}

```

See also

[Components for building charts](#)

List of indicators

List of Indicators

S# provides over 140 technical analysis indicators by default. This allows you to use ready-made indicators instead of creating them from scratch.

- **A/D** - Acceleration / Deceleration Indicator, measures the acceleration or deceleration of price movement
- **ADL** - Accumulation/Distribution Line (A/D Line), tracks the inflow or outflow of money into a security
- **ALF** - Adaptive Laguerre Filter, a filter that adjusts to market conditions to smooth price data
- **APZ** - Adaptive Price Zone, identifies potential support and resistance levels based on volatility
- **Alligator** - Alligator, a trend-following indicator using three moving averages to determine market trends
- **AFI** - Approval Flow Index, measures trend strength based on price and volume
- **ALMA** - Arnaud Legoux Moving Average, a moving average with reduced lag and noise
- **Aroon** - Aroon indicator, determines the strength and direction of a trend
- **Aroon Oscillator** - Aroon Oscillator, shows the spread between Aroon Up and Aroon Down
- **ADX** - Welles Wilder Average Directional Index, measures trend strength
- **ATR** - Average True Range, measures market volatility
- **AO** - Awesome Oscillator, measures market momentum
- **Bear Power** - Bear Power, measures bearish pressure relative to EMA
- **Bull Power** - Bull Power, measures bullish pressure relative to EMA
- **BMP** - Balance of Market Power, compares buying and selling pressure
- **BOP** - Balance of Power, shows the strength of buyers versus sellers
- **BV** - Balance Volume indicator, links volume to price changes
- **Bollinger** - Bollinger Bands, measures volatility and identifies overbought or oversold conditions
- **BBP** - Bollinger %b, shows the price position relative to Bollinger Bands
- **Pattern** - Pattern, recognizes specific price patterns on charts
- **CGO** - Center of Gravity Oscillator, identifies potential reversal points
- **CMF** - Chaikin Money Flow, measures buying and selling pressure based on volume
- **Chaikin's Volatility** - Chaikin volatility, measures the rate of change in the trading range
- **CKS** - Chande Kroll Stop, provides stop-loss levels based on volatility
- **CMO** - Chande Momentum Oscillator, measures the momentum of price changes
- **CHOP** - Choppiness Index, determines whether the market is trending or ranging
- **CCI** - Commodity Channel Index, identifies cyclical trends in commodity markets
- **CM** - Composite Momentum, combines multiple momentum indicators
- **CRSI** - Connors RSI, a composite indicator for identifying overbought and oversold conditions

- [CBCI](#) - Constance Brown Composite Index, measures market momentum and trend strength
- [DI](#) - Demand Index, compares buying and selling pressure
- [DPO](#) - Price oscillator without trend, removes the trend component to focus on cycles
- [DSP](#) - Detrended Synthetic Price, removes trends to analyze price fluctuations
- [DMI](#) - Welles Wilder Directional Movement Index, determines trend direction and strength
- [DPI](#) - Disparity Index, measures the difference between price and a moving average
- [DC](#) - Donchian Channels, identifies breakout levels based on highs and lows
- [DEMA](#) - Double Exponential Moving Average, reduces lag in moving average calculations
- [DeMarker](#) - DeMarker, compares consecutive highs and lows to gauge demand
- [DZRSI](#) - Dynamic Zones RSI, adapts RSI levels to market conditions
- [EMV](#) - Ease of Movement, links price changes to volume
- [EFT](#) - Ehlers Fisher Transform, normalizes prices to highlight extreme values
- [EIS](#) - Elder Impulse System, combines trend and momentum to find trading opportunities
- [Elder Ray](#) - Elder Ray Index, evaluates the balance between bulls and bears using EMA
- [EWO](#) - Elliot Wave Oscillator, identifies wave patterns in price movement
- [EPMA](#) - Endpoint Moving Average, a moving average with reduced lag
- [Envelope](#) - Envelope, creates bands around a moving average to identify overbought and oversold levels
- [EMA](#) - Exponential Moving Average, gives more weight to recent prices
- [FR](#) - Fibonacci Retracement, determines potential reversal levels based on Fibonacci ratios
- [FVE](#) - Finite Volume Element, measures the strength of price movements considering volume
- [FI](#) - Force Index, combines price and volume to measure buying and selling pressure
- [EFI](#) - Elder's Force Index, measures buying and selling pressure
- [FOSC](#) - Forecast Oscillator, compares actual prices to forecasted ones
- [FRAMA](#) - Fractal Adaptive Moving Average, adapts to market volatility
- [Fractals](#) - Fractals, identifies potential reversal points on price charts
- [FDI](#) - Fractal Dimension Index, measures market roughness
- [Gator](#) - Gator oscillator, measures convergence and divergence of moving averages
- [GAPO](#) - Gopalakrishnan Range Index, measures the variability of price movements
- [GMMA](#) - Guppy Multiple Moving Average, uses multiple moving averages to identify trends
- [HO](#) - Harmonic Oscillator, detects harmonic patterns in price data
- [Highest](#) - Maximum value for a period, determines the highest price over a specified period
- [HLI](#) - High Low Index, measures the number of new highs and lows
- [HVR](#) - Historical Volatility Ratio, compares current volatility to historical levels
- [H](#) - Hurst Exponent, measures the long-term memory of a series
- [HMA](#) - Hull Moving Average, a fast and smooth moving average

- [Ichimoku](#) - Ichimoku, a comprehensive indicator for trend identification
- [III](#) - Intraday Intensity Index, measures buying and selling pressure during the trading day
- [IMI](#) - Intraday Momentum Index, identifies overbought and oversold conditions intraday
- [JMA](#) - Jurik Moving Average, a smooth and responsive moving average
- [KPO](#) - Kase Peak Oscillator, identifies potential reversal points
- [KER](#) - Kaufman Efficiency Ratio, measures the efficiency of price movement
- [KAMA](#) - Kaufman adaptive moving average, adapts to market volatility
- [Kalman Filter](#) - Adaptive filter for tracking trends in noisy data
- [KC](#) - Keltner Channels indicator, defines trading ranges based on volatility
- [KVO](#) - Klinger Volume Oscillator, measures volume trends
- [KST](#) - Know Sure Thing, a momentum oscillator
- [LRSI](#) - Laguerre RSI, a modified RSI with reduced lag
- [LRC](#) - Linear regression, forecasts future prices based on past trends
- [LRF](#) - Linear Regression Forecast, predicts future prices
- [R-squared](#) - Linear regression R-squared, measures the strength of a linear relationship
- [LRS](#) - Linear regression gradient, indicates the slope of the regression line
- [Lowest](#) - Minimum value for a period, determines the lowest price over a specified period
- [LP](#) - Lunar Phase indicator, correlates market movements with lunar cycles
- [MFI](#) - Market Facilitation Index, measures the efficiency of price movement
- [MMI](#) - Market Meanness Index, determines mean-reverting behavior
- [MI](#) - Mass Index, identifies potential reversals based on range expansion
- [MCO](#) - McClellan Oscillator, measures market breadth
- [MGD](#) - McGinley Dynamic, a responsive moving average
- [MeanDev](#) - Average deviation, measures the average deviation from the mean
- [Moving Median](#) - Moving Median, smooths data using the median of the window
- [MedPr](#) - Median price, calculates the midpoint of the trading range
- [Momentum](#) - Momentum, measures the speed of price change
- [MOMA](#) - Momentum of Moving Average, measures the momentum of a moving average
- [MP](#) - Momentum Pinball indicator, identifies overbought and oversold conditions
- [MFI](#) - Money Flow Index, measures buying and selling pressure based on price and volume
- [MACD](#) - Convergence/divergence of moving averages, identifies trend changes
- [MACD Histogram](#) - Convergence/divergence of moving averages. Histogram, visualizes the difference between MACD and signal line
- [MACD Signal](#) - Convergence/divergence of moving averages with signal line, provides buy and sell signals
- [MAC](#) - Moving Average Crossover, identifies trend reversals using two moving averages

- [MAR](#) - Moving Average Ribbon, uses multiple moving averages to visualize trends
- [NVI](#) - Negative Volume Index, tracks price movements on days with decreasing volume
- [PVI](#) - Positive Volume Index, tracks price movements on days with increasing volume
- [NRTR](#) - Nick Rypock Trailing reverse, a trailing stop indicator
- [OBV](#) - On-Balance Volume (OBV), links volume to price changes
- [OBVM](#) - On Balance Volume Mean, a smoothed version of OBV
- [OptimalTracking](#) - Optimal Tracking Filter published by John Ehlers, filters price data to reduce noise
- [OMA](#) - Oscillator of Moving Average, measures the difference between a moving average and price
- [Parabolic SAR](#) - Parabolic SAR trend indicator implementation, identifies potential reversal points
- [Stub](#) - An indicator without logic. Used to draw lines, a placeholder for custom indicators
- [Peak](#) - Peak, identifies local highs in price data
- [PPO](#) - Percentage Price Oscillator, measures the percentage difference between two moving averages
- [PPOS](#) - Percentage Price Oscillator with signal line, smooths PPO with an EMA
- [PPOH](#) - Percentage Price Oscillator histogram, plots the difference between PPO and its signal line
- [PVO](#) - Percentage Volume Oscillator, measures the percentage difference in volume trends
- [PP](#) - Pivot Points, calculates support and resistance levels based on past price movements
- [PC](#) - Price Channels, show upper and lower price boundaries
- [PVT](#) - Price Volume Trend, multiplies price change by volume
- [PGO](#) - Pretty Good Oscillator, identifies overbought and oversold conditions
- [PSY](#) - Psychological Line, measures market sentiment based on price changes
- [QStick](#) - QStick, measures buying and selling pressure
- [RC](#) - Rainbow Charts, uses multiple moving averages to visualize trends
- [RCI](#) - Rank Correlation Index, measures trend using Spearman's rank correlation
- [RAVI](#) - Range Action Verification Index, measures trend strength
- [ROC](#) - Rate of change, measures the percentage change in price
- [RMI](#) - Relative Momentum Index, a variation of RSI incorporating momentum
- [RSI](#) - Relative Strength Index, measures the speed and change of price movements
- [RVI](#) - Relative Vigor Index, measures trend confidence
- [STC](#) - Schaff Trend Cycle, identifies trend cycles
- [SuperTrend](#) - SuperTrend indicator based on ATR
- [SMA](#) - Simple moving average, calculates the average price over a period
- [Shift](#) - Shift, offsets the data stream by a chosen number of periods
- [SW](#) - Sine Wave, identifies cyclical patterns in price data
- [SMMA](#) - Smoothed Moving Average, a moving average giving more weight to recent prices
- [Std Dev](#) - Standard deviation, measures the dispersion of price data

- [Standard error](#) - Standard error in linear regression, measures forecast accuracy
- [Stochastic %K](#) - Stochastic %K, measures momentum based on price position within a range
- [Stochastic Oscillator](#) - Stochastic Oscillator, identifies overbought and oversold conditions
- [Sum](#) - Sum of N last values, calculates the sum of the last N prices
- [T3MA](#) - T3 Moving Average, a smooth and responsive moving average
- [TWAP](#) - Time Weighted Average Price, calculates the average price weighted by time
- [TP](#) - Typical Price, average of High, Low, and Close
- [TEMA](#) - Triple Exponential Moving Average, reduces lag in moving average calculations
- [TRIX](#) - Triple Exponential Moving Average, identifies trend changes
- [Trough](#) - Trough, identifies local lows in price data
- [TR](#) - True range, measures the actual range of price movement
- [TSI](#) - True Strength Index, applies double-smoothed momentum analysis
- [TMF](#) - Twiggs Money Flow, measures buying and selling pressure
- [UltimateOsc](#) - Last oscillator, combines multiple timeframes to measure momentum
- [VHF](#) - Vertical-horizontal filter, distinguishes trending from ranging markets
- [VIDIA](#) - Variable Index Dynamic Average, adapts to market volatility
- [VMA](#) - Variable Moving Average, adjusts to volatility
- [Volume](#) - Candle volume, measures trading volume per candle
- [Volume profile](#) - Volume profile, shows trading activity at different price levels
- [VMA](#) - Volume weighted moving average, gives more weight to prices with higher volume
- [VWAP](#) - Volume Weighted Average Price, average price weighted by traded volume
- [VI](#) - Vortex, identifies the start of new trends
- [WTO](#) - Wave Trend Oscillator, identifies trend reversals
- [WMA](#) - Weighted moving average, gives more weight to recent prices
- [WCP](#) - Weighted Close Price, $(\text{High} + \text{Low} + 2 \times \text{Close}) / 4$
- [Wilder MA](#) - Welles Wilder Moving Average, a smoothed moving average
- [WAD](#) - Williams Accumulation/Distribution, measures buying and selling pressure
- [%R](#) - Williams Percent Range, identifies overbought and oversold conditions
- [WCCI](#) - Woodies CCI, a variation of CCI for trend identification
- [ZLEMA](#) - Zero Lag Exponential Moving Average, reduces lag in EMA calculations
- [Zig Zag](#) - ZigZag, filters out minor price movements to display significant trends

Notes

This list includes technical analysis indicators from the StockSharp library. Each indicator is designed for various aspects of financial market analysis:

- Trend identification

- Volatility assessment
- Reversal point detection
- Momentum measurement
- Trading volume analysis
- Support and resistance level identification

Indicators assist traders in making informed decisions by analyzing historical price data and identifying patterns.

Acceleration / Deceleration Indicator (A/D)

A/D

Acceleration/Deceleration (A/D) is an oscillator that was created by Bill Williams. It measures the acceleration and deceleration of the trend's momentum.

To use the indicator, the [Acceleration](#) class should be used.

Calculation

The A/D histogram is the difference between the value of the 5/34 histogram of driving force and the 5-period simple moving average taken from this histogram. The values are taken for the classic oscillator, and in the settings, it is always possible to specify your own parameters.

$$\text{MEDIAN PRICE} = (\text{HIGH} + \text{LOW}) / 2$$
$$\text{AO} = \text{SMA}(\text{MEDIAN PRICE}, 5) - \text{SMA}(\text{MEDIAN PRICE}, 34)$$
$$\text{A/D} = \text{AO} - \text{SMA}(\text{AO}, 5)$$

where:

MEDIAN PRICE — median price; HIGH — the highest price of the bar; LOW — the lowest price of the bar; SMA — simple moving average; AO — [Awesome Oscillator](#) indicator.

The parameters are set as the SMA periods values.

[image: IndicatorAcceleration]

See Also

[Alligator](#)

Accumulation/Distribution Line (ADL)

ADL

Accumulation/Distribution Line (ADL) is a volume indicator developed by Mark Chaikin. The indicator evaluates the supply and demand relationship in the market by analyzing the correlation between price and volume.

To use the indicator, you need to use the [AccumulationDistributionLine](#) class.

Description

The Accumulation/Distribution Line is a cumulative indicator that uses volume and price to determine whether a security is in an accumulation (buying) or distribution (selling) phase.

The ADL indicator helps confirm a trend or warn about its potential reversal:

- If the price is rising and ADL is falling, this may signal weakness in an upward trend.
- If the price is falling and ADL is rising, this may indicate a potential reversal of a downward trend.

Calculation

The Accumulation/Distribution Line calculation occurs in two steps:

1. Volume Multiplier (CLV - Close Location Value) Calculation:

$$CLV = ((Close - Low) - (High - Close)) / (High - Low)$$

2. ADL Calculation:

$$ADL = Previous\ ADL\ Value + CLV * Volume$$

Where:

- Close - closing price of the period
- Low - minimum price of the period
- High - maximum price of the period
- Volume - trading volume for the period

If (High - Low) is equal to zero, CLV is set to zero.

[image: IndicatorAccumulationDistributionLine]

See Also

[OBV](#)

Adaptive Laguerre Filter (ALF)

ALF

Adaptive Laguerre Filter (ALF) is an indicator developed to smooth price data with minimal lag, based on the mathematical principles of the Laguerre filter.

To use the indicator, you need to use the [AdaptiveLaguerreFilter](#) class.

Description

The Adaptive Laguerre Filter is an advanced market noise filtering tool. It provides a smoother representation of price movement while maintaining a quick response to real trend changes. This filter is particularly useful for reducing the lag often encountered in traditional smoothing indicators.

The main advantage of ALF over classical moving averages lies in its ability to more effectively separate market noise from genuine price movements, making it a valuable tool for traders seeking to reduce false signals.

Parameters

The indicator has the following parameters:

- **Gamma** - filtering coefficient (typically in the range of 0.1 to 0.9)

The Gamma parameter determines the degree of smoothing: lower values create a smoother line with more lag, while higher values result in less smoothing but a faster response to price changes.

Calculation

The Adaptive Laguerre Filter is based on Laguerre polynomials and represents a finite impulse response (FIR) filtering system. The calculation uses the following formulas:

1. Intermediate values L0, L1, L2, and L3 are calculated:

$$\begin{aligned}L_0(t) &= (1 - \gamma) * \text{price}(t) + \gamma * L_0(t-1) \\L_1(t) &= -\gamma * L_0(t) + L_0(t-1) + \gamma * L_1(t-1) \\L_2(t) &= -\gamma * L_1(t) + L_1(t-1) + \gamma * L_2(t-1) \\L_3(t) &= -\gamma * L_2(t) + L_2(t-1) + \gamma * L_3(t-1)\end{aligned}$$

2. The final ALF value is calculated as the average:

$$ALF = (L_0 + L_1 + L_2 + L_3) / 4$$

Where:

- γ (gamma) - filtering coefficient
- $\text{price}(t)$ - current price
- L0, L1, L2, L3 - intermediate filter values

[image: indicator_adaptive_laguerre_filter]

See Also

[LaguerreRSI ZLEMA](#)

Adaptive Price Zone (APZ)

APZ

Adaptive Price Zone (APZ) is a technical indicator developed by Lee Leibfarth that creates dynamic support and resistance zones, adapting to market volatility.

To use the indicator, you need to use the [AdaptivePriceZone](#) class.

Description

The APZ indicator consists of two lines (upper and lower) that form a price zone around the average price. This zone expands and contracts depending on the current market volatility. When the market becomes more volatile, the zone expands; when volatility decreases, the zone narrows.

APZ is particularly useful for:

- Identifying potential support and resistance levels
- Detecting possible trend reversal points
- Revealing periods of increased and decreased volatility
- Creating trading systems based on price zone breakouts

Parameters

The indicator has the following parameters:

- **Period** - calculation period (default value: 5)
- **BandPercentage** - percentage of range to define band width (default value: 2%)

Calculation

The APZ calculation is based on the Exponential Moving Average (EMA) and Average True Range (ATR):

1. First, calculate the EMA of price for the specified period:

```
EMA = Exponential Moving Average of price over Period
```

2. Then calculate volatility using ATR:

```
Volatility = Exponential Moving Average of ATR over Period
```

3. The upper and lower APZ lines are computed as follows:

```
Upper Line = EMA + (Volatility * BandPercentage)
Lower Line = EMA - (Volatility * BandPercentage)
```

When the price is above the upper APZ line, this can be considered an upward trend. When the price is below the lower APZ line, this may indicate a downward trend. When the price moves within the APZ zone, the market may be in a consolidation or sideways movement phase.

[image: indicator_adaptive_price_zone]

See Also

[BollingerBands](#) [KeltnerChannels](#) [DonchianChannels](#)

Alligator

Alligator

The **Alligator** indicator consists of a group of three moving averages. These moving averages have different periods and are also shifted forward on the chart.

To use the indicator, the [Alligator](#) class should be used.

Features of the "Alligator" indicator

The indicator consists of three differently colored lines:

- Blue, named the jaw, with a calculation period of 13 and a shift of 8 bars. When it is located under the price curve, it indicates a potential upward price movement. If the "jaw" rises above it, a decline is expected.
- Red, called the alligator's teeth, with a period of 8 and a shift of 5 bars. It belongs to the fast averages and shows the price behavior in the hourly range.
- Green, called the lips, with a period of 5 and a shift of 3 bars. It analyzes twelve-minute market trends.

The values are taken for the classic indicator, and in the settings, it is always possible to specify your own parameters.

[image: IndicatorAlligator]

See Also

[ADX](#)

Approval Flow Index (AFI)

AFI

Approval Flow Index (AFI) is an indicator that measures trend strength based on the relationship between volume and price movement.

To use the indicator, you need to use the [ApprovalFlowIndex](#) class.

Description

The Approval Flow Index (AFI) helps assess the intensity of order flow in the market and determine the strength of the current trend. This indicator analyzes the relationship between trading volume and price movement to identify potential reversal points or confirm trend continuation.

The AFI indicator can be used for:

- Determining the strength of the current trend
- Identifying divergences between price and the indicator
- Searching for potential market reversal points

Parameters

The indicator has the following parameters:

- **Length** - calculation period of the indicator

Calculation

The Approval Flow Index calculation is based on analyzing price change and volume over a specific period:

1. First, calculate the price change for the period
2. Then relate this change to trading volume
3. The resulting values are summed over the selected period (Length parameter)

AFI aims to determine how much trading volume "approves" the price movement.

Positive AFI values indicate a strong upward trend, while negative values suggest a downward trend. Values close to zero may indicate the absence of a pronounced trend.

[image: indicator_approval_flow_index]

See Also

[ADL](#) [OBV](#)

Arnaud Legoux Moving Average (ALMA)

ALMA

Arnaud Legoux Moving Average (ALMA) is an indicator developed by Arnaud Legoux and optimized to eliminate market noise and reduce signal lag.

To use the indicator, you need to use the [ArnaudLegouxMovingAverage](#) class.

Description

ALMA combines the advantages of two data smoothing approaches:

1. Eliminating market noise (like most moving averages)
2. Minimizing lag (typical of many smoothing indicators)

The ALMA indicator uses a normal (Gaussian) distribution as a weight function, which can be fine-tuned using offset and sigma parameters. This makes it a very flexible and effective tool for technical analysis.

ALMA is used for:

- Determining the current trend
- Identifying reversal points
- Creating trading systems based on crossovers

Parameters

The indicator has the following parameters:

- **Length** - calculation period (number of candles to analyze)
- **Sigma** - sigma, a parameter controlling the shape of the Gaussian curve (recommended value: 6)
- **Offset** - offset, a parameter controlling smoothing and response speed (recommended value: 0.85)

Calculation

The ALMA calculation occurs in several stages:

1. Determining weights for each data point in the window based on normal (Gaussian) distribution:

```
m = floor(Offset * (Length - 1))
s = Length / Sigma

For each i from 0 to Length-1:
w(i) = exp(-((i - m)^2) / (2 * s^2))
```

2. Normalizing weights:

```
Sum_of_weights = sum of all w(i)

For each i from 0 to Length-1:
w_norm(i) = w(i) / Sum_of_weights
```

3. Calculating ALMA as a weighted sum:

```
ALMA = sum(Price(t-i) * w_norm(i)) for all i from 0 to Length-1
```


Where:

- Length - ALMA period
- Offset - offset parameter (from 0 to 1)
- Sigma - sigma parameter (usually from 2 to 8)

[image: indicator_arnaud_legoux_moving_average]

See Also

[SMA](#) [EMA](#) [T3MA](#) [ZLEMA](#)

Aroon Indicator (Aroon)

Aroon

Aroon Indicator is a technical indicator developed by Tushar Chande in 1995 to identify trend changes and trend strength. The name "Aroon" comes from a Sanskrit word meaning "dawn of a new era".

To use the indicator, you need to use the [Aroon](#) class.

Description

The Aroon Indicator consists of two lines:

- **Aroon Up** - measures the strength of an upward trend
- **Aroon Down** - measures the strength of a downward trend

Aroon helps determine:

- The beginning of a new trend
- The strength of the current trend
- Consolidation and sideways movement
- Potential trend reversals

The indicator is particularly useful for identifying early stages of a new trend formation and for determining consolidation periods.

Parameters

The indicator has the following parameters:

- **Length** - calculation period (typically 14-25 periods are used)

Calculation

The Aroon Indicator calculation is based on determining the time (number of periods) elapsed since reaching the maximum and minimum prices within the specified period:

1. Aroon Up is calculated using the formula:

$$\text{Aroon Up} = ((\text{Length} - \text{Periods since high}) / \text{Length}) * 100$$

2. Aroon Down is calculated using the formula:

$$\text{Aroon Down} = ((\text{Length} - \text{Periods since low}) / \text{Length}) * 100$$

Where:

- Length - selected period
- "Periods since high" - number of periods since reaching the highest price within the Length period
- "Periods since low" - number of periods since reaching the lowest price within the Length period

Both Aroon lines oscillate between 0 and 100:

- A value of 100 means the high/low was reached in the most recent period

- A value of 0 means the high/low was reached Length periods ago

Interpretation

- **Strong Upward Trend:** Aroon Up is close to 100, and Aroon Down is close to 0
- **Strong Downward Trend:** Aroon Down is close to 100, and Aroon Up is close to 0
- **Sideways Movement:** both lines move parallel to each other at low levels
- **Potential Trend Reversal:** crossing of Aroon Up and Aroon Down lines
- **Consolidation:** both lines oscillate around 50

[image: indicator_aroon]

See Also

[ADX DMI](#)

Average Directional Index (ADX)

ADX

Welles Wilder's ADX (Average Directional Index) is a group of directional movement indicators. Lines compose the Directional Movement Index (DMI): ADX, DI+ and DI-. ADX indicates the trend strength, DI+ and DI- indicate the current price direction.

To use the indicator, you must use the [AverageDirectionalIndex](#).

[image: IndicatorAverageDirectionalIndex]

Recommended content

[ATR](#)

Average True Range (ATR)

ATR

Average True Range (ATR) is an indicator that shows the level of current volatility.

To use the indicator, the [AverageTrueRange](#) class should be used.

Calculation of the Indicator

The calculation of the indicator begins with determining the True Range (TR), which is calculated as the maximum of the following three values:

- the difference between the current maximum and minimum;
- the difference between the current maximum and the previous closing price (absolute value);
- the difference between the current minimum and the previous closing price (absolute value).

$$TR_t = \max(\text{High}(t) - \text{Low}(t); \text{High}(t) - \text{Close}(t-1); \text{Close}(t-1) - \text{Low}(t))$$

The absolute value is used to ensure positive values, as we are interested in the distance between two points, not the direction of price movement.

Based on this indicator, ATR is calculated. It has only one parameter - the period N. By default, a 14-period indicator is used, but it can be adjusted to fit your own strategy. Here is what the formula looks like (this is one of the forms of the exponential moving average)

$$ATR(t) = ((ATR(t-1) \times (N-1)) + TR(t)) / N$$

[image: IndicatorAverageTrueRange]

See Also

[AO](#)

Awesome Oscillator (AO)

AO

The **Awesome Oscillator (AO)** is a classic technical indicator constructed by subtracting moving averages (SMA) with different periods.

To use the indicator, the [AwesomeOscillator](#) class should be used.

Calculation

The Awesome Oscillator histogram is a 34-period simple moving average constructed on the central values of bars $(H+L) / 2$, subtracted from a 5-period simple moving average on the central points $(H+L) / 2$. Thus, the slow moving average line is subtracted from the fast one, to get an idea of the strength of price movement and its further intentions.

$\text{MEDIAN PRICE} = (\text{HIGH} + \text{LOW}) / 2$
 $\text{AO} = \text{SMA}(\text{MEDIAN PRICE}, 5) - \text{SMA}(\text{MEDIAN PRICE}, 34)$, where

MEDIAN PRICE – median price
 HIGH – the highest price of the bar
 LOW – the lowest price of the bar
 SMA – simple moving average

The values are taken for the classic indicator, and in the settings, it is always possible to specify your own parameters.

[image: IndicatorAwesomeOscillator]

See Also

[Bollinger Bands](#)

Balance of Market Power (BMP)

BMP

Balance of Market Power (BMP) is an indicator that measures the strength of buyers relative to sellers, based on an analysis of price movements and trading volumes.

To use the indicator, you need to use the [BalanceOfMarketPower](#) class.

Description

The Balance of Market Power indicator is designed to assess the current distribution of forces between buyers and sellers in the market. It analyzes how much the closing price deviates from its range (high-low) and correlates this with trading volume.

BMP helps traders:

- Determine the dominant market side (buyers or sellers)
- Identify potential trend reversals
- Detect divergences between price and the indicator
- Find overbought and oversold levels

Parameters

The indicator has the following parameters:

- **Length** - smoothing period (default value: 14)

Calculation

BMP calculation occurs in two stages:

1. Calculating BMP for each individual candle:

```
Raw BMP = ((Close Price - Open Price) / (High - Low)) * Volume
```

If (High - Low) is zero, the raw BMP is set to zero.

2. Smoothing BMP using a Simple Moving Average (SMA):

```
BMP = SMA(Raw BMP, Length)
```

Where:

- Close Price - closing price of the current candle
- Open Price - opening price of the current candle
- High - highest price of the current candle
- Low - lowest price of the current candle
- Volume - trading volume for the current candle period
- Length - selected smoothing period

Interpretation

- **Positive BMP values** indicate dominance of buyers (bulls) in the market
- **Negative BMP values** indicate dominance of sellers (bears) in the market
- **Crossing the zero line** can be considered a trend change signal
- **Extreme values** (above or below certain levels) may indicate market overbought or oversold conditions
- **Divergences** between BMP and price may signal a potential trend reversal

[image: indicator_balance_of_market_power]

See Also

[BalanceOfPower](#) [ForceIndex](#) [ADL](#) [OBV](#)

Balance of Power (BOP)

BOP

Balance of Power (BOP) is an indicator designed to measure the strength of bulls (buyers) relative to bears (sellers) by evaluating the bulls' ability to raise the price from the low to the high.

To use the indicator, you need to use the [BalanceOfPower](#) class.

Description

The Balance of Power (BOP) indicator displays the balance of forces between buyers and sellers in the market. It is based on the assumption that in a trend, buyers (bulls) or sellers (bears) can control the price throughout the session. By comparing the difference between closing and opening prices with the full price range (high-low), the indicator allows assessment of who is currently dominating the market.

BOP helps traders:

- Determine the direction and strength of the current trend
- Identify potential reversal points
- Detect divergences between price and the indicator
- Determine overbought and oversold levels

Calculation

The formula for calculating the Balance of Power (BOP) indicator is quite simple:

$$\text{BOP} = (\text{Close} - \text{Open}) / (\text{High} - \text{Low})$$

Where:

- Close - closing price
- Open - opening price
- High - highest price for the period
- Low - lowest price for the period

If (High - Low) is zero, BOP is set to zero to avoid division by zero.

BOP is often additionally smoothed using a moving average to reduce volatility and improve signal readability.

Interpretation

- **Positive BOP values** (above zero) indicate that buyers (bulls) are controlling the market, which may signal an upward trend.
- **Negative BOP values** (below zero) indicate that sellers (bears) are controlling the market, which may signal a downward trend.
- **Crossing the zero line** can be considered a signal of a potential trend direction change.
- **Extreme values** (strongly positive or strongly negative) may indicate market overbought or oversold conditions.

- **Divergences** between BOP and price can signal a possible trend reversal:
 - If the price is rising and BOP is falling, this may be a warning of a weakening upward trend.
 - If the price is falling and BOP is rising, this may indicate a potential end to the downward trend.

[image: indicator_balance_of_power]

See Also

[BalanceOfMarketPower](#) [ForceIndex](#) [ADL](#) [OBV](#)

Balance Volume Indicator (BV)

BV

Balance Volume (BV) is a technical indicator that tracks the accumulation and distribution of trading volume based on price changes.

To use the indicator, you need to use the [BalanceVolume](#) class.

Description

The Balance Volume (BV) indicator is designed to analyze the relationship between price change and trading volume. It helps traders determine how volume changes correspond to price movement, which can indicate the strength or weakness of the current trend.

The main idea of BV is that volume should confirm the price direction. If the price rises with increasing volume, this indicates a strong upward trend. Conversely, if the price falls with increasing volume, this suggests a strong downward trend.

The BV indicator is particularly useful for:

- Confirming the strength of the current trend
- Identifying potential trend reversals
- Detecting divergences between price and volume
- Determining accumulation and distribution levels

Calculation

The Balance Volume indicator calculation is based on comparing the closing price with the previous closing price and weighting the trading volume:

```
If Close > Previous Close:  
  BV = Previous BV + Volume  
If Close < Previous Close:  
  BV = Previous BV - Volume  
If Close = Previous Close:  
  BV = Previous BV
```

Where:

- Close - current closing price
- Previous Close - previous closing price
- Volume - current trading volume
- Previous BV - previous Balance Volume indicator value

Interpretation

- **BV rising with price increase** - confirmation of an upward trend, indicates strong buyer interest
- **BV falling with price decrease** - confirmation of a downward trend, indicates strong seller interest
- **BV rising with stable or falling price** - potential accumulation, may precede an upward reversal
- **BV falling with stable or rising price** - potential distribution, may precede a downward reversal

- **Divergence between BV and price** - warning of a possible trend reversal:
 - If price rises while BV falls, a quick downward reversal may be imminent
 - If price falls while BV rises, a quick upward reversal may be imminent

[image: indicator_balance_volume]

See Also

[OBV](#) [ADL](#) [Chaikin](#) [MoneyFlow](#) [ForceIndex](#)

Bear Power

Bear Power

Bear Power is part of Alexander Elder's Elder-ray system and shows how strong the sellers are compared with an exponential moving average (EMA). It measures how far the intraday lows fall below the average price and highlights moments when the bears lose control.

Use the [BearPower](#) class to access the indicator.

Description

The indicator is calculated as the difference between the bar low and the EMA value:

$\text{Bear Power} = \text{Low} - \text{EMA}$.

- Negative readings confirm selling pressure.
- Rising values toward zero or above zero indicate weakening bears and a possible bullish reversal.
- Deep troughs often precede rebounds, especially during panicky sell-offs.

Parameters

Bear Power inherits the configuration of [ExponentialMovingAverage](#):

- **Length** — EMA period.
- **Alpha** (optional) — smoothing coefficient if the EMA is configured this way.

Usage

- Look for reversals when Bear Power turns upward after an extreme low while the EMA starts to rise.
- Crossing the zero line may confirm a change in the prevailing trend.
- Combine Bear Power with [Bull Power](#) and the price EMA to build the full [Elder Ray](#) indicator.

[image: indicator_bear_power]

See also

[Bull Power](#) [Elder Ray](#) [ExponentialMovingAverage](#)

Bull Power

Bull Power

Bull Power is the bullish counterpart within the Elder-ray system. It measures how strongly buyers push prices above an exponential moving average (EMA) by comparing the bar high with the average price.

Use the [BullPower](#) class to work with this indicator.

Description

The indicator uses the formula:

$\text{Bull Power} = \text{High} - \text{EMA}$.

- Positive values confirm bullish pressure and support a rising trend.
- Falling values toward zero or below zero signal weakening bulls.
- Extreme peaks may precede corrections, especially when the EMA is pointing downward.

Parameters

Bull Power inherits its parameters from [ExponentialMovingAverage](#):

- **Length** — EMA period.
- **Alpha** (optional) — smoothing coefficient, when applicable.

Usage

- Rising Bull Power together with a rising EMA confirms trend strength.
- Price making new highs without higher Bull Power readings forms bearish divergence.
- Combine Bull and Bear Power with the price EMA to evaluate the full [Elder Ray](#) structure.

[image: indicator_bull_power]

See also

[Bear Power](#) [Elder Ray](#) [ExponentialMovingAverage](#)

Bollinger Bands (Bollinger)

Bollinger Bands

Bollinger Bands are an oscillating indicator used to measure market volatility. It allows assessing whether the price is high or low compared to the moving average. The central band corresponds to the simple moving average of the price. The upper and lower bands are levels at which the price can be considered high or low relative to the moving average.

To use the indicator, the [BollingerBands](#) class should be used.

Calculation

The following parameters with corresponding settings are used to calculate Bollinger Bands:

- type of standard deviation — usually double;
- period of Moving Average — at the trader's discretion.

Thus, the indicator is formed by three lines: central, upper, and lower, each with its formula:

Middle Line (ML) = Moving Average (SMA (Close, N)) Upper Band = ML + (D x Standard Deviation) Lower Band = ML - (D x Standard Deviation), where

D - the channel width set in settings, Standard Deviation (StdDev) - standard deviation, calculated by the formula: $\text{SQRT}(\text{Sum}(\text{Close}, n)^2, n)/n$, where Sum - the sum for n periods, n - calculation period, SQRT - square root, Close - closing price.

[image: IndicatorBollingerBands]

See Also

[CHV](#)

Bollinger %b (BBP)

BBP

Bollinger Percent B (BBP) is an indicator developed by John Bollinger as a complement to the Bollinger Bands indicator. BBP shows the price location relative to the upper and lower Bollinger Bands.

To use the indicator, you need to use the [BollingerPercentB](#) class.

Description

The Bollinger Percent B indicator determines the price position relative to the upper and lower Bollinger Bands as a percentage value from 0 to 1 (or from 0% to 100%). This allows for a more precise determination of the price position in the context of Bollinger Bands:

- A value of 1 (or 100%) means the price is at the upper Bollinger Band.
- A value of 0 (or 0%) means the price is at the lower Bollinger Band.
- A value of 0.5 (or 50%) means the price is at the middle Bollinger Band (SMA).

BBP can also take values outside the 0-1 range:

- Values above 1 indicate that the price is above the upper Bollinger Band.
- Values below 0 indicate that the price is below the lower Bollinger Band.

Parameters

The indicator has the following parameters:

- **Length** - SMA calculation period (default value: 20)
- **StdDevMultiplier** - standard deviation multiplier for calculating Bollinger Bands (default value: 2)

Calculation

The Bollinger Percent B calculation is based on the formula:

$$BBP = (\text{Price} - \text{Lower Bollinger Band}) / (\text{Upper Bollinger Band} - \text{Lower Bollinger Band})$$

Where:

- Price - current price (usually closing price)
- Lower Bollinger Band = $\text{SMA} - (\text{StdDevMultiplier} * \text{Standard Deviation})$
- Upper Bollinger Band = $\text{SMA} + (\text{StdDevMultiplier} * \text{Standard Deviation})$
- SMA - simple moving average over the Length period
- Standard Deviation - price standard deviation over the Length period

Usage

Bollinger Percent B can be used in various ways:

1. **Identifying Overbought/Oversold Conditions:**
 - Values above 1 indicate an overbought market

- Values below 0 indicate an oversold market
2. **Reversal Signals:**
 - When BBP returns to the 0-1 range after going outside it
 - Divergences between BBP and price
 3. **Trend Determination:**
 - BBP values consistently above 0.5 indicate an upward trend
 - BBP values consistently below 0.5 indicate a downward trend
 4. **Finding Hidden Support and Resistance Levels:**
 - Levels 0.8 and 0.2 are often used as additional support and resistance levels

[image: indicator_bollinger_percent_b]

See Also

[BollingerBands StdDev RSI](#)

Center of Gravity Oscillator (CGO)

CGO

Center of Gravity Oscillator (CGO) is a technical indicator developed by John Ehlers, based on the concept of center of gravity in physics and applied to analyzing price movement in the market.

To use the indicator, you need to use the [CenterOfGravityOscillator](#) class.

Description

The Center of Gravity Oscillator (CGO) is a leading indicator that attempts to identify market reversal points by treating the price series as a physical system and determining its "center of gravity". The indicator calculates where the "equilibrium" is in current price movements and uses this information to forecast future trend direction changes.

CGO is particularly useful for:

- Identifying potential reversal points before they appear on the price chart
- Revealing the strength and weakness of the current trend
- Detecting hidden divergences between price and the indicator
- Creating trading systems based on leading signals

Parameters

The indicator has the following parameters:

- **Length** - calculation period (default value: 10)

Calculation

The Center of Gravity Oscillator (CGO) calculation is based on the formula:

```
CGO = - Sum(Price(i) * (i + 1)) / Sum(Price(i))
```

Where:

- *i* - price value index in the period from 0 to (Length-1)
- Price(*i*) - price (usually closing price) for the corresponding index *i*
- Sum - sum of all values in the Length period

In this formula, each price is weighted by its position in the time series and then normalized by the total sum of prices. The minus sign before the formula is added to make the indicator rise when the price rises, making it more intuitive.

Interpretation

- **Zero Line Crossing:** When CGO crosses the zero line from bottom to top, this can be viewed as a bullish signal. Crossing from top to bottom may indicate a bearish signal.
- **Indicator Extremes:** When CGO reaches extremes (maximums or minimums), this can indicate a potential trend reversal.
- **Divergences:**

- **Bullish Divergence:** when the price forms a new low, but CGO does not confirm it, forming a higher low.
- **Bearish Divergence:** when the price reaches a new high, but CGO forms a lower high.
- **Indicator Movement:** Rapid CGO movement in one direction can indicate the beginning of a new trend. If the indicator moves slowly or oscillates around the zero line, this may indicate market consolidation.

Since CGO is a leading indicator, its signals often appear before corresponding changes on the price chart, giving traders an advantage in making trading decisions.

[image: indicator_center_of_gravity_oscillator]

See Also

[SineWave](#) [HarmonicOscillator](#) [FisherTransform](#) [RVI](#)

Chaikin Money Flow (CMF)

CMF

Chaikin Money Flow (CMF) is a technical indicator developed by Mark Chaikin that measures the strength of money flow (accumulation and distribution) in the market over a specific period.

To use the indicator, you need to use the [ChaikinMoneyFlow](#) class.

Description

Chaikin Money Flow (CMF) expands on the concept of the Accumulation/Distribution Line (A/D Line), focusing on a specific time period. The indicator measures money flow volume expressed as a percentage of total volume over the specified period.

CMF helps traders:

- Determine the strength of buying and selling pressure
- Identify accumulation (buying) and distribution (selling) trends
- Detect divergences between price movement and money flow
- Confirm the current trend or its weakness

The key idea of CMF is that in a strong upward trend, the closing price should be closer to the period's high, while in a strong downward trend, it should be closer to the period's low.

Parameters

The indicator has the following parameters:

- **Length** - calculation period (standard value: 20-21 days)

Calculation

CMF calculation involves the following steps:

1. Calculate Money Flow Multiplier for each period:

$$\text{Money Flow Multiplier} = ((\text{Close} - \text{Low}) - (\text{High} - \text{Close})) / (\text{High} - \text{Low})$$

If $(\text{High} - \text{Low}) = 0$, then Money Flow Multiplier = 0.

2. Calculate Money Flow Volume for the period:

$$\text{Money Flow Volume} = \text{Money Flow Multiplier} * \text{Volume}$$

3. Calculate Chaikin Money Flow:

$$\text{CMF} = \text{Sum}(\text{Money Flow Volume over Length period}) / \text{Sum}(\text{Volume over Length period})$$

Interpretation

CMF oscillates around the zero line and is typically within the range of -1 to +1:

- **Positive CMF values** (above zero):
 - Indicate buyer pressure (accumulation)

- The higher the value, the stronger the buyer pressure
- Particularly significant if sustained for a prolonged period
- **Negative CMF values** (below zero):
 - Indicate seller pressure (distribution)
 - The lower the value, the stronger the seller pressure
 - Prolonged stay in the negative zone confirms a downward trend
- **Zero Line Crossing:**
 - Crossing from bottom to top may indicate the start of an upward trend
 - Crossing from top to bottom may signal the beginning of a downward trend
- **Divergences:**
 - Bullish Divergence: price declines while CMF rises (potential upward reversal)
 - Bearish Divergence: price rises while CMF falls (potential downward reversal)
- **Extreme Levels:**
 - Values above +0.25 may indicate strong accumulation
 - Values below -0.25 may indicate strong distribution

[image: indicator_chaikin_money_flow]

See Also

[ADL](#) [OBV](#) [ForceIndex](#) [MFI](#)

Chaikin Volatility (Chaikin's Volatility)

CHV

Chaikin Volatility (CHV) indicates the difference between the maximum of purchases and the minimum of sales over a period of time. The indicator allows for qualitative analysis of price changes and the range width between the price maximum and minimum. CHV does not consider price gaps in its calculation, which can be considered a drawback to some extent.

To use the indicator, the [ChaikinVolatility](#) class should be used.

Calculation

The calculation of the Chaikin Volatility indicator begins with determining the spread – the difference between the maximum and minimum of the current bar. The obtained result is smoothed with an exponential moving average over the corresponding period. Volatility is calculated by the formula:

$$\text{CHV} = (\text{EMA}(\text{H-L}(i), n) - \text{EMA}(\text{H-L}(i-n), n)) / \text{EMA}(\text{H-L}(i-n), n) \times 100.$$

In this case, $\text{H-L}(i)$ represents the price difference on the current bar, and $\text{H-L}(i-n)$ – the price difference on a bar that occurred n periods earlier. Thus, we have the ability to visually see how the price has changed over time.

Main parameters of the indicator:

- **ROCPeriod** – the period number relative to which the calculation will be performed. Initially set to 5.
- **SmoothPeriod** – the period of the moving average. By default, it is set to 32.

[image: IndicatorChaikinVolatility]

See Also

[CMO](#)

Chande Kroll Stop (CKS)

CKS

Chande Kroll Stop (CKS) is an indicator for determining stop-loss levels, developed by Tushar Chande and Stanley Kroll, which adapts to market volatility and helps traders set position exit points.

To use the indicator, you need to use the [ChandeKrollStop](#) class.

Description

The Chande Kroll Stop indicator was developed as a dynamic tool for setting stop-loss levels that responds to changes in market volatility and trend. It consists of two lines: an upper stop line (for short positions) and a lower stop line (for long positions).

The main advantage of CKS lies in its ability to adapt to current market conditions. During periods of high volatility, stop lines are positioned further from the price, helping to avoid premature position closure due to market noise. During periods of low volatility, stop lines move closer to the price, providing tighter trend following.

CKS is particularly useful for:

- Determining stop-loss levels for long and short positions
- Following the trend with adaptive risk control
- Identifying potential trend reversal points
- Creating mechanical trading systems with clear exit rules

Parameters

The indicator has the following parameters:

- **Period** - main period for calculating extremums (default value: 10)
- **Multiplier** - multiplier for ATR, determining the distance from extremums (default value: 1.5)
- **StopPeriod** - period for calculating stop levels (default value: 20)

Calculation

Chande Kroll Stop calculation involves the following steps:

1. Determining high and low extremums over the Period:

```
HighestHigh = Highest High value over Period  
LowestLow = Lowest Low value over Period
```

2. Calculating Average True Range (ATR) over Period:

```
ATR = Average TR value over Period
```

3. Calculating upper and lower bands:

```
Upper Band = HighestHigh - (Multiplier * ATR)  
Lower Band = LowestLow + (Multiplier * ATR)
```

4. Determining final stop lines based on StopPeriod:

Upper Stop = Highest value of upper band over StopPeriod
Lower Stop = Lowest value of lower band over StopPeriod

Interpretation

- **Upper Stop** is used for short positions. If the closing price exceeds the upper stop, this can be considered a signal to close a short position or open a long position.
- **Lower Stop** is used for long positions. If the closing price falls below the lower stop, this can be considered a signal to close a long position or open a short position.
- **Price crossing stop lines** may indicate a potential trend reversal or the beginning of a new momentum.
- **Abrupt changes in stop lines** can occur with significant market volatility changes.
- **Use with other indicators:** CKS works best in combination with other trend and momentum indicators that help determine market entry direction.

[image: indicator_chande_kroll_stop]

See Also

[ATR](#) [ParabolicSAR](#) [DonchianChannels](#)

Chande Momentum Oscillator (CMO)

CMO

Chande Momentum Oscillator (CMO) is a modification of the Momentum indicator. The inventor of CMO is trader Tushar Chande.

To use the indicator, you must use the [ChandeMomentumOscillator](#) class.

[image: IndicatorChandeMomentumOscillator]

Recommended content

[CCI](#)

Choppiness Index (CHOP)

CHOP

Choppiness Index (CHOP) is an indicator designed to determine whether the market is in a sideways movement (within a range) or in a trending state.

To use the indicator, you need to use the [ChoppinessIndex](#) class.

Description

The Choppiness Index (CHOP) was created to quantitatively assess volatility and determine the nature of market movement. Unlike many other indicators, CHOP is not meant to identify trend direction or generate buy or sell signals. Instead, it helps traders determine whether the market is in consolidation (sideways movement) or in a directional trend.

The CHOP indicator oscillates between 0 and 100:

- Values closer to 100 indicate strong consolidation (high "choppiness")
- Values closer to 0 indicate a strong directional trend (low "choppiness")

CHOP is particularly useful for:

- Determining an appropriate trading strategy based on market character
- Identifying transitions from sideways movement to trend and vice versa
- Confirming or refuting signals from other indicators
- Avoiding false signals during consolidation

Parameters

The indicator has the following parameters:

- **Length** - calculation period (default value: 14)

Calculation

Choppiness Index calculation involves the following steps:

1. Calculate the sum of True Ranges over the selected period:

```
Sum of TR = Sum(TR(i)) for i from 1 to Length
```

2. Calculate the highest High and lowest Low over the selected period:

```
Highest High = maximum High value over Length period  
Lowest Low = minimum Low value over Length period
```

3. Calculate the CHOP index:

```
CHOP = 100 * LOG10(Sum TR / (Highest High - Lowest Low)) / LOG10(Length)
```

Where:

- TR - true range for each candle
- Length - selected period

- LOG10 - decimal logarithm

Interpretation

- **High CHOP values (above 60-70)** indicate that the market is in a sideways movement (consolidation). During this period, it's better to avoid trend strategies and consider range trading strategies.
- **Low CHOP values (below 30-40)** indicate a strong directional trend. This is a good time to use trend strategies and follow price movement.
- **Transitions between high and low values** can indicate a change in market character. A CHOP drop from high values may signal the beginning of a new trend. A CHOP rise from low values may warn of trend exhaustion and transition to consolidation.
- **Setting threshold levels:** Typically, the following threshold levels are used:
 - Above 60-70: High "choppiness" (sideways movement)
 - 30-60: Moderate "choppiness" (transitional state)
 - Below 30: Low "choppiness" (strong trend)

[image: indicator_choppiness_index]

See Also

[ATR](#) [ADX](#) [VHF](#) [TrueRange](#)

Commodity Channel Index (CCI)

CCI

Commodity Channel Index (CCI) is a technical indicator based on the analysis of the current price deviation from its average value and the statistically average absolute value.

To use the indicator, you must use the [CommodityChannelIndex](#) class.

[image: IndicatorCommodityChannelIndex]

Recommended content

[DPO](#)

Composite Momentum (CM)

CM

Composite Momentum (CM) is an indicator that combines multiple methods of measuring price momentum to obtain more reliable signals about trend strength and direction.

To use the indicator, you need to use the [CompositeMomentum](#) class.

Description

The Composite Momentum (CM) indicator is a comprehensive tool that integrates various aspects of price movement, including price change rate, relative strength, and other momentum components. Through this combined approach, CM provides a more complete picture of current market momentum compared to traditional one-dimensional momentum indicators.

CM is effective for:

- Determining the strength of the current trend
- Identifying potential reversal points
- Detecting divergences between price and momentum
- Filtering out false signals from other indicators

Composite Momentum is particularly useful in volatile markets where traditional momentum indicators may generate numerous false signals.

Calculation

Composite Momentum calculation involves several stages and components:

1. Calculating momentum components:
 - Price change relative to previous periods
 - Ratio between recent highs and lows
 - Volume analysis accompanying price movement
2. Normalizing each component to bring them to comparable scales.
3. Weighted summation of components to obtain the final CM value.

The final CM value is an oscillator that can fluctuate in both positive and negative areas:

- Positive values indicate upward momentum
- Negative values indicate downward momentum
- The magnitude of the value (absolute value) indicates momentum strength

Interpretation

- **Zero Line Crossing:**
 - Transition from negative to positive zone can be viewed as a bullish signal
 - Transition from positive to negative zone can be viewed as a bearish signal
- **Extreme Values:**
 - Very high positive values may indicate market overbought conditions

- Very low negative values may indicate market oversold conditions
- **Divergences:**
 - Bullish Divergence: price forms a new low, while CM forms a higher low
 - Bearish Divergence: price forms a new high, while CM forms a lower high
- **Trend Confirmation:**
 - Consistently positive CM values confirm the strength of an upward trend
 - Consistently negative CM values confirm the strength of a downward trend
- **Momentum Loss:**
 - Decrease in the absolute value of CM in the trend direction may signal momentum loss and a potential reversal

[image: indicator_composite_momentum]

See Also

[Momentum](#) [ROC](#) [RSI](#) [MACD](#)

Connors RSI (CRSI)

CRSI

Connors RSI (CRSI) is a comprehensive technical indicator developed by Larry Connors that combines three components to measure market overbought and oversold conditions.

To use the indicator, you need to use the [ConnorsRSI](#) class.

Description

Connors RSI is an advanced version of the traditional Relative Strength Index (RSI), which adds two additional components to provide more accurate overbought and oversold signals.

Unlike standard RSI, which only considers price change, Connors RSI also takes into account the streak (series of consecutive price movements in one direction) and rate of change (ROC), making it more sensitive to short-term changes and more reliable in identifying extreme market conditions.

CRSI is particularly useful for:

- Identifying short-term entry and exit opportunities
- Determining extreme overbought and oversold levels
- Creating trading systems based on mean reversion
- Filtering signals from other indicators

Parameters

The indicator has the following parameters:

- **RSIPeriod** - period for calculating the RSI component (default value: 3)
- **StreakRSIPeriod** - period for calculating the streak RSI component (default value: 2)
- **ROCRSIPeriod** - period for calculating the rate of change RSI component (default value: 100)

Calculation

Connors RSI calculation involves three components that are then averaged to obtain the final value:

1. **Price RSI Component** - standard RSI calculated over a short period (usually 3 days):

$$RSI = 100 - (100 / (1 + RS))$$

where $RS = \text{Average Positive Change} / \text{Average Negative Change}$

2. **Streak RSI Component:**

- First, calculate the streak (number of consecutive days of price rise or fall)
- Then apply RSI to this streak using the **StreakRSIPeriod**

3. **Rate of Change RSI Component (ROC RSI):**

- Calculate the Percentile Rank of the current ROC over the **ROCRSIPeriod**
- Scale the Percentile Rank from 0 to 100

4. **Final Connors RSI Value:**

$$CRSI = (RSI + \text{StreakRSI} + \text{ROCRSI}) / 3$$

Interpretation

Connors RSI oscillates between 0 and 100, similar to standard RSI:

- **Extremely high values (above 90)** indicate strong overbought conditions. This may be a signal to sell or take a short position.
- **Extremely low values (below 10)** indicate strong oversold conditions. This may be a signal to buy or close a short position.
- **Standard Levels:**
 - Above 70-80: overbought
 - Below 20-30: oversold
 - 40-60: neutral zone
- **Divergences:**
 - Bullish Divergence: price forms a new low, while CRSI forms a higher low
 - Bearish Divergence: price forms a new high, while CRSI forms a lower high

Connors RSI works best on charts with timeframes from daily to weekly and in mean reversion-oriented trading strategies.

[image: indicator_connors_rsi]

See Also

[RSI](#) [RMI](#) [LRSI](#)

Constance Brown Composite Index (CBCI)

CBCI

Constance Brown Composite Index (CBCI) is an indicator developed by Constance Brown that combines elements from various technical indicators to create a comprehensive market analysis tool.

To use the indicator, you need to use the [ConstanceBrownCompositeIndex](#) class.

Description

The Constance Brown Composite Index (CBCI) was created to merge the strengths of several indicators into one comprehensive tool. It incorporates elements of the stochastic oscillator, RSI, and other oscillators to provide more accurate signals about potential market reversals and trend movements.

CBCI is designed for:

- Identifying potential trend reversal points
- Determining overbought and oversold levels
- Detecting hidden divergences
- Confirming the strength of the current trend

The indicator works well across various timeframes and market types, including stocks, forex, and commodity markets.

Parameters

The indicator has the following parameters:

- **Length** - main calculation period for the index (default value: 14)
- **StochasticKPeriod** - period for calculating the stochastic oscillator %K (default value: 5)
- **StochasticDPeriod** - period for calculating the stochastic oscillator %D (default value: 3)

Calculation

CBCI calculation involves the following steps:

1. Calculate RSI over the Length period:

```
RSI = 100 - (100 / (1 + RS))
RS = Average Gain / Average Loss
```

2. Calculate the stochastic oscillator:

```
%K = ((Close - Lowest Low) / (Highest High - Lowest Low)) * 100
%D = SMA(%K, StochasticDPeriod)
```

3. Combine RSI and stochastic oscillator:

```
CBCI = (RSI + %K + %D) / 3
```

This combined index can then be smoothed to reduce noise.

Interpretation

- **Overbought and Oversold Levels:**
 - Values above 80 may indicate market overbought conditions
 - Values below 20 may indicate market oversold conditions
- **Centerline Crossovers:**
 - Crossing from below to above the 50 line can be viewed as a bullish signal
 - Crossing from above to below the 50 line can be viewed as a bearish signal
- **Divergences:**
 - Classic divergences: when price and CBI move in opposite directions
 - Hidden divergences: when price and CBI create different types of highs or lows
- **Trend Movement:**
 - If CBI consistently remains above 50, it may indicate uptrend strength
 - If CBI consistently remains below 50, it may indicate downtrend strength

[image: indicator_constance_brown_composite_index]

See Also

[RSI](#) [StochasticOscillator](#) [StochasticK](#) [CCI](#)

DeMarker

The **DeMarker (DeM)** indicator evaluates buying and selling pressure by comparing the extrema of the current bar with the previous one. It highlights overbought and oversold zones and helps spot potential turning points.

Use the [DeMarker](#) class to work with this indicator.

Calculation

1. For each bar compute intermediate values: $\text{DeMax} = \max(\text{High} - \text{PreviousHigh}, 0)$ $\text{DeMin} = \max(\text{PreviousLow} - \text{Low}, 0)$
2. Smooth DeMax and DeMin with a moving average of length **Length**.
3. Calculate the final value: $\text{DeMarker} = \text{SMA}(\text{DeMax}, \text{Length}) / (\text{SMA}(\text{DeMax}, \text{Length}) + \text{SMA}(\text{DeMin}, \text{Length}))$.

The output is normalised between 0 and 1.

Parameters

- **Length** — smoothing period controlling the responsiveness of the indicator.

Interpretation

- **Above 0.7** — overbought conditions, potential downward correction.
- **Below 0.3** — oversold conditions, potential upward reversal.
- **Divergence** between price and the indicator warns about a trend change.

DeMarker can be used for counter-trend entries as well as confirming signals from momentum oscillators.

[image: indicator_demarker]

See also

[RSI Stochastic Oscillator Momentum](#)

Demand Index (DI)

DI

Demand Index (DI) is a technical indicator developed by James Sibbett that analyzes the relationship between price and volume to assess demand strength and buyer pressure in the market.

To use the indicator, you need to use the [DemandIndex](#) class.

Description

The Demand Index (DI) is a comprehensive volume indicator that evaluates the relationship between price and volume to determine how strong buyer pressure (demand) is compared to seller pressure. The indicator is based on the assumption that the ratio of price change to volume change allows for a more accurate assessment of market demand than simply observing price or volume individually.

DI aims to identify the following market situations:

- Strong demand (buyer pressure)
- Weak demand (seller pressure)
- Imbalance between price and volume (potential reversal points)
- Confirmation or refutation of the current trend

Parameters

The indicator has the following parameters:

- **Length** - calculation period (default value: 13)

Calculation

The Demand Index calculation is quite complex and involves several stages:

1. Calculating the price component based on price change:

```
Price Component = ((High + Low + Close) / 3) - ((Previous High + Previous Low + Previous Close) / 3)
```

2. Calculating the volume component, taking into account relative volume change.
3. Computing demand as the ratio of price and volume components:

```
Raw Demand = Price Component / Volume Component
```

4. Smoothing the obtained values to reduce noise:

```
Smoothed Demand = EMA(Raw Demand, Length)
```

5. Normalizing the result to obtain the final index:

```
Demand Index = 100 * Normalized(Smoothed Demand)
```

Interpretation

The Demand Index can be interpreted in various ways:

1. **Extreme Levels:**

- High positive values indicate strong demand (buyer pressure)
- High negative values indicate weak demand (seller pressure)

2. **Zero Line Crossovers:**

- Crossing from bottom to top can be viewed as a bullish signal
- Crossing from top to bottom can be viewed as a bearish signal

3. **Divergences:**

- Bullish Divergence: price forms a new low, but DI forms a higher low
- Bearish Divergence: price forms a new high, but DI forms a lower high

4. **DI Trends:**

- Sustained positive DI values confirm an upward trend
- Sustained negative DI values confirm a downward trend

5. **Extreme Values:**

- Very high or very low values may indicate market overbought or oversold conditions

Using the Demand Index is most effective when combined with other indicators and analysis methods to filter out false signals.

[image: indicator_demand_index]

See Also

[OBV](#) [ADL](#) [ChaikinMoneyFlow](#) [BalanceOfPower](#)

Detrended Price Oscillator (DPO)

DPO

Detrended Price Oscillator (DPO) is an oscillator that eliminates price trends in an attempt to estimate the duration of price cycles from peak to peak or trough to trough. Unlike other oscillators, such as stochastic convergence or moving average convergence divergence (MACD), DPO is not a momentum indicator. It highlights peaks and troughs in price, which are used to estimate entry and exit points.

To use the indicator, the [DetrendedPriceOscillator](#) class should be used.

The Detrended Price Oscillator is calculated by subtracting a simple moving average (SMA) from the current price value. The length of the moving average is determined by the user.

[image: IndicatorDetrendedPriceOscillator]

See Also

[DMI](#)

Detrended Synthetic Price (DSP)

DSP

Detrended Synthetic Price (DSP) is a technical indicator that removes the overall trend from a price series, allowing traders to focus on short-term price fluctuations.

To use the indicator, you need to use the [DetrendedSyntheticPrice](#) class.

Description

The DSP indicator was developed to eliminate the long-term trend from a price chart, enabling traders to more clearly see short-term cycles and oscillations. It is particularly useful for identifying short-term trading opportunities that may be hidden by the dominant trend.

The main idea of DSP is that by removing the trend from the price series, it becomes easier to identify the cyclical components of price movement. This makes the indicator especially valuable for traders specializing in short-term trading and using cyclical patterns.

DSP is useful for:

- Identifying short-term market cycles
- Determining potential reversal points
- Detecting divergences with price
- Creating trading systems based on the cyclical nature of markets

Parameters

The indicator has the following parameters:

- **Length** - calculation period (default value: 10-20 periods)

Calculation

Calculating the Detrended Synthetic Price involves the following steps:

1. Calculate the moving average of price over the specified period:

```
MA = SMA(Price, Length)
```

2. Determine the offset for synthetic price calculation:

```
Offset = (Length / 2) + 1
```

3. Calculate synthetic price by subtracting the shifted moving average from the current price:

```
DSP = Price - MA[shifted (Length/2) + 1 periods back]
```

Where:

- Price - current price (usually closing price)
- MA - simple moving average
- Length - selected calculation period

Interpretation

The DSP indicator oscillates around the zero line and can be interpreted as follows:

1. **Zero Line Crossovers:**
 - When DSP crosses the zero line from bottom to top, it can be viewed as a bullish signal
 - When DSP crosses the zero line from top to bottom, it can be viewed as a bearish signal
2. **Indicator Extremes:**
 - When DSP reaches extremely high values, it may indicate market overbought conditions
 - When DSP reaches extremely low values, it may indicate market oversold conditions
3. **Cyclical Analysis:**
 - Regular DSP oscillations can be used to determine market cycle periodicity
 - Changes in oscillation amplitude may indicate shifts in market dynamics
4. **Divergences:**
 - Bullish Divergence: price forms a new low, while DSP forms a higher low
 - Bearish Divergence: price forms a new high, while DSP forms a lower high
5. **Pattern Formation:**
 - Technical patterns (head and shoulders, double bottom, etc.) can form on the DSP chart, potentially providing additional trading signals

[image: indicator_detrended_synthetic_price]

See Also

[DetrendedPriceOscillator](#) [CenterOfGravityOscillator](#) [SineWave](#)

Directional Movement Index (DMI)

DMI

Directional Movement Index (DMI) indicator is a technical tool for determining the start and direction of a trend. It is the first version of the famous ADX.

To use the indicator, you must use the [DirectionalIndex](#) class.

[image: IndicatorDirectionalIndex]

Recommended content

[DEMA](#)

Disparity Index (DPI)

DPI

Disparity Index (DPI) is a technical indicator that measures the relative deviation of the current price from a moving average over a specified period, expressed as a percentage.

To use the indicator, you need to use the [DisparityIndex](#) class.

Description

The Disparity Index (DPI) is designed to measure the degree of price deviation from its moving average. This indicator helps determine how far the price is "stretched" relative to its average value and can be used to identify potential overbought or oversold conditions.

DPI is based on the assumption that price tends to return to its mean value after a significant deviation. The larger the deviation, the higher the probability of subsequent price movement in the opposite direction, bringing it closer to the average.

The Disparity Index is useful for:

- Identifying extreme price deviations from its mean
- Detecting potential reversal points
- Measuring the strength of the current trend
- Creating trading strategies based on mean reversion

Parameters

The indicator has the following parameters:

- **Length** - period for calculating the moving average (default value: 14)

Calculation

The Disparity Index calculation formula is quite simple:

$$\text{DPI} = ((\text{Price} / \text{MA}) - 1) * 100$$

Where:

- Price - current price (usually closing price)
- MA - moving average of price over the Length period
- Result is expressed as a percentage

Positive DPI values indicate that the price is above its moving average, while negative values indicate that the price is below its moving average.

Interpretation

The Disparity Index can be interpreted as follows:

1. Extreme Levels:

- High positive values (e.g., above +10%) may indicate market overbought conditions
- High negative values (e.g., below -10%) may indicate market oversold conditions

2. Zero Line Crossovers:

- Crossing from bottom to top (from negative to positive values) indicates that the price has crossed its moving average from bottom to top, which can be viewed as a bullish signal
- Crossing from top to bottom (from positive to negative values) indicates that the price has crossed its moving average from top to bottom, which can be viewed as a bearish signal

3. Divergences:

- Bullish Divergence: price reaches a new low, but DPI forms a higher low
- Bearish Divergence: price reaches a new high, but DPI forms a lower high

4. Trend Analysis:

- Consistently positive DPI values indicate a strong upward trend
- Consistently negative DPI values indicate a strong downward trend
- Oscillations around zero may indicate a sideways trend or consolidation

5. Mean Reversion Strategies:

- Extreme DPI values can be used to open positions against the current price movement, expecting a return to the mean

[image: indicator_disparity_index]

See Also

[SMA](#) [RSI](#) [StochasticOscillator](#) [BollingerBands](#)

Donchian Channels (DC)

DC

Donchian Channels (DC) is a technical indicator developed by trader Richard Donchian, consisting of an upper and lower band (channel boundaries) based on the maximum and minimum price values over a specific period.

To use the indicator, you need to use the [DonchianChannels](#) class.

Description

Donchian Channels are a simple yet effective volatility and trend indicator. The indicator consists of three lines:

- Upper line: highest high over the selected period
- Lower line: lowest low over the selected period
- Middle line: average value between the upper and lower lines

This indicator was first used by Richard Donchian in his 4-week channel rule, according to which a buy signal occurs when the price exceeds the highest high of 4 weeks, and a sell signal occurs when the price falls below the lowest low of 4 weeks.

Donchian Channels are useful for:

- Identifying market volatility
- Determining support and resistance levels
- Generating breakout signals
- Defining the current trading range

Parameters

The indicator has the following parameters:

- **Length** - calculation period (default value: 20)

Calculation

Donchian Channels calculation is quite simple:

1. Upper channel line:

```
Upper = Highest High over Length period
```

2. Lower channel line:

```
Lower = Lowest Low over Length period
```

3. Middle channel line:

```
Middle = (Upper + Lower) / 2
```

Interpretation

Donchian Channels can be used in various ways:

1. **Breakout Strategies:**

- Breaking above the upper channel line can be viewed as a buy signal
- Breaking below the lower channel line can be viewed as a sell signal

2. **Trend Determination:**

- If the price is in the upper half of the channel (above the middle line), an upward trend can be inferred
- If the price is in the lower half of the channel (below the middle line), a downward trend can be inferred

3. **Support and Resistance Levels:**

- The upper channel line can serve as a resistance level
- The lower channel line can serve as a support level

4. **Volatility Measurement:**

- Channel width (difference between upper and lower lines) indicates market volatility
- Channel expansion indicates increased volatility
- Channel contraction indicates decreased volatility

5. **Counter-Trend Strategies:**

- Some traders use opposite signals, expecting the price to return to the middle line after reaching the channel edges

[image: indicator_donchian_channels]

See Also

[BollingerBands](#) [KeltnerChannels](#) [Highest](#) [Lowest](#)

Double Exponential Moving Average (DEMA)

DEMA

Double Exponential Moving Average (DEMA) uses multiple (double) Exponential Moving Averages (EMA) to eliminate lags when forecasting prices.

To use the indicator, you must use the [DoubleExponentialMovingAverage](#) class.

[image: IndicatorDoubleExponentialMovingAverage]

Recommended content

[Envelope](#)

Dynamic Zones RSI (DZRSI)

DZRSI

Dynamic Zones RSI (DZRSI) is a modification of the classic Relative Strength Index (RSI) that uses dynamically changing overbought and oversold levels instead of static ones.

To use the indicator, you need to use the [DynamicZonesRSI](#) class.

Description

The Dynamic Zones RSI (DZRSI) is based on the traditional RSI, but with an important improvement: instead of using fixed overbought and oversold levels (usually 70 and 30), DZRSI adapts these levels to current market conditions.

The main idea of DZRSI is that different market conditions require different thresholds for determining overbought and oversold states. In a strong upward trend, RSI can remain above the traditional overbought level of 70 for an extended period without providing accurate entry or exit signals. Similarly, in a strong downward trend, RSI can stay below the oversold level of 30 for a long time.

DZRSI solves this problem by dynamically adjusting these levels based on the historical behavior of the RSI itself, making the indicator more adaptive to various market regimes.

Parameters

The indicator has the following parameters:

- **Length** - period for calculating the base RSI (default value: 14)
- **OverboughtLevel** - initial overbought level (default value: 70)
- **OversoldLevel** - initial oversold level (default value: 30)

Calculation

DZRSI calculation involves several steps:

1. Calculate standard RSI over the specified Length period:

```
RSI = 100 - (100 / (1 + RS))  
RS = Average Positive Change / Average Negative Change
```

2. Determine the RSI oscillation range over a specific historical period.
3. Adapt overbought and oversold levels based on this range:

```
Dynamic Overbought Level = Base Overbought Level + Adjustment Based on Historical Data  
Dynamic Oversold Level = Base Oversold Level - Adjustment Based on Historical Data
```

4. Adjust dynamic zones based on the strength of the current trend.

Interpretation

DZRSI is interpreted similarly to traditional RSI, but with consideration of dynamic zones:

1. **Overbought and Oversold Signals:**
 - When DZRSI rises above the current dynamic overbought level, it may indicate market overbought conditions

- When DZRSI falls below the current dynamic oversold level, it may indicate market oversold conditions
2. **Reversal Signals:**
 - Reversal from the dynamic overbought level downward can be viewed as a sell signal
 - Reversal from the dynamic oversold level upward can be viewed as a buy signal
 3. **Divergences:**
 - Bullish Divergence: price forms a new low, while DZRSI forms a higher low
 - Bearish Divergence: price forms a new high, while DZRSI forms a lower high
 4. **Trend Analysis:**
 - In an upward trend, the dynamic oversold level may be higher than the traditional 30
 - In a downward trend, the dynamic overbought level may be lower than the traditional 70
 5. **Centerline (50) Crossovers:**
 - Crossing from bottom to top can be viewed as a bullish signal
 - Crossing from top to bottom can be viewed as a bearish signal

[image: indicator_dynamic_zones_rsi]

See Also

[RSI ConnorsRSI LRSI](#)

Ease of Movement (EMV)

EMV

Ease of Movement (EMV) is a technical indicator developed by Richard Arms that correlates price change with volume to assess the ease with which price moves up or down.

To use the indicator, you need to use the [EaseOfMovement](#) class.

Description

The Ease of Movement (EMV) indicator was created to measure the relationship between price movement and volume. The main concept of the indicator is that in an upward trend, price should move up easily with little volume, while in a downward trend, price should move down easily also with little volume.

EMV combines information about price range, price change, and volume to create a measure of price movement "ease". Positive EMV values indicate that price is rising with relative ease, while negative values indicate that price is falling with relative ease.

The indicator is particularly useful for:

- Confirming the strength or weakness of the current trend
- Identifying potential reversal points
- Detecting divergences with price
- Evaluating the "quality" of price movement considering volume

Parameters

The indicator has the following parameters:

- **Length** - smoothing period (default value: 14)

Calculation

Ease of Movement indicator calculation involves the following steps:

1. Calculate Midpoint Move:

```
Midpoint = (High + Low) / 2
Midpoint Move = Midpoint[current] - Midpoint[previous]
```

2. Calculate Box Ratio (volume-distance coefficient):

```
Box Ratio = Volume / (High - Low)
```

3. Calculate single-period EMV:

```
1-Period EMV = Midpoint Move / Box Ratio
```

4. Smooth to obtain final EMV:

```
EMV = SMA(1-Period EMV, Length)
```

Where:

- High - candle's highest price

- Low - candle's lowest price
- Volume - trading volume
- SMA - simple moving average

Interpretation

The EMV indicator can be interpreted as follows:

1. Zero Line Crossovers:

- Crossing from bottom to top (from negative to positive values) can be viewed as a bullish signal, indicating that price is beginning to move up with ease
- Crossing from top to bottom (from positive to negative values) can be viewed as a bearish signal, indicating that price is beginning to move down with ease

2. Extreme Values:

- High positive values indicate that price is moving up very easily (with little volume)
- High negative values indicate that price is moving down very easily (with little volume)

3. Divergences:

- Bullish Divergence: price forms a new low, while EMV forms a higher low (may indicate a potential upward reversal)
- Bearish Divergence: price forms a new high, while EMV forms a lower high (may indicate a potential downward reversal)

4. EMV Trends:

- Sustained positive values confirm an upward trend
- Sustained negative values confirm a downward trend
- Oscillations around zero may indicate a sideways trend or consolidation

5. Volume Analysis:

- If price rises with a positive EMV, this confirms the strength of the upward movement
- If price falls with a negative EMV, this confirms the strength of the downward movement
- If price rises with a negative EMV or falls with a positive EMV, this may indicate the instability of the current movement

[image: indicator_ease_of_movement]

See Also

[ForceIndex](#) [BalanceOfPower](#) [ADL](#) [OBV](#)

Ehlers Fisher Transform (EFT)

EFT

Ehlers Fisher Transform (EFT) is a technical indicator developed by John Ehlers that uses the Fisher statistical transformation to convert price data into a normally distributed form.

To use the indicator, you need to use the [EhlersFisherTransform](#) class.

Description

The Ehlers Fisher Transform is based on the concept that market prices do not have a normal (Gaussian) distribution. Instead, they often demonstrate asymmetrical distributions. The indicator applies a mathematical Fisher transformation formula to convert these asymmetrical distributions into normally distributed values.

This transformation makes extreme price movements more noticeable and helps more clearly identify market reversal points. When the Fisher transformation is applied, peak values increase sharply, making market behavior extremes more obvious.

EFT is particularly useful for:

- Determining potential market reversal points
- Identifying overbought and oversold conditions
- Detecting hidden divergences between price and the indicator
- Generating more accurate entry and exit signals

Parameters

The indicator has the following parameters:

- **Length** - calculation period (default value: 10)

Calculation

Ehlers Fisher Transform calculation involves several steps:

1. Transform price data to values between -1 and +1 (typically using normalized price rank or another oscillator):

```
Value = (2 * ((Price - Min) / (Max - Min))) - 1
```

where Min and Max are the minimum and maximum prices over the Length period.

2. Apply the Fisher transformation:

```
If Value >= 0.999, then Value = 0.999
If Value <= -0.999, then Value = -0.999

Fisher = 0.5 * ln((1 + Value) / (1 - Value))
```

where ln is the natural logarithm.

3. Smooth to reduce noise:

```
EFT = EMA(Fisher, Period)
```

where EMA is the exponential moving average.

Interpretation

The Ehlers Fisher Transform can be interpreted as follows:

1. **Extreme Values:**
 - Values above +2 often indicate market overbought conditions
 - Values below -2 often indicate market oversold conditions
2. **Zero Line Crossovers:**
 - Crossing the zero line from bottom to top can be viewed as a bullish signal
 - Crossing the zero line from top to bottom can be viewed as a bearish signal
3. **Indicator Reversal:**
 - Indicator reversal from extreme values often precedes price reversal
4. **Divergences:**
 - Bullish Divergence: price forms a new low, while EFT forms a higher low
 - Bearish Divergence: price forms a new high, while EFT forms a lower high
5. **Indicator Line Slope:**
 - Steep upward slope indicates strong upward momentum
 - Steep downward slope indicates strong downward momentum

The Ehlers Fisher Transform differs from many other oscillators in that it can reach extreme values and remain there for some time without necessarily reversing immediately. This makes it useful for identifying strong trend movements.

[image: indicator_ehlers_fisher_transform]

See Also

[CenterOfGravityOscillator](#) [SineWave](#) [HarmonicOscillator](#) [RSI](#)

Elder Impulse System (EIS)

EIS

Elder Impulse System (EIS) is a technical indicator developed by Dr. Alexander Elder that combines a trend indicator and a momentum oscillator to determine the direction and strength of market movement.

To use the indicator, you need to use the [ElderImpulseSystem](#) class.

Description

The Elder Impulse System (EIS) is a simple yet powerful tool for visualizing market momentum. It combines two indicators:

1. **Exponential Moving Average (EMA)** - to determine trend direction
2. **MACD Histogram** - to measure price movement strength and momentum

EIS classifies each candle on the price chart into one of three categories (usually denoted by different colors):

- **Green (strong bullish impulse)** - when both indicators are rising
- **Red (strong bearish impulse)** - when both indicators are falling
- **Blue or neutral (no clear impulse)** - when indicators move in opposite directions

EIS is particularly useful for:

- Quickly visually determining trend direction and strength
- Identifying entry and exit points in the direction of the main trend
- Identifying potential reversal points
- Filtering out false signals

Calculation

Elder Impulse System calculation involves the following steps:

1. Calculate 13-period exponential moving average (EMA):

```
EMA = EMA(Close, 13)
```

2. Calculate MACD Histogram (standard values: 12, 26, 9):

```
MACD Line = EMA(Close, 12) - EMA(Close, 26)
Signal Line = EMA(MACD Line, 9)
MACD Histogram = MACD Line - Signal Line
```

3. Determine color classification for the current candle:

```
If EMA[current] > EMA[previous] AND MACD Histogram[current] > MACD Histogram[previous], then
Green (Bullish Impulse)
If EMA[current] < EMA[previous] AND MACD Histogram[current] < MACD Histogram[previous], then Red
(Bearish Impulse)
Otherwise Blue (No Impulse)
```

Interpretation

The Elder Impulse System is interpreted as follows:

1. **Green Candles (strong bullish impulse):**
 - Indicate a strong upward momentum
 - Best time to buy or hold long positions
 - A series of green candles indicates a strong upward trend
2. **Red Candles (strong bearish impulse):**
 - Indicate a strong downward momentum
 - Best time to sell or hold short positions
 - A series of red candles indicates a strong downward trend
3. **Blue Candles (no clear impulse):**
 - Indicate uncertainty or consolidation
 - Signal a possible slowdown or trend reversal
 - Often appear during consolidation periods or before a trend change
4. **Trading Strategies:**
 - Buy when candles change color from blue to green
 - Sell when candles change color from blue to red
 - Close long positions when candles change color from green to any other
 - Close short positions when candles change color from red to any other
5. **Trend Confirmation:**
 - A sequence of green candles confirms an upward trend
 - A sequence of red candles confirms a downward trend
 - Color alternation indicates a sideways trend or uncertainty

[image: indicator_elder_impulse_system]

See Also

[EMA](#) [MACD](#) [MACDHistogram](#) [ForceIndex](#)

Elder's Force Index (EFI)

Elder's Force Index

Elder's Force Index (EFI) measures buying and selling pressure using price change and volume.

To use the indicator, you must use the [ElderForceIndex](#) class.

[image: IndicatorElderForceIndex]

Recommended content

[Force Index](#)

Elliot Wave Oscillator (EWO)

EWO

Elliot Wave Oscillator (EWO) is a technical indicator based on Elliott Wave Theory that helps traders determine wave structure and potential market reversal points.

To use the indicator, you need to use the [ElliotWaveOscillator](#) class.

Description

The Elliot Wave Oscillator (EWO) was developed to help traders apply Elliott Wave Theory in market analysis. Elliott Wave Theory assumes that markets move in predictable cycles consisting of five waves in the trend direction (impulse waves) and three waves against the trend (corrective waves).

EWO is based on the difference between fast and slow moving averages and is designed to identify impulse and corrective waves according to Elliott's theory. It helps determine when the market is in an impulse or corrective phase and suggests potential reversal points.

The Elliot Wave Oscillator is particularly useful for:

- Identifying the current Elliott wave structure
- Determining potential endings of impulse and corrective waves
- Confirming manual wave analysis
- Forecasting potential reversal points

Parameters

The indicator has the following parameters:

- **ShortPeriod** - period for the short moving average (default value: 5)
- **LongPeriod** - period for the long moving average (default value: 35)

Calculation

The Elliot Wave Oscillator calculation is quite simple:

```
EWO = EMA(Close, ShortPeriod) - EMA(Close, LongPeriod)
```

Where:

- EMA - exponential moving average
- Close - closing price
- ShortPeriod - short period (usually 5)
- LongPeriod - long period (usually 35)

Interpretation

The Elliot Wave Oscillator can be interpreted as follows:

1. Positive and Negative Values:

- Positive values (EWO above zero) indicate that the short EMA is above the long EMA, often

corresponding to a bullish trend or upward impulse wave

- Negative values (EWO below zero) indicate that the short EMA is below the long EMA, often corresponding to a bearish trend or downward impulse wave

2. Zero Line Crossovers:

- Crossing the zero line from bottom to top may signal the start of a new upward impulse wave
- Crossing the zero line from top to bottom may signal the start of a new downward impulse wave

3. Oscillator Extremes:

- Peaks and troughs of the oscillator may correspond to the end of impulse waves
- A corrective phase often follows reaching an extreme

4. Divergences:

- Bullish divergence (price forms a new low, while EWO forms a higher low) may indicate a potential end of a downward impulse wave
- Bearish divergence (price forms a new high, while EWO forms a lower high) may indicate a potential end of an upward impulse wave

5. Wave Structure:

- In impulse waves (waves 1, 3, 5), EWO typically shows strong values in the trend direction
- In corrective waves (waves 2, 4, A, B, C), EWO typically shows weaker values or moves in a direction opposite to the main trend

6. Wave 3 Identification:

- Wave 3, which is usually the strongest impulse wave in Elliott Wave Theory, is often characterized by the highest EWO values

[image: indicator_elliott_wave_oscillator]

See Also

[EMA MACD ZigZag WaveTrendOscillator](#)

Endpoint Moving Average (EPMA)

EPMA

Endpoint Moving Average (EPMA) is a technical indicator that is a modification of the standard moving average, designed to reduce lag in trend identification.

To use the indicator, you need to use the [EndpointMovingAverage](#) class.

Description

The Endpoint Moving Average (EPMA) is a special form of moving average that focuses on the endpoint data points. Unlike standard moving averages that evenly weight all points in a given period, EPMA gives more weight to the endpoint points, allowing for faster response to trend changes.

The main purpose of EPMA is to reduce the lag inherent in traditional moving averages while maintaining the ability to filter market noise. Due to its calculation methodology, EPMA often reacts more quickly to price direction changes, making it a valuable tool for traders seeking to identify trend reversals earlier.

EPMA is particularly useful for:

- Earlier identification of trend changes
- Reducing signal lag
- Creating more sensitive trading systems
- Confirming signals from other indicators with less delay

Parameters

The indicator has the following parameters:

- **Length** - calculation period (default value: 14)

Calculation

Endpoint Moving Average calculation is based on linear regression method, focusing on the endpoint points of the considered period:

1. Determining the linear trend between the initial and final points of the period:

```
Start Value = Price[current - Length + 1]
End Value = Price[current]
```

2. Calculating the slope of the trend line:

```
Slope = (End Value - Start Value) / (Length - 1)
```

3. Computing EPMA as a projection of the trend line to the current point:

```
EPMA = Start Value + Slope * (Length - 1)
```

In fact, EPMA is equal to the last price value (End Value) in the considered period, but conceptually it is a projection of the linear trend defined by the endpoint points.

Interpretation

The Endpoint Moving Average is interpreted similarly to other moving averages, but with consideration of its increased sensitivity:

1. EPMA Direction:

- Upward EPMA indicates an upward trend
- Downward EPMA indicates a downward trend

2. Crossovers with Price:

- When price crosses EPMA from bottom to top, it can be viewed as a bullish signal
- When price crosses EPMA from top to bottom, it can be viewed as a bearish signal

3. Multiple EPMA Crossovers:

- Crossing of a short EPMA with a long EPMA from bottom to top may indicate the start of an upward trend
- Crossing of a short EPMA with a long EPMA from top to bottom may indicate the start of a downward trend

4. Divergence with Other Moving Averages:

- Due to its increased sensitivity, EPMA may react to trend changes earlier than traditional moving averages
- Divergence between EPMA and other types of moving averages can serve as an early warning of a potential trend change

5. Signal Filtering:

- Due to its increased sensitivity, EPMA may generate more false signals during sideways consolidation periods
- It is recommended to use additional filters or confirmations from other indicators to improve signal reliability

[image: indicator_endpoint_moving_average]

See Also

[SMA](#) [EMA](#) [ZLEMA](#) [DEMA](#)

Envelope

Envelope

Envelope is an indicator that forms a channel created by offsetting a moving average by a specific value. The method of constructing the indicator exactly replicates the construction of Bollinger Bands, except for the calculation of the distance that the outer lines are from the average. If Bollinger Bands use standard deviation for this calculation, in **Envelope**, this distance is manually set in the settings. The parameters set are the period of the moving average and the size of the deviation.

To use the indicator, the [Envelope](#) class should be used.

[image: IndicatorEnvelope]

See Also

[EMA](#)

Elder Ray Index

Elder Ray

The **Elder Ray Index** is a composite indicator by Alexander Elder that combines an exponential moving average with the Bull Power and Bear Power oscillators. It visualises the balance between buyers and sellers and helps identify when one side loses control.

Use the [ElderRay](#) class to access the indicator.

Components

The indicator returns a [ElderRayValue](#) structure containing:

- **EMA** — the baseline exponential moving average of closing prices;
- **Bull Power** — the distance between the bar high and the EMA;
- **Bear Power** — the distance between the bar low and the EMA.

Parameters

Elder Ray inherits the settings of [ExponentialMovingAverage](#):

- **Length** — EMA period;
- **Alpha** — smoothing coefficient, when configured directly.

Interpretation

- **Bull Power > 0** together with a rising EMA confirms an uptrend.
- **Bear Power < 0** with a falling EMA confirms a downtrend.
- Shrinking Bull Power on rising prices or rising Bear Power on falling prices form divergences and warn about reversals.
- Zero-line crossings of Bull Power or Bear Power mark the shift in market control.

Trading decisions are made by analysing the EMA and both oscillators simultaneously. For example, a buying opportunity appears when the EMA is rising, Bear Power recovers from a fresh low, and Bull Power breaks above zero.

[image: indicator_elder_ray]

See also

[Bull Power](#) [Bear Power](#) [ExponentialMovingAverage](#)

Exponential Moving Average (EMA)

EMA

Exponential Moving Average (EMA) is a type of moving average (MA) that gives more weight and significance to the most recent data.

To use the indicator, you must use the [ExponentialMovingAverage](#) class.

[image: IndicatorExponentialMovingAverage]

Recommended content

[Fractals](#)

Fibonacci Retracement (FR)

FR

Fibonacci Retracement (FR) is a technical indicator based on Fibonacci numbers that helps identify potential support and resistance levels based on previous price movement.

To use the indicator, you need to use the [FibonacciRetracement](#) class.

Description

Fibonacci Retracement is a popular technical tool that uses horizontal lines to indicate areas of possible support or resistance on a price chart. These levels are based on Fibonacci numbers and their corresponding percentage ratios.

The indicator is based on the Fibonacci mathematical sequence, where each number is the sum of the two preceding ones (1, 1, 2, 3, 5, 8, 13, 21, 34...). From this sequence, key ratios used in technical analysis are derived: 23.6%, 38.2%, 50%, 61.8%, and 78.6%.

Fibonacci Retracement is applied to a significant price movement (trend) and shows the levels where a correction (pullback) may occur before continuing in the direction of the main trend.

The indicator is particularly useful for:

- Identifying potential support levels in an upward trend
- Identifying potential resistance levels in a downward trend
- Setting entry targets after a correction
- Determining levels for stop-loss placement

Parameters

The indicator has the following parameters:

- **Length** - period for determining a significant price movement (default value depends on the timeframe)

Calculation

Calculating Fibonacci Retracement levels involves the following steps:

1. Determining a significant price movement (trend):
 - In an upward trend: from low to high
 - In a downward trend: from high to low
2. Calculating correction levels based on the range of this movement:

```
Range = |High - Low|
```

```
Level 0% = High (for upward trend) or Low (for downward trend)
```

```
Level 23.6% = High - (Range * 0.236) or Low + (Range * 0.236)
```

```
Level 38.2% = High - (Range * 0.382) or Low + (Range * 0.382)
```

```
Level 50.0% = High - (Range * 0.5) or Low + (Range * 0.5)
```

```
Level 61.8% = High - (Range * 0.618) or Low + (Range * 0.618)
```

```
Level 78.6% = High - (Range * 0.786) or Low + (Range * 0.786)
```

```
Level 100% = Low (for upward trend) or High (for downward trend)
```

Interpretation

Fibonacci Retracement levels are interpreted as follows:

1. Main Correction Levels:

- 23.6% - weak level, often broken during a strong trend
- 38.2% - moderate level, corresponds to a typical correction
- 50.0% - psychologically significant level (though not a Fibonacci number)
- 61.8% - "golden ratio", the most important Fibonacci level
- 78.6% - deep correction, may signal a potential trend reversal

2. In an Upward Trend:

- Fibonacci levels serve as potential support levels
- Bouncing off a Fibonacci level can be a signal to enter a long position
- Breaking several Fibonacci levels downward may indicate trend weakening

3. In a Downward Trend:

- Fibonacci levels serve as potential resistance levels
- Bouncing off a Fibonacci level can be a signal to enter a short position
- Breaking several Fibonacci levels upward may indicate trend weakening

4. Coincidence with Other Levels:

- Fibonacci levels are more significant when they coincide with other important levels (previous highs/lows, moving averages, etc.)

5. Plotting on Different Timeframes:

- Fibonacci levels plotted on different timeframes can create clustering zones where the probability of reversal increases

[image: indicator_fibonacci_retracement]

See Also

[PivotPoints](#)

Finite Volume Element (FVE)

FVE

Finite Volume Element (FVE) is a technical indicator developed to analyze the relationship between price and volume, helping to assess buyer and seller pressure in the market.

To use the indicator, you need to use the [FiniteVolumeElement](#) class.

Description

The Finite Volume Element (FVE) analyzes the relationship between price changes and trading volumes to determine the potential strength of price movement. It is based on the assumption that price changes are most significant when they are confirmed by corresponding volumes.

The FVE indicator converts price change, weighted by volume, into an oscillator that helps determine the relative balance between buyers and sellers in the market. Positive FVE values indicate buyer predominance, while negative values indicate seller predominance.

FVE is particularly useful for:

- Assessing the strength and sustainability of the current trend
- Identifying potential reversal points
- Determining divergences between price and volume
- Confirming signals from other indicators

Parameters

The indicator has the following parameters:

- **Length** - smoothing period (default value: 22)

Calculation

FVE indicator calculation involves several steps:

1. Calculate typical price for current and previous periods:

$$\text{Typical Price} = (\text{High} + \text{Low} + \text{Close}) / 3$$

2. Calculate typical price change:

$$\text{Price Change} = \text{Typical Price}[\text{current}] - \text{Typical Price}[\text{previous}]$$

3. Calculate volume-weighted price change:

$$\text{Volume-Weighted Price Change} = \text{Price Change} * \text{Volume}[\text{current}]$$

4. Normalize to account for market scale:

$$\text{Normalized Value} = \text{Volume-Weighted Price Change} / (\text{Average Volume over period} * \text{Price Volatility})$$

5. Cumulative summation and smoothing:

$$\text{FVE} = \text{SMA}(\text{Cumulative Sum of Normalized Values}, \text{Length})$$

Where:

- High, Low, Close - highest, lowest, and closing prices
- Volume - trading volume
- SMA - simple moving average
- Length - smoothing period

Interpretation

The FVE indicator can be interpreted as follows:

1. Zero Line Crossovers:

- Transition from negative to positive values indicates increased buyer pressure and can be viewed as a bullish signal
- Transition from positive to negative values indicates increased seller pressure and can be viewed as a bearish signal

2. Extreme Values:

- High positive values (above +3) may indicate market overbought conditions
- High negative values (below -3) may indicate market oversold conditions

3. Divergences:

- Bullish divergence (price forms a new low, while FVE forms a higher low) may signal a potential upward reversal
- Bearish divergence (price forms a new high, while FVE forms a lower high) may signal a potential downward reversal

4. Trend Confirmation:

- Consistently positive FVE values confirm the strength of an upward trend
- Consistently negative FVE values confirm the strength of a downward trend

5. FVE Change Rate:

- Rapid increase or decrease in FVE values may indicate strong price movement momentum
- Slowing of FVE value changes may signal potential momentum deceleration

6. Support and Resistance Levels:

- Historical reversal points on the FVE chart can serve as guidelines for future reversals

[image: indicator_finite_volume_element]

See Also

[OBV](#) [ADL](#) [ForceIndex](#) [ChaikinMoneyFlow](#)

Force Index (FI)

FI

Force Index (FI) is a technical indicator developed by Dr. Alexander Elder that measures the strength of each price movement based on its direction, magnitude, and trading volume.

To use the indicator, you need to use the [ForceIndex](#) class.

Description

The Force Index is an oscillator that measures the strength of "bulls" (buyers) or "bears" (sellers) in each price movement. It combines three important elements of market information: price movement direction, movement magnitude, and trading volume.

The main idea of the indicator is that the larger the price change and the larger the trading volume, the stronger the market movement. Positive Force Index values indicate buyer predominance (bullish pressure), while negative values indicate seller predominance (bearish pressure).

The Force Index is particularly useful for:

- Determining the strength of the current trend
- Identifying potential reversal points
- Confirming breakouts
- Detecting divergences between price and movement strength

Parameters

The indicator has the following parameters:

- **Length** - smoothing period (default value: 13)

Calculation

Force Index calculation involves the following steps:

1. Calculating single-period Force Index:

$$\text{1-Period Force Index} = (\text{Close}[\text{current}] - \text{Close}[\text{previous}]) * \text{Volume}[\text{current}]$$

2. Smoothing using Exponential Moving Average (EMA):

$$\text{Force Index} = \text{EMA}(\text{1-Period Force Index}, \text{Length})$$

Where:

- Close - closing price
- Volume - trading volume
- EMA - exponential moving average
- Length - smoothing period

Interpretation

The Force Index can be interpreted in various ways:

1. Zero Line Crossovers:

- Transition from negative to positive values indicates increased bullish pressure and can be viewed as a buy signal
- Transition from positive to negative values indicates increased bearish pressure and can be viewed as a sell signal

2. Extreme Values:

- High positive values indicate strong bullish pressure that may lead to market overbought conditions
- High negative values indicate strong bearish pressure that may lead to market oversold conditions

3. Divergences:

- Bullish Divergence (price forms a new low, while Force Index forms a higher low) may signal a potential upward reversal
- Bearish Divergence (price forms a new high, while Force Index forms a lower high) may signal a potential downward reversal

4. Trend Confirmation:

- Consistently positive Force Index values confirm the strength of an upward trend
- Consistently negative Force Index values confirm the strength of a downward trend

5. Triple Use (according to Elder):

- Short-term Force Index (2-day): for identifying short-term opportunities
- Medium-term Force Index (13-day): for determining medium-term trends and corrections
- Long-term Force Index (100-day): for identifying the main trend

6. Correction Identification:

- In an upward trend, days with negative Force Index may indicate temporary corrections
- In a downward trend, days with positive Force Index may indicate temporary bounces

[image: indicator_force_index]

See Also

[EMA](#) [OBV](#) [ADL](#)

Forecast Oscillator (FOSC)

FOSC

Forecast Oscillator (FOSC) is a technical indicator that measures the deviation of price from its predicted value obtained through linear regression, representing this deviation as a percentage.

To use the indicator, you need to use the [ForecastOscillator](#) class.

Description

The Forecast Oscillator (FOSC) is based on linear regression and designed to measure the degree of deviation of the current price from its predicted value. It helps traders assess how closely the current price corresponds to the expected trend or deviates from it.

The indicator calculates a trend line using linear regression over a specified period, then compares the actual closing price with the predicted value on this line. The difference is expressed as a percentage, making FOSC an oscillator that fluctuates around the zero line.

The Forecast Oscillator is particularly useful for:

- Determining the degree of price alignment with the expected trend
- Identifying potential reversal points
- Detecting extreme price deviations from the trend
- Identifying periods when price moves faster or slower than expected

Parameters

The indicator has the following parameters:

- **Length** - period for linear regression calculation (default value: 14)

Calculation

Forecast Oscillator calculation involves the following steps:

1. Calculate the forecast line using linear regression over the specified period:

```
Forecast = Linear Regression Line(Close, Length)
```

2. Calculate the oscillator as a percentage ratio between current price and forecast value:

```
FOSC = ((Close - Forecast) / Forecast) * 100
```

Where:

- Close - current closing price
- Forecast - predicted value obtained through linear regression
- Length - period for linear regression calculation

Interpretation

The Forecast Oscillator is interpreted as follows:

1. **Deviation from Zero:**

- Positive values ($FOSC > 0$) indicate that the current price is above the predicted value, which may suggest a stronger upward movement than expected
 - Negative values ($FOSC < 0$) indicate that the current price is below the predicted value, which may suggest a stronger downward movement than expected
2. **Extreme Values:**
 - Very high positive values may indicate market overbought conditions relative to the trend
 - Very low negative values may indicate market oversold conditions relative to the trend
 3. **Return to Zero:**
 - Movement of FOSC from extreme values towards zero may signal a potential return of price to its trend line
 4. **Zero Line Crossovers:**
 - Crossing the zero line from bottom to top can be viewed as a bullish signal
 - Crossing the zero line from top to bottom can be viewed as a bearish signal
 5. **Divergences:**
 - Bullish Divergence (price forms a new low, while FOSC forms a higher low) may indicate a potential upward reversal
 - Bearish Divergence (price forms a new high, while FOSC forms a lower high) may indicate a potential downward reversal
 6. **Trend Confirmation:**
 - If FOSC moves in the same direction as price, this confirms the strength of the current trend
 - If FOSC moves in the opposite direction to price, this may indicate a weakening of the current trend

[image: indicator_forecast_oscillator]

See Also

[LinearRegression](#) [StandardError](#) [DisparityIndex](#)

Fractal Adaptive Moving Average (FRAMA)

FRAMA

Fractal Adaptive Moving Average (FRAMA) is a technical indicator developed by John Ehlers that adapts the rate of reaction to price changes based on the market's fractal dimension.

To use the indicator, you need to use the [FractalAdaptiveMovingAverage](#) class.

Description

The Fractal Adaptive Moving Average (FRAMA) is an advanced type of Exponential Moving Average (EMA) that automatically adjusts its sensitivity to price changes based on the market's fractal dimension. The indicator was developed by John Ehlers and introduced in the Technical Analysis of Stocks & Commodities magazine in October 2000.

FRAMA uses the concept of fractal geometry to analyze market structure. It determines how "fractal" or chaotic the current market is, and based on this, adjusts the indicator's response rate:

- In trending (less fractal) market conditions, FRAMA responds quickly to price changes, similar to a short EMA
- In sideways (more fractal) market conditions, FRAMA responds more slowly, similar to a long EMA

This allows FRAMA to respond faster to significant price movements and ignore market noise, making it more effective compared to traditional moving averages.

Parameters

The indicator has the following parameters:

- **Length** - calculation period (default value: 10-20)

Calculation

FRAMA calculation involves several steps:

1. Calculate fractal dimension (D) based on the logarithmic ratio of high-low price length to the number of periods:

```
N1 = High(1...Length/2) - Low(1...Length/2)
N2 = High(Length/2+1...Length) - Low(Length/2+1...Length)
N3 = High(1...Length) - Low(1...Length)

D = (log(N1 + N2) - log(N3)) / log(2)
```

2. Convert fractal dimension to alpha factor for exponential smoothing:

```
Smoothing Factor = exp(-4.6 * (D - 1))
Alpha = Smoothing Factor * Smoothing Factor
```

3. Apply alpha factor to current price and previous FRAMA value:

```
FRAMA = Alpha * Price + (1 - Alpha) * FRAMA[previous]
```

Where:

- High - maximum price for the period

- Low - minimum price for the period
- log - natural logarithm

Interpretation

FRAMA can be interpreted similarly to other moving averages, but with consideration of its adaptive nature:

1. **FRAMA Direction:**
 - Upward FRAMA indicates an upward trend
 - Downward FRAMA indicates a downward trend
2. **Crossovers with Price:**
 - When price crosses FRAMA from bottom to top, it can be viewed as a bullish signal
 - When price crosses FRAMA from top to bottom, it can be viewed as a bearish signal
3. **Multiple FRAMA Crossovers:**
 - Crossing of a short FRAMA with a long FRAMA from bottom to top may indicate the start of an upward trend
 - Crossing of a short FRAMA with a long FRAMA from top to bottom may indicate the start of a downward trend
4. **FRAMA Slope Angle:**
 - Steep slope angle indicates a strong trend
 - Shallow slope angle indicates a weak trend
 - Horizontal movement indicates a sideways trend
5. **Signal Filtering:**
 - Due to its adaptive nature, FRAMA creates fewer false signals than traditional moving averages
 - The shorter the FRAMA period, the more sensitive the indicator will be to price changes
6. **Support and Resistance Levels:**
 - FRAMA can serve as a dynamic support level in an upward trend
 - FRAMA can serve as a dynamic resistance level in a downward trend

[image: indicator_fractal_adaptive_moving_average]

See Also

[EMA KAMA VIDYA](#)

Fractals

Fractals

Fractals is a sequence of several candles, the maximum (minimum) of the central one is above the maximums (minimums) of adjacent ones.

To use the indicator, you must use the [Fractals](#) class.

[image: IndicatorFractals]

Recommended content

[Gator Oscillator](#)

Fractal Dimension Index (FDI)

FDI

Fractal Dimension Index (FDI) quantifies the roughness of a price series.

To use the indicator, you need to use the [FractalDimension](#) class.

Description

The FDI ranges from 1 to 2 and reflects market behavior:

- Values near 1 indicate a persistent trend (smoother path).
- Values around 1.5 correspond to a random walk.
- Values close to 2 denote a ranging or noisy market.

The indicator is based on fractal geometry and measures how complex the price path is.

Parameters

The indicator has the following parameter:

- **Length** – calculation period (default value: 30)

Calculation

FDI is computed by comparing the total price path length with the overall high-low range:

1. Sum absolute differences between consecutive prices over the period to get the price path length.
2. Find the difference between the maximum high and minimum low for the period.
3. Calculate FDI using:

$$\text{FDI} = 1 + (\log(\text{PathLength}) - \log(\text{Range})) / \log(2 * (\text{Length} - 1))$$

4. Clamp the result between 1 and 2.

Interpretation

- **FDI near 1** – strong trending behavior.
- **FDI around 1.5** – random walk; trend strength is neutral.
- **FDI closer to 2** – choppy or sideways market.

[image: indicator_fractal_dimension]

See Also

[Hurst Exponent](#)

[Fractal Adaptive Moving Average](#)

Gator Oscillator (Gator)

Gator Oscillator

Gator Oscillator is an oscillator developed by Bill Williams. It is closely related to the similar Alligator indicator. As a trend indicator, it is most useful in markets that demonstrate strong directional movement.

To use the indicator, you must use the [GatorOscillator](#) class.

[image: IndicatorGatorOscillator]

Recommended content

[Highest](#)

Gopalakrishnan Range Index (GAPO)

GAPO

Gopalakrishnan Range Index (GAPO) is a technical indicator developed by Tushar Gopalakrishnan to measure market volatility using a logarithmic scale.

To use the indicator, you need to use the [GopalakrishnanRangeIndex](#) class.

Description

The Gopalakrishnan Range Index (GAPO) is a volatility indicator that uses a logarithmic scale to measure the overall price range over a specific period. It was developed by Tushar Gopalakrishnan and presented in the "Technical Analysis of Stocks & Commodities" magazine.

GAPO evaluates extreme market movements by measuring the logarithmic ratio between the maximum and minimum prices over a given period. This approach allows the indicator to more accurately reflect increased volatility, especially during periods of sharp price movements.

The GAPO indicator is particularly useful for:

- Identifying periods of high and low volatility
- Detecting potential reversal points after extreme movements
- Adjusting parameters for other volatility-based indicators
- Adapting trading strategies to current market conditions

Parameters

The indicator has the following parameters:

- **Length** - calculation period (default value: 10)

Calculation

The Gopalakrishnan Range Index calculation is quite simple:

$$\text{GAPO} = \log(N) * \log(\text{Highest High} - \text{Lowest Low})$$

Where:

- $\log$ - natural logarithm
- N - number of periods (Length)
- Highest High - highest high over the Length period
- Lowest Low - lowest low over the Length period

Interpretation

The Gopalakrishnan Range Index can be interpreted as follows:

1. Absolute Values:

- High GAPO values indicate periods of high volatility
- Low GAPO values indicate periods of low volatility

- Extremely high values may indicate possible market overextension and potential reversal
2. **GAPO Trends:**
 - Increasing GAPO values indicate increasing volatility
 - Decreasing GAPO values indicate decreasing volatility
 - A sharp GAPO jump may signal the beginning of a new trend movement
 3. **Relative Levels:**
 - Comparing the current GAPO value with its historical levels allows assessment of relative volatility
 - Values above the 95th percentile of the historical range may indicate extreme volatility
 - Values below the 5th percentile of the historical range may indicate unusually low volatility
 4. **Trading Strategies:**
 - During high volatility periods (high GAPO values), it may be appropriate to increase stop-loss and target profit sizes
 - During low volatility periods (low GAPO values), range trading strategies may be more suitable
 - Extreme GAPO values can be used as contrary indicators for finding reversal points
 5. **Combination with Other Indicators:**
 - GAPO can be used to filter signals from other indicators
 - During high volatility periods, trend indicator signals may be more reliable
 - During low volatility periods, oscillator signals may be more effective

[image: indicator_gopalakrishnan_range_index]

See Also

[ATR](#) [ChoppinessIndex](#) [TrueRange](#)

Guppy Multiple Moving Average (GMMA)

GMMA

Guppy Multiple Moving Average (GMMA) is a technical indicator developed by Daryl Guppy that uses two groups of exponential moving averages (EMA) to reveal the interaction between short-term traders and long-term investors.

To use the indicator, you need to use the [GuppyMultipleMovingAverage](#) class.

Description

The Guppy Multiple Moving Average (GMMA) consists of two groups of exponential moving averages (EMA):

1. **Short-term group** (typically 3, 5, 8, 10, 12, and 15 periods) - represents short-term traders' activity
2. **Long-term group** (typically 30, 35, 40, 45, 50, and 60 periods) - represents long-term investors' activity

GMMA allows visualization of the interaction between these two market participant groups and determines whether the market is in a trending or consolidation state. The indicator also helps identify moments when short-term traders begin to follow the same direction as long-term investors, which often indicates the formation or strengthening of a trend.

GMMA is particularly useful for:

- Determining the strength and direction of the current trend
- Identifying potential trend reversals
- Recognizing when the market transitions from consolidation to a trend
- Determining optimal entry points into an existing trend

Calculation

GMMA calculation involves computing two groups of exponential moving averages:

1. Short-term EMA group:

```
EMA_3 = EMA(Price, 3)
EMA_5 = EMA(Price, 5)
EMA_8 = EMA(Price, 8)
EMA_10 = EMA(Price, 10)
EMA_12 = EMA(Price, 12)
EMA_15 = EMA(Price, 15)
```

2. Long-term EMA group:

```
EMA_30 = EMA(Price, 30)
EMA_35 = EMA(Price, 35)
EMA_40 = EMA(Price, 40)
EMA_45 = EMA(Price, 45)
EMA_50 = EMA(Price, 50)
EMA_60 = EMA(Price, 60)
```

Where:

- EMA - exponential moving average
- Price - price (usually closing price)

Interpretation

GMMA interpretation involves analyzing both individual groups and their interaction:

1. **Group Positioning:**
 - When the short-term group is above the long-term group, it indicates an upward trend
 - When the short-term group is below the long-term group, it indicates a downward trend
2. **Distance Between Groups:**
 - Large distance between groups indicates a strong trend
 - Small distance or group crossover indicates a weak trend or consolidation
3. **Compression and Expansion:**
 - Compression (convergence) of lines within a group indicates uncertainty and possible consolidation
 - Expansion (divergence) of lines within a group indicates trend strengthening
4. **Crossovers:**
 - Short-term group crossing through long-term group from bottom to top - strong bullish signal
 - Short-term group crossing through long-term group from top to bottom - strong bearish signal
5. **Direction Changes:**
 - When the long-term group begins to change direction, it indicates a significant change in long-term investor sentiment
 - Short-term group reversal without changes in the long-term group often indicates a temporary correction
6. **Optimal Entry Points:**
 - After strong expansion, compression may occur, indicating a correction within the trend
 - The end of such compression (new expansion) can be a good entry point in the direction of the main trend
7. **Early Reversal Warning:**
 - Short-term averages change direction first, then changes begin to appear in long-term averages
 - Crossover between groups can serve as a trend reversal confirmation

[image: indicator_guppy_multiple_moving_average]

See Also

[EMA](#) [MovingAverage](#) [Ribbon](#) [RainbowCharts](#) [MACD](#)

Harmonic Oscillator (HO)

HO

Harmonic Oscillator (HO) is a technical indicator based on harmonic oscillation theory that helps identify cyclical components in price movement.

To use the indicator, you need to use the [HarmonicOscillator](#) class.

Description

The Harmonic Oscillator (HO) is an indicator developed to identify periodicity and cyclical nature of market price movements. It is based on the principle that many price movements contain harmonic (periodic) components that can be isolated and used to forecast future price movements.

The indicator applies spectral analysis methods to decompose the price series into harmonic components, highlighting dominant cycles. It then displays these cyclical components as an oscillator that helps traders determine when price may reach local maximums or minimums within the identified cycles.

HO is particularly useful for:

- Determining the cyclical nature of the market
- Identifying potential reversal points
- Filtering market noise
- Predicting moments when price may change direction

Parameters

The indicator has the following parameters:

- **Length** - analysis period (default value: 30)

Calculation

Harmonic Oscillator calculation involves the following steps:

1. Price series preprocessing (detrending):

```
Detrended Price = Price - SMA(Price, Length)
```

2. Applying spectral analysis to identify dominant cycles:

```
Spectral Components = FFT(Detrended Price)
```

3. Extracting the most significant harmonic components:

```
Dominant Cycles = Extract Top N Spectral Components based on amplitude
```

4. Synthesizing the Harmonic Oscillator based on dominant cycles:

```
HO = Reconstruction of Dominant Cycles through Inverse FFT
```

Where:

- Price - price (usually closing price)

- SMA - simple moving average
- FFT - Fast Fourier Transform
- Length - analysis period

Interpretation

The Harmonic Oscillator can be interpreted as follows:

1. **Zero Line Crossovers:**
 - When HO crosses the zero line from bottom to top, it can be viewed as a bullish signal
 - When HO crosses the zero line from top to bottom, it can be viewed as a bearish signal
2. **Oscillator Extremes:**
 - When HO reaches a local maximum, it may indicate a potential price peak
 - When HO reaches a local minimum, it may indicate a potential price bottom
3. **Divergences:**
 - Bullish Divergence: price forms a new low, while HO forms a higher low
 - Bearish Divergence: price forms a new high, while HO forms a lower high
4. **Cycle Projection:**
 - Regular HO peaks and troughs can be used to project future reversal points
 - Analyzing the duration between peaks/troughs can help determine the dominant cycle length
5. **Amplitude Changes:**
 - Increased HO oscillation amplitude may indicate strengthening of the cyclical component
 - Decreased HO oscillation amplitude may indicate attenuation of the cyclical component
6. **Combination with Other Indicators:**
 - HO works best in combination with trend indicators
 - In trending markets, HO signals can be used to determine entry points in the trend direction

[image: indicator_harmonic_oscillator]

See Also

[SineWave](#) [CenterOfGravityOscillator](#) [FisherTransform](#) [DetrendedSyntheticPrice](#)

Highest Value (Highest)

Highest

The indicator indicates the **Maximum value for the period**.

To use the indicator, you must use the [Highest](#) class.

[image: IndicatorHighest]

Recommended content

[HMA](#)

High Low Index (HLI)

HLI

High Low Index (HLI) is a technical indicator that measures the ratio of the number of stocks reaching new highs to the number of stocks reaching new lows over a specific period of time.

To use the indicator, you need to use the [HighLowIndex](#) class.

Description

The High Low Index (HLI) is a market breadth indicator that analyzes overall market activity by comparing the number of instruments reaching new highs with the number reaching new lows. This allows for assessing the internal strength or weakness of the market.

The main idea of the indicator is that a healthy market is characterized by more securities reaching new highs than new lows. Conversely, a weakening market will have more securities reaching new lows.

HLI is particularly useful for:

- Assessing the overall market condition
- Identifying divergences between the index and individual market instruments
- Determining potential market reversal points
- Confirming signals from other indicators

Parameters

The indicator has the following parameters:

- **Length** - calculation period (default value: 14)

Calculation

High Low Index calculation involves the following steps:

1. Count the number of securities reaching new highs over the Length period:

```
New Highs = Number of instruments reaching new highs over the Length period
```

2. Count the number of securities reaching new lows over the Length period:

```
New Lows = Number of instruments reaching new lows over the Length period
```

3. Calculate High Low Index as the ratio of the difference between new highs and lows to their sum:

```
HLI = ((New Highs - New Lows) / (New Highs + New Lows)) * 100
```

Note: If (New Highs + New Lows) equals zero, HLI is set to zero to avoid division by zero.

Interpretation

The High Low Index is interpreted as follows:

1. **Value Range:**
 - HLI oscillates between -100 and +100

- Positive values indicate more securities reaching new highs than new lows
 - Negative values indicate more securities reaching new lows than new highs
2. **Zero Line Crossovers:**
 - Transition from negative to positive values can be viewed as a bullish signal
 - Transition from positive to negative values can be viewed as a bearish signal
 3. **Extreme Values:**
 - Values close to +100 indicate a strong bullish market (possible overbought condition)
 - Values close to -100 indicate a strong bearish market (possible oversold condition)
 4. **Divergences:**
 - Bullish Divergence: market index reaches a new low, but HLI forms a higher low
 - Bearish Divergence: market index reaches a new high, but HLI forms a lower high
 5. **HLI Trends:**
 - Sustained HLI growth indicates strengthening of a bullish market
 - Sustained HLI decline indicates strengthening of a bearish market
 6. **Market Trend Confirmation:**
 - If the market index rises and HLI also rises, this confirms the strength of an upward trend
 - If the market index falls and HLI also falls, this confirms the strength of a downward trend

[image: indicator_high_low_index]

See Also

[McClellanOscillator](#)

Historical Volatility Ratio (HVR)

HVR

Historical Volatility Ratio (HVR) is a technical indicator that compares short-term historical volatility with long-term historical volatility to assess changes in market activity.

To use the indicator, you need to use the [HistoricalVolatilityRatio](#) class.

Description

The Historical Volatility Ratio (HVR) is a relative volatility indicator that compares short-term volatility with long-term market volatility. The indicator helps determine whether current volatility is increasing or decreasing relative to its historical level.

HVR is calculated as the ratio of short-term historical volatility to long-term historical volatility. Values above 1.0 indicate that current (short-term) volatility is higher than long-term volatility, which may signal increased market activity or a potential trend change.

The indicator is particularly useful for:

- Identifying periods of high and low volatility
- Determining potential trend reversal points
- Adapting trading strategies to current market conditions
- Assessing market risk and setting appropriate position sizes

Parameters

The indicator has the following parameters:

- **ShortPeriod** - period for calculating short-term volatility (default value: 5)
- **LongPeriod** - period for calculating long-term volatility (default value: 20)

Calculation

Historical Volatility Ratio calculation involves the following steps:

1. Calculate short-term historical volatility:

```
Short-term Volatility = Standard Deviation of Log Returns over ShortPeriod * Sqrt(Trading Days Per Year)
```

2. Calculate long-term historical volatility:

```
Long-term Volatility = Standard Deviation of Log Returns over LongPeriod * Sqrt(Trading Days Per Year)
```

3. Calculate HVR as the ratio of short-term volatility to long-term volatility:

```
HVR = Short-term Volatility / Long-term Volatility
```

Where:

- Log Returns - logarithmic returns ($\ln(\text{Price}[i] / \text{Price}[i-1])$)
- Standard Deviation - standard deviation

- Trading Days Per Year - number of trading days in a year (usually 252 for stock markets)
- ShortPeriod - short period for volatility calculation
- LongPeriod - long period for volatility calculation

Interpretation

The Historical Volatility Ratio can be interpreted as follows:

1. Level 1.0:

- $HVR = 1.0$ means short-term volatility is equal to long-term volatility
- $HVR > 1.0$ indicates short-term volatility is higher than long-term volatility
- $HVR < 1.0$ indicates short-term volatility is lower than long-term volatility

2. Extreme Values:

- Very high HVR values (e.g., > 2.0) may indicate a sharp volatility increase, often occurring during market panics or strong movements
- Very low HVR values (e.g., < 0.5) may indicate a volatility compression period, often preceding strong movements

3. HVR Trends:

- Rising HVR indicates an increase in current volatility
- Falling HVR indicates a decrease in current volatility

4. Trading Strategies:

- When HVR is high, it may be appropriate to use breakout-based strategies
- When HVR is low, mean reversion or range trading strategies may be more suitable

5. Risk Management:

- High HVR values may signal the need to reduce position sizes due to increased volatility
- Low HVR values may allow increased position sizes due to reduced volatility

6. Potential Reversals:

- Extreme HVR values often precede significant price movements
- A sharp HVR increase after a low volatility period may signal the start of a new trend

[image: indicator_historical_volatility_ratio]

See Also

[ATR](#) [StandardDeviation](#) [ChoppinessIndex](#)

Hurst Exponent (H)

HurstExponent

Hurst Exponent is a statistical measure used to evaluate the tendency of a time series to either trend or revert to the mean.

To use the indicator, you need to use the [HurstExponent](#) class.

Description

The Hurst exponent (H) ranges between 0 and 1 and describes the long-term memory of a series:

- $H > 0.5$ indicates persistent trending behaviour.
- $H < 0.5$ shows a tendency toward mean reversion.
- $H \approx 0.5$ corresponds to a random walk.

It can help estimate market efficiency and reveal emerging cycles or trends.

Parameters

- **Length** – the number of bars used in the calculation.

Calculation

A common method is based on the rescaled range (R/S) approach:

1. For each window of Length points, compute cumulative deviations from the mean.
2. Find the range R as the difference between maximum and minimum cumulative deviation.
3. Calculate the standard deviation S of the window.
4. Evaluate the rescaled range R/S.
5. Estimate H as the slope of $\log(R/S)$ against $\log(\text{Length})$.

Higher values of the exponent suggest stronger trending behaviour, while lower values imply greater mean reversion.

[image: indicator_hurst_exponent]

See Also

[Fractal Adaptive Moving Average Market Meanness Index](#)

Hull Moving Average (HMA)

HMA

Hull Moving Average (HMA) - the indicator shows the direction of the market trend and represents an improved moving average.

To use the indicator, you must use the [HullMovingAverage](#) class.

[image: IndicatorHullMovingAverage]

Recommended content

[Ichimoku](#)

Ichimoku

Ichimoku is an indicator represented by a combination of five lines, three of which are moving averages, and two are derivatives of these. Ichimoku identifies the presence of a trend, as well as indicates support/resistance zones and trend retracements.

To use the indicator, the [Ichimoku](#) class should be used.

Description of the Ichimoku Indicator

Graphically, the indicator consists of five colored lines similar to simple moving averages:

- Tenkan (conversion line) — the fastest line, it reacts first to price changes. Its main purpose is to determine the direction of the short-term trend. In the classic version, it takes a segment of 9 bars back. It is constructed as half the sum of the highest and lowest prices.
- Kijun (base line) — indicates the medium-term trend, with a period of 26.
- Senkou A and Senkou B — projected and displayed 26 periods into the future, together they form what is called the cloud (Kumo), which shows areas of support and resistance and is a key component of the indicator.
- Chikou (lagging span) — represents the last closing price, shifted 26 periods back. It helps confirm signals: if it crosses the chart from bottom to top, it's a buy signal, and from top to bottom — a sell signal. Essentially, Chikou acts as a trend filter.

[image: IndicatorIchimoku]

See Also

[JMA](#)

Intraday Intensity Index (III)

III

Intraday Intensity Index (III) is a technical indicator developed by David Bostian that assesses the relationship between closing price, price range, and trading volume within a trading day.

To use the indicator, you need to use the [IntradayIntensityIndex](#) class.

Description

The Intraday Intensity Index (III) combines information about price movement and trading volume to evaluate the intensity of buying or selling pressure within a trading day. The indicator is based on the assumption that the position of the closing price relative to the day's price range, combined with volume, can indicate the direction and strength of market movement.

III is particularly useful for identifying intraday changes in market sentiment and determining potential reversal points. Positive indicator values indicate buying pressure (closing price closer to the day's high), while negative values indicate selling pressure (closing price closer to the day's low).

The Intraday Intensity Index is especially effective for:

- Identifying intraday changes in market sentiment
- Determining potential reversal points
- Confirming signals from other indicators
- Assessing the strength of the current trend

Parameters

The indicator has the following parameters:

- **Length** - smoothing period (default value: 14)

Calculation

Intraday Intensity Index calculation involves the following steps:

1. Calculate individual III value for each period:

```
III raw = ((2 * Close - High - Low) / ((High - Low) * Volume)) * Volume
```

2. Smooth using a simple moving average:

```
III = SMA(III raw, Length)
```

Where:

- Close - closing price
- High - period's highest price
- Low - period's lowest price
- Volume - trading volume
- SMA - simple moving average

- Length - smoothing period

Interpretation

The Intraday Intensity Index can be interpreted as follows:

1. **Zero Line Crossovers:**

- Transition from negative to positive values can be viewed as a bullish signal, indicating increased buying pressure
- Transition from positive to negative values can be viewed as a bearish signal, indicating increased selling pressure

2. **Extreme Values:**

- High positive values indicate strong buying pressure
- High negative values indicate strong selling pressure
- Extreme values may indicate market overbought or oversold conditions

3. **Divergences:**

- Bullish Divergence: price forms a new low, while III forms a higher low
- Bearish Divergence: price forms a new high, while III forms a lower high

4. **III Trends:**

- Consistently positive III values confirm an upward trend
- Consistently negative III values confirm a downward trend
- Oscillations around the zero line may indicate a sideways trend or uncertainty

5. **Combining with Other Indicators:**

- III is often used in combination with other technical indicators to confirm signals
- Particularly effective when combined with trend and volume indicators

6. **Changes in Values:**

- Rapid change from negative to positive values may indicate a sharp shift in market sentiment
- Gradual convergence towards the zero line may indicate a weakening of current momentum

[image: indicator_intraday_intensity_index]

See Also

[IntradayMomentumIndex](#) [BalanceOfPower](#) [ForceIndex](#) [ChaikinMoneyFlow](#)

Intraday Momentum Index (IMI)

IMI

Intraday Momentum Index (IMI) is a technical indicator developed by Tushar Chande that combines intraday pricing principles and the RSI concept to measure intraday momentum.

To use the indicator, you need to use the [IntradayMomentumIndex](#) class.

Description

The Intraday Momentum Index (IMI) was created as a modification of the classic Relative Strength Index (RSI), specifically adapted for analyzing intraday market dynamics. Instead of using sequential closing prices, like in traditional RSI, IMI compares the closing price with the opening price for each period.

IMI evaluates how often and how strongly the closing price exceeds the opening price (positive momentum) or falls below the opening price (negative momentum) over a given period. This allows identifying the predominant direction and strength of intraday movement.

The indicator is particularly useful for:

- Determining the intraday market direction
- Identifying potential reversal points
- Determining overbought and oversold levels
- Detecting divergences between price and momentum

Parameters

The indicator has the following parameters:

- **Length** - calculation period (default value: 14)

Calculation

Intraday Momentum Index calculation involves the following steps:

1. Determine intraday price movement:

```
Gain = Close - Open, if Close > Open
Loss = Open - Close, if Close < Open
```

2. Calculate the sum of positive and negative movements over the Length period:

```
Sum Gains = Sum of all Gains over Length period
Sum Losses = Sum of all Losses over Length period
```

3. Calculate IMI using a formula similar to RSI:

```
IMI = 100 * (Sum Gains / (Sum Gains + Sum Losses))
```

Note: If (Sum Gains + Sum Losses) equals zero, IMI is set to 50 to avoid division by zero.

Interpretation

The Intraday Momentum Index is interpreted similarly to RSI:

1. **Value Range:**
 - IMI oscillates between 0 and 100
 - Values above 50 indicate predominance of positive intraday momentum
 - Values below 50 indicate predominance of negative intraday momentum
2. **Overbought and Oversold Levels:**
 - Values above 70 are typically viewed as indicating market overbought conditions
 - Values below 30 are typically viewed as indicating market oversold conditions
3. **Centerline Crossovers:**
 - Crossing the 50 line from bottom to top can be viewed as a bullish signal
 - Crossing the 50 line from top to bottom can be viewed as a bearish signal
4. **Divergences:**
 - Bullish Divergence: price forms a new low, while IMI forms a higher low
 - Bearish Divergence: price forms a new high, while IMI forms a lower high
5. **Failed Swings:**
 - If IMI cannot reach the overbought level during an upward trend, this may indicate trend weakness
 - If IMI cannot reach the oversold level during a downward trend, this may indicate trend weakness
6. **Trend Confirmation:**
 - Sustained IMI values above 50 confirm an upward trend
 - Sustained IMI values below 50 confirm a downward trend

[image: indicator_intraday_momentum_index]

See Also

[RSI Intraday](#)[IntensityIndex](#) [Momentum](#) [RelativeMomentumIndex](#)

Jurik Moving Average (JMA)

JMA

Jurik's Moving Average (JMA) is an indicator that is a variation of the moving average. The curve of the indicator is characterized by good smoothing, as well as minimal lead time after the end of price movements and minimal lag from strong price movements.

To use the indicator, the [JurikMovingAverage](#) class should be used.

Indicator Settings.

- Phase - determines the speed at which the moving average will unfold;
- Period - determines the period of the moving average.

[image: IndicatorJurikMovingAverage]

See Also

[KAMA](#)

Kase Peak Oscillator (KPO)

KPO

Kase Peak Oscillator (KPO) is a technical indicator developed by Celia Kase that combines momentum and volatility to identify potential market peaks and troughs.

To use the indicator, you need to use the [KasePeakOscillator](#) class.

Description

The Kase Peak Oscillator (KPO) is a tool for determining overbought and oversold market conditions and identifying potential reversal points. It was developed by trader and engineer Celia Kase as part of her trading methodology.

KPO is based on the concept that market peaks and troughs form when the price movement momentum begins to exhaust. The oscillator uses a combination of momentum and volatility indicators to identify these key turning points.

The indicator is a dimensionless oscillator that fluctuates around the zero line. Positive values indicate upward momentum, while negative values indicate downward momentum. Extreme oscillator values often coincide with peaks and troughs on the price chart.

Parameters

The indicator has the following parameters:

- **ShortPeriod** - short period for momentum calculation (default value: 10)
- **LongPeriod** - long period for momentum calculation (default value: 30)

Calculation

Kase Peak Oscillator calculation involves several steps:

1. Calculate short-term momentum based on the short period:

```
Short Momentum = EMA(Price, ShortPeriod) - EMA(Price, ShortPeriod)[previous]
```

2. Calculate long-term momentum based on the long period:

```
Long Momentum = EMA(Price, LongPeriod) - EMA(Price, LongPeriod)[previous]
```

3. Calculate current volatility:

```
Volatility = ATR(ShortPeriod)
```

4. Normalize momentum relative to volatility:

```
Normalized Short Momentum = Short Momentum / Volatility  
Normalized Long Momentum = Long Momentum / Volatility
```

5. Final KPO calculation:

```
KPO = Normalized Short Momentum - Normalized Long Momentum
```

Where:

- Price - usually closing price
- EMA - exponential moving average
- ATR - average true range
- ShortPeriod - short calculation period
- LongPeriod - long calculation period

Interpretation

The Kase Peak Oscillator can be interpreted as follows:

1. Zero Line Crossovers:

- Crossing from bottom to top can be viewed as a bullish signal
- Crossing from top to bottom can be viewed as a bearish signal

2. Extreme Values:

- High positive values may indicate market overbought conditions and a potential downward reversal
- High negative values may indicate market oversold conditions and a potential upward reversal

3. Divergences:

- Bullish Divergence (price forms a new low, while KPO forms a higher low) may signal an upcoming upward reversal
- Bearish Divergence (price forms a new high, while KPO forms a lower high) may signal an upcoming downward reversal

4. Component Crossovers:

- When short-term momentum crosses long-term momentum from bottom to top, it can be viewed as a bullish signal
- When short-term momentum crosses long-term momentum from top to bottom, it can be viewed as a bearish signal

5. Acceleration and Deceleration:

- Increased KPO slope indicates momentum acceleration
- Decreased KPO slope indicates momentum deceleration, which may precede a reversal

6. Combining with Other Indicators:

- KPO is often used with other technical indicators to confirm signals
- Particularly effective when combined with trend indicators and support/resistance levels

[image: indicator_kase_peak_oscillator]

See Also

[MomentumOscillator](#) [MACD](#) [PrettyGoodOscillator](#) [ATR](#)

Kaufman Efficiency Ratio (KER)

KER

Kaufman Efficiency Ratio (KER) is a technical indicator developed by Perry Kaufman that measures price movement efficiency by comparing directional price movement to overall volatility.

To use the indicator, you need to use the [KaufmanEfficiencyRatio](#) class.

Description

The Kaufman Efficiency Ratio (KER) evaluates how "efficiently" price moves in a specific direction compared to the total path it travels. It represents the ratio of net directional price movement to the sum of all price changes over a specific period.

KER was developed by Perry Kaufman and was originally used as a component of the Adaptive Moving Average (KAMA). However, KER itself is a valuable tool that helps determine whether the market is in a trending or oscillating state.

KER values oscillate between 0 and 1:

- Values close to 1 indicate highly efficient price movement (strong trend)
- Values close to 0 indicate inefficient price movement (sideways market or high volatility)

Parameters

The indicator has the following parameters:

- **Length** - period for efficiency calculation (default value: 10)

Calculation

Kaufman Efficiency Ratio calculation involves the following steps:

1. Calculate directional movement (net change) over the period:

```
Direction = |Price[current] - Price[current - Length]|
```

2. Calculate total movement (sum of all changes) over the period:

```
Volatility = Sum(|Price[i] - Price[i-1]|) for i from (current - Length + 1) to current
```

3. Calculate efficiency ratio:

```
KER = Direction / Volatility
```

Where:

- Price - usually closing price
- Length - calculation period
- || - denotes absolute value

If Volatility is zero (which is unlikely), KER is set to zero to avoid division by zero.

Interpretation

The Kaufman Efficiency Ratio can be interpreted as follows:

1. Efficiency Levels:

- High KER values (>0.6) indicate a strong trend
- Medium KER values ($0.3-0.6$) indicate a moderate trend
- Low KER values (<0.3) indicate a sideways market or high volatility

2. KER Changes:

- KER growth may signal the formation or strengthening of a trend
- KER decline may signal trend weakening or transition to sideways movement

3. Trading Strategies:

- During high efficiency periods (high KER), trend strategies are preferable
- During low efficiency periods (low KER), range trading strategies are preferable

4. Signal Filtering:

- KER can be used to filter signals from other indicators
- Trend indicator signals are more reliable at high KER
- Oscillator signals are more reliable at low KER

5. Market Condition Adaptation:

- KER allows adapting trading strategies to changing market conditions
- Traders can dynamically adjust parameters of other indicators based on KER values

6. Change Precursor:

- Sharp KER changes often precede new price movements
- KER decline after a period of high values may warn of a potential trend reversal

[image: indicator_kaufman_efficiency_ratio]

See Also

[KAMA ADX VHF VIDYA](#)

Kaufman Adaptive Moving Average (KAMA)

KAMA

Kaufman's Adaptive Moving Average (AMA, KAMA, AMkA) was developed by Perry Kaufman to account for market noise and volatility. The KAMA indicator can be used to identify reversal points in time, determine the overall trend, and filter price movements.

To use the indicator, the [KaufmanAdaptiveMovingAverage](#) class should be used.

Indicator Settings.

- Fast MA - the fast smoothing constant;
- Slow MA - the slow smoothing constant;
- Period - the period of Kaufman's moving average.

[image: IndicatorKaufmanAdaptiveMovingAverage]

See Also

[KAMA](#)

Kalman Filter (KF)

KalmanFilter

Kalman Filter is a recursive algorithm that estimates the underlying state of a system from noisy observations.

To use the indicator, you need to use the [KalmanFilter](#) class.

Description

The Kalman Filter applies a prediction-correction cycle to smooth price data and reduce market noise. It adapts dynamically as new information becomes available, making it useful for tracking trends in volatile markets.

Parameters

- **ProcessNoise** – expected variance in the underlying process.
- **ObservationNoise** – expected variance in the observed data.

Calculation

At each step the filter performs:

1. **Prediction** of the next state based on the previous estimate.
2. **Update** of this prediction using the newest price observation and the noise estimates.

This yields an optimized estimate that reacts quickly to price changes while filtering out short-term fluctuations.

[image: indicator_kalman_filter]

See Also

[Kaufman Adaptive Moving Average](#) [Adaptive Laguerre Filter](#)

Keltner Channels (KC)

KC

Keltner Channels (KC) is a technical indicator consisting of a set of volatility bands that uses an exponential moving average (EMA) as the central line and the average true range (ATR) to determine channel width.

To use the indicator, you need to use the [KeltnerChannels](#) class.

Description

Keltner Channels consist of three lines:

1. **Central Line:** typically represented by a 20-period EMA
2. **Upper Band:** central line plus a multiplier of ATR
3. **Lower Band:** central line minus the same ATR multiplier

The indicator was developed by Chester Keltner in the 1960s and later modified by Linda Raschke, who replaced the simple moving average (SMA) with an exponential moving average (EMA) and began using ATR instead of the High-Low range to calculate channel width.

Keltner Channels help traders determine trend direction and potential support and resistance levels. They are also used to identify overbought and oversold conditions when price touches or breaks through the upper or lower band, respectively.

Parameters

The indicator has the following parameters:

- **Length** - period for calculating EMA and ATR (default value: 20)
- **Multiplier** - multiplier for ATR, determining channel width (default value: 2.0)

Calculation

Keltner Channels calculation involves the following steps:

1. Calculate exponential moving average:

```
Middle Line = EMA(Price, Length)
```

2. Calculate average true range:

```
ATR = Average True Range over Length period
```

3. Calculate upper and lower bands:

```
Upper Band = Middle Line + (Multiplier * ATR)  
Lower Band = Middle Line - (Multiplier * ATR)
```

Where:

- Price - usually closing price
- EMA - exponential moving average
- ATR - average true range

- Length - period for EMA and ATR calculation
- Multiplier - multiplier determining channel width

Interpretation

Keltner Channels can be interpreted as follows:

1. Trend Direction:

- When all three lines are pointing upward, this indicates an upward trend
- When all three lines are pointing downward, this indicates a downward trend
- Horizontal line movement indicates a sideways trend

2. Breakouts:

- Price breaking above the upper band may indicate strong upward momentum
- Price breaking below the lower band may indicate strong downward momentum
- Breakouts are often used as entry signals in the direction of the breakout

3. Returns to Middle Line:

- After breaking the upper or lower band, price often returns to the middle line
- The middle line can serve as a support or resistance level

4. Overbought and Oversold Conditions:

- Price near or beyond the upper band may indicate overbought conditions
- Price near or beyond the lower band may indicate oversold conditions
- In trending markets, price can remain in "extreme" zones for extended periods

5. Channel Contraction and Expansion:

- Channel narrowing (decreasing distance between bands) indicates decreased volatility, often preceding a strong price movement
- Channel expansion indicates increased volatility

6. Trading Strategies:

- "Edge to Middle" strategy: opening a position when price touches the upper or lower band, targeting the middle line
- Breakout strategy: opening a position when price breaks the upper or lower band, expecting continued movement in the same direction
- "Middle to Edge" strategy: opening a position when price bounces from the middle line, targeting the upper or lower band

[image: indicator_keltner_channels]

See Also

[BollingerBands](#) [DonchianChannels](#) [EMA](#) [ATR](#)

Klinger Volume Oscillator (KVO)

KVO

Klinger Volume Oscillator (KVO) is a technical indicator developed by Stephen Klinger that uses volume and price to identify long-term trends and short-term reversals in the market.

To use the indicator, you need to use the [KlingerVolumeOscillator](#) class.

Description

The Klinger Volume Oscillator (KVO) was created by Stephen Klinger to measure the divergence between volume and price. The indicator is based on the concept that price movement is confirmed by volume. KVO seeks to determine not only the trend direction but also its strength and potential reversal points.

KVO combines price information with volume using a Volume Force indicator that considers both the direction and magnitude of price movement, as well as trading volume. It then applies exponential moving averages (EMA) with two different periods to this money flow and calculates the difference between them.

The indicator is an oscillator that fluctuates above and below the zero line. Positive KVO values indicate that buyers control the market, while negative values indicate that sellers have an advantage.

Parameters

The indicator has the following parameters:

- **ShortPeriod** - period for calculating short EMA (default value: 34)
- **LongPeriod** - period for calculating long EMA (default value: 55)

Calculation

Klinger Volume Oscillator calculation involves several steps:

1. Determine trend for each period:

```
Trend = +1, if (High + Low + Close) > (High[previous] + Low[previous] + Close[previous])
Trend = -1, otherwise
```

2. Calculate Volume Force indicator:

```
Volume Force = Volume * Trend * abs(2 * ((Close - Low) - (High - Close)) / (High - Low))
```

If (High - Low) is zero, Volume Force is set to volume multiplied by trend.

3. Calculate EMA for two periods:

```
Short EMA = EMA(Volume Force, ShortPeriod)
Long EMA = EMA(Volume Force, LongPeriod)
```

4. Final KVO calculation:

```
KVO = Short EMA - Long EMA
```

5. Calculate signal line (optional):

```
Signal Line = EMA(KVO, 13)
```

Where:

- High, Low, Close - highest, lowest, and closing prices
- Volume - trading volume
- EMA - exponential moving average
- ShortPeriod - period for short EMA
- LongPeriod - period for long EMA

Interpretation

The Klinger Volume Oscillator can be interpreted as follows:

1. **Zero Line Crossovers:**
 - KVO crossing the zero line from bottom to top can be viewed as a bullish signal
 - KVO crossing the zero line from top to bottom can be viewed as a bearish signal
2. **Signal Line Crossovers:**
 - KVO crossing the signal line from bottom to top can be viewed as a bullish entry signal
 - KVO crossing the signal line from top to bottom can be viewed as a bearish entry signal
3. **Divergences:**
 - Bullish Divergence: price forms a new low, while KVO forms a higher low
 - Bearish Divergence: price forms a new high, while KVO forms a lower high
4. **Trend Confirmation:**
 - Positive KVO values confirm an upward trend
 - Negative KVO values confirm a downward trend
5. **Trend Strength:**
 - Increasing KVO value (both positive and negative) indicates strengthening of the current trend
 - Decreasing KVO value indicates weakening of the current trend
6. **Potential Reversals:**
 - Extreme KVO values may indicate market overbought or oversold conditions and potential reversal
 - Slowdown in KVO rise or fall may precede a trend reversal
7. **Volume and Price:**
 - KVO allows assessment of price and volume movement consistency
 - Strong volume in the trend direction leads to more extreme KVO values

[image: indicator_klinger_volume_oscillator]

See Also

[OBV](#) [ChaikinMoneyFlow](#) [ADL](#) [ForceIndex](#)

Know Sure Thing (KST)

KST

Know Sure Thing (KST) is a technical indicator developed by Martin Pring that represents the sum of four smoothed rates of change (ROC) with different periods to identify long-term market cycles.

To use the indicator, you need to use the [KnowSureThing](#) class.

Description

The Know Sure Thing (KST) indicator is an oscillator developed by Martin Pring to identify trends by measuring price momentum across various time periods. The indicator combines four rate of change (ROC) measurements with different periods, giving more significance to longer periods.

KST is based on the theory that market cycles of different durations collectively influence price movement. By combining ROC from different periods, KST aims to identify long-term cyclical trends and determine potential reversal points.

The indicator is typically accompanied by a signal line (moving average of KST), and their crossovers can be used to generate trading signals.

Calculation

KST indicator calculation involves the following steps:

1. Calculate four rate of change (ROC) measurements with different periods:

```
ROC1 = ((Close / Close[n1 periods ago]) - 1) * 100
ROC2 = ((Close / Close[n2 periods ago]) - 1) * 100
ROC3 = ((Close / Close[n3 periods ago]) - 1) * 100
ROC4 = ((Close / Close[n4 periods ago]) - 1) * 100
```

2. Smooth each ROC using a simple moving average (SMA):

```
RCMA1 = SMA(ROC1, m1)
RCMA2 = SMA(ROC2, m2)
RCMA3 = SMA(ROC3, m3)
RCMA4 = SMA(ROC4, m4)
```

3. Weighted summation to obtain KST:

```
KST = (RCMA1 * 1) + (RCMA2 * 2) + (RCMA3 * 3) + (RCMA4 * 4)
```

4. Calculate signal line:

```
Signal Line = SMA(KST, signal period)
```

Where:

- Close - closing price
- n1, n2, n3, n4 - periods for ROC calculation (default values: 10, 15, 20, 30)
- m1, m2, m3, m4 - periods for ROC smoothing (default values: 10, 10, 10, 15)
- signal period - signal line period (default value: 9)

Interpretation

The KST indicator can be interpreted as follows:

1. **Zero Line Crossovers:**

- When KST crosses the zero line from bottom to top, it can be viewed as a bullish signal
- When KST crosses the zero line from top to bottom, it can be viewed as a bearish signal

2. **Signal Line Crossovers:**

- When KST crosses the signal line from bottom to top, it can be viewed as a bullish signal (more sensitive than zero line crossing)
- When KST crosses the signal line from top to bottom, it can be viewed as a bearish signal (more sensitive than zero line crossing)

3. **Divergences:**

- Bullish Divergence: price forms a new low, while KST forms a higher low
- Bearish Divergence: price forms a new high, while KST forms a lower high

4. **Extreme Values:**

- High positive KST values may indicate market overbought conditions
- High negative KST values may indicate market oversold conditions

5. **Movement Direction:**

- Rising KST trend indicates an overall bullish market sentiment
- Falling KST trend indicates an overall bearish market sentiment

6. **Trend Confirmation:**

- KST can be used to confirm signals from other indicators
- Consistency between KST direction and price confirms the strength of the current trend

7. **Market Sentiment Analysis:**

- Positive KST values indicate predominance of bullish sentiment
- Negative KST values indicate predominance of bearish sentiment

[image: indicator_kst]

See Also

[ROC](#) [MACD](#) [Momentum](#) [RSI](#)

Laguerre RSI (LRSI)

LRSI

Laguerre RSI (LRSI) is a technical indicator based on Laguerre filter mathematical principles, developed by John Ehlers as an advanced version of the traditional Relative Strength Index (RSI).

To use the indicator, you need to use the [LaguerreRSI](#) class.

Description

Laguerre RSI (LRSI) is an innovative oscillator that uses Laguerre filter mathematics to create a more sensitive and less lagging indicator compared to traditional RSI. John Ehlers developed this indicator to address the signal delay problem inherent in many technical indicators.

LRSI combines Laguerre polynomial principles with the Relative Strength Index concept. This allows the indicator to respond more quickly to trend changes and form clearer trading signals. Like traditional RSI, LRSI oscillates between 0 and 1 (or 0 to 100 when multiplied by 100), but has a less noisy structure and more distinct turns.

The main advantage of LRSI is its ability to quickly identify trend changes while maintaining signal stability. This makes it particularly useful for short-term trading and determining entry and exit points.

Parameters

The indicator has the following parameters:

- **Gamma** - filtering coefficient (default value: 0.4, range from 0.1 to 0.9)

The Gamma parameter determines the degree of filtering and affects the indicator's sensitivity. Lower Gamma values result in a smoother and less sensitive indicator, while higher values make the indicator more sensitive to price changes, but potentially more noisy.

Calculation

Laguerre RSI calculation involves several steps:

1. Initialize four Laguerre filter values (L0, L1, L2, L3) on first run:

```
L0 = L1 = L2 = L3 = 0
```

2. Update filter values for each new price:

```
L0_new = (1 - Gamma) * Price + Gamma * L0_old  
L1_new = -Gamma * L0_new + L0_old + Gamma * L1_old  
L2_new = -Gamma * L1_new + L1_old + Gamma * L2_old  
L3_new = -Gamma * L2_new + L2_old + Gamma * L3_old
```

3. Calculate "cumulative multiplication" of filtered values:

```
CU = (L0_new + L1_new + L2_new + L3_new) / 4
```

4. Separate into "up" and "down" components:

```
If CU >= CU_old, then:  
    UP = CU - CU_old  
    DN = 0  
Otherwise:
```

```
UP = 0
DN = CU_old - CU
```

5. Final LRSI calculation:

```
LRSI = UP / (UP + DN)
```

If (UP + DN) is zero, LRSI is set to the previous value.

Where:

- Price - input price (usually closing price)
- Gamma - filtering parameter
- CU - "cumulative multiplication"

Interpretation

Laguerre RSI is interpreted similarly to traditional RSI, but with its increased sensitivity:

1. **Overbought and Oversold Levels:**

- Values above 0.8 (or 80) are typically considered overbought
- Values below 0.2 (or 20) are typically considered oversold
- Due to Laguerre filter characteristics, these levels may be adjusted based on market volatility

2. **Centerline Crossovers:**

- Crossing the 0.5 (or 50) level from bottom to top can be viewed as a bullish signal
- Crossing the 0.5 (or 50) level from top to bottom can be viewed as a bearish signal

3. **Divergences:**

- Bullish Divergence: price forms a new low, while LRSI forms a higher low
- Bearish Divergence: price forms a new high, while LRSI forms a lower high

4. **Rebounds from Extremes:**

- LRSI reversal from overbought or oversold levels can serve as a market entry signal

5. **Trend Confirmation:**

- LRSI values above 0.5 confirm an upward trend
- LRSI values below 0.5 confirm a downward trend

6. **Gamma Parameter Tuning:**

- For faster signals - increase Gamma (closer to 0.9)
- For smoother signals - decrease Gamma (closer to 0.1)

[image: indicator_laguerre_rsi]

See Also

[RSI AdaptiveLaguerreFilter](#) [ConnorsRSI](#)

Linear Regression (LRC)

LRC

Linear Regression Curve (LRC) is a type of moving average based on the linear regression equation. The calculation result is a straight line with the best fit for different prices for the period. The end point of this line is used to plot the LRC.

To use the indicator, you must use the [LinearReg](#) class.

[image: IndicatorLinearReg]

Recommended content

[Logging](#)

Linear Regression Gradient (LRS)

Linear Regression Slope

Interpretation of the **Linear Regression Slope** indicator shows the slope value of regression lines, which include the current price bar and the previous $n-1$ price bar (where n = regression periods).

To use the indicator, you must use the [LinearRegSlope](#) class.

[image: IndicatorLinearRegSlope]

Recommended content

[Logging](#)

Linear Regression Forecast (LRF)

Linear Regression Forecast

Linear Regression Forecast (LRF) predicts future price values based on linear regression over a selected period.

To use the indicator, you must use the [LinearRegressionForecast](#) class.

[image: IndicatorLinearRegressionForecast]

Recommended content

[Linear Regression](#)

R-squared

R-Squared

R-Squared in Linear Regression (Linear Regression R-Squared) is a technical indicator that measures how well linear regression approximates price data and determines the strength of the market trend.

To use the indicator, you need to use the [LinearRegRSquared](#) class.

Description

R-Squared in Linear Regression (R^2) is a statistical measure used to evaluate the degree of correspondence between price data and a linear regression line drawn through these data. In the context of technical analysis, R^2 shows how well the current price movement corresponds to a linear trend.

R^2 values range from 0 to 1 (or 0% to 100%):

- A value close to 1 (100%) indicates that prices align very well along the trend line, signifying a strong trend
- A value close to 0 indicates the absence of a linear trend and is characteristic of sideways, chaotic, or cyclical markets

The indicator helps traders distinguish between periods of strong trends and periods of consolidation or sideways movements, allowing them to choose an appropriate trading strategy.

Parameters

The indicator has the following parameters:

- **Length** - period for linear regression calculation (default value: 14)

Calculation

R-Squared in Linear Regression calculation involves the following steps:

1. Constructing a linear regression line for price data over the Length period:

$$y = a + b \cdot x$$

Where:

- y - price (dependent variable)
- x - period sequence number (independent variable)
- a - free term (y-axis intercept)
- b - slope coefficient

2. Calculating the sum of squared deviations from regression (SSE):

$$SSE = \text{Sum}((\text{Actual Price} - \text{Predicted Price})^2)$$

Where:

- Actual Price - actual price
- Predicted Price - predicted price from the regression equation

3. Calculating the total sum of squares (SST):

$$SST = \text{Sum}((\text{Actual Price} - \text{Average Price})^2)$$

Where Average Price is the mean price over the Length period

4. Calculating R^2 :

$$R^2 = 1 - (SSE / SST)$$

Interpretation

R-Squared in Linear Regression can be interpreted as follows:

1. Trend Strength Assessment:

- Values above 0.7 (70%) indicate a strong trend
- Values between 0.3 and 0.7 (30-70%) indicate a moderate trend
- Values below 0.3 (30%) indicate a weak trend or no trend

2. Trading Strategy Selection:

- With high R^2 values (strong trend), trend-following strategies are effective
- With low R^2 values (sideways movement), range trading strategies are effective

3. Transition Point Search:

- Increasing R^2 may signal the formation of a new trend
- Decreasing R^2 may signal trend weakening and possible consolidation or reversal

4. Signal Filtering:

- Signals from trend indicators are more reliable at high R^2 values
- Oscillator signals are more reliable at low R^2 values

5. Combining with Other Indicators:

- R^2 is often used to determine market mode, after which appropriate indicators are applied
- For example, use moving averages with high R^2 , and stochastic oscillator with low R^2

6. Market Predictability Assessment:

- High R^2 values indicate more predictable price movement in the short term
- Low R^2 values indicate more chaotic, unpredictable movement

7. Timeframes:

- R^2 can yield different results on different timeframes
- Comparing R^2 across timeframes can provide additional information about market structure

[image: indicator_linear_reg_r_squared]

See Also

[LinearRegression](#) [StandardError](#) [ChoppinessIndex](#)

Lowest Value (Lowest)

Lowest

The indicator indicates the **Minimum value for the period**.

To use the indicator, you must use the [Lowest](#) class.

[image: IndicatorLowest]

Recommended content

[Lowest](#)

Lunar Phase Indicator (LP)

LP

Lunar Phase (LP) is an unconventional technical indicator based on astronomical data about moon phases to analyze the potential influence of lunar cycles on financial markets.

To use the indicator, you need to use the [LunarPhase](#) class.

Description

The Lunar Phase (LP) indicator is an unusual technical analysis tool that uses information about the current moon phase to potentially forecast market trends. The indicator is based on the theory that lunar cycles may influence market participant behavior and, consequently, financial instrument price movements.

The lunar cycle lasts approximately 29.53 days and is traditionally divided into four main phases:

1. New Moon
2. First Quarter (Waxing Moon)
3. Full Moon
4. Last Quarter (Waning Moon)

The indicator tracks the current moon phase and represents this information as a numerical value from 0 to 1, where:

- 0 corresponds to the new moon
- 0.25 corresponds to the first quarter
- 0.5 corresponds to the full moon
- 0.75 corresponds to the last quarter

Calculation

Lunar Phase indicator calculation is based on astronomical algorithms for determining the current moon phase:

1. Determine the number of days passed since the start of the lunar cycle (new moon):

```
Current_Cycle_Position = (Current_Date - Last_New_Moon_Date) % 29.53
```

2. Convert this value to a phase from 0 to 1:

```
Moon_Phase = Current_Cycle_Position / 29.53
```

The resulting value is the Lunar Phase (LP) indicator.

Interpretation

Lunar Phase indicator interpretation can vary, as it is an unconventional technical analysis tool. However, there are some commonly accepted approaches:

1. **Potential Reversal Points:**
 - Some traders believe that new and full moons may coincide with market reversal points
 - Transitions between main phases can also be viewed as potential periods of increased volatility

2. Market Sentiment Cycles:

- There is a theory that moon phases can influence mass psychology and, consequently, market sentiment
- Some studies suggest that the full moon period may lead to more emotional and irrational trader behavior

3. Volatility Correlation:

- Some studies show that volatility may be higher during full moon and new moon periods
- This can be used when adjusting parameters of other indicators and strategies

4. Seasonal Patterns:

- LP can be used in combination with seasonal pattern analysis to identify potential market periodicities

5. Signal Filtering:

- Some traders use LP as an additional filter for their trading strategies
- For example, they may avoid certain types of trades during specific lunar phases if historical statistics show low efficiency

6. Combining with Other Indicators:

- LP is typically not used as a standalone tool for making trading decisions
- It is recommended to combine it with traditional technical indicators to confirm signals

It is important to note that there is insufficient scientific evidence of direct lunar phase influence on financial markets, and many professional traders approach such tools skeptically. However, some market participants find value in including LP in their analytical arsenal.

[image: indicator_lunar_phase]

See Also

[SineWave HarmonicOscillator](#)

Market Facilitation Index (MFI)

Market Facilitation Index

Market Facilitation Index (MFI) is an indicator that assesses the market's readiness to move the price. It analyzes the ratio of price changes to volume changes over a certain period of time to measure the market's response to volume changes. It calculates MFI using the formula $(High - Low) / Volume$, providing an understanding of market dynamics. The absolute values of the indicator cannot provide any trading signals, unlike its dynamics relative to the volume dynamics.

To use the indicator, the [MarketFacilitationIndex](#) class should be used.

[image: IndicatorMarketFacilitationIndex]

See Also

[Mean Deviation](#)

Market Meanness Index (MMI)

MMI

Market Meanness Index (MMI) is a technical indicator developed to determine whether the market is in a trending or sideways (chaotic) state.

To use the indicator, you need to use the [MarketMeannessIndex](#) class.

Description

The Market Meanness Index (MMI) is a tool that helps traders determine the nature of the current market - whether it is trending or sideways. The name "Meanness" reflects the idea that the market sometimes behaves "meanly" or unpredictably towards traders, especially when in a sideways movement.

MMI is based on counting the number of price value pairs (usually closing prices) that do not follow a simple linear pattern, and their ratio to the total number of analyzed pairs. The indicator measures the "chaos" or "randomness" of price movement over a specific period.

The index oscillates from 0 to 100:

- Low values (usually below 50) indicate predominance of trending movement
- High values (usually above 50) indicate predominance of sideways or chaotic movement

Parameters

The indicator has the following parameters:

- **Length** - calculation period (default value: 20)

Calculation

Market Meanness Index calculation involves the following steps:

1. Create a set of consecutive closing price (Close) pairs within the given Length period.
2. Count the number of "non-sequential" pairs. A pair is considered non-sequential if it does not follow the linear pattern typical of a trend. If two consecutive pairs (P1, P2) and (P2, P3) have opposite directions (different difference signs), the pair is considered non-sequential.
3. Calculate MMI as a percentage ratio:

$$\text{MMI} = (\text{Number of non-sequential pairs} / \text{Total number of pairs}) * 100$$

Formally, this can be represented as:

1. For each trio of consecutive prices (Close[i-2], Close[i-1], Close[i]), check:
 - If $(\text{Close}[i-1] - \text{Close}[i-2]) * (\text{Close}[i] - \text{Close}[i-1]) < 0$, the pair is considered non-sequential
 - Count the total number of such pairs
2. $\text{MMI} = (\text{Number of non-sequential pairs} / (\text{Length} - 2)) * 100$

Interpretation

The Market Meanness Index can be interpreted as follows:

1. **Indicator Levels:**

- $MMI > 50$: Market is in a sideways or chaotic state
 - $MMI < 50$: Market is in a trending state
 - The closer MMI is to 100, the more chaotic the market
 - The closer MMI is to 0, the more pronounced the trend
2. **Trading Strategy Application:**
 - When MMI is high (>50), use strategies oriented towards sideways market (e.g., range trading, oscillators)
 - When MMI is low (<50), use trend strategies (e.g., trend following)
 3. **Dynamics of Changes:**
 - Decrease in MMI from high levels may signal the formation of a new trend
 - Increase in MMI from low levels may indicate trend completion and transition to consolidation
 4. **Extreme Values:**
 - Very low values ($MMI < 20$) may indicate a strong trend, but also potential overbought/oversold conditions
 - Very high values ($MMI > 80$) indicate an extremely chaotic market where it's difficult to apply any strategies
 5. **Signal Filtering:**
 - MMI is often used as a filter for other indicators:
 - Trend indicator signals (MA, MACD) are more reliable at low MMI
 - Oscillator signals (RSI, Stochastic) are more reliable at high MMI
 6. **Combining with Other Indicators:**
 - MMI works well in combination with ADX (Average Directional Index)
 - Low MMI and high ADX confirm a strong trend
 - High MMI and low ADX confirm a sideways market
 7. **Timeframes:**
 - MMI can be used on different timeframes to determine market character
 - Long-term MMI helps determine the market's primary state
 - Short-term MMI helps choose an appropriate strategy for current conditions

[image: indicator_market_meanness_index]

See Also

[ChoppinessIndex](#) [ADX](#) [VHF](#) [BalanceOfPower](#)

Mass Index (MI)

MI

Mass Index (MI) is a technical indicator developed by Donald Dorsey that identifies potential trend reversals by analyzing price range expansion and contraction.

To use the indicator, you need to use the [MassIndex](#) class.

Description

The Mass Index (MI) is a technical analysis tool that helps detect potential trend reversals by tracking changes in the price range (difference between maximum and minimum prices). The indicator was developed by Donald Dorsey based on the assumption that trend reversals are typically preceded by expansion and subsequent contraction of the price range.

MI measures volatility using exponential moving averages (EMA) of the price range. It does not predict the reversal direction, only its probability. This is why MI is often used in conjunction with other directional indicators.

The main concept is that when the mass index reaches a certain threshold and then falls below this level, the likelihood of a current trend reversal increases.

Parameters

The indicator has the following parameters:

- **Length** - main calculation period (default value: 25)
- **EmaLength** - period for price range EMA (default value: 9)

Calculation

Mass Index calculation involves the following steps:

1. Calculate High-Low range for each period:

$$\text{Range} = \text{High} - \text{Low}$$

2. Calculate 9-period EMA of the range:

$$\text{EMA1} = \text{EMA}(\text{Range}, \text{EmaLength})$$

3. Calculate 9-period EMA of the range's EMA:

$$\text{EMA2} = \text{EMA}(\text{EMA1}, \text{EmaLength})$$

4. Calculate the ratio:

$$\text{Ratio} = \text{EMA1} / \text{EMA2}$$

5. Sum ratios over 25 periods:

$$\text{MI} = \text{Sum}(\text{Ratio over last Length periods})$$

Where:

- High - period's highest price
- Low - period's lowest price
- EMA - exponential moving average
- Length - summation period (usually 25)
- EmaLength - EMA period (usually 9)

Interpretation

The Mass Index is interpreted as follows:

1. "Reversal Hump":

- The classic "reversal hump" signal forms when the mass index rises above 27 and then falls below 26.5
- This pattern indicates a possible trend reversal, though it does not predict its direction

2. Index Levels:

- Values above 27 indicate price range expansion and increased volatility
- High values followed by a decline may precede a trend reversal
- Low values (below 20) indicate price range contraction and reduced volatility

3. Trend Direction:

- The mass index does not indicate trend direction or its reversal
- Additional indicators or analysis methods are needed to determine direction (e.g., moving averages or support/resistance levels)

4. Divergences:

- Divergences between price and mass index are less significant than the "reversal hump" pattern
- However, discrepancies between price's new highs/lows and declining mass index highs/lows may indicate trend weakening

5. Combining with Other Indicators:

- Mass index works best when combined with trend direction indicators
- Popular combinations include EMA, MACD, or RSI to determine potential reversal direction

6. Volatility Changes:

- Sharp mass index increase indicates significant price range expansion, which may precede a strong movement
- Gradual index decline suggests range narrowing and possible consolidation

7. Parameter Tuning:

- Standard parameters (9 for EMA, 25 for summation) work well on most timeframes
- Reducing periods can create faster signals but may increase false signals

[image: indicator_mass_index]

See Also

[ATR](#) [BollingerBands](#) [ChoppinessIndex](#) [TrueRange](#)

McClellan Oscillator (MCO)

MCO

McClellan Oscillator (MCO) is a technical indicator developed by Sherman and Marian McClellan that measures market breadth by analyzing the difference between moving averages of advancing and declining stocks.

To use the indicator, you need to use the [McClellanOscillator](#) class.

Description

The McClellan Oscillator (MCO) is one of the most well-known market breadth indicators that helps assess the overall market condition and identify potential reversal points. Developed in 1969, it has since become a crucial tool for many technical analysts.

MCO is based on analyzing the ratio between the number of rising and falling stocks in the market. The indicator calculates the difference between 19-period and 39-period exponential moving averages of net advances (difference between the number of advancing and declining stocks).

The McClellan Oscillator is particularly useful for:

- Determining the overall market direction
- Identifying overbought and oversold conditions
- Identifying potential reversal points
- Confirming the strength or weakness of the current trend

Calculation

McClellan Oscillator calculation involves the following steps:

1. Calculate Net Advances for each trading day:

$$\text{Net Advances} = \text{Advances} - \text{Declines}$$

where Advances is the number of rising stocks, Declines is the number of falling stocks.

2. Calculate 19-period exponential moving average of Net Advances:

$$\text{EMA19} = \text{EMA}(\text{Net Advances}, 19)$$

3. Calculate 39-period exponential moving average of Net Advances:

$$\text{EMA39} = \text{EMA}(\text{Net Advances}, 39)$$

4. Compute McClellan Oscillator as the difference between these two EMAs:

$$\text{MCO} = \text{EMA19} - \text{EMA39}$$

Interpretation

The McClellan Oscillator can be interpreted as follows:

1. **Zero Line Crossovers:**

- MCO crossing the zero line from bottom to top can be viewed as a bullish signal, indicating a

potential upward trend start

- MCO crossing the zero line from top to bottom can be viewed as a bearish signal, indicating a potential downward trend start

2. Extreme Values:

- Values above +100 often indicate market overbought conditions
- Values below -100 often indicate market oversold conditions
- Extreme values (+150/-150 and above/below) may signal a potential market reversal

3. Divergences:

- Bullish Divergence: index forms a new low, while MCO forms a higher low
- Bearish Divergence: index forms a new high, while MCO forms a lower high

4. Market Breadth State:

- Positive MCO values indicate that most stocks in the market are rising
- Negative MCO values indicate that most stocks in the market are falling

5. Movement Acceleration/Deceleration:

- Increasing MCO values (positive or negative) indicate acceleration of the current market movement
- Decreasing MCO values indicate deceleration of the current market movement

6. Combining with McClellan Summation Index:

- McClellan Summation Index (MSI) is the cumulative sum of MCO values
- MSI crossing zero can confirm MCO signals and indicate long-term trend changes

7. Bullish/Bearish Patterns:

- "Bullish Tail" - rapid MCO decline followed by quick recovery, often indicating a potential market bottom
- "Bearish Tail" - rapid MCO rise followed by quick decline, often indicating a potential market top

[image: indicator_mcclellan_oscillator]

See Also

[HighLowIndex](#)

McGinley Dynamic (MGD)

MGD

McGinley Dynamic (MGD) is a technical indicator developed by John R. McGinley that represents an advanced form of moving average, automatically adjusting its speed based on market velocity changes.

To use the indicator, you need to use the [McGinleyDynamic](#) class.

Description

McGinley Dynamic (MGD) was created by John McGinley to overcome some drawbacks of traditional moving averages, such as lagging and inability to adapt to market speed changes. The indicator automatically adjusts its reaction period depending on price movement speed, making it more sensitive to rapid changes and less prone to false signals.

Unlike simple and exponential moving averages, MGD incorporates a tuning constant and the ratio of price to the previous indicator value. This allows MGD to respond more quickly to significant price changes while maintaining stability during slower movements.

The main idea is that MGD "accelerates" during rapid market movements and "slows down" during consolidation periods, providing more accurate price tracking compared to traditional moving averages.

Parameters

The indicator has the following parameters:

- **Length** - calculation period (default value: 14)

Calculation

McGinley Dynamic calculation is performed recursively using the following formula:

```
MGD = MGD[previous] + (Price - MGD[previous]) / (Length * ((Price / MGD[previous])^4))
```

Where:

- Price - current price (usually closing price)
- MGD[previous] - previous indicator value
- Length - period parameter

For the initial MGD value, a simple moving average over the specified period is typically used:

```
For first calculation: MGD = SMA(Price, Length)
```

Interpretation

McGinley Dynamic can be interpreted similarly to other moving averages, but with its improved characteristics:

1. Trend Determination:

- When price is above MGD, it indicates an upward trend
- When price is below MGD, it indicates a downward trend
- A steep MGD slope indicates a strong trend

2. Price Crossovers:

- Price crossing MGD from bottom to top can be viewed as a bullish signal
- Price crossing MGD from top to bottom can be viewed as a bearish signal
- Due to its adaptive nature, these crossovers typically form earlier than with traditional moving averages

3. Multiple MGD Crossovers:

- Multiple MGDs with different periods can be used (e.g., MGD(14) and MGD(30))
- Short MGD crossing long MGD from bottom to top can be viewed as bullish trend confirmation
- Short MGD crossing long MGD from top to bottom can be viewed as bearish trend confirmation

4. Support and Resistance Levels:

- MGD often serves as a dynamic support level in an upward trend
- MGD often serves as a dynamic resistance level in a downward trend
- Multiple bounces off MGD confirm trend strength

5. Price Relationship:

- Distance between price and MGD can indicate market overbought or oversold conditions
- When price significantly deviates from MGD, it may signal a potential reversal or correction

6. Combining with Other Indicators:

- MGD works well with oscillators (RSI, Stochastic)
- Can be used as a trend filter for other trading systems

7. Length Parameter Selection:

- Smaller Length values (e.g., 8-12) make MGD more sensitive to price changes and suit short-term trading
- Larger Length values (e.g., 20-50) make MGD smoother and suit long-term trading

[image: indicator_mcginley_dynamic]

See Also

[SMA](#) [EMA](#) [DEMA](#) [HMA](#)

Mean Deviation (MeanDev)

Mean Deviation

The **Mean Deviation** indicator shows the average deviation of the price from its simple moving average (SMA) over a certain period.

To use the indicator, the [MeanDeviation](#) class should be used.

[image: IndicatorMeanDeviation]

See Also

[Median Price](#)

Median

Moving Median

The **Moving Median** indicator calculates the median of the most recent N values. Compared with moving averages it is less sensitive to outliers and preserves abrupt price changes, which makes it useful in noisy environments.

Use the [Median](#) class to access the indicator.

Description

A median filter sorts the prices inside the moving window and selects the middle value. As a result:

- Single spikes or drops do not distort the output.
- The lag is smaller than with many smoothing filters.
- The shape of the signal remains more angular, helping to capture reversals.

Parameters

- **Length** — window size used to compute the median. Larger windows deliver stronger smoothing but increase lag.

Usage

- Apply the Moving Median as an alternative to moving averages when price data contains significant noise.
- Crossovers between price and the median can be treated as trend-change signals.
- Combine the median with other filters to extract trends while keeping important price jumps.

[image: indicator_median]

See also

[SMA EMA Smoothed Moving Average](#)

Median Price (MedPr)

Median Price

Median Price indicator shows the median price for the candle.

To use the indicator, you must use the [MedianPrice](#) class.

[image: IndicatorMedianPrice]

Recommended content

[Momentum](#)

Momentum

Momentum

The **Momentum** indicator measures the magnitude of the price change of a financial instrument over a certain period. It indicates overbought and oversold moments when the curve reaches maximum or minimum values. Adding a smoothed moving average to the indicator improves the interpretation of trend changes.

To use the indicator, the [Momentum](#) class should be used.

Calculation

Momentum is defined as the ratio of today's price to the price n periods ago:

$$\text{MOMENTUM} = \text{CLOSE}(i) / \text{CLOSE}(i - n) * 100$$

where: CLOSE(i) — the closing price of the current bar; CLOSE(i - n) — the closing price of n bars back.

[image: IndicatorMomentum]

See Also

[Money Flow Index](#)

Momentum of Moving Average (MOMA)

MOMA

Momentum of Moving Average (MOMA) is a technical indicator that measures the rate of change of the moving average price, combining the concepts of momentum and moving averages.

To use the indicator, you need to use the [MomentumOfMovingAverage](#) class.

Description

Momentum of Moving Average (MOMA) is a combination of two indicators - a momentum indicator and a moving average. First, a moving average of the price series is calculated, and then the momentum (rate of change) of this moving average is measured.

The main idea of MOMA is to first smooth the price series using a moving average, thereby removing market noise, and then analyze the speed and direction of change in this smoothed curve. This allows obtaining a cleaner signal about trend momentum change than calculating momentum directly from the price.

MOMA helps determine trend strength and potential reversal points by focusing on changes in moving average dynamics, rather than the price itself. Positive MOMA values indicate an upward moving average momentum, while negative values indicate a downward momentum.

Parameters

The indicator has the following parameters:

- **Length** - period for moving average calculation (default value: 14)
- **MomentumPeriod** - period for momentum calculation (default value: 10)

Calculation

Momentum of Moving Average calculation involves the following steps:

1. Calculate moving average for the price series:

```
MA = SMA(Price, Length)
```

2. Calculate moving average momentum:

```
MOMA = MA[current] - MA[current - MomentumPeriod]
```

Where:

- Price - price (usually closing price)
- SMA - simple moving average
- Length - period for moving average
- MomentumPeriod - period for momentum calculation

Note: Other types of moving averages such as EMA (exponential moving average), WMA (weighted moving average), etc., can also be used instead of SMA.

Interpretation

Momentum of Moving Average can be interpreted as follows:

1. **Zero Line Crossovers:**

- MOMA crossing the zero line from bottom to top can be viewed as a bullish signal, indicating the start or strengthening of an upward trend
- MOMA crossing the zero line from top to bottom can be viewed as a bearish signal, indicating the start or strengthening of a downward trend

2. **Absolute Values:**

- High positive MOMA values indicate strong upward moving average momentum
- High negative MOMA values indicate strong downward moving average momentum
- Values close to zero indicate no pronounced momentum or a sideways trend

3. **Divergences:**

- Bullish Divergence: price forms a new low, while MOMA forms a higher low
- Bearish Divergence: price forms a new high, while MOMA forms a lower high

4. **Direction Change:**

- When MOMA changes direction of movement (from rising to falling or vice versa), this may signal a potential change in the moving average trend
- Such reversals often precede changes in price direction

5. **Trend Confirmation:**

- Positive MOMA values confirm an upward trend
- Negative MOMA values confirm a downward trend
- Increasing MOMA values indicate strengthening of the current trend
- Decreasing MOMA values indicate weakening of the current trend

6. **Signal Filtering:**

- MOMA can be used to filter signals from other indicators
- For example, consider only bullish signals when MOMA is positive, and only bearish signals when MOMA is negative

7. **Parameter Selection:**

- Shorter periods for Length and MomentumPeriod make MOMA more sensitive, but also more prone to false signals
- Longer periods make MOMA smoother but may lead to lagging signals

[image: indicator_momentum_of_moving_average]

See Also

[Momentum SMA EMA ROC](#)

Momentum Pinball (MP)

MP

Momentum Pinball (MP) is a technical indicator that analyzes price momentum and its changes to identify potential reversal points and trend strength in the market.

To use the indicator, you need to use the [MomentumPinball](#) class.

Description

The Momentum Pinball (MP) indicator is a specialized oscillator designed to track price momentum and identify potential reversal points. The name "pinball" reflects the indicator's ability to identify moments when price, like a ball in a pinball machine, bounces off extreme positions.

MP analyzes the relationship between current momentum and its historical extremes, determining when the market reaches overbought or oversold conditions. The indicator also helps identify moments when momentum begins to weaken, which may precede a trend reversal.

The main idea is that extreme momentum values are often unstable, and after reaching such extremes, a correction or reversal usually follows. MP helps visualize this process by tracking both momentum itself and its changes.

Parameters

The indicator has the following parameters:

- **Length** - calculation period (default value: 14)

Calculation

Momentum Pinball Indicator calculation involves the following steps:

1. Calculate base momentum as the difference between current price and price N periods ago:

```
Momentum = Price[current] - Price[current - Length]
```

2. Determine historical maximum and minimum momentum over the specified period:

```
Max_Momentum = Maximum(Momentum) over Length period  
Min_Momentum = Minimum(Momentum) over Length period
```

3. Normalize current momentum relative to historical extremes:

```
Normalized_Momentum = (Momentum - Min_Momentum) / (Max_Momentum - Min_Momentum)
```

4. Calculate rate of change for normalized momentum:

```
Momentum_Change = Normalized_Momentum[current] - Normalized_Momentum[current - 1]
```

5. Final MP calculation as a combination of normalized momentum and its change:

```
MP = Normalized_Momentum + Momentum_Change
```

Where:

- Price - price (usually closing price)

- Length - calculation period
- Momentum - base momentum
- Normalized_Momentum - normalized momentum
- Momentum_Change - momentum change

Interpretation

The Momentum Pinball indicator can be interpreted as follows:

- 1. Extreme Levels:**
 - Values above 0.8 indicate market overbought conditions
 - Values below 0.2 indicate market oversold conditions
 - When MP reaches these levels, the probability of a reversal or correction increases
- 2. Divergences:**
 - Bullish Divergence: price forms a new low, while MP forms a higher low
 - Bearish Divergence: price forms a new high, while MP forms a lower high
 - Divergences often precede significant trend reversals
- 3. Centerline Crossovers:**
 - MP crossing the 0.5 level from bottom to top can be viewed as a bullish signal
 - MP crossing the 0.5 level from top to bottom can be viewed as a bearish signal
- 4. Rebounds from Extremes:**
 - MP reversal from overbought or oversold levels can generate market entry signals
 - Particularly strong signals form when such reversals are accompanied by divergences
- 5. Trend Analysis:**
 - Sustained MP values above 0.5 confirm an upward trend
 - Sustained MP values below 0.5 confirm a downward trend
 - MP oscillations around the 0.5 level indicate a sideways trend or uncertainty
- 6. Momentum Strength:**
 - Steep MP slope indicates strong momentum
 - Shallow MP slope indicates weak momentum
 - Slowdown in MP rise or fall may precede a trend reversal
- 7. Combining with Other Indicators:**
 - MP is often used in combination with trend indicators
 - For example, moving averages can be used to determine trend direction, while MP can be used for entry and exit points

[image: indicator_momentum_pinball]

See Also

[Momentum](#) [RSI](#) [StochasticOscillator](#) [PrettyGoodOscillator](#)

Money Flow Index (MFI)

Money Flow Index

Money Flow Index (MFI) calculates the difference between incoming and outgoing money flows. If the base value of the asset is below the difference, it indicates an increase in incoming money mass, meaning the market is bullish. If the opposite is observed, it means investors are leaving the instrument, and the market is bearish. The indicator is a technical oscillator for determining overbought or oversold conditions in an asset. It can also be used to identify divergences that warn of a trend change in price. The oscillator moves between 0 and 100.

To use the indicator, the [MoneyFlowIndex](#) class should be used.

[image: IndicatorMoneyFlowIndex]

See Also

[MACD](#)

Moving Average Convergence Divergence (MACD)

MACD

Moving Averages Convergence-Divergence (MACD) is a momentum indicator that shows the relationship between two moving averages of a security's price.

The indicator is calculated as the difference between a short moving average and a long moving average. By default, the periods for these averages are set to 12 and 26, respectively.

To use the indicator, the [MovingAverageConvergenceDivergence](#) class should be used.

[image: IndicatorMovingAverageConvergenceDivergence]

See Also

[MACD Histogram](#)

MACD Histogram

MACD Histogram

Moving Averages Convergence-Divergence (MACD) is a momentum indicator that shows the relationship between two moving averages of a security's price, presented as a histogram.

To use the indicator, the [MovingAverageConvergenceDivergenceHistogram](#) class should be used.

Three exponential moving averages with different periods are used to calculate the indicator. The fast moving average with a shorter period (EMA_s) is subtracted from the slow moving average with a longer period (EMA_l). The MACD line is constructed from the obtained values.

$$\text{MACD} = \text{EMA}_s(P) - \text{EMA}_l(P)$$

The default periods are 12 and 26. This line is then smoothed by a third exponential moving average (EMA_a), typically with a period of 9, resulting in the so-called MACD signal line (Signal).

$$\text{Signal} = \text{EMA}_a(\text{EMA}_s(P) - \text{EMA}_l(P))$$

These two resulting curves represent the regular linear MACD. Also, the zero line, relative to which the curves fluctuate, is usually marked in the indicator window.

When constructing the MACD Histogram (MACD Histogram), the histogram bars show the difference between the Signal and MACD lines, further simplifying the perception of the indicator.

[image: IndicatorMovingAverageConvergenceDivergenceHistogram]

See Also

[MACD with Signal Line](#)

MACD Signal

MACD with Signal Line

Moving Averages Convergence-Divergence (MACD) is a momentum indicator that shows the relationship between two moving averages of a security's price, presented with a signal line.

For a detailed description of the indicator calculation, see [MACD Histogram](#).

To use the indicator, the [MovingAverageConvergenceDivergenceSignal](#) class should be used.

[image: IndicatorMovingAverageConvergenceDivergenceSignal]

See Also

[NRTR](#)

Moving Average Crossover (MAC)

MAC

Moving Average Crossover (MAC) is a technical indicator that tracks crossovers between short and long moving averages to identify potential market entry and exit points.

To use the indicator, you need to use the [MovingAverageCrossover](#) class.

Description

The Moving Average Crossover (MAC) indicator is one of the most widely used and easy-to-understand indicators in technical analysis. It is based on the concept that when a short-term moving average crosses a long-term moving average, it may signal a trend change or significant price movement.

MAC uses two moving averages with different periods:

1. Short Moving Average (FastMA) - reflects recent price movement
2. Long Moving Average (SlowMA) - reflects longer-term price movement

The indicator is typically represented as the difference between the short and long moving averages, which allows for easy identification of the crossover moment (when the indicator value crosses the zero line).

MAC is widely used both in standalone trading strategies and as part of more complex systems, such as MACD (Moving Average Convergence Divergence).

Parameters

The indicator has the following parameters:

- **ShortPeriod** - period for the short moving average (default value: 9)
- **LongPeriod** - period for the long moving average (default value: 26)

Calculation

Moving Average Crossover indicator calculation involves the following steps:

1. Calculate the short moving average:

```
FastMA = SMA(Price, ShortPeriod)
```

2. Calculate the long moving average:

```
SlowMA = SMA(Price, LongPeriod)
```

3. Calculate MAC value as the difference between short and long moving averages:

```
MAC = FastMA - SlowMA
```

Where:

- Price - price (usually closing price)
- SMA - simple moving average
- ShortPeriod - period for the short moving average
- LongPeriod - period for the long moving average

Note: Other types of moving averages such as EMA (exponential moving average), WMA (weighted moving average), etc., can also be used instead of SMA.

Interpretation

The Moving Average Crossover indicator can be interpreted as follows:

1. Zero Line Crossovers:

- Crossing MAC zero line from bottom to top (FastMA crosses SlowMA from bottom to top) generates a bullish signal, indicating a potential upward trend start
- Crossing MAC zero line from top to bottom (FastMA crosses SlowMA from top to bottom) generates a bearish signal, indicating a potential downward trend start

2. Indicator Value:

- Positive MAC value indicates that the short moving average is above the long moving average, often interpreted as a bullish market state
- Negative MAC value indicates that the short moving average is below the long moving average, often interpreted as a bearish market state

3. Distance Between Moving Averages:

- Increasing distance between moving averages (increasing MAC absolute value) indicates trend strengthening
- Decreasing distance between moving averages (decreasing MAC absolute value) may indicate trend weakening and potential reversal

4. False Signals:

- During sideways consolidation periods, MAC may generate multiple false signals due to frequent moving average crossovers
- Additional indicators or rules are often used to filter false signals (e.g., requiring price to be above/below both moving averages)

5. Combining with Other Indicators:

- MAC is often used in combination with momentum indicators (RSI, Stochastic) to confirm signals
- It can also be combined with trend and volatility indicators to create more comprehensive trading systems

6. Parameter Selection:

- Shorter periods (e.g., 5 and 20) are more sensitive and suitable for short-term trading
- Longer periods (e.g., 50 and 200) are less sensitive and suitable for long-term trading

[image: indicator_moving_average_crossover]

See Also

[SMA](#) [EMA](#) [MACD](#) [MovingAverageRibbon](#)

Moving Average Ribbon (MAR)

MAR

Moving Average Ribbon (MAR) is a technical indicator that displays multiple moving averages with progressively increasing periods to visualize trend strength and direction.

To use the indicator, you need to use the [MovingAverageRibbon](#) class.

Description

The Moving Average Ribbon (MAR) is a set of multiple moving averages displayed on a chart in a "ribbon" or "fan" formation. This indicator helps traders visualize the current trend state and its strength more intuitively than using one or two moving averages.

MAR includes several moving averages (typically 5 to 10) with progressively increasing periods. The interval between periods can be uniform (e.g., 10, 20, 30, 40...) or exponential (e.g., 5, 10, 20, 40...).

The main idea is that the mutual positioning and shape of these moving averages can provide valuable information about the current trend's state and strength, and help identify potential reversal points.

Parameters

The indicator has the following parameters:

- **ShortPeriod** - initial (minimum) period for moving averages (default value: 10)
- **LongPeriod** - final (maximum) period for moving averages (default value: 100)
- **RibbonCount** - number of moving averages in the ribbon (default value: 10)

Calculation

Moving Average Ribbon calculation involves the following steps:

1. Determine the sequence of periods for moving averages:

```
Step = (LongPeriod - ShortPeriod) / (RibbonCount - 1)
Periods = [ShortPeriod, ShortPeriod + Step, ShortPeriod + 2*Step, ..., LongPeriod]
```

2. Calculate moving average for each period:

```
MAs = [SMA(Price, Period) for each Period in Periods]
```

Where:

- Price - price (usually closing price)
- SMA - simple moving average
- ShortPeriod - initial period
- LongPeriod - final period
- RibbonCount - number of moving averages

Note: Other types of moving averages such as EMA (exponential moving average), WMA (weighted moving average), etc., can be used instead of SMA.

Interpretation

The Moving Average Ribbon can be interpreted as follows:

1. Mutual Positioning of Moving Averages:

- When all lines are arranged in ascending order of periods (shortest on top, longest at bottom), this indicates a strong upward trend
- When all lines are arranged in descending order of periods (shortest at bottom, longest on top), this indicates a strong downward trend
- When lines intersect and have no clear order, this indicates a sideways trend or uncertainty

2. Ribbon Shape:

- Expanding ribbon (increasing distance between lines) indicates trend strengthening
- Contracting ribbon (decreasing distance between lines) indicates trend weakening
- Tight grouping of lines indicates consolidation or lack of a pronounced trend

3. Moving Average Crossovers:

- Beginning of line intersections may signal a potential trend change
- When short moving averages start crossing long ones, this can be an early signal of trend reversal

4. Ribbon Slope Angle:

- Steep angle indicates a strong trend
- Shallow angle indicates a weak trend
- Horizontal ribbon positioning indicates a sideways trend

5. Price Position Relative to Ribbon:

- When price is above the entire ribbon, this confirms a strong upward trend
- When price is below the entire ribbon, this confirms a strong downward trend
- When price moves within the ribbon, this may indicate a transitional state or consolidation

6. Trading Strategies:

- Enter a position when price bounces from the ribbon's edge in the trend direction
- Exit a position when moving averages begin to cross in the opposite direction
- Use ribbon width to set stop-losses and take-profits

[image: indicator_moving_average_ribbon]

See Also

[SMA](#) [EMA](#) [MovingAverageCrossover](#) [Guppy](#) [MultipleMovingAverage](#)

Negative Volume Index (NVI)

NVI

Negative Volume Index (NVI) is a technical indicator developed by Paul Dysart that focuses on price changes on days when trading volume decreases compared to the previous day.

To use the indicator, you need to use the [NegativeVolumeIndex](#) class.

Description

The Negative Volume Index (NVI) is based on the idea that "smart money" is active on low-volume days, while the "crowd" (non-professional traders) is more active on high-volume days. NVI changes only on days when the current volume is lower than the previous day's volume.

The indicator's concept suggests that price movements on reduced volume are more significant and often reflect the actions of informed investors. NVI seeks to track these movements, ignoring price changes on days with increased volume.

NVI is often used together with the complementary Positive Volume Index (PVI), which, in contrast, considers only days when volume increases.

Calculation

Negative Volume Index calculation involves the following steps:

1. Set initial NVI value (usually 1000):

```
NVI[initial] = 1000
```

2. For each subsequent period:

```
If Volume[current] < Volume[previous], then:  
    NVI[current] = NVI[previous] * (1 + (Price[current] - Price[previous]) / Price[previous])  
Otherwise:  
    NVI[current] = NVI[previous]
```

Where:

- Price - price (usually closing price)
- Volume - trading volume

In other words, NVI changes only on days when trading volume decreases and remains unchanged on days with increasing or unchanged volume.

Interpretation

The Negative Volume Index can be interpreted as follows:

1. **Trend Analysis:**
 - Rising NVI indicates that "smart money" is buying, which may foretell future price increase
 - Falling NVI indicates that "smart money" is selling, which may foretell future price decline
2. **Moving Average Crossovers:**
 - NVI is often compared to its 255-day moving average (approximately one year of trading)
 - When NVI is above its 255-day SMA, it is considered a bullish signal

- When NVI is below its 255-day SMA, it is considered a bearish signal
3. **Divergences:**
 - Bullish Divergence: price forms a new low, while NVI forms a higher low
 - Bearish Divergence: price forms a new high, while NVI forms a lower high
 4. **Combining with PVI:**
 - When both NVI and PVI are rising, it's a strong bullish signal
 - When both NVI and PVI are falling, it's a strong bearish signal
 - When NVI rises and PVI falls, it may indicate "smart money" buying while the "crowd" sells (potentially bullish scenario)
 - When NVI falls and PVI rises, it may indicate "smart money" selling while the "crowd" buys (potentially bearish scenario)
 5. **Long-Term Changes:**
 - NVI is often viewed as a long-term indicator
 - Sustained change in NVI direction may signal a significant shift in market sentiment
 6. **Confirming Other Indicators:**
 - NVI works best when combined with other technical indicators and analysis methods
 - NVI signals become more reliable when confirmed by other indicators
 7. **Setting Thresholds:**
 - Some traders set threshold levels for NVI (e.g., 5% above or below the moving average)
 - Crossing these thresholds may be considered a stronger signal than simple crossovers

[image: indicator_negative_volume_index]

See Also

[OBV](#) [ADL](#) [Chaikin](#) [MoneyFlow](#) [ForceIndex](#)

Positive Volume Index (PVI)

PVI

Positive Volume Index (PVI) is a technical indicator developed by Paul Dysart that focuses on price changes on days when trading volume increases compared to the previous day.

To use the indicator, you need to use the [PositiveVolumeIndex](#) class.

Description

The Positive Volume Index (PVI) is based on the idea that the "crowd" (non-professional traders) is more active on high-volume days, while "smart money" acts on low-volume days. PVI changes only on days when the current volume is higher than the previous day's volume.

The indicator suggests that price movements on increased volume are significant and often reflect mass-market sentiment. PVI tracks these movements, ignoring price changes on days with lower volume.

PVI is often used together with the complementary Negative Volume Index (NVI), which, in contrast, considers only days when volume decreases.

Calculation

Positive Volume Index calculation involves the following steps:

1. Set initial PVI value (usually 1000):

```
PVI[initial] = 1000
```

2. For each subsequent period:

```
If Volume[current] > Volume[previous], then:  
    PVI[current] = PVI[previous] * (1 + (Price[current] - Price[previous]) / Price[previous])  
Otherwise:  
    PVI[current] = PVI[previous]
```

Where:

- Price - price (usually closing price)
- Volume - trading volume

In other words, PVI changes only on days when trading volume increases and remains unchanged on days with decreasing or unchanged volume.

Interpretation

The Positive Volume Index can be interpreted as follows:

1. **Trend Analysis:**
 - Rising PVI indicates that the "crowd" is buying, which may foretell future price increase
 - Falling PVI indicates that the "crowd" is selling, which may foretell future price decline
2. **Moving Average Crossovers:**
 - PVI is often compared to its 255-day moving average (approximately one year of trading)
 - When PVI is above its 255-day SMA, it is considered a bullish signal
 - When PVI is below its 255-day SMA, it is considered a bearish signal

3. Divergences:

- Bullish Divergence: price forms a new low, while PVI forms a higher low
- Bearish Divergence: price forms a new high, while PVI forms a lower high

4. Combining with NVI:

- When both PVI and NVI are rising, it's a strong bullish signal
- When both PVI and NVI are falling, it's a strong bearish signal
- When PVI rises and NVI falls, it may indicate the "crowd" buying while "smart money" sells (potentially bullish scenario)
- When PVI falls and NVI rises, it may indicate the "crowd" selling while "smart money" buys (potentially bearish scenario)

5. Long-Term Changes:

- PVI is often viewed as a long-term indicator
- Sustained change in PVI direction may signal a significant shift in market sentiment

6. Confirming Other Indicators:

- PVI works best when combined with other technical indicators and analysis methods
- PVI signals become more reliable when confirmed by other indicators

7. Setting Thresholds:

- Some traders set threshold levels for PVI (e.g., 5% above or below the moving average)
- Crossing these thresholds may be considered a stronger signal than simple crossovers

[image: indicator_positive_volume_index]

See Also

[OBV](#) [ADL](#) [ChaikinMoneyFlow](#) [ForceIndex](#) [NegativeVolumeIndex](#)

Nick Rypock Trailing Reverse (NRTR)

NRTR

Nick Rypock Trailing reverse (NRTR) — the indicator essence is that it is always on the segment from the reached price extremes. The following idea was implemented here: small corrective movements against the main trend should be ignored, and the movement against the main trend above some level signals an inverse trend.

To use the indicator, you must use the [NickRypockTrailingReverse](#) class.

[image: IndicatorNickRypockTrailingReverse]

Recommended content

[Parabolic SAR](#)

On-Balance Volume (OBV)

OBV

On-Balance Volume (OBV) is a technical indicator developed by Joseph Granville that uses trading volume to forecast price changes by accumulating volume based on price direction.

To use the indicator, you need to use the [OnBalanceVolume](#) class.

Description

On-Balance Volume (OBV) is a cumulative indicator that adds volume when the closing price rises and subtracts volume when the closing price falls. The indicator is based on the concept that volume changes precede price changes. According to this theory, when volume increases significantly without a corresponding price change, it should be expected that the price will eventually rise, and vice versa.

OBV aims to detect moments when "smart money" (large institutional investors) are accumulating or distributing positions, which may foretell future price movement. The indicator is particularly useful for identifying divergences between price and volume that may signal potential market reversals.

The OBV indicator was first introduced by Joseph Granville in 1963 in his book "Granville's New Key to Stock Market Profits" and has since become one of the most widely used volume indicators.

Calculation

On-Balance Volume calculation is very simple:

1. Set initial OBV value (usually 0 or an arbitrary number):

```
OBV[initial] = 0
```

2. For each subsequent period:

```
If Close[current] > Close[previous], then:  
    OBV[current] = OBV[previous] + Volume[current]  
If Close[current] < Close[previous], then:  
    OBV[current] = OBV[previous] - Volume[current]  
If Close[current] = Close[previous], then:  
    OBV[current] = OBV[previous]
```

Where:

- Close - closing price
- Volume - trading volume

Interpretation

On-Balance Volume can be interpreted as follows:

1. **Trend Analysis:**
 - Rising OBV indicates volume entering the market (accumulation), which may foretell price increase
 - Falling OBV indicates volume leaving the market (distribution), which may foretell price decline
 - Flat OBV indicates no directional volume movement, which may correspond to a sideways trend
2. **Price Trend Confirmation:**
 - If OBV moves in the same direction as price, this confirms the current price trend

- If OBV and price move in opposite directions, this may signal a potential trend reversal
3. **Divergences:**
 - Bullish Divergence: price forms a new low, while OBV forms a higher low (buy signal)
 - Bearish Divergence: price forms a new high, while OBV forms a lower high (sell signal)
 4. **OBV Breakouts:**
 - OBV breakout of resistance or support level often precedes a similar breakout on the price chart
 - Traders can use OBV trendline breakouts to forecast future price movements
 5. **Baseline:**
 - Some traders use OBV moving averages as "baseline" lines
 - OBV crossing its moving average can generate trading signals
 6. **Technical Analysis Patterns:**
 - Classic technical analysis patterns can form on the OBV chart, such as "head and shoulders", "double bottom", etc.
 - These patterns can provide additional trading signals
 7. **Volume Spikes:**
 - Sudden sharp changes in OBV may indicate significant changes in market sentiment
 - Such "volume spikes" often precede substantial price movements

It's important to note that OBV is a cumulative indicator, so its absolute value is not of great significance. The direction of OBV movement and its relationship to price movement are what matter.

[image: indicator_on_balance_volume]

See Also

[ADL](#) [Chaikin](#) [MoneyFlow](#) [ForceIndex](#) [NegativeVolumeIndex](#)

On Balance Volume Mean (OBVM)

OBVM

On Balance Volume Mean (OBVM) is a technical indicator that represents a moving average of the On Balance Volume (OBV) indicator, allowing for clearer trend signals based on volume.

To use the indicator, you need to use the [OnBalanceVolumeMean](#) class.

Description

On Balance Volume Mean (OBVM) is a modification of the classic On Balance Volume (OBV) indicator that applies a moving average to OBV values to smooth out fluctuations and identify clearer trends. The indicator maintains the core concept of OBV - volume accumulation based on price direction change, but adds an additional layer of filtering.

OBVM helps eliminate noise present in the original OBV and makes long-term volume flow trends more noticeable. This is particularly useful in volatile markets or when analyzing instruments with irregular trading volumes.

The main advantage of OBVM is its ability to generate clearer and less prone to false signals trading signals compared to the classic OBV. The indicator can also be used to identify crossovers between OBV and its mean value, providing additional trading opportunities.

Parameters

The indicator has the following parameters:

- **Length** - period for moving average calculation (default value: 20)

Calculation

On Balance Volume Mean calculation involves the following steps:

1. Calculate the base On Balance Volume (OBV) indicator:

```
If Close[current] > Close[previous]:  
    OBV[current] = OBV[previous] + Volume[current]  
If Close[current] < Close[previous]:  
    OBV[current] = OBV[previous] - Volume[current]  
If Close[current] = Close[previous]:  
    OBV[current] = OBV[previous]
```

2. Apply moving average to OBV values:

```
OBVM = SMA(OBV, Length)
```

Where:

- Close - closing price
- Volume - trading volume
- OBV - On Balance Volume
- SMA - simple moving average
- Length - moving average period

Note: Other types of moving averages such as EMA (exponential moving average), WMA (weighted moving

average), etc., can be used instead of SMA.

Interpretation

On Balance Volume Mean can be interpreted as follows:

1. **Trend Analysis:**

- Rising OBVM indicates a bullish trend with strong volume support
- Falling OBVM indicates a bearish trend with strong volume support
- Flat OBVM indicates no pronounced trend

2. **OBV and OBVM Crossovers:**

- When OBV crosses OBVM from bottom to top, it can be viewed as a bullish signal
- When OBV crosses OBVM from top to bottom, it can be viewed as a bearish signal
- These crossovers often indicate the start of new trends or significant price movements

3. **Divergences:**

- Bullish Divergence: price forms a new low, while OBVM forms a higher low
- Bearish Divergence: price forms a new high, while OBVM forms a lower high
- Divergences often precede significant trend reversals

4. **Price Trend Confirmation:**

- If OBVM moves in the same direction as price, this confirms the current price trend
- If OBVM and price move in opposite directions, this may signal a potential trend reversal

5. **Support and Resistance Levels:**

- The OBVM chart can form its own support and resistance levels
- Breakout of these levels may precede similar breakouts on the price chart

6. **Comparison with Other Volume Indicators:**

- OBVM can be compared with other volume indicators to confirm signals
- Consistency of signals from multiple volume indicators increases their reliability

7. **Length Parameter Selection:**

- Shorter periods (e.g., 10-15) make OBVM more sensitive to short-term changes
- Longer periods (e.g., 30-50) better identify long-term trends
- Optimal period depends on trading time horizon and specific instrument characteristics

[image: indicator_on_balance_volume_mean]

See Also

[OBV](#) [ADL](#) [ChaikinMoneyFlow](#) [ForceIndex](#)

Optimal Tracking Filter (OptimalTracking)

Optimal Tracking

Optimal Tracking Filter indicator adjusts fast price changes and is relatively smooth when the price has a side effect. This filter's effect is similar to the Kaufman's Adaptive Moving Average.

To use the indicator, you must use the [OptimalTracking](#) class.

[image: IndicatorOptimalTracking]

Recommended content

[Alligator](#)

Oscillator of Moving Average (OMA)

OMA

Oscillator of Moving Average (OMA) is a technical indicator that measures the difference between two moving averages with different periods to determine momentum and potential reversal points.

To use the indicator, you need to use the [OscillatorOfMovingAverage](#) class.

Description

The Oscillator of Moving Average (OMA) represents the difference between a short and a long moving average. This indicator helps determine trend strength and its potential changes by analyzing the relationship between moving averages of different periods.

OMA works on a principle similar to MACD (Moving Average Convergence Divergence), but in a simpler form, as it does not include a signal line. The indicator oscillates around the zero line, where positive values indicate that the short moving average is above the long moving average (bullish state), and negative values indicate that the short moving average is below the long moving average (bearish state).

The main strength of OMA lies in its ability to identify changes in trend momentum and generate trading signals based on zero line crossovers and price divergences.

Parameters

The indicator has the following parameters:

- **ShortPeriod** - period for the short moving average (default value: 12)
- **LongPeriod** - period for the long moving average (default value: 26)

Calculation

Oscillator of Moving Average calculation involves the following steps:

1. Calculate the short moving average:

```
Short MA = SMA(Price, ShortPeriod)
```

2. Calculate the long moving average:

```
Long MA = SMA(Price, LongPeriod)
```

3. Calculate OMA as the difference between short and long moving averages:

```
OMA = Short MA - Long MA
```

Where:

- Price - price (usually closing price)
- SMA - simple moving average
- ShortPeriod - period for the short moving average
- LongPeriod - period for the long moving average

Note: Other types of moving averages such as EMA (exponential moving average), WMA (weighted moving average), etc., can be used instead of SMA.

Interpretation

The Oscillator of Moving Average can be interpreted as follows:

1. Zero Line Crossovers:

- OMA crossing the zero line from bottom to top (short MA crosses long MA from bottom to top) can be viewed as a bullish signal
- OMA crossing the zero line from top to bottom (short MA crosses long MA from top to bottom) can be viewed as a bearish signal

2. Extreme Values:

- High positive OMA values indicate that the market may be overbought
- High negative OMA values indicate that the market may be oversold
- Extreme values often precede corrections or trend reversals

3. Divergences:

- Bullish Divergence: price forms a new low, while OMA forms a higher low
- Bearish Divergence: price forms a new high, while OMA forms a lower high
- Divergences often precede significant trend reversals

4. Trend Confirmation:

- Positive OMA values confirm an upward trend
- Negative OMA values confirm a downward trend
- Increasing OMA absolute value indicates strengthening of the current trend

5. Centerline (0):

- When OMA oscillates around the zero line, it may indicate the absence of a pronounced trend or consolidation

6. Rate of Change:

- OMA slope indicates the rate of trend change
- Steep slope indicates rapid trend change
- Shallow slope indicates slow trend change

7. Combining with Other Indicators:

- OMA is often used in combination with other indicators to confirm signals
- Particularly effective when combined with overbought/oversold indicators like RSI or Stochastic

[image: indicator_oscillator_of_moving_average]

See Also

[MACD MovingAverageCrossover SMA EMA](#)

Parabolic SAR

Parabolic SAR

Parabolic SAR (SAR) - A trend indicator that indicates price stop and reversal points, as well as the trend direction.

To use the indicator, the [ParabolicSar](#) class should be used.

Indicator Calculation

The price of the indicator point (SAR) for the next period (candle) is calculated using the following formulas:

$SAR(n+1) = SAR(n) + a * (high - SAR(n))$, for an uptrend; $SAR(n+1) = SAR(n) + a * (low - SAR(n))$, for a downtrend, where:

$SAR(n+1)$ — price for the period $n+1$; $SAR(n)$ — price for the period n ; high and low — new maximum and minimum respectively (extremes). They are considered for the time interval between the activation of the previous indicator signal and the current moment;

a — acceleration factor.

The acceleration factor is a floating coefficient, characterized by minimum, maximum values, and a step of change.

The factor takes a minimum value equal to one step at the reversal point, and as soon as the price reaches a new extreme value according to the trend (high or low), the factor is increased by a step. When the factor reaches its maximum value, its growth is paused.

[image: IndicatorParabolicSar]

See Also

[Peak](#)

Stub (Pass Through)

Pass-Through

The **Pass-Through** indicator does not have logic and is used only for passing data to the chart, for example, for drawing lines.

Peak

Peak

Peak indicator shows the peak value for the period.

To use the indicator, you must use the [Peak](#) class.

[image: IndicatorPeak]

Recommended content

[QStick](#)

Percentage Price Oscillator (PPO)

PPO

Percentage Price Oscillator (PPO) is a technical indicator similar to MACD, but expressing the difference between two exponential moving averages as a percentage rather than in absolute values.

To use the indicator, you need to use the [PercentagePriceOscillator](#) class.

Description

The Percentage Price Oscillator (PPO) is a variation of the more well-known MACD (Moving Average Convergence Divergence) indicator. The main difference is that PPO expresses the difference between two exponential moving averages as a percentage, rather than in absolute units. This makes PPO particularly useful when comparing different instruments with varying price levels or analyzing a single instrument over a long period when its price has changed significantly.

PPO consists of three components:

1. **PPO Line** - difference between fast and slow EMA, expressed as a percentage
2. **Signal Line** - EMA of the PPO line
3. **Histogram** - difference between the PPO line and signal line

The PPO indicator oscillates around the zero line, where positive values indicate a bullish market sentiment, and negative values indicate a bearish sentiment. The magnitude of deviation from zero reflects the strength of the current trend.

Parameters

The indicator has the following parameters:

- **ShortPeriod** - period for calculating short EMA (default value: 12)
- **LongPeriod** - period for calculating long EMA (default value: 26)

Calculation

Percentage Price Oscillator calculation involves the following steps:

1. Calculate short and long exponential moving averages:

```
Short EMA = EMA(Price, ShortPeriod)
Long EMA = EMA(Price, LongPeriod)
```

2. Calculate PPO line as a percentage difference between short and long EMA:

```
PPO Line = ((Short EMA - Long EMA) / Long EMA) * 100
```

3. Calculate signal line (typically 9-period EMA of PPO line):

```
Signal Line = EMA(PPO Line, 9)
```

4. Calculate histogram:

```
Histogram = PPO Line - Signal Line
```

Where:

- Price - price (usually closing price)
- EMA - exponential moving average
- ShortPeriod - period for short EMA
- LongPeriod - period for long EMA

Interpretation

The Percentage Price Oscillator can be interpreted as follows:

1. **Zero Line Crossovers:**

- PPO line crossing the zero line from bottom to top can be viewed as a bullish signal
- PPO line crossing the zero line from top to bottom can be viewed as a bearish signal

2. **Signal Line Crossovers:**

- PPO line crossing the signal line from bottom to top can be viewed as a bullish signal
- PPO line crossing the signal line from top to bottom can be viewed as a bearish signal

3. **Divergences:**

- Bullish Divergence: price forms a new low, while PPO forms a higher low
- Bearish Divergence: price forms a new high, while PPO forms a lower high

4. **Overbought/Oversold:**

- Extremely high positive PPO values may indicate market overbought conditions
- Extremely low negative PPO values may indicate market oversold conditions

5. **Histogram Analysis:**

- Histogram expansion indicates strengthening of the current trend
- Histogram contraction indicates weakening of the current trend
- Change in histogram color (or sign) indicates a change in short-term momentum

6. **Instrument Comparison:**

- Unlike MACD, PPO can be used for direct comparison of different instruments
- Higher PPO values for one instrument compared to another may indicate stronger relative momentum

7. **Signal Filtering:**

- Signal line crossover signals are more reliable when PPO is in line with the main trend
- For example, bullish signals are more reliable when PPO is positive, and bearish signals are more reliable when PPO is negative

[image: indicator_percentage_price_oscillator]

See Also

[MACD](#) [EMA](#) [Percentage Price Oscillator](#) [Signal](#) [Percentage Price Oscillator Histogram](#)
[PercentageVolumeOscillator](#) [TRIX](#)

Percentage Price Oscillator Signal

Percentage Price Oscillator Signal

The **Percentage Price Oscillator Signal (PPOS)** indicator augments the standard Percentage Price Oscillator by adding the accompanying signal line that is typically used to filter trades.

To use the indicator, employ the [PercentagePriceOscillatorSignal](#) class.

Description

The Percentage Price Oscillator (PPO) measures the percentage difference between two exponential moving averages (EMAs). The signal version focuses on smoothing the PPO line with an additional EMA, helping traders react only to the more persistent shifts in momentum.

The indicator is formed by the following components:

1. **PPO line** – the percentage difference between the fast and slow EMAs.
2. **Signal line** – an EMA calculated from the PPO line (9 periods by default).

When the PPO line crosses above the signal line, it suggests increasing bullish momentum; crossing below signals strengthening bearish momentum. Remaining above or below the signal line can confirm the strength of the prevailing trend.

Calculation

1. Compute the fast and slow EMAs of the selected price series.
2. Calculate the PPO line as the percentage distance between the fast and slow EMAs.
3. Smooth the PPO line with an EMA to obtain the signal line.

```
FastEMA = EMA(Price, ShortPeriod)
SlowEMA = EMA(Price, LongPeriod)
PPO = ((FastEMA - SlowEMA) / SlowEMA) * 100
Signal = EMA(PPO, SignalPeriod)
```

Interpretation

- **Signal crossovers.** A bullish signal occurs when the PPO line crosses the signal line from below; the opposite crossover points to bearish momentum.
- **Trend confirmation.** Holding above the signal line confirms an uptrend, while staying below supports a downtrend.
- **Divergences.** Divergence between price action and the PPO line while it interacts with the signal line can anticipate reversals.

[image: indicator_percentage_price_oscillator_signal]

See Also

- [Percentage Price Oscillator](#)
- [Percentage Price Oscillator Histogram](#)

Percentage Price Oscillator Histogram

Percentage Price Oscillator Histogram

The **Percentage Price Oscillator Histogram (PPOH)** displays the distance between the PPO line and its signal line as a histogram, helping traders immediately evaluate the momentum balance.

To use the indicator, employ the [PercentagePriceOscillatorHistogram](#) class.

Description

The PPO histogram is derived from the standard Percentage Price Oscillator (PPO). Instead of plotting both the PPO and signal lines, it visualises their difference as bars around the zero level. Positive bars indicate that the PPO line is above the signal line (bullish momentum), whereas negative bars show that the PPO line is below the signal line (bearish momentum).

The histogram reacts quickly to changes in the spread between the PPO and the signal line, making it suitable for spotting early shifts in trend strength or identifying momentum divergences.

Calculation

1. Calculate the PPO line and its signal line using the desired periods.
2. Subtract the signal line from the PPO line to obtain the histogram value.

```
Histogram = PPO - Signal
```

Values above zero highlight bullish pressure, while those below zero reflect bearish pressure. The rate at which the histogram bars expand or contract provides clues about the momentum acceleration or deceleration.

Interpretation

- **Zero line crossovers.** Moving above zero confirms that the PPO line has crossed above the signal line, suggesting a bullish shift. Falling below zero indicates a bearish crossover.
- **Momentum surges.** Rapid growth of positive bars suggests strengthening bullish momentum; shrinking bars hint at weakening strength and a possible reversal.
- **Divergences.** Divergence between price action and the histogram can alert traders to potential trend exhaustion before it becomes visible on price charts.

[image: indicator_percentage_price_oscillator_histogram]

See Also

- [Percentage Price Oscillator](#)
- [Percentage Price Oscillator Signal](#)

Percentage Volume Oscillator (PVO)

PVO

Percentage Volume Oscillator (PVO) is a technical indicator similar to MACD, but applied to trading volume instead of price, expressing the difference between fast and slow exponential moving averages of volume as a percentage.

To use the indicator, you need to use the [PercentageVolumeOscillator](#) class.

Description

The Percentage Volume Oscillator (PVO) is a modification of the MACD (Moving Average Convergence Divergence) indicator, applied to trading volume instead of price. Similar to PPO (Percentage Price Oscillator), PVO expresses the difference between fast and slow exponential moving averages as a percentage, rather than in absolute units. This makes PVO particularly useful when comparing different instruments with varying volume levels or analyzing a single instrument over an extended period.

PVO consists of three components:

1. **PVO Line** - percentage difference between fast and slow volume EMA
2. **Signal Line** - EMA of the PVO line
3. **Histogram** - difference between the PVO line and signal line

The PVO indicator helps identify volume anomalies that may precede significant price movements. It is also useful for confirming price trends and identifying potential reversal points.

Parameters

The indicator has the following parameters:

- **ShortPeriod** - period for calculating short volume EMA (default value: 12)
- **LongPeriod** - period for calculating long volume EMA (default value: 26)

Calculation

Percentage Volume Oscillator calculation involves the following steps:

1. Calculate short and long exponential moving averages of volume:

```
Short EMA = EMA(Volume, ShortPeriod)
Long EMA = EMA(Volume, LongPeriod)
```

2. Calculate PVO line as a percentage difference between short and long EMA:

```
PVO Line = ((Short EMA - Long EMA) / Long EMA) * 100
```

3. Calculate signal line (typically 9-period EMA of PVO line):

```
Signal Line = EMA(PVO Line, 9)
```

4. Calculate histogram:

```
Histogram = PVO Line - Signal Line
```

Where:

- Volume - trading volume
- EMA - exponential moving average
- ShortPeriod - period for short EMA
- LongPeriod - period for long EMA

Interpretation

The Percentage Volume Oscillator can be interpreted as follows:

1. Zero Line Crossovers:

- PVO line crossing the zero line from bottom to top indicates volume acceleration above average, which may herald a bullish movement
- PVO line crossing the zero line from top to bottom indicates volume deceleration below average, which may herald a bearish movement

2. Signal Line Crossovers:

- PVO line crossing the signal line from bottom to top can be viewed as a bullish signal
- PVO line crossing the signal line from top to bottom can be viewed as a bearish signal

3. Divergences:

- Bullish Divergence: price forms a new low, while PVO forms a higher low
- Bearish Divergence: price forms a new high, while PVO forms a lower high

4. Extreme Values:

- Very high PVO values may indicate excessive volume, often occurring during market peaks or panic
- Very low PVO values may indicate insufficient volume, often occurring during market lulls

5. Histogram Analysis:

- Increasing positive histogram indicates strengthening bullish volume momentum
- Increasing negative histogram indicates strengthening bearish volume momentum
- Histogram contraction indicates weakening of current volume momentum

6. Price Trend Confirmation:

- Rising PVO confirms an upward price trend
- Falling PVO confirms a downward price trend
- Divergence between PVO direction and price may signal a potential reversal

7. Volume Spikes:

- Sharp PVO jumps indicate significant volume changes, often accompanying important market events
- Such spikes may precede or accompany breakouts of key price levels

[image: indicator_percentage_volume_oscillator]

See Also

[PPO](#) [OBV](#) [MACD](#) [Chaikin](#)[MoneyFlow](#)

Pivot Points (PP)

PP

Pivot Points (PP) is a technical indicator that uses previous high, low, and closing prices to determine potential support and resistance levels for the current trading period.

To use the indicator, you need to use the [PivotPoints](#) class.

Description

Pivot Points (pivot points) are one of the oldest and most widely used methods for identifying key market levels. The indicator calculates the central pivot point (PP) and several support (S1, S2, S3) and resistance (R1, R2, R3) levels based on the previous period's data.

Originally, Pivot Points were used by traders on exchange floors to determine key levels for the current trading day based on the previous day's data. Today, this method has been adapted for various timeframes - from intraday to monthly.

The main idea behind Pivot Points is that the market tends to react to these pre-calculated levels, using them as reversal points or zones where consolidation may occur. Traders use these levels to make decisions about market entry and exit, as well as to set target levels and stop-losses.

Calculation

There are several methods for calculating Pivot Points, including standard, Fibonacci, Woodie, Camarilla, and DeMark. Below is the standard calculation method:

1. Calculate the main pivot point (PP):

$$PP = (High + Low + Close) / 3$$

2. Calculate the first resistance (R1) and support (S1) levels:

$$\begin{aligned} R1 &= (2 * PP) - Low \\ S1 &= (2 * PP) - High \end{aligned}$$

3. Calculate the second resistance (R2) and support (S2) levels:

$$\begin{aligned} R2 &= PP + (High - Low) \\ S2 &= PP - (High - Low) \end{aligned}$$

4. Calculate the third resistance (R3) and support (S3) levels:

$$\begin{aligned} R3 &= High + 2 * (PP - Low) \\ S3 &= Low - 2 * (High - PP) \end{aligned}$$

Where:

- High - highest price of the previous period
- Low - lowest price of the previous period
- Close - closing price of the previous period

Interpretation

Pivot Points can be interpreted as follows:

1. **Main Pivot Point (PP):**
 - PP serves as the primary reference for determining overall market sentiment
 - If the price is above PP, it indicates a bullish sentiment
 - If the price is below PP, it indicates a bearish sentiment
 - PP can also serve as a support or resistance level
2. **Resistance Levels (R1, R2, R3):**
 - These levels represent potential resistance zones in a bullish market
 - A breakout of a level may lead to continued movement to the next level
 - A bounce from a level may lead to a downward reversal
3. **Support Levels (S1, S2, S3):**
 - These levels represent potential support zones in a bearish market
 - A breakout of a level may lead to continued movement to the next level
 - A bounce from a level may lead to an upward reversal
4. **Trading Strategies:**
 - **Bounce Trading:** Enter a position when bouncing off a support or resistance level
 - **Breakout Trading:** Enter a position after a confirmed level breakout
 - **Range Trading:** Buy at support levels and sell at resistance levels
 - **Target Setting:** Use the next level as a profit-taking target
 - **Stop-Loss Placement:** Place stop-losses beyond the corresponding levels
5. **Combining with Other Indicators:**
 - Pivot Points are often used in combination with other technical indicators to confirm signals
 - Particularly effective when combined with momentum indicators (RSI, Stochastic) and trend indicators (MA, MACD)
6. **Timeframes:**
 - Daily Pivot Points are calculated based on the previous trading day
 - Weekly Pivot Points are calculated based on the previous week
 - Monthly Pivot Points are calculated based on the previous month
 - Timeframe selection depends on trading style and time horizon

[image: indicator_pivot_points]

See Also

[FibonacciRetracement](#)

Price Channels (PC)

Price Channels

Price Channels (PC) display upper and lower boundaries of price movement over a specified period.

To use the indicator, you must use the [PriceChannels](#) class.

[image: IndicatorPriceChannels]

Recommended content

[Donchian Channels](#)

Price Volume Trend (PVT)

PVT

Price Volume Trend (PVT) is a cumulative indicator that multiplies price change by volume to show buying or selling pressure.

To use the indicator, you must use the [PriceVolumeTrend](#) class.

[image: IndicatorPVT]

Recommended content

[On-Balance Volume](#)

Pretty Good Oscillator (PGO)

PGO

Pretty Good Oscillator (PGO) is a technical indicator developed by Mark Johnson that compares the current closing price with previous prices, considering volatility, to determine market overbought or oversold conditions.

To use the indicator, you need to use the [PrettyGoodOscillator](#) class.

Description

The Pretty Good Oscillator (PGO) is an indicator that evaluates the strength of the current closing price relative to its historical values over a specific period. PGO takes into account not only the current price's position in the historical range but also the volatility of this range, making it more adaptive to changing market conditions.

The name "Pretty Good Oscillator" reflects its creator's pragmatic approach - the indicator does not claim to be a perfect tool, but offers a "pretty good" way of assessing the current market situation.

PGO is particularly useful for identifying overbought and oversold conditions, as well as detecting divergences that may precede trend reversals.

Parameters

The indicator has the following parameters:

- **Length** - calculation period (default value: 14)

Calculation

Pretty Good Oscillator calculation involves the following steps:

1. Determine the highest high (Highest High) and lowest low (Lowest Low) over the specified period:

```
Highest High = Highest(High, Length)
Lowest Low = Lowest(Low, Length)
```

2. Calculate the standard deviation of closing prices over the specified period:

```
Standard Deviation = StdDev(Close, Length)
```

3. Calculate the Pretty Good Oscillator:

```
PGO = (Close - (Highest High + Lowest Low) / 2) / Standard Deviation
```

Where:

- Close - current closing price
- High - highest price
- Low - lowest price
- Length - calculation period
- StdDev - standard deviation

Interpretation

The Pretty Good Oscillator can be interpreted as follows:

1. Overbought and Oversold Levels:

- Values above +2 often indicate market overbought conditions
- Values below -2 often indicate market oversold conditions
- Extreme values (+3/-3 and above/below) may signal significant overbought/oversold conditions and potential reversal

2. Zero Line Crossovers:

- PGO crossing the zero line from bottom to top can be viewed as a bullish signal
- PGO crossing the zero line from top to bottom can be viewed as a bearish signal

3. Divergences:

- Bullish Divergence: price forms a new low, while PGO forms a higher low
- Bearish Divergence: price forms a new high, while PGO forms a lower high
- Divergences often precede significant trend reversals

4. Trend Confirmation:

- Positive PGO values indicate that the price is above the average range, characteristic of an upward trend
- Negative PGO values indicate that the price is below the average range, characteristic of a downward trend

5. Trend Strength Assessment:

- The further the PGO value is from zero, the stronger the current trend
- PGO converging with the zero line may indicate trend weakening

6. Signal Filtering:

- PGO can be used to filter signals from other indicators
- For example, consider only bullish signals when PGO is positive, and only bearish signals when PGO is negative

7. Position Exits:

- Extreme PGO values can be used as signals to take profits
- For example, exit long positions when PGO exceeds +2, and short positions when PGO falls below -2

[image: indicator_pretty_good_oscillator]

See Also

[RSI](#) [StochasticOscillator](#) [CCI](#) [StandardDeviation](#)

Psychological Line (PSY)

PSY

Psychological Line (PSY) is a technical indicator that measures the proportion of rising periods (candles, bars) relative to the total number of periods over a specified time interval.

To use the indicator, you need to use the [PsychologicalLine](#) class.

Description

The Psychological Line (PSY) is a simple but effective indicator that reflects market sentiment by calculating the percentage of price-rising periods compared to the total number of periods considered. The indicator is based on the assumption that market psychology and investor sentiment play a crucial role in price movements.

PSY is an oscillator with values ranging from 0 to 100, where:

- A value of 100 means the price rose in all considered periods
- A value of 0 means the price fell in all considered periods
- A value of 50 means an equal number of rising and falling periods

The PSY indicator helps determine whether the market is in an overbought or oversold condition and can predict potential trend reversals.

Parameters

The indicator has the following parameters:

- **Length** - calculation period (default value: 12-14)

Calculation

The Psychological Line calculation is very simple:

$$\text{PSY} = (\text{Number of rising periods over Length periods} / \text{Length}) * 100$$

Where:

- A rising period is defined as a period where the closing price is higher than the previous period's closing price
- Length - number of periods considered

Interpretation

The Psychological Line can be interpreted as follows:

1. Overbought and Oversold Levels:

- Values above 70-80 indicate market overbought conditions (too many periods were rising)
- Values below 20-30 indicate market oversold conditions (too many periods were falling)
- Extreme values often precede trend reversals

2. Centerline (50):

- Crossing the 50 level from bottom to top can be viewed as a bullish signal

- Crossing the 50 level from top to bottom can be viewed as a bearish signal
 - Sustained movement above 50 indicates bull dominance
 - Sustained movement below 50 indicates bear dominance
3. **Divergences:**
- Bullish Divergence: price forms a new low, while PSY forms a higher low
 - Bearish Divergence: price forms a new high, while PSY forms a lower high
4. **Rebounds from Extreme Levels:**
- PSY reversal from the overbought zone may signal a potential bearish reversal
 - PSY reversal from the oversold zone may signal a potential bullish reversal
5. **Trend Analysis:**
- In a strong upward trend, PSY often remains above 50, with periodic rebounds from the overbought zone
 - In a strong downward trend, PSY often remains below 50, with periodic rebounds from the oversold zone
6. **Length Parameter Tuning:**
- Shorter periods (e.g., 5-8) make PSY more sensitive and suitable for short-term trading
 - Longer periods (e.g., 20-25) make PSY smoother and suitable for long-term trading
7. **Combining with Other Indicators:**
- PSY is often used in combination with other indicators to confirm signals
 - Particularly useful when combined with trend indicators and volume indicators

[image: indicator_psychological_line]

See Also

[RSI](#) [StochasticOscillator](#) [UltimateOscillator](#) [MomentumOscillator](#)

QStick

Qstick is a technical indicator used to determine trends on candle charts. It is calculated by taking the moving average for N periods from the difference between open and close prices. The positive Qstick value means that the buying pressure increases.

To use the indicator, you must use the [QStick](#) class.

[image: IndicatorQStick]

Recommended content

[RAVI](#)

Rainbow Charts (RC)

RC

Rainbow Charts (RC) is a technical analysis indicator consisting of a set of moving averages with different periods displayed on a single chart. Visually, the indicator resembles a rainbow, hence its name.

To use the indicator, you need to use the [RainbowCharts](#) class.

Description

Rainbow Charts are based on using multiple moving averages (usually simple SMA) with progressively increasing periods. Different moving average lines are colored in different colors, creating a rainbow effect on the chart.

The indicator helps determine trend direction and strength:

- When lines diverge, this indicates trend strengthening
- When lines converge, this may signal trend weakening or a potential reversal
- When the price is above all lines, it indicates a strong upward trend
- When the price is below all lines, it indicates a strong downward trend

Parameters

- **Lines** - number of SMA moving averages used in the rainbow chart.

Calculation

Rainbow Charts consist of multiple moving averages (SMA), with each subsequent line's period increasing by a certain step. For n lines with a base period p, periods are calculated as:

$$\text{Period}(i) = p + i * \text{step}$$

Where:

- i - line number (from 0 to n-1)
- step - period increase step (usually 1)

[image: IndicatorRainbowCharts]

See Also

[SMA](#)

Range Action Verification Index (RAVI)

RAVI

Range Action Verification Index (Ravi) is a technical analysis indicator that determines the presence of a trend in the market and its direction, based on a pair of simple moving averages with customizable periods.

To use the indicator, you must use the [RangeActionVerificationIndex](#) class.

[image: IndicatorRangeActionVerificationIndex]

Recommended content

[RoC](#)

Rank Correlation Index (RCI)

Rank Correlation Index

The **Rank Correlation Index (RCI)** is an oscillator based on Spearman's rank correlation coefficient. It compares the price ranks with the time ranks inside the moving window and shows how close the recent movement is to a perfectly rising or falling sequence.

Use the [RankCorrelationIndex](#) class to access the indicator.

Calculation

1. Assign each data point inside the **Length** window a time rank (1 for the oldest value, Length for the most recent).
2. Rank the prices by value (1 for the lowest price, Length for the highest).
3. Compute the difference $d = \text{RankTime} - \text{RankPrice}$ for each bar.
4. Apply the Spearman formula: $\text{RCI} = 1 - (6 \times \sum d^2) / (\text{Length} \times (\text{Length}^2 - 1))$.

When multiplied by 100 the indicator ranges between -100 and +100.

Parameters

- **Length** — window size for the ranking procedure.

Interpretation

- **RCI $\approx$ +100** — perfectly rising sequence (strong uptrend).
- **RCI $\approx$ -100** — perfectly falling sequence (strong downtrend).
- **RCI around 0** — random or sideways market.
- Divergences between price and the RCI warn of possible reversals.

The indicator is helpful for short-term trend assessment and spotting turning points, especially when combined with momentum tools.

[image: indicator_rank_correlation_index]

See also

[Momentum ROC RSI](#)

Rate of Change (ROC)

RoC

Rate of Change (RoC) indicator shows the difference between the current price and the price for n candles back.

To use the indicator, you must use the [RateOfChange](#) class.

[image: IndicatorRateOfChange]

Recommended content

[RSI](#)

Relative Momentum Index (RMI)

RMI

Relative Momentum Index (RMI) is a modification of the traditional RSI indicator, proposed by Roger Altman. Unlike the classic RSI, which calculates the ratio of price increases and decreases over a certain period, RMI takes into account the relative price change over a selected momentum period.

To use the indicator, you need to use the [RelativeMomentumIndex](#) class.

Description

The Relative Momentum Index (RMI) improves the classic RSI by adding a momentum period parameter. This allows traders to adjust the indicator's sensitivity without changing the main calculation period.

Like RSI, RMI oscillates between 0 and 100:

- Values above 70 usually indicate an overbought market
- Values below 30 indicate an oversold market
- The central line at 50 serves as a reference point for determining the main market movement direction

RMI is particularly useful for identifying potential trend reversal points and confirming the strength of the current trend.

Parameters

- **MomentumPeriod** - momentum period that defines the time lag for price comparison.
- **Length** - main period for calculating the indicator (similar to the period in RSI).

Calculation

The RMI calculation is performed in several steps:

1. Calculate momentum as the difference between the current price and the price n periods ago:

```
Momentum = Price(current) - Price(current - MomentumPeriod)
```

2. Divide momentums into positive (U) and negative (D):

```
If Momentum > 0, then U = Momentum, D = 0  
If Momentum < 0, then U = 0, D = |Momentum|
```

3. Calculate the average values of positive and negative momentums over the specified period:

```
AverageU = SMA(U, Length)  
AverageD = SMA(D, Length)
```

4. Calculate relative strength:

```
RS = AverageU / AverageD
```

5. Convert to Relative Momentum Index:

```
RMI = 100 - (100 / (1 + RS))
```


[image: IndicatorRelativeMomentumIndex]

See Also

[RSI Momentum](#)

Relative Strength Index (RSI)

RSI

Relative Strength Index (RSI) is a momentum indicator that measures the amount of recent price changes to assess the conditions of overbought or oversold prices of a share or other asset.

To use the indicator, you must use the [RelativeStrengthIndex](#) class.

[image: IndicatorRelativeStrengthIndex]

Recommended content

[RVI](#)

Relative Vigor Index (RVI)

RVI

Relative Vigor Index (RVI, relative energy index) indicator compares the price with a specific price range and provides an indication on some strength of price movement. Higher RVI values indicate an increase in the trend strength, while lower values indicate a trend decrease.

To use the indicator, you must use the [RelativeVigorIndex](#) class.

[image: IndicatorRelativeVigorIndex]

Schaff Trend Cycle (STC)

STC

Schaff Trend Cycle (STC) is a momentum indicator developed by Doug Schaff. STC is based on the assumption that market cycles more frequently move between overbought and oversold conditions rather than in a true trend.

To use the indicator, you need to use the [SchaffTrendCycle](#) class.

Description

The Schaff Trend Cycle combines the advantages of the Stochastic Oscillator, MACD, and cyclical analysis. This indicator can react to trend changes faster than traditional indicators such as MACD or Stochastic.

STC oscillates between 0 and 100:

- Values above 75 usually indicate overbought conditions
- Values below 25 indicate oversold conditions
- Crossing the 50 level may signal a trend change

Main indicator signals:

- Buy when STC crosses the 25 level from bottom to top (exiting oversold zone)
- Sell when STC crosses the 75 level from top to bottom (exiting overbought zone)

Parameters

- **Length** - main period for calculating the indicator.

Calculation

The STC calculation is performed in several steps:

1. Calculate MACD:

```
MACD = EMA(Close, Fast) - EMA(Close, Slow)
Signal = EMA(MACD, Signal)
```

where Fast, Slow, and Signal are typically 23, 50, and 10 respectively.

2. Calculate Stochastic Oscillator based on MACD:

```
Stoch_K = 100 * ((MACD - Lowest(MACD, Length)) / (Highest(MACD, Length) - Lowest(MACD, Length)))
Stoch_D = EMA(Stoch_K, 3)
```

3. Repeat stochastic calculation to obtain STC:

```
STC = 100 * ((Stoch_D - Lowest(Stoch_D, Length)) / (Highest(Stoch_D, Length) - Lowest(Stoch_D, Length)))
```

The result is an oscillator that is smoother than the classic Stochastic and reacts faster to trend changes than MACD.

[image: IndicatorSchaffTrendCycle]

See Also

[MACD Stochastic](#)

Shift

Shift

Shift is a helper indicator that offsets the incoming value stream by a specified number of periods. It does not transform the data; it only delays or aligns it for use in composite calculations.

Use the [Shift](#) class to access the indicator.

Description

The indicator keeps a buffer of the latest values and outputs the one that arrived **Length** bars ago. If the data history is shorter than the required offset, the value is considered undefined.

Parameters

- **Length** — number of periods by which the data is shifted.

Usage

- Align signals from different indicators in time.
- Build custom indicators and strategies that require delayed inputs.
- Create synthetic series, for example, to compute the difference between current prices and past values.

[image: indicator_shift]

See also

[PassThrough Sum](#)

SuperTrend

SuperTrend

The **SuperTrend indicator** is a trend-following indicator based on the Average True Range (ATR). It helps identify the current trend direction and possible reversal points.

To use the indicator, the [SuperTrend](#) class should be used.

Description

SuperTrend is built using the average price and the ATR value. The indicator line switches from above the price to below (and vice versa) when the trend changes. In this way, SuperTrend visually highlights the current trend until the price crosses the indicator line.

Parameters

- **ATR Length** – the period used for ATR calculation.
- **Multiplier** – the factor that defines how far the line is offset from the average price.

Calculation

1. Calculate ATR over the chosen period.
2. Compute two boundaries:

```
UpperBand = (High + Low) / 2 + Multiplier * ATR  
LowerBand = (High + Low) / 2 - Multiplier * ATR
```

3. SuperTrend initially equals one of the bands depending on the current trend.
4. If the closing price crosses the SuperTrend line, the trend direction changes and the line moves to the opposite side.

[image: IndicatorSuperTrend]

See Also

[ATR Parabolic SAR](#)

Simple Moving Average (SMA)

SMA

Simple Moving Average (SMA) is an arithmetic moving average calculated by adding the last closing prices and dividing this number by the number of time periods.

To use the indicator, you must use the [SimpleMovingAverage](#) class.

[image: IndicatorSimpleMovingAverage]

Recommended content

[Smoothed MA](#)

Sine Wave (SW)

SW

Sine Wave (SW) is a technical indicator that uses the sine mathematical function to identify cyclical patterns in price movement. The indicator aims to identify and predict periodic market oscillations.

To use the indicator, you need to use the [SineWave](#) class.

Description

The Sine Wave indicator is based on the idea that market movements have a cyclical nature and can be modeled using sinusoidal functions. This indicator is particularly useful in markets that move in a sideways range or have predictable cyclical fluctuations.

Key characteristics of the indicator:

- Helps identify potential market reversal points
- Allows determining the current position in the cycle
- Can be used to forecast future price movements

Indicator signals:

- Potential buy when the sine wave line reaches a minimum and begins to turn upward
- Potential sell when the line reaches a maximum and begins to turn downward

Parameters

- **Length** - period of the sine wave cycle, defining the cycle length in price bars.

Calculation

The Sine Wave indicator calculation is based on using the sine function and determining the dominant cycle in price movement:

1. Determining the dominant cycle using spectral analysis or another method of cycle identification.
2. Applying the sine function to model the identified cycle:

$$\text{SineWave}(t) = A * \sin(2\pi * t / \text{Length} + \phi)$$

where:

- A - amplitude (wave height)
 - t - current time or bar
 - Length - cycle length
 - ϕ - phase shift to align the sine wave with the actual price cycle
3. Additionally, a lead indicator may be calculated, which leads the main sine wave by a quarter cycle:

$$\text{Lead}(t) = A * \sin(2\pi * t / \text{Length} + \phi + \pi/2)$$

The indicator may include additional components such as a trend line or filter to improve signal accuracy.

[image: IndicatorSineWave]

See Also

[Schaff Trend Cycle](#)

Smoothed Moving Average (SMMA)

Smoothed MA

Smoothed Moving Average (SMA) - the indicator displays the smoothed average price direction for a certain time period.

To use the indicator, you must use the [SmoothedMovingAverage](#) class.

[image: IndicatorSmoothedMovingAverage]

Recommended content

[Standard Deviation](#)

Standard Deviation (Std Dev)

Standard Deviation

Standard Deviation is an indicator that shows the security price range, characterizing its volatility. Typically, the indicator value is expressed in price units of the analyzed asset. Standard Deviation is a function of standard price deviation, which is actively used in mathematical statistics.

To use the indicator, you must use the [StandardDeviation](#) class.

[image: IndicatorStandardDeviation]

Recommended content

[Stochastic Oscillator](#)

Standard Error (Standard error)

Standard Error

Standard Error indicator is an estimate of the mean square deviation of the regression coefficient from its true value.

To use the indicator, you must use the [StandardError](#) class.

[image: IndicatorStandardError]

Recommended content

[Alligator](#)

Stochastic %K (Stochastic %K)

Stochastic Oscillator K%

Stochastic Oscillator K% — K% line of [Stochastic Oscillator](#).

[image: IndicatorStochasticK]

Stochastic Oscillator

Stochastic Oscillator

Stochastic Oscillator is a momentum indicator that compares a specific closing price of a security with a range of its prices for a certain time period. The oscillator sensitivity to market movements can be reduced by adjusting this time period or taking the moving average from the result.

To use the indicator, you must use the [StochasticOscillator](#) class.

[image: IndicatorStochasticOscillator]

Recommended content

[Sum N](#)

Sum of N Last Values (Sum)

Sum N

Sum N last values indicator shows the sum of N last values for the period.

To use the indicator, you must use the [Sum](#) class.

[image: IndicatorSum]

Recommended content

[TEMA](#)

T3 Moving Average (T3MA)

T3MA

T3 Moving Average (T3MA) is an advanced type of moving average developed by Tim Tillson. T3 represents a thrice-smoothed exponential moving average (EMA) with a volume factor, making it smoother and less prone to false signals compared to traditional moving averages.

To use the indicator, you need to use the [T3MovingAverage](#) class.

Description

T3 Moving Average was developed to eliminate the drawbacks of traditional moving averages, such as lagging and false signals. Through multiple smoothing and an adjustable volume factor, T3MA provides a smoother curve that more accurately follows the price trend.

Main advantages of T3MA:

- Less lagging compared to ordinary moving averages
- Smoother curve with fewer false signals
- Adaptability to various market conditions due to the adjustable volume factor

T3MA can be used for:

- Determining trend direction
- Finding entry and exit points when price crosses the indicator line
- Building trading systems based on crossovers of multiple T3MAs with different periods

Parameters

- **VolumeFactor** - volume factor determining the degree of smoothing (usually a value between 0 and 1, recommended value is 0.7).
- **Length** - calculation period, similar to the period in ordinary moving averages.

Calculation

The T3 Moving Average calculation is performed in several steps:

1. Calculate six consecutive exponential moving averages with the same period:

```
EMA1 = EMA(Price, Length)
EMA2 = EMA(EMA1, Length)
EMA3 = EMA(EMA2, Length)
EMA4 = EMA(EMA3, Length)
EMA5 = EMA(EMA4, Length)
EMA6 = EMA(EMA5, Length)
```

2. Calculate T3 based on the obtained EMAs and volume factor:

```
c1 = -VolumeFactor^3
c2 = 3 * VolumeFactor^2 + 3 * VolumeFactor^3
c3 = -6 * VolumeFactor^2 - 3 * VolumeFactor - 3 * VolumeFactor^3
c4 = 1 + 3 * VolumeFactor + VolumeFactor^3 + 3 * VolumeFactor^2

T3 = c1 * EMA6 + c2 * EMA5 + c3 * EMA4 + c4 * EMA3
```

When VolumeFactor = 0, T3 becomes equivalent to EMA3 (triple EMA). When VolumeFactor = 1, T3 is maximally smoothed.

[image: IndicatorT3MovingAverage]

See Also

[EMA](#) [DEMA](#) [TEMA](#)

Time Weighted Average Price (TWAP)

TWAP

Time-Weighted Average Price (TWAP) is an indicator that calculates the average price of a financial instrument weighted by time over a specific period. TWAP is widely used by institutional investors to execute large orders with minimal market impact.

To use the indicator, you need to use the [TimeWeightedAveragePrice](#) class.

Description

TWAP is one of the most common order execution algorithms that breaks a large order into a series of smaller orders evenly distributed over time. The goal of TWAP is to obtain an average price over a specific time interval while minimizing market impact.

Main applications of TWAP:

- Benchmark price for evaluating order execution quality
- Execution algorithm to minimize market impact
- Tool for market analysis and trading decision-making

Unlike VWAP (Volume Weighted Average Price), TWAP does not consider trading volumes, focusing exclusively on the time aspect.

Calculation

TWAP calculation is performed by summing prices at equal time intervals and dividing this sum by the number of time intervals:

$$\text{TWAP} = (P_1 + P_2 + P_3 + \dots + P_n) / n$$

where:

- $P_1, P_2, \dots, P_n$ - prices at successive time moments
- n - number of time intervals

In practical implementation, typical prices for each period (candle) are most often used:

$$\begin{aligned} \text{Typical Price} &= (\text{High} + \text{Low} + \text{Close}) / 3 \\ \text{TWAP} &= \text{Sum}(\text{Typical Price}) / \text{Number of Periods} \end{aligned}$$

A recursive formula can also be used to determine the current TWAP value in real-time:

$$\text{TWAP}(\text{current}) = (\text{TWAP}(\text{previous}) * (n-1) + P(\text{current})) / n$$

where n is the number of observations in the TWAP window.

[image: IndicatorTimeWeightedAveragePrice]

Typical Price (TP)

Typical Price

Typical Price calculates the average of high, low, and close values for a candle.

To use the indicator, you must use the [TypicalPrice](#) class.

[image: IndicatorTypicalPrice]

Recommended content

[Median Price](#)

Triple Exponential Moving Average (TEMA)

TEMA

Triple Exponential Moving Average (TEMA) uses multiple (triple) Exponential Moving Averages (EMA) to eliminate lags when forecasting prices.

To use the indicator, you must use the [TripleExponentialMovingAverage](#) class.

[image: IndicatorTripleExponentialMovingAverage]

Recommended content

[TRIX](#)

TRIX

TRIX

TRIX is an oscillator used to determine oversold and overbought markets or as a momentum indicator. Like many oscillators, TRIX oscillates around the zero line. When it is used as an oscillator, a positive value indicates the overbought market and a negative value indicates the oversold market. When TRIX is used as a momentum indicator, a positive value indicates an increase in the momentum and a negative value indicates a decrease in the momentum.

To use the indicator, you must use the [Trix](#) class.

[image: IndicatorTrix]

Recommended content

[Trough](#)

Trough

Trough

Trough indicator shows the minimum value for the period.

To use the indicator, you must use the [Trough](#) class.

[image: IndicatorTrough]

Recommended content

[True Range](#)

True Range (TR)

True Range

True Range (TR) shows the current market volatility, determining the range of fluctuations of maximum, minimum and closing prices.

To use the indicator, you must use the [TrueRange](#) class.

[image: IndicatorTrueRange]

Recommended content

[UO](#)

True Strength Index (TSI)

True Strength Index

The **True Strength Index (TSI)** is a momentum oscillator created by William Blau. It applies double smoothing to the difference between consecutive closing prices, helping identify trends and turning points with less lag compared with many classic oscillators.

Use the [TrueStrengthIndex](#) class to access the indicator.

Calculation

1. Compute the price change $m = \text{Close} - \text{PreviousClose}$.
2. Apply two exponential moving averages with periods `Length1` and `Length2` to both m and $|m|$.
3. Calculate the ratio of the double-smoothed momentum to the double-smoothed absolute momentum:
$$\text{TSI} = 100 \times \text{EMA}(\text{EMA}(m, \text{Length1}), \text{Length2}) / \text{EMA}(\text{EMA}(|m|, \text{Length1}), \text{Length2}).$$
4. Optionally derive a signal line by taking an EMA of the TSI with period **Signal**.

Parameters

- **Length1** — first smoothing period.
- **Length2** — second smoothing period.
- **Signal** — period of the signal line (optional).

Interpretation

- **TSI > 0** — bullish momentum.
- **TSI < 0** — bearish momentum.
- **Signal-line crossovers** provide trade entries.
- **Divergences** between TSI and price warn about potential reversals.

Because of the double smoothing and normalisation, the indicator filters out noise yet remains responsive compared with simple momentum calculations.

[image: indicator_true_strength_index]

See also

[Momentum](#) [MACD](#) [RSI](#)

Twigg's Money Flow (TMF)

TMF

Twigg's Money Flow (TMF) is a volume indicator developed by Colin Twigg's as an improved version of the Chaikin Money Flow indicator. TMF is more sensitive to changes in market sentiment and has fewer false signals.

To use the indicator, you need to use the [Twigg'sMoneyFlow](#) class.

Description

Twigg's Money Flow analyzes the relationship between price and volume to determine the direction of money flow into or out of the market. Unlike traditional volume indicators, TMF eliminates noise by normalizing values between -1 and +1.

Key features of TMF:

- Positive values indicate money inflow into the instrument (bullish sentiment)
- Negative values indicate money outflow from the instrument (bearish sentiment)
- A value of 0 shows balance between supply and demand

The indicator is useful for:

- Confirming the current trend or identifying trend weakness
- Detecting divergences between price and money flow
- Identifying potential market reversal points

Parameters

- **Length** - calculation period for the exponential moving average, typically using a value of 21.

Calculation

Twigg's Money Flow calculation is performed in several steps:

1. Calculate True Range:

$$TR = \text{Max}(\text{High} - \text{Low}, |\text{High} - \text{Previous Close}|, |\text{Low} - \text{Previous Close}|)$$

2. Determine Twigg's Money Flow Volume (TMFV):

$$\text{TMFV} = \text{Volume} * ((\text{Close} - \text{Low} - (\text{High} - \text{Close})) / \text{TR})$$

When (High - Low = 0), TMFV = 0

3. Calculate exponential moving average of TMFV and volume:

$$\begin{aligned} \text{EMA\_TMFV} &= \text{EMA}(\text{TMFV}, \text{Length}) \\ \text{EMA\_Volume} &= \text{EMA}(\text{Volume}, \text{Length}) \end{aligned}$$

4. Final TMF value:

$$\text{TMF} = \text{EMA\_TMFV} / \text{EMA\_Volume}$$

TMF values range from -1 (strong bearish signal) to +1 (strong bullish signal).

[image: IndicatorTwiggMoneyFlow]

See Also

[ADL Money Flow Index OBV](#)

Ultimate Oscillator (UltimateOsc)

UO

Ultimate Oscillator (UO) is a technical indicator developed by Larry Williams in 1976 to measure a momentum across multiple timeframes. Using a weighted average for three different timeframes, the indicator has less volatility and less trading signals compared to other indicators that rely on a single timeframe.

To use the indicator, you must use the [UltimateOscillator](#) class.

[image: IndicatorUltimateOscillator]

Recommended content

[VHF](#)

Vertical-Horizontal Filter (VHF)

VHF

Vertical Horizontal Filter (VHF) - The vertical horizontal filter indicator was developed by Adam White in 1991. This indicator is used in technical analysis to recognize periods when the price is in a trend (uptrend or downtrend) or this indicator is in an overload phase (side trend).

To use the indicator, you must use the [VerticalHorizontalFilter](#) class.

[image: IndicatorVerticalHorizontalFilter]

Recommended content

[Volume](#)

Variable Index Dynamic Average (VIDYA)

VIDYA

Variable Index Dynamic Average (VIDYA) is an indicator with the original method of calculating the exponential moving average (EMA) with dynamically changing averaging period. The averaging period depends on the market volatility; a momentum oscillator (CMO) was chosen as a volatility measure.

To use the indicator, you must use the [Vidya](#) class.

[image: IndicatorVidya]

Recommended content

[JMA](#)

Variable Moving Average (VMA)

Variable MA

Variable Moving Average (VMA) adapts its smoothing period based on market volatility.

To use the indicator, you must use the [VariableMovingAverage](#) class.

[image: IndicatorVariableMovingAverage]

Recommended content

[VIDYA](#)

Volume

Volume

Candle volume indicator shows candle volume.

To use the indicator, you must use the [VolumeIndicator](#) class.

[image: IndicatorVolumeIndicator]

Recommended content

[Volume Profile](#)

Volume Profile

Volume Profile

Volume Profile is an advanced indicator that displays trading activity for a certain time period and at certain price levels. The indicator is a histogram on the chart that shows prevailing or important price levels based on volume.

To use the indicator, you must use the [VolumeProfileIndicator](#) class.

[image: Gui ClusterChart]

Recommended content

[Volume Weighted MA](#)

Volume Weighted Average Price (VWAP)

VWAP

Volume Weighted Average Price (VWAP) shows the average price weighted by traded volume.

To use the indicator, you must use the [VolumeWeightedAveragePrice](#) class.

[image: IndicatorVWAP]

Recommended content

[Time Weighted Average Price](#)

Volume Weighted Moving Average (VMA)

Volume Weighted MA

Volume Weighted Moving Average (VWMA) - the indicator displays the volume weighted price for a certain time period. This indicator follows the price as a wavy line, has a certain deviation relative to the price chart.

To use the indicator, you must use the [VolumeWeightedMovingAverage](#) class.

[image: IndicatorVolumeWeightedMovingAverage]

Recommended content

[Weighted MA](#)

Vortex Indicator (VI)

VI

Vortex Indicator (VI) is a technical indicator developed by Etienne and Julia Boisse in 2009. The indicator consists of two lines, VI+ and VI-, which show upward and downward price movement, helping to identify the beginning of new trends and confirm existing ones.

To use the indicator, you need to use the [VortexIndicator](#) class.

Description

The Vortex Indicator is inspired by the principles of vortex movement in nature and aims to reflect the cyclical nature of market movements. It consists of two lines:

- **VI+** (positive Vortex Indicator) - measures upward price movement
- **VI-** (negative Vortex Indicator) - measures downward price movement

Main indicator signals:

- Buy when VI+ crosses VI- from bottom to top
- Sell when VI- crosses VI+ from bottom to top
- The degree of separation between the lines indicates trend strength

The Vortex Indicator is particularly useful for:

- Determining the start of new trends
- Assessing the strength of an existing trend
- Identifying potential reversal points

Parameters

- **Length** - calculation period, typically using a value of 14.

Calculation

The Vortex Indicator calculation is performed in several steps:

1. Calculate positive and negative movement:

```
VM+ = |Current High - Previous Low|  
VM- = |Current Low - Previous High|
```

2. Calculate True Range:

```
TR = Max(High - Low, |High - Previous Close|, |Low - Previous Close|)
```

3. Sum VM+ and VM- values over the Length period:

```
Sum_VM+ = Sum(VM+, Length)  
Sum_VM- = Sum(VM-, Length)
```

4. Sum True Range over the Length period:

```
Sum_TR = Sum(TR, Length)
```

5. Calculate normalized VI+ and VI- values:

```
VI+ = Sum_VM+ / Sum_TR  
VI- = Sum_VM- / Sum_TR
```

The crossing of these two lines generates trading signals: when VI+ rises above VI-, it signals a bullish trend, and conversely, when VI- rises above VI+, it signals a bearish trend.

[image: IndicatorVortexIndicator]

See Also

[ADX DMI](#)

Wave Trend Oscillator (WTO)

WTO

Wave Trend Oscillator (WTO) is a technical indicator developed to identify overbought and oversold market levels, as well as to detect cyclical price fluctuations. WTO combines elements of channels and oscillators, making it an effective tool for identifying market momentum and potential reversal points.

To use the indicator, you need to use the [WaveTrendOscillator](#) class.

Description

Wave Trend Oscillator is designed to filter market noise and highlight primary price movements. The indicator oscillates around the zero line, creating wave patterns that correlate with cyclical price movements.

Key characteristics of WTO:

- Oscillations around the zero line, where positive values indicate an upward trend and negative values indicate a downward trend
- Overbought levels (usually above +60) and oversold levels (usually below -60)
- Ability to filter price noise and highlight primary movements

Key indicator signals:

- Zero line crossing (trend direction change)
- Exiting overbought/oversold zones
- Divergences between WTO and price (potential reversals)
- Specific wave patterns

Parameters

- **EsaPeriod** - EMA period for calculating ESA value (typically 10)
- **DPeriod** - period for calculating deviation (typically 21)
- **AveragePeriod** - period for calculating the average of the final oscillator (typically 4)

Calculation

Wave Trend Oscillator calculation involves several steps:

1. Calculate typical price:

$$AP = (High + Low + Close) / 3$$

2. Create smoothed and absolute first measurement value:

$$\begin{aligned} ESA &= EMA(AP, EsaPeriod) \\ D &= EMA(Abs(AP - ESA), DPeriod) \end{aligned}$$

3. Calculate the first oscillator line:

$$CI = (AP - ESA) / (0.015 * D)$$

4. Smooth the oscillator to obtain the final WTO value:

```
WTO = EMA(CI, AveragePeriod)
```

Typical values for the indicator parameters are: EsaPeriod = 10, DPeriod = 21, AveragePeriod = 4, but they can be adapted to different timeframes and instruments.

[image: IndicatorWaveTrendOscillator]

See Also

[MACD Stochastic Oscillator](#)

Weighted Close Price (WCP)

Weighted Close Price

Weighted Close Price (WCP) is calculated as $(\text{High} + \text{Low} + 2 \times \text{Close}) / 4$ for each candle.

To use the indicator, you must use the [WeightedClosePrice](#) class.

[image: IndicatorWeightedClosePrice]

Recommended content

[Typical Price](#)

Weighted Moving Average (WMA)

Weighted MA

Weighted Moving Average (WMA) - the indicator displays the weighted average price for a certain time period. This indicator follows the price as a wavy line, has a certain deviation relative to the price chart.

To use the indicator, you must use the [WeightedMovingAverage](#) class.

[image: IndicatorWeightedMovingAverage]

Recommended content

[%R](#)

Welles Wilder Moving Average (Wilder MA)

Wilder MA

Wilder Moving Average (WilderMA) is a weighted moving average indicator. Like all other moving average indicators, the WMA indicator smoothes market noises and displays market trends more clearly. To achieve the goal, the indicator filters out market fluctuations (noises), averaging the price values of the periods by which it is calculated. At the same time, some additional value (weight) is added to the average prices, as is done when calculating all weighted indicators such as EMA, LWMA and SMMA.

To use the indicator, you must use the [WilderMovingAverage](#) class.

[image: IndicatorWilderMovingAverage]

Recommended content

[Alligator](#)

Williams Accumulation/Distribution (WAD)

WAD

Williams Accumulation/Distribution (WAD) is a volume indicator developed by Larry Williams. Unlike the traditional Accumulation/Distribution line, the WAD indicator focuses on the relationship between the closing price of the current period and the closing price of the previous period to determine buyer or seller pressure.

To use the indicator, you need to use the [WilliamsAccumulationDistribution](#) class.

Description

The Williams Accumulation/Distribution indicator is designed to identify discrepancies between price and volume that may signal potential trend reversals. WAD is particularly useful for revealing weakness in the current price movement.

Key features of WAD:

- Positive values indicate accumulation (buying pressure)
- Negative values indicate distribution (selling pressure)
- Divergences between WAD and price may precede price reversals

Main applications of the indicator:

- Confirming the current trend
- Identifying potential price reversals
- Determining buyer or seller pressure

Calculation

The Williams Accumulation/Distribution indicator is calculated using the following logic:

1. Determine the True Range Protection (TRP) for the current period:

```
TRP = Max(High - Low, |High - Close_prev|, |Low - Close_prev|)
```

2. Calculate the Accumulation/Distribution (AD) value for the current period:

- If $\text{Close} > \text{Close\_prev}$ (market up):

```
AD = Close - Min(Low, Close_prev)
```

- If $\text{Close} < \text{Close\_prev}$ (market down):

```
AD = Close - Max(High, Close_prev)
```

- If $\text{Close} = \text{Close\_prev}$:

```
AD = 0
```

3. Compute the WAD value by accumulating AD values:

```
WAD = Previous WAD value + AD
```

The indicator accumulates positive and negative values, forming a cumulative line that can be used to

compare with price movement.

[image: IndicatorWilliamsAccumulationDistribution]

See Also

[ADL OBV](#)

Williams %R (%R)

%R

Williams %R (%R, Williams— Percent Range) is a momentum indicator that fluctuates between 0 and -100 and displays overbought and oversold levels.

To use the indicator, the [WilliamsR](#) class should be used.

Calculation

The formula for calculating the Williams— Percent Range indicator is similar to that used for calculating the Stochastic Oscillator:

$$\%R = - (\text{MAX}(\text{HIGH}(i - n)) - \text{CLOSE}(i)) / (\text{MAX}(\text{HIGH}(i - n)) - \text{MIN}(\text{LOW}(i - n))) * 100$$

where:

CLOSE(i) - today's closing price; MAX(HIGH(i - n)) - the highest maximum of the past n periods; MIN(LOW(i - n)) - the lowest minimum of the past n periods.

The value of n is set as the indicator parameter.

[image: IndicatorWilliamsR]

See Also

[ZigZag](#)

Woodies CCI (WCCI)

WCCI

Woodies CCI (WCCI) is a modification of the standard Commodity Channel Index (CCI), developed by trader Ken Wood (known as "Woodies"). This CCI variation includes additional smoothing and is used as part of a comprehensive Woodies CCI trading system.

To use the indicator, you need to use the [WoodiesCCI](#) class.

Description

Woodies CCI is a modified version of the classic CCI indicator that includes two lines:

- The main CCI line with a selected period (typically 14)
- A smoothed CCI line, which is a simple moving average of the main CCI line

The Woodies CCI system uses these two lines, along with several key levels to generate trading signals. The main levels include:

- +100 and -100 (traditional overbought and oversold levels)
- +200 and -200 (strong overbought and oversold conditions)
- Zero line (important for trend determination)

Key signals in the Woodies CCI system:

- "Zero-line Reject" - when CCI approaches the zero line and then bounces off it, continuing in the previous direction
- "Trend-line Break" - when CCI breaks a significant trend line
- "Reverse Divergence" - a specific type of divergence between price and CCI

Parameters

- **Length** - calculation period for the main CCI line (typically 14)
- **SMALength** - period for smoothing the main CCI line to obtain the second line (typically 9)

Calculation

Woodies CCI calculation is performed in several steps:

1. First, calculate the standard CCI:

```
Typical Price (TP) = (High + Low + Close) / 3
Average Value (SMA) = SMA(TP, Length)
Mean Deviation (MD) = Sum(|TP - SMA|) / Length
CCI = (TP - SMA) / (0.015 * MD)
```

2. Then calculate the smoothed CCI line:

```
Smooth CCI = SMA(CCI, SMALength)
```

Woodies CCI uses a combination of these two lines to create trading signals. In the classic Woodies system, the crossing of these lines, their interaction with key levels, and various patterns form the basis for trading decisions.

[image: IndicatorWoodiesCCI]

See Also

[CCI](#)

Zero Lag Exponential Moving Average (ZLEMA)

ZLEMA

Zero Lag Exponential Moving Average (ZLEMA) is a modified version of the exponential moving average (EMA), developed by John Ehlers. ZLEMA is designed to eliminate or significantly reduce the lag inherent in traditional moving averages.

To use the indicator, you need to use the [ZeroLagExponentialMovingAverage](#) class.

Description

The Zero Lag Exponential Moving Average (ZLEMA) was created to solve the primary problem of most moving averages - signal lag. Traditional moving averages lag behind price movements due to the time window used for their calculation. ZLEMA minimizes this lag by using a correction mechanism based on the difference between the current price and the price in the past.

Main advantages of ZLEMA:

- Faster reaction to price changes
- Less lag compared to traditional moving averages
- Preservation of the smoothing effect characteristic of EMA

ZLEMA can be used for:

- Determining trend direction
- Finding entry and exit points
- Identifying support and resistance levels
- Creating trading systems based on crossovers

Parameters

- **Length** - calculation period that determines the degree of smoothing (similar to the period in EMA).

Calculation

ZLEMA calculation is based on eliminating lag through forecasting and includes the following steps:

1. Calculate lag as half the period:

$$\text{lag} = (\text{Length} - 1) / 2$$

2. Compute the "detrended" price:

$$\text{detrendedPrice} = 2 * \text{Price} - \text{Price}[\text{lag}]$$

This is a key step that allows "looking ahead" and eliminating lag.

3. Apply exponential smoothing to the detrended price:

$$\begin{aligned} k &= 2 / (\text{Length} + 1) \\ \text{ZLEMA} &= k * \text{detrendedPrice} + (1 - k) * \text{ZLEMA}[\text{previous}] \end{aligned}$$

The result is a moving average that follows the price much closer than an ordinary EMA with the same period while maintaining the smoothing effect.

[image: IndicatorZeroLagExponentialMovingAverage]

See Also

[EMA](#) [DEMA](#) [TEMA](#) [T3MA](#)

ZigZag

ZigZag

ZigZag — the indicator displays points on the chart whenever prices change by a percentage that exceeds the preset value. Then straight lines are drawn connecting these points. The indicator is used to determine price trends.

To use the indicator, you must use the [ZigZag](#) class.

[image: IndicatorZigZag]

Williams Variable Accumulation Distribution (WVAD)

WVAD

Williams Variable Accumulation Distribution (WVAD) is a cumulative volume indicator developed by Larry Williams. It evaluates buying and selling pressure by analyzing the relationship between the open price, close price, high, low, and trading volume.

To use the indicator, use the [WilliamsVariableAccumulationDistribution](#) class.

Description

The WVAD indicator measures how much buyers or sellers control price movement within each bar, and weights this value by volume. If the close price is higher than the open price, it indicates buyer dominance, and vice versa. The High-Low range is used as a normalizing factor.

Main applications of the indicator:

- Confirming the current trend
- Identifying divergences between the indicator and price
- Determining buying or selling pressure
- Assessing the strength of price movement taking volume into account

Calculation

The WVAD indicator is calculated using the following formula:

$$\text{WVAD} = \text{WVAD}(\text{previous}) + ((\text{Close} - \text{Open}) / (\text{High} - \text{Low})) * \text{Volume}$$

where:

- Close - closing price of the current period
- Open - opening price of the current period
- High - highest price of the current period
- Low - lowest price of the current period
- Volume - trading volume of the current period
- WVAD(previous) - previous indicator value

If High = Low (the range is zero), the value for that period is not added.

The indicator is cumulative -- values accumulate with each new period.

See Also

[WAD ADL OBV](#)

Custom indicator

Custom Indicator

To create your own indicator, you need to implement the [IIndicator](#) interface. As an example, you can look at the source code of other indicators located in the [GitHub/StockSharp](#) repository. Here's what the implementation of the Simple Moving Average [SimpleMovingAverage](#) looks like:

```
/// <summary>
/// Simple Moving Average.
/// </summary>
[DisplayName("SMA")]
[Description("Simple Moving Average.")]
public class SimpleMovingAverage : LengthIndicator<decimal>
{
    /// <summary>
    /// Create <see cref="SimpleMovingAverage"/>.
    /// </summary>
    public SimpleMovingAverage()
    {
        Length = 32;
    }

    /// <summary>
    /// Process input value.
    /// </summary>
    /// <param name="input">Input value.</param>
    /// <returns>Resulting value.</returns>
    protected override IIndicatorValue OnProcess(IIndicatorValue input)
    {
        var newValue = input.GetValue<decimal>();
        if (input.IsFinal)
        {
            Buffer.Add(newValue);
            if (Buffer.Count > Length)
                Buffer.RemoveAt(0);
        }

        if (input.IsFinal)
            return new DecimalIndicatorValue(this, Buffer.Sum() / Length);

        return new DecimalIndicatorValue(this, (Buffer.Skip(1).Sum() + newValue) / Length);
    }
}
```

[SimpleMovingAverage](#) inherits from the [LengthIndicator<TResult>](#), from which all indicators with a period length parameter must inherit.

Important Indicator Properties and Methods

When creating a custom indicator, special attention should be paid to the following properties and methods:

NumValuesToInitialize

The [NumValuesToInitialize](#) property indicates how many values the indicator requires for initialization (formation or "warm-up"). This value is used to determine when the indicator can be considered formed and ready to use:

```
/// <inheritdoc />
public override int NumValuesToInitialize => Length;
```

For more complex indicators consisting of multiple components, this value is usually determined as the maximum of all constituent parts:

```
/// <inheritdoc />
public override int NumValuesToInitialize =>
    _shortEma.NumValuesToInitialize.Max(_longEma.NumValuesToInitialize);
```

Measure

The [Measure](#) property defines the type of measurement and dimensionality provided by the indicator:

```
/// <inheritdoc />
public override IndicatorMeasures Measure => IndicatorMeasures.Percent;
```

Available measurement types:

- `IndicatorMeasures.Price` - indicator measures price (e.g., moving averages)
- `IndicatorMeasures.Percent` - indicator uses percentage scale from 0 to 100 (e.g., RSI)
- `IndicatorMeasures.MinusOnePlusOne` - indicator uses scale from -1 to +1
- `IndicatorMeasures.Volume` - indicator measures volume (e.g., OBV)

This property is critically important for correctly displaying indicators on a chart. When multiple indicators with different dimensions are overlaid on the same panel, separate Y-axes are created for indicators with different Measure types. This allows visually displaying all indicators in their natural scale, even if one has values in thousands (e.g., price), while another is measured in fractions of a unit (e.g., oscillator).

Save and Load

The [Save](#) and [Load](#) methods are necessary for saving and loading indicator settings:

```
/// <inheritdoc />
public override void Save(SettingsStorage storage)
{
    base.Save(storage);

    storage.SetValue(nameof(ShortPeriod), ShortPeriod);
    storage.SetValue(nameof(LongPeriod), LongPeriod);
}

/// <inheritdoc />
public override void Load(SettingsStorage storage)
{
    base.Load(storage);

    ShortPeriod = storage.GetValue<int>(nameof(ShortPeriod));
    LongPeriod = storage.GetValue<int>(nameof(LongPeriod));
}
```

Composite Indicators

Some indicators are composite and use other indicators in their calculations. Therefore, indicators can be reused from each other, as demonstrated in the example implementation of the Chaikin Volatility indicator [ChaikinVolatility](#):

```
/// <summary>
/// Chaikin Volatility.
/// </summary>
[DisplayName("Volatility")]
```

```

[Description("Chaikin Volatility.")]
public class ChaikinVolatility : BaseIndicator<IIIndicatorValue>
{
    /// <summary>
    /// Create <see cref="ChaikinVolatility"/>.
    /// </summary>
    public ChaikinVolatility()
    {
        Ema = new ExponentialMovingAverage();
        Roc = new RateOfChange();
    }

    /// <summary>
    /// Moving Average.
    /// </summary>
    [ExpandableObject]
    [DisplayName("MA")]
    [Description("Moving Average.")]
    [Category("Main")]
    public ExponentialMovingAverage Ema { get; private set; }

    /// <summary>
    /// Rate of Change.
    /// </summary>
    [ExpandableObject]
    [DisplayName("ROC")]
    [Description("Rate of Change.")]
    [Category("Main")]
    public RateOfChange Roc { get; private set; }

    /// <summary>
    /// Is the indicator formed.
    /// </summary>
    public override bool IsFormed
    {
        get { return Roc.IsFormed; }
    }

    /// <summary>
    /// Process input value.
    /// </summary>
    /// <param name="input">Input value.</param>
    /// <returns>Resulting value.</returns>
    protected override IIIndicatorValue OnProcess(IIIndicatorValue input)
    {
        var candle = input.GetValue<Candle>();
        var emaValue = Ema.Process(input.SetValue(this, candle.HighPrice - candle.LowPrice));

        if (Ema.IsFormed)
        {
            return Roc.Process(emaValue);
        }

        return input;
    }
}

```

Indicators with Multiple Lines

The last type of indicators is those that not only consist of other indicators but are also graphically displayed with multiple states simultaneously (multiple lines). For example, [AverageDirectionalIndex](#):

```

/// <summary>
/// Welles Wilder's Average Directional Index.
/// </summary>
[DisplayName("ADX")]
[Description("Welles Wilder's Average Directional Index.")]

```

```

public class AverageDirectionalIndex : BaseComplexIndicator
{
    /// <summary>
    /// Create <see cref="AverageDirectionalIndex"/>.
    /// </summary>
    public AverageDirectionalIndex()
        : this(new DirectionalIndex { Length = 14 }, new WilderMovingAverage { Length = 14 })
    {
    }

    /// <summary>
    /// Create <see cref="AverageDirectionalIndex"/>.
    /// </summary>
    /// <param name="dx">Welles Wilder's Directional Movement Index.</param>
    /// <param name="movingAverage">Moving Average.</param>
    public AverageDirectionalIndex(DirectionalIndex dx, LengthIndicator<decimal> movingAverage)
    {
        if (dx == null)
            throw new ArgumentNullException(nameof(dx));
        if (movingAverage == null)
            throw new ArgumentNullException(nameof(movingAverage));

        InnerIndicators.Add(Dx = dx);
        InnerIndicators.Add(MovingAverage = movingAverage);
        Mode = ComplexIndicatorModes.Sequence;
    }

    /// <summary>
    /// Welles Wilder's Directional Movement Index.
    /// </summary>
    [Browsable(false)]
    public DirectionalIndex Dx { get; private set; }

    /// <summary>
    /// Moving Average.
    /// </summary>
    [Browsable(false)]
    public LengthIndicator<decimal> MovingAverage { get; private set; }

    /// <summary>
    /// Period length.
    /// </summary>
    [DisplayName("Period")]
    [Description("Indicator period.")]
    [Category("Main")]
    public virtual int Length
    {
        get { return MovingAverage.Length; }
        set
        {
            MovingAverage.Length = Dx.Length = value;
            Reset();
        }
    }
}

```

Such indicators should inherit from the [BaseComplexIndicator](#) class and pass the indicator's components to [BaseComplexIndicator.InnerIndicators](#). In addition, each complex indicator must declare its own value type derived from [ComplexIndicatorValue](#).

Example of a Complex Indicator with SaveLoad Implementation

Below is an example of implementing the Percentage Volume Oscillator (PVO), which demonstrates the implementation of [NumValuesToInitialize](#), [Measure](#), and [Save](#) and [Load](#) methods:

```

/// <summary>

```

```

/// Percentage Volume Oscillator (PVO).
/// </summary>
[Display(
    ResourceType = typeof(LocalizedStrings),
    Name = LocalizedStrings.PVOKey,
    Description = LocalizedStrings.PercentageVolumeOscillatorKey)]
[IndicatorIn(typeof(CandleIndicatorValue))]
[Doc("topics/api/indicators/list_of_indicators/percentage_volume_oscillator.html")]
[IndicatorOut(typeof(PercentageVolumeOscillatorValue))]
public class PercentageVolumeOscillator : BaseComplexIndicator<PercentageVolumeOscillatorValue>
{
    private readonly ExponentialMovingAverage _shortEma;
    private readonly ExponentialMovingAverage _longEma;

    /// <summary>
    /// Initializes a new instance of the <see cref="PercentageVolumeOscillator"/>.
    /// </summary>
    public PercentageVolumeOscillator()
        : this(new(), new())
    {
        ShortPeriod = 12;
        LongPeriod = 26;
    }

    /// <summary>
    /// Initializes a new instance of the <see cref="PercentageVolumeOscillator"/>.
    /// </summary>
    /// <param name="shortEma">The short-term EMA.</param>
    /// <param name="longEma">The long-term EMA.</param>
    public PercentageVolumeOscillator(ExponentialMovingAverage shortEma, ExponentialMovingAverage
longEma)
        : base(shortEma, longEma)
    {
        _shortEma = shortEma;
        _longEma = longEma;
    }

    /// <summary>
    /// Short period.
    /// </summary>
    [Display(
        ResourceType = typeof(LocalizedStrings),
        Name = LocalizedStrings.ShortPeriodKey,
        Description = LocalizedStrings.ShortMaDescKey,
        GroupName = LocalizedStrings.GeneralKey)]
    public int ShortPeriod
    {
        get => _shortEma.Length;
        set => _shortEma.Length = value;
    }

    /// <summary>
    /// Long period.
    /// </summary>
    [Display(
        ResourceType = typeof(LocalizedStrings),
        Name = LocalizedStrings.LongPeriodKey,
        Description = LocalizedStrings.LongMaDescKey,
        GroupName = LocalizedStrings.GeneralKey)]
    public int LongPeriod
    {
        get => _longEma.Length;
        set => _longEma.Length = value;
    }

    /// <inheritdoc />
    public override IndicatorMeasures Measure => IndicatorMeasures.Volume;

    /// <inheritdoc />

```



```

public override int NumValuesToInitialize =>
_shortEma.NumValuesToInitialize.Max(_longEma.NumValuesToInitialize);

/// <inheritdoc />
protected override bool CalcIsFormed() => _shortEma.IsFormed && _longEma.IsFormed;

/// <inheritdoc />
protected override IIndicatorValue OnProcess(IIndicatorValue input)
{
    var volume = input.ToCandle().TotalVolume;

    var result = new PercentageVolumeOscillatorValue(this, input.Time);

    var shortValue = _shortEma.Process(input, volume);
    var longValue = _longEma.Process(input, volume);

    result.Add(_shortEma, shortValue);
    result.Add(_longEma, longValue);

    if (_longEma.IsFormed)
    {
        var den = longValue.ToDecimal();
        var pvo = den == 0 ? 0 : ((shortValue.ToDecimal() - den) / den) * 100;
        result.Add(this, new DecimalIndicatorValue(this, pvo, input.Time));
    }

    return result;
}

/// <inheritdoc />
public override void Save(SettingsStorage storage)
{
    base.Save(storage);

    storage.SetValue(nameof(ShortPeriod), ShortPeriod);
    storage.SetValue(nameof(LongPeriod), LongPeriod);
}

/// <inheritdoc />
public override void Load(SettingsStorage storage)
{
    base.Load(storage);

    ShortPeriod = storage.GetValue<int>(nameof(ShortPeriod));
    LongPeriod = storage.GetValue<int>(nameof(LongPeriod));
}

/// <inheritdoc />
public override string ToString() => base.ToString() + $" S={ShortPeriod},L={LongPeriod}";

/// <inheritdoc />
protected override PercentageVolumeOscillatorValue CreateValue(DateTimeOffset time)
=> new(this, time);
}

```

```

/// <summary>
/// <see cref="PercentageVolumeOscillator"/> indicator value.
/// </summary>
public class PercentageVolumeOscillatorValue : ComplexIndicatorValue<PercentageVolumeOscillator>
{
    /// <summary>
    /// Initializes a new instance of the <see cref="PercentageVolumeOscillatorValue"/> class.
    /// </summary>
    /// <param name="indicator">Indicator.</param>
    /// <param name="time">Value time.</param>
    public PercentageVolumeOscillatorValue(PercentageVolumeOscillator indicator, DateTimeOffset time)
        : base(indicator, time)
    {
    }
}

```

```
}  
}
```

This example demonstrates:

1. Implementation of `NumValuesToInitialize` for a complex indicator
2. Specifying measurement type through the `Measure` property
3. Implementation of a dedicated value type for the complex indicator
4. Correct implementations of `Save` and `Load` methods for saving and loading parameters
5. Overriding `ToString()` for convenient display of indicator configuration

Graphic renderer of indicators

Graphic Renderer of Indicators

Some indicators require a special drawing style, such as two lines for the [BollingerBands](#) indicator. Or points for the [Fractals](#) indicator. In these cases, you need to explicitly specify the graphic renderer of indicators in [ChartIndicatorElement](#).

```
var chartIndicatorElement = new ChartIndicatorElement()
{
    IndicatorPainter = new BollingerBandsPainter(),
};
```

Let's consider how to create a custom IndicatorPainter using the [Fractals](#) indicator as an example.

All IndicatorPainters must inherit from the base class [BaseChartIndicatorPainter<TIndicator>](#) or implement the interface [IChartIndicatorPainter](#):

```
/// <summary>
/// The chart element for <see cref="Fractals"/>.
/// </summary>
[Indicator(typeof(Fractals))]
public class FractalsPainter : BaseChartIndicatorPainter<Fractals>
{
    ...
}
```

Let's define the chart elements [ChartLineElement](#) that will represent the upper and lower fractals:

```
/// <summary>
/// The chart element for <see cref="Fractals"/>.
/// </summary>
[Indicator(typeof(Fractals))]
public class FractalsPainter : BaseChartIndicatorPainter<Fractals>
{
    /// <summary>
    /// <see cref="Fractals.Up"/> dots color.
    /// </summary>
    [Display(
        ResourceType = typeof(LocalizedStrings),
        Name = LocalizedStrings.Str2035Key,
        Description = LocalizedStrings.Str2036Key)]
    public ChartLineElement Up { get; }

    /// <summary>
    /// <see cref="Fractals.Down"/> dots color.
    /// </summary>
    [Display(
        ResourceType = typeof(LocalizedStrings),
        Name = LocalizedStrings.Str2037Key,
        Description = LocalizedStrings.Str2038Key)]
    public ChartLineElement Down { get; }
    ...
}
```

In the constructor of [FractalsPainter](#), we set their values and main properties such as color, thickness, and drawing style. After that, we add them as child elements of the chart:

```
...
/// <summary>
```

```

/// Create instance.
/// </summary>
public FractalsPainter()
{
    Up = new ChartLineElement { Color = Colors.Green };
    Down = new ChartLineElement { Color = Colors.Red };
    Up.Style = Down.Style = DrawStyles.Dot;
    Up.StrokeThickness = Down.StrokeThickness = 4;
    AddChildElement(Up);
    AddChildElement(Down);
}

...

```

Override the [OnDraw](#) method, in which we draw the indicator using the [DrawValues](#) method:

```

...
/// <inheritdoc />
protected override bool OnDraw(Fractals ind, IDictionary<IIndicator,
IList<ChartDrawData.IndicatorData>> data)
{
    var result = false;
    result |= DrawValues(data[ind.Down], Down);
    result |= DrawValues(data[ind.Up], Up);
    return result;
}
...

```

Complete code of [FractalsPainter](#):

```

/// <summary>
/// The chart element for <see cref="Fractals"/>.
/// </summary>
[Indicator(typeof(Fractals))]
public class FractalsPainter : BaseChartIndicatorPainter<Fractals>
{
    /// <summary>
    /// <see cref="Fractals.Up"/> dots color.
    /// </summary>
    [Display(
        ResourceType = typeof(LocalizedStrings),
        Name = LocalizedStrings.Str2035Key,
        Description = LocalizedStrings.Str2036Key)]
    public ChartLineElement Up { get; }
    /// <summary>
    /// <see cref="Fractals.Down"/> dots color.
    /// </summary>
    [Display(
        ResourceType = typeof(LocalizedStrings),
        Name = LocalizedStrings.Str2037Key,
        Description = LocalizedStrings.Str2038Key)]
    public ChartLineElement Down { get; }
    /// <summary>
    /// Create instance.
    /// </summary>
    public FractalsPainter()
    {
        Up = new ChartLineElement { Color = Colors.Green };
        Down = new ChartLineElement { Color = Colors.Red };
        Up.Style = Down.Style = DrawStyles.Dot;
        Up.StrokeThickness = Down.StrokeThickness = 4;
        AddChildElement(Up);
        AddChildElement(Down);
    }
    /// <inheritdoc />
    protected override bool OnDraw(Fractals ind, IDictionary<IIndicator,

```

```

IList<ChartDrawData.IndicatorData>> data)
{
    var result = false;
    result |= DrawValues(data[ind.Down], Down);
    result |= DrawValues(data[ind.Up], Up);
    return result;
}
#region IPersistable
/// <inheritdoc />
public override void Load(SettingsStorage storage)
{
    base.Load(storage);
    Up.Load(storage.GetValue<SettingsStorage>(nameof(Up)));
    Down.Load(storage.GetValue<SettingsStorage>(nameof(Down)));
}
/// <inheritdoc />
public override void Save(SettingsStorage storage)
{
    base.Save(storage);
    storage.SetValue(nameof(Up), Up.Save());
    storage.SetValue(nameof(Down), Down.Save());
}
#endregion
}

```

Patterns

Patterns

Pattern (from English: pattern — model, sample) — in technical analysis, refers to stable recurring combinations of price data, volume, or indicators. Pattern analysis is based on one of the axioms of technical analysis: "history repeats itself" — it is believed that recurring data combinations lead to similar results.

Patterns are also called "**templates**" or "**figures**" of technical analysis.

Patterns are conventionally divided into:

- Indeterminate (can lead to both continuation and change of the current trend).
- Continuation patterns of the current trend.
- Patterns of existing trend reversal.

Using Patterns

In Designer

Designer has built-in preset candlestick patterns that can be used in your trading strategy. Patterns are called through the **indicator** cube with the subsequent selection of the corresponding value. The pattern itself is selected from the drop-down list in the window on the right.

[image: IndicatorPatternCommon]

It is also possible to edit existing and add your own custom patterns. To do this, you need to click on the button *[image: Designer edit button]* after which the pattern editing window will be shown.

[image: IndicatorPatternCommon01]

To create your own pattern, click the *[image: DesignerPlusButton]* button at the top of the window. Clicking the *[image: DesignerDeleteButton]* button deletes the pattern.

In Terminal

In **Terminal**, patterns are added to the chart like any other indicator. To do this, simply right-click on the chart and select the appropriate indicator from the list of available ones.

In StockSharp API

When using **S#** (or when creating **strategies from code** in Designer), working with patterns is done as with any other indicator. Example of use:

```
// Creating a pattern indicator
var patternIndicator = new CandlePatternIndicator
{
    // Setting the desired pattern
    Pattern = new ExpressionCandlePattern("My pattern", new[]
    {
        new CandleExpressionCondition(Paths.FileSystem, "C > 0"), // Current candle is rising
        new CandleExpressionCondition(Paths.FileSystem, "pC < p0") // Previous candle is falling
    })
};

// Adding the indicator to the collection
Indicators.Add(patternIndicator);
```

```
// Processing a candle
var result = patternIndicator.Process(candle);

// Checking the result
if (result.GetValue<bool>())
{
    // Pattern detected, perform necessary actions
}
```

Pattern Description Format

When editing a pattern, each line represents a separate candle. The topmost line is the current candle, accordingly, the second line is one candle back, the third and subsequent lines are minus 2 and more candles.

The editor uses the following parameters:

- O - opening price,
- H - high,
- L - low,
- C - closing price,
- V - volume,
- OI - open interest,
- B - candle body,
- LEN - length of the candle (from high to low),
- BS - lower shadow of the candle,
- TS - upper shadow of the candle.

With parameters, it is possible to use the following indices (references) to the desired values. For example, for the closing price:

- C: closing price of the current candle,
- C1: closing price of the 1st candle after the current one,
- C2: closing price of the 2nd candle after the current one,
- pC: closing price of the previous candle,
- pC1: closing price of the candle before the previous one, All references must be within the range of the current pattern. For example, the range of the 3 Black Crows pattern consists of the current and two previous candles, so referring to the third previous candle is not allowed.

For additional verification of parameters in correlation, the expression && is used, representing a logical AND.

When describing a pattern, it is also possible to use the following functions: abs, acos, asin, atan, ceiling, cos, exp, floor, log, log10, max, min, pow, round, sign, sin, sqrt, tan, truncate. More about using functions is explained in the description of the [formula](#) cube.

When using [ExpressionCandlePattern](#) in code, formulas are created by the same rules as described above and use the same variables.

Standard Patterns

For quick creation of patterns based on existing ones, you can use the section at the bottom of the pattern editor window. Clicking the *[image: DesignerPlusButton]* button at the bottom of the window adds the logic of the pattern selected from the drop-down list opposite to the editing window. The *[image: DesignerDeleteButton]* button at the bottom of the window deletes the selected line in the editing window.

Advanced Features

- [ComplexCandlePattern](#) — allows combining multiple patterns into a single composite pattern for more complex analysis.
- [ICandlePatternProvider](#) — pattern provider interface that allows loading and saving custom patterns.

3 Outside Down 3 Outside Up

Pattern 3 Outside Down and 3 Outside Up

The terms 3 Outside Down and 3 Outside Up refer to three-candle reversal patterns. For a pattern to form, three candles must form in a specific sequence, indicating that the current trend has lost momentum and may signal a reversal of the existing trend. Specifically, a pattern forms when a bearish candle (which closes lower than it opened) is followed by two instances of a bullish candle (which closes higher than it opened), or vice versa.

Key Features:

- The 3 Outside Down/Up patterns are three-candle patterns often signaling a trend reversal.
- 3 Outside Down and 3 Outside Up patterns are characterized by one candle immediately followed by two candles of the opposite color.
- Each aims to use market psychology to understand short-term mood changes.

3 Outside Up

This bullish candlestick pattern has the following characteristics:

1. The market is in a downtrend.
2. The first candle is bearish.
3. The second candle is bullish with a long body and completely encompasses the first candle.
4. The third candle is bullish with a higher close than the second candle.

[image: IndicatorPattern3OU]

3 Outside Down

This pattern variation is a bearish candlestick model with the following characteristics:

1. The market is in an uptrend.
2. The first candle is bullish.
3. The second candle is bearish with a long body that fully encompasses the first candle.
4. The third candle is bearish with a close lower than the second candle.

The first candle signifies the beginning of the end of the prevailing trend as the second candle engulfs the first candle. The third candle then signifies an acceleration of the reversal.

[image: IndicatorPattern3oD]

See Also

[Pattern 3 Inside Down and 3 Inside Up](#)

3 Inside Down and 3 Inside Up

Pattern 3 Inside Down and 3 Inside Up

The terms 3 Inside Down and 3 Inside Up refer to a pair of reversal candlestick patterns (each containing three separate candles), which are displayed on candlestick charts. The pattern requires the formation of three candles in a specific sequence, indicating that the current trend has lost momentum and may be starting to move in the opposite direction.

Key Features:

- The 3 Inside Up pattern is a bullish reversal pattern consisting of a large downward candle, a smaller upward candle contained within the previous candle, followed by another upward candle that closes above the second candle's close.
- The 3 Inside Down pattern is a bearish reversal pattern consisting of a large upward candle, a smaller downward candle contained within the previous candle, followed by another downward candle that closes below the second candle's close.
- These patterns are short-term in nature and may not always lead to a significant or even minor trend change.
- Consider using these patterns in the context of the overall trend. For example, use 3 Inside Up during a pullback in an overall upward trend.

3 Inside Up

The upward version of the pattern is bullish, indicating that the downward price movement may be ending, and an upward movement is starting. Here are the characteristics of the pattern:

1. The market is in a downward trend or moving lower.
2. The first candle is a black (downward) candle with a large body.
3. The second candle is a white (upward) candle with a small body, which opens and closes within the body of the first candle.
4. The third candle is a white (upward) candle, which closes above the second candle's close.

[image: IndicatorPattern3IU]

3 Inside Down

The downward version of the pattern is bearish. It shows that the upward price movement is ending, and the price begins to move down. Here are the characteristics of the pattern:

1. The market is in an upward trend or moving higher.
2. The first candle is a white candle with a large body.
3. The second candle is a black candle with a small body, which opens and closes within the body of the first candle.
4. The third candle is a black candle, which closes below the second candle's close.

These patterns are essentially harami patterns followed by a confirming candle, which many traders wait for in the case of haramis.

[image: IndicatorPattern3ID]

See Also

[Pattern 3 Outside Down and 3 Outside Up](#)

3 Black Crows and 3 White Soldiers

Pattern 3 Black Crows and 3 White Soldiers

3 Black Crows

Three Black Crows is a bearish candlestick pattern that can predict a reversal of an ascending trend.

The 3 Black Crows pattern consists of three consecutive candles that opened within the body of the previous candle and closed below the previous candle. Traders often use this indicator in combination with other technical indicators or chart patterns as confirmation of a reversal.

Key Features:

- Three Black Crows is a bearish candlestick pattern used to forecast the reversal of the current upward trend.
- Traders use it alongside other technical indicators, such as the Relative Strength Index (RSI).
- The pattern's candle size and shadows can indicate the risk of reversal to pullback.
- The opposite pattern to Three Black Crows is Three White Soldiers, indicating a downward trend reversal.

[image: IndicatorPattern3BC]

3 White Soldiers

Three White Soldiers is a bullish candlestick pattern used to forecast the reversal of the current downward trend on the price chart. The pattern consists of three consecutive long candles that open within the body of the previous candle and close surpassing the previous candle's high. These candles should not have very long shadows and ideally open within the body of the previous candle in the pattern.

Key Features

- Three White Soldiers are considered a reliable reversal model if confirmed by other technical indicators, such as the Relative Strength Index (RSI).
- The size of candles and the length of shadows are used to assess whether there is a risk of pullback.
- The opposite pattern to Three White Soldiers is Three Black Crows, indicating an upward trend reversal.

[image: IndicatorPattern3WS]

Three White Soldiers

Pattern Three White Soldiers

Three White Soldiers is a powerful bullish reversal candlestick pattern consisting of three consecutive candles that forms in a downtrend. This pattern indicates a decisive shift in control from sellers to buyers, signaling a potential reversal of a downtrend.

Key Features:

- Three consecutive white (bullish) candles with opening price lower than closing price ($O < C$).
- Each subsequent candle opens within the body of the previous candle ($O > pO$).
- Each candle closes higher than the closing of the previous candle.
- All three candles have relatively long bodies and short shadows.
- Forms in a downtrend.

[image: Three White Soldiers Pattern]

Interpretation

Three White Soldiers is considered one of the most reliable signals of a downtrend reversal:

- The sequence of three rising candles shows a steady increase in bullish pressure.
- The opening of each subsequent candle within the body of the previous one indicates some consolidation, then continuation of the bullish movement.
- The closing of each candle higher than the previous demonstrates the ability of buyers to consistently raise the price.
- Relatively long candle bodies with short shadows indicate decisive bull control over the market.
- The more uniform the sizes of the three candles, the stronger the signal.

Trading Strategies

Three White Soldiers provides reliable opportunities for entering a long position:

- Enter a long position after the formation of the complete pattern, usually at the opening of the fourth candle.
- Place a stop-loss below the low of the third candle or below the low of the entire pattern.
- Target profit can be set based on Fibonacci levels or previous resistance levels.
- Pay attention to volume - increasing volume with each candle confirms the strength of the signal.
- Take a more cautious approach with extremely long candle bodies, as a short-term correction may follow due to overbought conditions.
- Combine with other technical indicators, such as RSI, to increase the probability of a successful trade.

See also

[Pattern Three Black Crows](#)

[Pattern Rising Three Methods](#)

Three Black Crows

Pattern Three Black Crows

Three Black Crows is a powerful bearish reversal candlestick pattern consisting of three consecutive candles that forms in an uptrend. This pattern indicates a decisive shift in control from buyers to sellers, signaling a potential reversal of an uptrend.

Key Features:

- Three consecutive black (bearish) candles with opening price higher than closing price ($O > C$).
- Each subsequent candle opens within the body of the previous candle ($O < pO$).
- Each candle closes lower than the closing of the previous candle.
- All three candles have relatively long bodies and short shadows.
- Forms in an uptrend.

[image: Three Black Crows Pattern]

Interpretation

Three Black Crows is considered one of the most reliable signals of an uptrend reversal:

- The sequence of three declining candles shows a steady increase in bearish pressure.
- The opening of each subsequent candle within the body of the previous one indicates some consolidation, then continuation of the bearish movement.
- The closing of each candle lower than the previous demonstrates the ability of sellers to consistently lower the price.
- Relatively long candle bodies with short shadows indicate decisive bear control over the market.
- The more uniform the sizes of the three candles, the stronger the signal.

Trading Strategies

Three Black Crows provides reliable opportunities for entering a short position:

- Enter a short position after the formation of the complete pattern, usually at the opening of the fourth candle.
- Place a stop-loss above the high of the third candle or above the high of the entire pattern.
- Target profit can be set based on Fibonacci levels or previous support levels.
- Pay attention to volume - increasing volume with each candle confirms the strength of the signal.
- Take a more cautious approach with extremely long candle bodies, as a short-term correction may follow due to oversold conditions.
- Combine with other technical indicators, such as RSI in the overbought zone, to increase the probability of a successful trade.
- Particularly strong signal if the pattern forms after a prolonged uptrend or at an important resistance level.

See also

Pattern Three White Soldiers

Pattern Falling Three Methods

Pattern On-Neck

On-Neck is a bearish trend continuation candlestick pattern consisting of two candles that forms in a downtrend. This pattern shows a temporary unsuccessful attempt by bulls to reverse the trend, after which the downward movement continues.

Key Features:

- First candle is black (bearish) with opening price higher than closing price ($O > C$) and a relatively long body.
- Second candle is white (bullish) with opening price lower than closing price ($(C > O) \ \&\& \ (ABS(pL - C) \leq 1)$).
- Closing price of the second candle is around (or at the level of) the low of the first candle.
- Forms in a downtrend.

[image: On-Neck Pattern]

Interpretation

On-Neck is considered a signal of downtrend continuation:

- The first candle confirms the existing downtrend.
- The second candle starts with a gap down, indicating continued bearish pressure.
- However, during the session, bulls attempt to reverse the movement, but can only push the price up to the low of the previous candle.
- The closing of the second candle at the level of the low of the first candle indicates that buyers could not overcome this resistance level.
- This signals a likely continuation of the downtrend.

Trading Strategies

On-Neck provides opportunities to enter or strengthen short positions:

- Enter a short position or add to an existing short position after the pattern formation, usually at the opening of the third candle.
- Place a stop-loss above the high of the second candle or, for more protection, above the high of the first candle.
- Target profit can be set based on previous support levels or using technical measurement methods, such as projecting the height of the first candle.
- Pay attention to volume - decreasing volume on the second candle and increasing on subsequent bearish candles enhances the credibility of the signal.
- Combine with other indicators and patterns to increase the probability of a successful trade.

Piercing

Pattern Piercing

Piercing is a bullish reversal candlestick pattern consisting of two candles that forms in a downtrend. The first candle is black (bearish), followed by a white (bullish) candle that opens below the low of the previous candle and closes above the midpoint of the body of the previous candle.

Key Features:

- First candle is black with opening price higher than closing price ($O > C$).
- Second candle is white with opening price lower than closing price ($O < C$).
- Closing price of the second candle penetrates deeply into the body of the first candle, closing above the midpoint of the first candle ($C > (pB / 2 + pC)$).
- Forms in a downtrend.

[image: Piercing Pattern]

Interpretation

Piercing is considered a reliable signal of a potential reversal of a downtrend:

- The opening of the second candle below the low of the first candle (gap down) shows continued bearish pressure.
- However, during the trading session, bulls take control and push the price significantly higher, closing the candle above the midpoint of the body of the previous candle.
- This demonstrates a sharp shift in sentiment from bearish to bullish.
- The deeper the penetration into the second half of the body of the first candle, the stronger the signal.
- If the closing price of the second candle is above the opening price of the first candle, the pattern transitions to the "Bullish Engulfing" category, which is considered an even stronger signal.

Trading Strategies

Piercing provides good opportunities for entering a long position:

- Enter a position after confirming the pattern (usually at the opening of the third candle or when the high of the second candle breaks).
- Place a stop-loss below the low of the second candle or the low of the entire pattern.
- Target profit can be set based on previous resistance levels or Fibonacci levels.
- Consider trading volume - high volume during the formation of the second candle enhances the credibility of the signal.
- Additional confirmation from oversold indicators, such as RSI or Stochastic, increases the probability of a successful trade.

See also

[Pattern Bullish Engulfing](#)

Bearish

Pattern Bearish Candle

Bearish Candle is a candlestick pattern characterized by a closing price lower than the opening price. This pattern shows a bearish market sentiment.

Key Features:

- Opening price is higher than closing price ($O > C$).
- Indicates bearish pressure in the market.

[image: Bearish Candle Pattern]

Interpretation

The Bearish Candle signals a bearish market sentiment with several characteristics:

- A long upper shadow indicates that buyers tried to push the price up, but sellers took control.
- Closing below the opening shows the predominance of sellers at the end of the period.
- The ratio between the candle body and the upper shadow demonstrates the strength of sellers after testing upper levels.
- In an uptrend, it may signal a potential reversal or correction.
- In a downtrend, it confirms the strength of the trend, especially after a rebound.

Trading Strategies

Bearish Candle can be used in various trading strategies:

- Enter a short position after the formation of a Bearish Candle at a resistance level or in an overbought zone.
- Set a stop-loss above the maximum of the candle to protect against further rise.
- Combine with other technical indicators or patterns to increase the probability of a successful trade.
- Use to confirm downtrend signals from indicators such as MACD or Moving Averages.
- Pay attention to trading volume - high volume enhances the significance of the signal.
- Possible use for closing long positions or partial profit-taking when this pattern appears within an uptrend.

See also

[Pattern Bullish Candle](#)

[Pattern Black Candle](#)

Bearish Engulfing

Pattern Bearish Engulfing

Bearish Engulfing is a powerful bearish reversal candlestick pattern consisting of two candles that forms in an uptrend. The first candle is white (bullish), followed by a black (bearish) candle, the body of which completely engulfs (covers) the body of the previous candle.

Key Features:

- First candle is white with opening price lower than closing price ($O < C$).
- Second candle is black with opening price higher than closing price ($O > C$).
- Opening price of the second candle is higher than the closing price of the first candle ($O > pC$).
- Closing price of the second candle is lower than the opening price of the first candle ($C < pO$).
- The body of the second candle completely engulfs the body of the first candle.
- Forms in an uptrend.

[image: Bearish Engulfing Pattern]

Interpretation

Bearish Engulfing is considered one of the most reliable signals of an uptrend reversal:

- The first candle confirms the existing uptrend and shows control by buyers.
- The second candle demonstrates a sharp transition of control to sellers, who not only erase the profit of the previous period but also create a significant decline.
- The complete engulfing of the previous candle body symbolizes a complete shift in prevailing sentiment from bullish to bearish.
- The larger the size of the second candle compared to the first, the stronger the signal.
- If the pattern forms at a resistance level or after a prolonged uptrend, its significance increases.

Trading Strategies

Bearish Engulfing provides excellent opportunities for entering a short position:

- Enter a short position after the pattern formation, usually at the opening of the next candle.
- Place a stop-loss above the high of the second candle of the pattern.
- Target profit can be set based on previous support levels, risk/reward ratio, or using technical indicators.
- High trading volume during the formation of the second candle significantly increases the reliability of the signal.
- Combine with other indicators, such as RSI in the overbought zone or divergence on oscillators, to increase the probability of a successful trade.
- Can also be used as a signal to close existing long positions.

Bearish Harami

Pattern Bearish Harami

Bearish Harami is a reversal candlestick pattern consisting of two candles that forms in an uptrend. The term "harami" comes from the Japanese word meaning "pregnancy," as the small candle (child) is contained within the large candle (mother).

Key Features:

- First candle is white (bullish) with opening price lower than closing price ($O < C$) and a long body.
- Second candle is black (bearish) with opening price higher than closing price ($O > C$) and a smaller body.
- The body of the second candle is completely contained within the body of the first candle ($O < pC$) and ($C > pO$).
- Forms in an uptrend.

[image: Bearish Harami Pattern]

Interpretation

Bearish Harami signals a potential end to an uptrend:

- The first candle confirms the existing uptrend and the strength of buyers.
- The second candle, completely contained within the first, indicates a loss of bullish momentum and possible emergence of sellers.
- The smaller the body of the second candle compared to the first, the more pronounced the uncertainty and potential for reversal.
- If the second candle is a doji (with a very small body), the pattern is called a "Harami Cross" and is considered a stronger signal of uncertainty.
- This pattern is often considered a weaker signal compared to Bearish Engulfing, but may be an earlier indicator of a possible reversal.

Trading Strategies

Bearish Harami typically requires additional confirmation for position entry:

- Wait for a confirming bearish candle after the pattern formation before entering a short position.
- Place a stop-loss above the high of the pattern or the high of the first candle.
- Use trading volume as additional confirmation - decreasing volume on the second candle and increasing on subsequent bearish candles strengthens the signal.
- Combine with other technical indicators, such as RSI in the overbought zone or oscillator divergence.
- Consider the pattern at resistance levels or in overbought zones to increase the probability of a successful trade.
- Possible use for closing existing long positions, even if the signal is not strong enough to open short positions.

See also

[Pattern Bullish Harami](#)

Black Candle

Pattern Black Candle

Black Candle (bearish candle) is a classic candlestick pattern that forms when the closing price is lower than the opening price. This candle reflects a bearish market sentiment, where sellers controlled the price during the period of candle formation.

Key Features:

- Opening price is higher than closing price ($O > C$).
- The candle body is usually colored black (or red in modern charts).
- Indicates the predominance of sellers over buyers.
- The size of the candle body shows the strength of the bearish movement.

[image: Black Candle Pattern]

Interpretation

Black Candle signals bearish pressure in the market:

- The longer the candle body, the stronger the bearish pressure.
- A long black candle after an uptrend may indicate a potential reversal.
- The presence of short shadows indicates that bears controlled the price throughout the period.
- Consecutive black candles indicate a steady downtrend.

Trading Strategies

Although a single black candle is not usually an independent trading signal, it can be used as part of a broader strategy:

- Confirmation of a downtrend or reversal after an upward movement.
- Looking for long black candles at resistance levels for potential short positions.
- Using in combination with other candlestick patterns, for example, a black candle after a bearish engulfing.
- Identifying support levels after a series of consecutive black candles.

See also

[Pattern White Candle](#)

[Pattern Black Marubozu](#)

Black Marubozu

Pattern Black Marubozu

Black Marubozu is a bearish candlestick pattern characterized by the absence of shadows on both ends of the candle. The term "marubozu" comes from the Japanese word meaning "bald-headed" or "shaved," reflecting the appearance of the candle without shadows.

Key Features:

- Opening price is higher than closing price ($O > C$).
- The candle body is completely filled, without upper and lower shadows.
- Opening price equals the high of the candle, and closing price equals the low of the candle.
- Represents a strong bearish movement, where sellers controlled the price throughout the period.

[image: Black Marubozu Pattern]

Interpretation

Black Marubozu is considered a strong bearish signal:

- The absence of shadows indicates complete seller domination - the price opened at the high and continuously fell until the period closed.
- A long Black Marubozu indicates very strong bearish pressure.
- The appearance of this pattern after an uptrend may signal a reversal.
- Within a downtrend, it confirms the strength of the movement.

Trading Strategies

Black Marubozu provides a stronger signal than a regular black candle:

- Opportunity to enter a short position after the formation of a Black Marubozu, especially if it appears at an important resistance level.
- Using the closing price of Black Marubozu as a resistance level when setting stop-losses.
- Combining with other technical indicators to confirm the signal.
- Paying attention to trading volume - high volume enhances the significance of the signal.

See also

[Pattern White Marubozu](#)

[Pattern Black Candle](#)

Bullish

Pattern Bullish Candle

Bullish Candle is a candlestick pattern characterized by a closing price higher than the opening price. This pattern shows a bullish market sentiment.

Key Features:

- Opening price is lower than closing price ($O < C$).
- Indicates bullish pressure in the market.

[image: Bullish Candle Pattern]

Interpretation

The Bullish Candle signals a bullish market sentiment with several characteristics:

- A long lower shadow indicates that sellers tried to push the price down, but buyers took control.
- Closing above the opening shows the predominance of buyers at the end of the period.
- The ratio between the candle body and the lower shadow demonstrates the strength of buyers after testing lower levels.
- In a downtrend, it may signal a potential reversal.
- In an uptrend, it confirms the strength of the trend, especially after a correction.

Trading Strategies

Bullish Candle can be used in various trading strategies:

- Enter a long position after the formation of a Bullish Candle at a support level or in an oversold zone.
- Set a stop-loss below the minimum of the candle to protect against further decline.
- Combine with other technical indicators or patterns to increase the probability of a successful trade.
- Use to confirm uptrend signals from indicators such as MACD or Moving Averages.
- Pay attention to trading volume - high volume enhances the significance of the signal.

See also

[Pattern Bearish Candle](#)

[Pattern White Candle](#)

Bullish Engulfing

Pattern Bullish Engulfing

Bullish Engulfing is a powerful bullish reversal candlestick pattern consisting of two candles that forms in a downtrend. The first candle is black (bearish), followed by a white (bullish) candle, the body of which completely engulfs (covers) the body of the previous candle.

Key Features:

- First candle is black with opening price higher than closing price ($O > C$).
- Second candle is white with opening price lower than closing price ($O < C$).
- Opening price of the second candle is lower than the closing price of the first candle ($O < pC$).
- Closing price of the second candle is higher than the opening price of the first candle ($C > pO$).
- The body of the second candle completely engulfs the body of the first candle.
- Forms in a downtrend.

[image: Bullish Engulfing Pattern]

Interpretation

Bullish Engulfing is considered one of the most reliable signals of a downtrend reversal:

- The first candle confirms the existing downtrend and shows control by sellers.
- The second candle demonstrates a sharp transition of control to buyers, who not only reverse the losses of the previous period but also create a significant gain.
- The complete engulfing of the previous candle body symbolizes a complete shift in prevailing sentiment from bearish to bullish.
- The larger the size of the second candle compared to the first, the stronger the signal.
- If the pattern forms at a support level or after a prolonged downtrend, its significance increases.

Trading Strategies

Bullish Engulfing provides excellent opportunities for entering a long position:

- Enter a long position after the pattern formation, usually at the opening of the next candle.
- Place a stop-loss below the low of the second candle of the pattern.
- Target profit can be set based on previous resistance levels, risk/reward ratio, or using technical indicators.
- High trading volume during the formation of the second candle significantly increases the reliability of the signal.
- Combine with other indicators, such as RSI in the oversold zone or moving average convergence, to increase the probability of a successful trade.

See also

[Pattern Bearish Engulfing](#)

[Pattern Piercing](#)

Bullish Harami

Pattern Bullish Harami

Bullish Harami is a reversal candlestick pattern consisting of two candles that forms in a downtrend. The term "harami" comes from the Japanese word meaning "pregnancy," as the small candle (child) is contained within the large candle (mother).

Key Features:

- First candle is black (bearish) with opening price higher than closing price ($O > C$) and a long body.
- Second candle is white (bullish) with opening price lower than closing price ($O < C$) and a smaller body.
- The body of the second candle is completely contained within the body of the first candle ($O > pC$) and ($C < pO$).
- Forms in a downtrend.

[image: Bullish Harami Pattern]

Interpretation

Bullish Harami signals a potential end to a downtrend:

- The first candle confirms the existing downtrend and the strength of sellers.
- The second candle, completely contained within the first, indicates a loss of bearish momentum and possible emergence of buyers.
- The smaller the body of the second candle compared to the first, the more pronounced the uncertainty and potential for reversal.
- If the second candle is a doji (with a very small body), the pattern is called a "Harami Cross" and is considered a stronger signal of uncertainty.
- This pattern is often considered a weaker signal compared to Bullish Engulfing, but may be an earlier indicator of a possible reversal.

Trading Strategies

Bullish Harami typically requires additional confirmation for position entry:

- Wait for a confirming bullish candle after the pattern formation before entering a long position.
- Place a stop-loss below the low of the pattern or the low of the first candle.
- Use trading volume as additional confirmation - decreasing volume on the second candle and increasing on subsequent bullish candles strengthens the signal.
- Combine with other technical indicators, such as RSI in the oversold zone or oscillator divergence.
- Consider the pattern at support levels or in oversold zones to increase the probability of a successful trade.
- Possible use for closing existing short positions, even if the signal is not strong enough to open long positions.

See also

[Pattern Bearish Harami](#)

Pattern Dragonfly

Dragonfly is a candlestick pattern characterized by equal opening and closing prices, no upper shadow, and a long lower shadow. The candle resembles the shape of the letter "T," which gave it the name "dragonfly."

Key Features:

- Opening price equals closing price ($O == C$).
- No upper shadow ($TS == 0$).
- Long lower shadow.
- Similar to a Hammer, but with a neutral body (doji).

[image: Dragonfly Pattern]

Interpretation

Dragonfly Doji is considered a potential reversal signal, especially in a downtrend:

- The long lower shadow indicates that sellers controlled the market for most of the period, but then buyers returned the price to the opening level.
- The rejection of lower prices may signal the end of bearish pressure.
- Unlike a regular Hammer, the equality of opening and closing prices (doji) indicates a more pronounced balance of power.
- In a downtrend, this pattern has bullish implications and may foreshadow a reversal.
- In an uptrend, it may signal a potential correction.

Trading Strategies

Dragonfly requires additional confirmation for making trading decisions:

- Wait for a confirming bullish candle in the next period before entering a long position.
- Place a stop-loss below the minimum of the Dragonfly.
- Use in conjunction with support levels or oversold conditions on indicators.
- Pay attention to volume - high volume during the formation of a Dragonfly increases the credibility of the signal.
- Possible use for determining an exit point from short positions, even without entering a long position.

See also

[Pattern Gravestone](#)

[Pattern Hammer](#)

Evening Star

Pattern Evening Star

Evening Star is a bearish reversal candlestick pattern consisting of three candles that forms in an uptrend. This pattern shows a transition from bullish to bearish sentiment through a period of uncertainty or consolidation.

Key Features:

- First candle is white (bullish) with opening price lower than closing price ($O < C$).
- Second candle has a small body (can be either bullish or bearish) and forms a gap up from the first candle. The body is significantly smaller than the first candle's body ($B < pB * 0.5m$).
- Third candle is black (bearish) with opening price higher than closing price and a large body ($(O > C) \&\& (B > pB * 2)$), which closes deep inside the body of the first candle.
- Forms in an uptrend.

[image: Evening Star Pattern]

Interpretation

Evening Star is considered a strong signal of a potential reversal of an uptrend:

- The first candle confirms the strength of the bullish trend.
- The second candle (star) shows weakening of bullish pressure and uncertainty in the market.
- The third candle demonstrates the return of sellers and a shift in control from bulls to bears.
- The deeper the third candle penetrates into the body of the first candle, the stronger the reversal signal.
- If the second candle is a doji (with a very small body), the pattern is called "Evening Doji Star" and is considered an even stronger signal.

Trading Strategies

Evening Star provides good opportunities for entering a short position:

- Enter a short position after the pattern formation, usually at the opening of the fourth candle or when the low of the third candle breaks.
- Place a stop-loss above the high of the second candle or the high of the entire pattern.
- Target profit can be set based on Fibonacci levels relative to the previous upward movement or previous support levels.
- Pay attention to volume - increasing volume on the third candle confirms the strength of the bearish reversal.
- Combine with other technical indicators, such as RSI in the overbought zone or resistance lines, to increase the reliability of the signal.
- Can also be used as a signal to exit existing long positions.

See also

[Pattern Morning Star](#)

Falling Three Methods

Pattern Falling Three Methods

Falling Three Methods is a bearish trend continuation pattern consisting of five candles that forms in a downtrend. This pattern shows a temporary consolidation or rest within an existing downtrend before its continuation.

Key Features:

- First candle is black (bearish) with opening price higher than closing price ($O > C$) and a long body.
- The next three candles are white (bullish) with opening price lower than closing price ($O < C$), with small bodies: $(B < pB * 0.5m)$, $(B < ppB * 0.5m)$, $(B < pppB * 0.5m)$.
- The bodies of the three middle candles do not extend beyond the range of the first candle.
- Fifth candle is black (bearish) with opening price higher than closing price ($O > C$) and a long body ($B > pB * 2$).
- The fifth candle breaks through the low of the first candle and closes lower.
- Forms in a downtrend.

[image: Falling Three Methods Pattern]

Interpretation

Falling Three Methods is considered a reliable signal of downtrend continuation:

- The first long black candle shows the strength of the downtrend.
- Three small white candles represent a temporary consolidation or correction, during which buyers were unable to significantly change the trend.
- The fifth long black candle confirms the return of control to sellers and the continuation of the downtrend.
- This pattern can be viewed as a flag or pennant in classical technical analysis.
- Such a sequence of candles indicates that the correction was used to accumulate short positions before continuing the downward movement.

Trading Strategies

Falling Three Methods provides good opportunities for entering or strengthening short positions:

- Enter a short position at the opening after the fifth candle or when the low of the first candle breaks.
- Place a stop-loss above the high of the correction candles or above the high of the first candle.
- Target profit can be set using a projection equal to the distance from the beginning of the trend to the first candle of the pattern.
- Pay attention to volume - ideally, volume decreases on the three middle candles and significantly increases on the fifth candle.
- Combine with other technical indicators to confirm the signal.
- Consider important support levels below the current price that may affect the development of the movement.

See also

[Pattern Rising Three Methods](#)

[Pattern Three Black Crows](#)

Flat Candle

Pattern Flat (Neutral) Candle

Flat (neutral) candle is a candlestick pattern that forms when the opening and closing prices are identical or very close. This candle reflects market indecision, when the forces of buyers and sellers are balanced.

Key Features:

- Opening price equals closing price ($O == C$).
- Upper and lower shadows can be of different lengths.
- Indicates neutrality or indecision in the market.
- May signal consolidation before continuation of the existing trend or a possible reversal.

[image: Flat Candle Pattern]

Interpretation

A flat candle by itself does not provide a clear signal about market direction, but can be useful in the context of previous candles and the overall trend:

- After a strong upward or downward movement, a flat candle may signal a weakening of momentum and possible reversal.
- During a sideways movement, a flat candle confirms the continuation of consolidation.
- The size of the shadows can provide additional information about market sentiment - long shadows indicate rejected attempts at price movement, while short shadows indicate low volatility.

Trading Strategies

Flat candles are rarely used as standalone signals for position entry, but can help in making trading decisions:

- Look for confirmation from subsequent candles or other technical indicators before making a trade.
- Use flat candles to identify support or resistance levels in combination with other analysis methods.
- Increase stop-loss size or take profit if flat candles appear in a trend, which may indicate weakening movement.

See also

[Pattern White Candle](#)

[Pattern Black Candle](#)

Pattern Gravestone

Gravestone is a candlestick pattern characterized by equal opening and closing prices, a long upper shadow, and no lower shadow. The candle resembles the shape of an inverted letter "T," which gave it the name "gravestone."

Key Features:

- Opening price equals closing price ($O == C$).
- Long upper shadow.
- No lower shadow ($BS == 0$).
- Similar to a Shooting Star, but with a neutral body (doji).

[image: Gravestone Pattern]

Interpretation

Gravestone Doji is considered a potential reversal signal, especially in an uptrend:

- The long upper shadow indicates that buyers controlled the market for most of the period, but then sellers returned the price to the opening level.
- The rejection of higher prices may signal the end of bullish momentum.
- Unlike a regular Shooting Star, the equality of opening and closing prices (doji) indicates a more pronounced balance of power.
- In an uptrend, this pattern has bearish implications and may foreshadow a reversal.
- In a downtrend, it may indicate temporary resistance or trend continuation.

Trading Strategies

Gravestone requires additional confirmation for making trading decisions:

- Wait for a confirming bearish candle in the next period before entering a short position.
- Place a stop-loss above the maximum of the Gravestone.
- Use in conjunction with resistance levels or overbought conditions on indicators.
- Pay attention to volume - high volume during the formation of a Gravestone increases the credibility of the signal.
- Possible use for determining an exit point from long positions, even without entering a short position.

See also

[Pattern Dragonfly](#)

[Pattern Shooting Star](#)

Hammer

Pattern Hammer

Hammer is a bullish candlestick pattern that forms during a downtrend. The candle has a small body at the upper part and a long lower shadow, with the upper shadow being absent or very short. The name comes from the candle's resemblance to a hammer.

Key Features:

- Opening price is lower than closing price ($O < C$), although it can be the opposite.
- Small candle body at the upper part of the price range.
- Long lower shadow, typically 2-3 times longer than the body.
- No upper shadow or a very short one.
- Forms in a downtrend.

[image: Hammer Pattern]

Interpretation

Hammer is considered a strong signal of a potential reversal of a downtrend:

- The long lower shadow shows that the price fell significantly during the period, but then buyers intervened and pushed the price back up.
- This indicates the market's rejection of lower prices and a potential shift in sentiment from bearish to bullish.
- The longer the lower shadow, the stronger the potential reversal signal.
- The color of the candle body is less important, although a white/green hammer is considered more bullish than a black/red one.

Trading Strategies

Hammer provides opportunities for entering a long position:

- Wait for confirmation from the next candle - a bullish candle after a Hammer strengthens the reversal signal.
- Place a stop-loss level below the low of the Hammer.
- Set a target profit based on previous resistance levels or risk/reward ratio.
- Combine with other technical indicators, such as RSI or MACD, to confirm trend reversal.
- Higher trading volume during the formation of a Hammer increases the reliability of the signal.

See also

[Pattern Inverted Hammer](#)

[Pattern Hanging Man](#)

Hanging Man

Pattern Hanging Man

Hanging Man is a bearish reversal candlestick pattern that forms in an uptrend. The candle has a small body at the upper part and a long lower shadow, with the upper shadow being absent or very short. The shape of the candle resembles the figure of a man with hanging legs, hence the name.

Key Features:

- Opening price is higher than closing price ($O > C$), although it can be the opposite.
- Small candle body at the upper part of the price range.
- Long lower shadow, typically 2-3 times longer than the body.
- No upper shadow or a very short one ($TS == 0$).
- Forms in an uptrend.

[image: Hanging Man Pattern]

Interpretation

Hanging Man is considered a warning of a potential end to an uptrend:

- The long lower shadow shows that the price dropped significantly during the trading session, indicating the emergence of sellers.
- Despite the fact that buyers were able to push the price back to the upper part of the range, the very fact of a significant price drop in an uptrend is a warning signal.
- The color of the candle body is less important, although a black/red body is considered more bearish than a white/green one.
- The longer the lower shadow and the smaller the candle body, the stronger the signal.
- This pattern requires confirmation from subsequent candles.

Trading Strategies

Hanging Man usually requires additional confirmation before making trading decisions:

- Wait for a confirming bearish candle after the Hanging Man formation before entering a short position.
- Place a stop-loss above the high of the Hanging Man.
- Consider trading volume - high volume during pattern formation and on the confirming candle strengthens the bearish signal.
- Combine with other technical indicators, such as RSI in the overbought zone or divergence on oscillators.
- Possible use for partial closing of existing long positions even without opening a short position.
- Pay special attention to the pattern if it forms at important resistance levels or after a prolonged upward movement.

See also

[Pattern Hammer](#)

Inverted Hammer

Pattern Inverted Hammer

Inverted Hammer is a bullish candlestick pattern that forms during a downtrend. The candle has a small body at the lower part and a long upper shadow, with the lower shadow being absent or very short. It looks like an upside-down hammer.

Key Features:

- Opening price is lower than closing price ($O < C$), although it can be the opposite.
- Small candle body at the lower part of the price range.
- Long upper shadow, typically 2-3 times longer than the body.
- No lower shadow or a very short one.
- Forms in a downtrend.

[image: Inverted Hammer Pattern]

Interpretation

Inverted Hammer is considered a potential signal of a downtrend reversal:

- The long upper shadow shows that buyers tried to push the price significantly higher but couldn't maintain it at high levels.
- Despite the inability to close at the highs, the appearance of buyers after a prolonged downtrend may signal a change in sentiment.
- This pattern is not as strong as the classic Hammer and requires confirmation from subsequent candles.
- The color of the body is less important, although a white/green Inverted Hammer is considered more bullish.

Trading Strategies

Inverted Hammer requires a cautious approach and confirmation:

- Mandatory waiting for confirmation from the next candle - a strong bullish candle after an Inverted Hammer significantly increases the probability of a reversal.
- Placing a stop-loss level below the low of the Inverted Hammer.
- Using a smaller position size compared to more reliable reversal patterns.
- Combining with oversold indicators such as RSI or Stochastic to increase the probability of a successful trade.
- Paying attention to trading volume - high volume increases the significance of the signal.

See also

[Pattern Hammer](#)

[Pattern Shooting Star](#)

Morning Star

Pattern Morning Star

Morning Star is a bullish reversal candlestick pattern consisting of three candles that forms in a downtrend. This pattern shows a transition from bearish to bullish sentiment through a period of uncertainty or consolidation.

Key Features:

- First candle is black (bearish) with opening price higher than closing price ($O > C$).
- Second candle has a small body (can be either bullish or bearish) and forms a gap down from the first candle. The body is significantly smaller than the first candle's body ($B < pB * 0.5m$).
- Third candle is white (bullish) with opening price lower than closing price ($O < C$), which closes deep inside the body of the first candle.
- Forms in a downtrend.

[image: Morning Star Pattern]

Interpretation

Morning Star is considered a strong signal of a potential reversal of a downtrend:

- The first candle confirms the strength of the bearish trend.
- The second candle (star) shows weakening of bearish pressure and uncertainty in the market.
- The third candle demonstrates the return of buyers and a shift in control from bears to bulls.
- The deeper the third candle penetrates into the body of the first candle, the stronger the reversal signal.
- If the second candle is a doji (with a very small body), the pattern is called "Morning Doji Star" and is considered an even stronger signal.

Trading Strategies

Morning Star provides good opportunities for entering a long position:

- Enter a long position after the pattern formation, usually at the opening of the fourth candle or when the high of the third candle breaks.
- Place a stop-loss below the low of the second candle or the low of the entire pattern.
- Target profit can be set based on Fibonacci levels relative to the previous downward movement or previous resistance levels.
- Pay attention to volume - increasing volume on the third candle confirms the strength of the bullish reversal.
- Combine with other technical indicators, such as RSI in the oversold zone or support lines, to increase the reliability of the signal.

See also

[Pattern Evening Star](#)

[Pattern Three White Soldiers](#)

Rising Three Methods

Pattern Rising Three Methods

Rising Three Methods is a bullish trend continuation pattern consisting of five candles that forms in an uptrend. This pattern shows a temporary consolidation or rest within an existing uptrend before its continuation.

Key Features:

- First candle is white (bullish) with opening price lower than closing price ($O < C$) and a long body.
- The next three candles are black (bearish) with opening price higher than closing price ($O > C$), with small bodies: ($B < pB * 0.5m$), ($B < ppB * 0.5m$), ($B < pppB * 0.5m$).
- The bodies of the three middle candles do not extend beyond the range of the first candle.
- Fifth candle is white (bullish) with opening price lower than closing price ($O < C$) and a long body ($B > pB * 2$).
- The fifth candle breaks through the high of the first candle and closes higher.
- Forms in an uptrend.

[image: Rising Three Methods Pattern]

Interpretation

Rising Three Methods is considered a reliable signal of uptrend continuation:

- The first long white candle shows the strength of the uptrend.
- Three small black candles represent a temporary consolidation or correction, during which sellers were unable to significantly change the trend.
- The fifth long white candle confirms the return of control to buyers and the continuation of the uptrend.
- This pattern can be viewed as a flag or pennant in classical technical analysis.
- Such a sequence of candles indicates that the correction was used to accumulate long positions before continuing the upward movement.

Trading Strategies

Rising Three Methods provides good opportunities for entering or strengthening long positions:

- Enter a long position at the opening after the fifth candle or when the high of the first candle breaks.
- Place a stop-loss below the low of the correction candles or below the low of the first candle.
- Target profit can be set using a projection equal to the distance from the beginning of the trend to the first candle of the pattern.
- Pay attention to volume - ideally, volume decreases on the three middle candles and significantly increases on the fifth candle.
- Combine with other technical indicators to confirm the signal.
- Consider important resistance levels above the current price that may affect the development of the movement.

See also

[Pattern Falling Three Methods](#)

[Pattern Three White Soldiers](#)

Shooting Star

Pattern Shooting Star

Shooting Star is a bearish reversal candlestick pattern that forms in an uptrend. The candle has a small body at the lower part and a long upper shadow, with the lower shadow being absent or very short. It resembles a star with an upward-pointing tail.

Key Features:

- Opening price is higher than closing price ($O > C$), although it can be the opposite.
- Small candle body at the lower part of the price range.
- Long upper shadow, typically 2-3 times longer than the body.
- No lower shadow or a very short one ($BS == 0$).
- Forms in an uptrend.

[image: Shooting Star Pattern]

Interpretation

Shooting Star is considered a strong signal of a potential reversal of an uptrend:

- The long upper shadow shows that the price rose significantly during the trading session, but then sellers intervened and pushed the price down.
- This indicates the market's rejection of higher prices and a potential shift in sentiment from bullish to bearish.
- The longer the upper shadow, the stronger the potential reversal signal.
- The color of the candle body is less important, although a black/red Shooting Star is considered more bearish than a white/green one.
- The pattern is similar to an Inverted Hammer, but forms in an uptrend and has the opposite meaning.

Trading Strategies

Shooting Star provides opportunities for entering a short position:

- Wait for confirmation from the next candle - a bearish candle after a Shooting Star strengthens the reversal signal.
- Place a stop-loss level above the high of the Shooting Star.
- Set a target profit based on previous support levels or risk/reward ratio.
- Combine with other technical indicators, such as RSI in the overbought zone or MACD with bearish divergence, to confirm trend reversal.
- Higher trading volume during the formation of a Shooting Star increases the reliability of the signal.
- Particularly strong signal if the Shooting Star forms at an important resistance level or after a rapid price rise.

See also

[Pattern Inverted Hammer](#)

Spinning Top

Pattern Spinning Top

Spinning Top is a candlestick pattern characterized by a small candle body and long upper and lower shadows of approximately equal length. This pattern indicates market indecision and a struggle between buyers and sellers.

Key Features:

- Small candle body (can be either white/green or black/red).
- Long upper and lower shadows of approximately equal length.
- Indicates balance between bulls and bears.
- Reflects market indecision and potential trend change.

[image: Spinning Top Pattern]

Interpretation

Spinning Top signals indecision in the market:

- Long shadows show that during the period, the price fluctuated significantly both up and down.
- The small body means that the closing price didn't move far from the opening price, despite high volatility.
- Within an uptrend, it may signal a weakening of bullish momentum.
- Within a downtrend, it may indicate exhaustion of bearish pressure.
- At the top of a trend or before an important resistance level, it may foreshadow a reversal.

Trading Strategies

Spinning Top by itself is not a strong trading signal, but can be used in combination with other factors:

- Look for additional confirmation from subsequent candles before making trading decisions.
- Use as part of more complex candlestick patterns, such as morning or evening stars.
- Increase attention to subsequent price movements after the formation of a Spinning Top at key support or resistance levels.
- Possible tightening of stop-losses when a Spinning Top appears after a prolonged trend.

See also

[Pattern Hammer](#)

[Pattern Inverted Hammer](#)

Tweezer Bottom

Pattern Tweezer Bottom

Tweezer Bottom is a bullish reversal candlestick pattern consisting of two candles that forms in a downtrend. A characteristic feature of the pattern is that both candles have the same or nearly the same low, resembling tweezers with two identical ends.

Key Features:

- First candle is black (bearish) with opening price higher than closing price ($O > C$).
- Second candle is white (bullish) with opening price lower than closing price ($O < C$).
- The second candle has no lower shadow or a very short one ($BS == 0$).
- The upper shadow of the second candle is larger than its body ($TS > B$).
- Both candles have the same or very close lows.
- Forms in a downtrend.

[image: Tweezer Bottom Pattern]

Interpretation

Tweezer Bottom is considered a signal of a potential reversal of a downtrend:

- The first candle confirms the existing downtrend.
- The second candle with the same low shows that bears twice tried to break through the same level but failed.
- This inability to overcome a certain level indicates weakening of bearish momentum and possible beginning of bullish movement.
- The absence of a lower shadow on the second candle with a long upper shadow indicates strong bullish pressure from the low level.
- The pattern is particularly significant if it forms at an important support level or after a prolonged downward movement.

Trading Strategies

Tweezer Bottom requires a cautious approach and often additional confirmation:

- Wait for a confirming bullish candle after the pattern formation before entering a long position.
- Place a stop-loss slightly below the common low of the pattern.
- Consider trading volume - decreasing volume on the first candle and increasing on the second and subsequent bullish candles strengthens the signal.
- Combine with other technical indicators, such as RSI in the oversold zone or bullish divergence on oscillators.
- Possible use for partial or complete closing of existing short positions.
- Pay special attention to subsequent price movements - rapid price growth after the pattern confirms its significance.

See also

Pattern Tweezer Top

Pattern Morning Star

Tweezer Top

Pattern Tweezer Top

Tweezer Top is a bearish reversal candlestick pattern consisting of two candles that forms in an uptrend. A characteristic feature of the pattern is that both candles have the same or nearly the same high, resembling tweezers with two identical ends.

Key Features:

- First candle is white (bullish) with opening price lower than closing price ($O < C$).
- Second candle is black (bearish) with opening price higher than closing price ($O > C$).
- Both candles have the same or very close highs ($H \approx pH$).
- The body of the second candle is significantly larger than the body of the first candle ($B > (pB * 3)$).
- Forms in an uptrend.

[image: Tweezer Top Pattern]

Interpretation

Tweezer Top is considered a signal of a potential reversal of an uptrend:

- The first candle confirms the existing uptrend.
- The second candle with the same high shows that bulls twice tried to break through the same level but failed.
- This inability to overcome a certain level indicates weakening of bullish momentum and possible beginning of bearish movement.
- The more significant body of the second candle compared to the first enhances the bearish signal.
- The pattern is particularly significant if it forms at an important resistance level or after a prolonged upward movement.

Trading Strategies

Tweezer Top requires a cautious approach and often additional confirmation:

- Wait for a confirming bearish candle after the pattern formation before entering a short position.
- Place a stop-loss slightly above the common high of the pattern.
- Consider trading volume - decreasing volume on the first candle and increasing on the second and subsequent bearish candles strengthens the signal.
- Combine with other technical indicators, such as RSI in the overbought zone or divergence on oscillators.
- Possible use for partial or complete closing of existing long positions.
- Pay special attention to subsequent price movements - rapid price decline after the pattern confirms its significance.

See also

[Pattern Tweezer Bottom](#)

White Candle

Pattern White Candle

White candle (bullish candle) is a classic candlestick pattern that forms when the closing price is higher than the opening price. This candle reflects a bullish market sentiment, where buyers controlled the price during the period of candle formation.

Key Features:

- Opening price is lower than closing price ($O < C$).
- The candle body is usually colored white (or green in modern charts).
- Indicates the predominance of buyers over sellers.
- The size of the candle body shows the strength of the bullish movement.

[image: White Candle Pattern]

Interpretation

White candle signals bullish pressure in the market:

- The longer the candle body, the stronger the bullish pressure.
- A long white candle after a downtrend may indicate a potential reversal.
- The presence of short shadows indicates that bulls controlled the price throughout the period.
- Consecutive white candles indicate a steady uptrend.

Trading Strategies

Although a single white candle is not usually an independent trading signal, it can be used as part of a broader strategy:

- Confirmation of an uptrend or reversal after a downward movement.
- Looking for long white candles at support levels for potential long positions.
- Using in combination with other candlestick patterns, for example, a white candle after a bullish engulfing.
- Identifying resistance levels after a series of consecutive white candles.

See also

[Pattern Black Candle](#)

[Pattern White Marubozu](#)

White Marubozu

Pattern White Marubozu

White Marubozu is a bullish candlestick pattern characterized by the absence of shadows on both ends of the candle. The term "marubozu" comes from the Japanese word meaning "bald-headed" or "shaved," reflecting the appearance of the candle without shadows.

Key Features:

- Opening price is lower than closing price ($O < C$).
- The candle body is completely filled, without upper and lower shadows.
- Opening price equals the low of the candle, and closing price equals the high of the candle.
- Represents a strong bullish movement, where buyers controlled the price throughout the period.

[image: White Marubozu Pattern]

Interpretation

White Marubozu is considered a strong bullish signal:

- The absence of shadows indicates complete buyer domination - the price opened at the low and continuously rose until the period closed.
- A long White Marubozu indicates very strong bullish pressure.
- The appearance of this pattern after a downtrend may signal a reversal.
- Within an uptrend, it confirms the strength of the movement.

Trading Strategies

White Marubozu provides a stronger signal than a regular white candle:

- Opportunity to enter a long position after the formation of a White Marubozu, especially if it appears at an important support level.
- Using the closing price of White Marubozu as a support level when setting stop-losses.
- Combining with other technical indicators to confirm the signal.
- Paying attention to trading volume - high volume enhances the significance of the signal.

See also

[Pattern Black Marubozu](#)

[Pattern White Candle](#)

Complex Patterns

Complex Candle Patterns

Overview

The `ComplexCandlePattern` class allows creating complex candle patterns by combining multiple simple patterns (`ICandlePattern`) into one. When recognizing a complex pattern, each inner pattern is checked sequentially on its segment of candles. The pattern is considered recognized only if all inner patterns match.

ICandlePattern

The base interface for all candle patterns:

```
public interface ICandlePattern : IPersistable
{
    // Pattern name
    string Name { get; }

    // Number of candles required for recognition
    int CandlesCount { get; }

    // Check whether the pattern is recognized on the given candles
    bool Recognize(ReadOnlySpan<ICandleMessage> candles);
}
```

The `CandlePatternRegistry` registry contains a set of built-in patterns: Flat, White, Black, Hammer, BullishEngulfing, MorningStar, ThreeWhiteSoldiers, and others.

ComplexCandlePattern

The `ComplexCandlePattern` class implements `ICandlePattern` and combines multiple inner patterns:

```
public class ComplexCandlePattern : ICandlePattern
{
    // Create an empty pattern
    public ComplexCandlePattern() { }

    // Create a pattern with a name and set of inner patterns
    public ComplexCandlePattern(string name, IEnumerable<ICandlePattern> inner);

    // Complex pattern name
    public string Name { get; }

    // Inner patterns
    public IEnumerable<ICandlePattern> Inner { get; }

    // Total number of candles (sum of CandlesCount for all inner patterns)
    public int CandlesCount { get; }
}
```

When `Recognize` is called, the candle array is split into sequential segments according to each inner pattern's `CandlesCount`. If at least one inner pattern does not match, the method returns `false`.

Example: Creating a Complex Pattern

```

using StockSharp.Algo.Candles.Patterns;

// Create a complex pattern: first a bearish candle, then bullish engulfing
var complex = new ComplexCandlePattern(
    "Reversal Up",
    new ICandlePattern[]
    {
        CandlePatternRegistry.Black,           // 1 candle: bearish
        CandlePatternRegistry.BullishEngulfing, // 2 candles: bullish engulfing
    }
);

// 3 candles are required for recognition (1 + 2)
Console.WriteLine($"Candles required: {complex.CandlesCount}"); // 3

```

ICandlePatternProvider

The ICandlePatternProvider interface manages pattern storage and lookup:

```

public interface ICandlePatternProvider
{
    // Events for pattern creation, replacement, and deletion
    event Action<ICandlePattern> PatternCreated;
    event Action<ICandlePattern, ICandlePattern> PatternReplaced;
    event Action<ICandlePattern> PatternDeleted;

    // Initialize storage
    ValueTask InitAsync(CancellationToken cancellationToken);

    // All available patterns
    IEnumerable<ICandlePattern> Patterns { get; }

    // Find a pattern by name
    bool TryFind(string name, out ICandlePattern pattern);

    // Remove a pattern
    bool Remove(ICandlePattern pattern);

    // Save (create or replace) a pattern
    void Save(ICandlePattern pattern);
}

```

Implementations

- InMemoryCandlePatternProvider -- stores patterns in memory. On initialization, it loads all built-in patterns from CandlePatternRegistry.All.
- CandlePatternFileStorage -- saves custom patterns to a file (JSON). Built-in patterns from InMemoryCandlePatternProvider are also available through this provider.

Example: Working with ICandlePatternProvider

```

using StockSharp.Algo.Candles.Patterns;
using StockSharp.Configuration;

// Create file-based pattern storage
var executor = new ChannelExecutor();
var provider = new CandlePatternFileStorage(
    Paths.FileSystem,
    Paths.CandlePatternsFile,
    executor
);

```

```
// Initialize (loads built-in + custom patterns from file)
await provider.InitAsync(Cancellation.Token.None);

// Subscribe to new pattern creation event
provider.PatternCreated += pattern =>
{
    Console.WriteLine($"Pattern created: {pattern.Name}");
};

// Create and save a complex pattern
var myPattern = new ComplexCandlePattern(
    "My Pattern",
    new ICandlePattern[]
    {
        CandlePatternRegistry.Hammer,
        CandlePatternRegistry.White,
    }
);

provider.Save(myPattern);

// Find a pattern by name
if (provider.TryFind("My Pattern", out var found))
{
    Console.WriteLine($"Found: {found.Name}, candles: {found.CandlesCount}");
}
```

ExpressionCandlePattern

For creating patterns based on formulas, ExpressionCandlePattern is used. Each candle in the pattern is described by a CandleExpressionCondition expression, with the following variables available:

Variable	Description
O	Open price
H	High price
L	Low price
C	Close price
V	Volume
B	Candle body
LEN	Candle length
BS	Bottom shadow
TS	Top shadow

The p prefix refers to the previous candle (p0, pC), pp -- to two candles back, and so on.

All built-in patterns in CandlePatternRegistry are built using ExpressionCandlePattern.

Position Management

Position Management

StockSharp provides a flexible position management system that allows you to track the current state of positions, calculate them based on orders or trades, and maintain a lifecycle history (opening, closing, reversals).

PositionManager

The [PositionManager](#) class implements the [IPositionManager](#) interface and serves as the primary component for calculating current positions based on incoming messages.

Creating the Manager

The constructor accepts two parameters:

```
var state = new PositionManagerState();
var manager = new PositionManager(byOrders: false, state);
```

- `byOrders = true` -- the position is calculated based on order balance changes. Suitable when the trading system receives order state updates but not individual trades.
- `byOrders = false` -- the position is calculated based on trade volumes (recommended mode). Provides more accurate accounting of executed operations.

Processing Messages

The `ProcessMessage` method accepts an incoming message ([Message](#)) and returns a [PositionChangeMessage](#) when the position changes, or `null` if the position has not changed:

```
var posChange = manager.ProcessMessage(executionMsg);

if (posChange != null)
{
    Console.WriteLine($"Position: {posChange.CurrentValue}");
}
```

IPositionManagerState

The [IPositionManagerState](#) interface describes the internal state of the position manager. The [PositionManagerState](#) implementation stores information about current orders and positions.

Main Methods

Method	Description
AddOrGetOrder	Registers a new order or returns an existing one by <code>transactionId</code>
TryGetOrder	Retrieves order parameters (instrument, portfolio, direction, balance)
UpdateOrderBalance	Updates the current order balance after partial execution
RemoveOrder	Removes a completed order from tracking

UpdatePosition	Updates the position by instrument and portfolio, returns the new value
Clear	Resets all manager state

Working with State Example

```
var state = new PositionManagerState();

// Register an order
state.AddOrGetOrder(
    transactionId: 12345,
    securityId: secId,
    portfolioName: "MyPortfolio",
    side: Sides.Buy,
    volume: 100,
    balance: 100
);

// Update after partial execution
state.UpdateOrderBalance(12345, newBalance: 60);

// Update position directly
var newPosition = state.UpdatePosition(secId, "MyPortfolio", diff: 40);
Console.WriteLine($"Current position: {newPosition}");

// Clear
state.Clear();
```

PositionLifecycleTracker

The [PositionLifecycleTracker](#) class tracks the complete lifecycle of positions -- from opening to closing (round-trip). This is useful for analyzing individual trades, calculating profit for each position, and generating reports.

Key Features

- **History:** the History property ([IReadOnlyList<ReportPosition>](#)) contains all completed round-trip positions.
- **RoundTripClosed event:** fires when a position is closed (value reached zero) or reversed (position sign changed).
- **ProcessPosition method:** accepts a [Position](#) object and updates the internal state.

Detected States

State	Description
Opening	Position transitions from zero to a non-zero value
Closing	Position value reaches zero
Reversal	Position sign changes (e.g., from long to short)

Usage Example

```
var tracker = new PositionLifecycleTracker();

tracker.RoundTripClosed += report =>
{
```

```

        Console.WriteLine($"Round-trip completed:");
        Console.WriteLine($"  Opened: {report.OpenTime}");
        Console.WriteLine($"  Closed: {report.CloseTime}");
    };

    // Process position updates
    tracker.ProcessPosition(position);

    // View history
    foreach (var report in tracker.History)
    {
        Console.WriteLine($"  {report.OpenTime} -> {report.CloseTime}");
    }

```

PositionMessageAdapter

The [PositionMessageAdapter](#) class is a wrapper around a message adapter that automatically calculates positions from the message stream. It is used within the internal connector infrastructure.

How It Works

```

var innerAdapter = connector.Adapter;
var posManager = new PositionManager(byOrders: false, new PositionManagerState());
var posAdapter = new PositionMessageAdapter(innerAdapter, posManager);

```

The adapter intercepts order execution and trade messages, calls `PositionManager.ProcessMessage`, and generates corresponding `PositionChangeMessage` instances for upstream handlers.

Positions in Strategies

In the [Strategy](#) class, the current position is accessed through the `Position` property:

```

// Current position for the primary instrument
decimal currentPosition = Position;

// Close position
if (Position > 0)
    SellMarket(Math.Abs(Position));
else if (Position < 0)
    BuyMarket(Math.Abs(Position));

// Or through a built-in method
ClosePosition();

```

For more details about trading operations in strategies, see the [Trading Operations](#) section.

See Also

- [Trading Operations](#)
- [Position Protection](#)
- [Target Position Management](#)
- [Reporting](#)

Adapter Routing

Adapter Routing

StockSharp supports simultaneous connections to multiple exchanges and brokers. The routing system (basket routing) manages which messages are directed to which adapters, ensuring transparent operation with multiple connections.

General Architecture

When using multiple adapters, the connector automatically creates a basket that combines all connections. The router determines which adapter each specific message should be directed to -- market data subscriptions, transactions, portfolio requests, etc.

AdapterRouter

The [IAdapterRouter](#) interface defines the logic for routing messages between adapters.

Main Methods

Method	Description
GetAdapters	Returns a list of adapters suitable for processing the given message
GetSubscriptionAdaptersAsync	Asynchronously determines adapters for market data subscriptions
GetPortfolioAdapter	Returns the adapter bound to a specific portfolio
TryGetOrderAdapter	Finds the adapter through which an order was registered
SetSecurityAdapter	Binds an instrument to a specific adapter
SetPortfolioAdapter	Binds a portfolio to a specific adapter

Routing Priorities

The system determines the target adapter in the following priority order:

1. **Explicit specification** -- if an adapter is specified in the message via the `message.Adapter` property, that adapter is used.
2. **Instrument binding** -- mapping set via `SetSecurityAdapter`.
3. **Data type binding** -- adapters registered for a specific message type.
4. **Supported type filtering** -- adapters that support the given message type are selected.

Configuring Routing

```
var router = connector.Adapter.InnerAdapters;  
  
// Bind instrument to adapter for receiving tick data  
router.SetSecurityAdapter(  
    secId,  
    DataType.Ticks,  
    binanceAdapter
```

```
);

// Bind portfolio to adapter for transactions
router.SetPortfolioAdapter(
    "MyPortfolio",
    interactiveBrokersAdapter
);
```

Managing Connections

Connection States

Each adapter in the basket goes through standard connection states:

- **Disconnected** -- disconnected
- **Connecting** -- connection in progress
- **Connected** -- connected
- **Disconnecting** -- disconnection in progress

The basket aggregates the states of all nested adapters.

Aggregation Parameters

The `ConnectDisconnectEventOnFirstAdapter` property determines when the basket is considered connected:

- `true` -- the connection event fires when the **first** adapter connects (default). Allows starting work without waiting for all connections.
- `false` -- the event fires only after **all** adapters have connected.

```
// Wait for all adapters to connect
connector.Adapter.InnerAdapters.ConnectDisconnectEventOnFirstAdapter = false;

connector.Connected += () =>
{
    Console.WriteLine("All adapters connected");
};

connector.Connect();
```

Parent and Child Subscriptions

When working with multiple adapters, a single subscription can be split into multiple child subscriptions, each directed to its own adapter. The system automatically:

- Creates child subscriptions for each suitable adapter
- Aggregates responses before notifying the parent subscription
- Handles partial errors (if one adapter fails to subscribe, the others continue working)

Multi-Exchange Example

```
// Adding adapters
connector.Adapter.InnerAdapters.Add(binanceAdapter);
connector.Adapter.InnerAdapters.Add(bybitAdapter);

connector.Connect();
```

```
// Subscribing to ticks -- will be automatically routed
// to all adapters supporting the given instrument
var subscription = new Subscription(DataType.Ticks, security);
connector.Subscribe(subscription);
```

Pending Message Queue

If no adapter is connected when a message is sent, the message is placed in a pending queue (IPendingMessageState). When an adapter connects, all accumulated messages are automatically sent.

```
// Registering an order before connecting -- the order will be sent
// automatically after the connection is established
connector.RegisterOrder(order);
connector.Connect();
```

Configuring Multiple Connections

Programmatic Configuration

```
// Creating adapters
var binance = new BinanceMessageAdapter(connector.TransactionIdGenerator)
{
    Key = "<API_KEY>",
    Secret = "<API_SECRET>.Secure()",
};

var ib = new InteractiveBrokersMessageAdapter(connector.TransactionIdGenerator)
{
    Address = InteractiveBrokersMessageAdapter.DefaultAddress,
};

// Adding to the basket
connector.Adapter.InnerAdapters.Add(binance);
connector.Adapter.InnerAdapters.Add(ib);

// Configuring routing
connector.Adapter.InnerAdapters.SetPortfolioAdapter("BinancePortfolio", binance);
connector.Adapter.InnerAdapters.SetPortfolioAdapter("IBPortfolio", ib);

connector.Connect();
```

Graphical Configuration

For visual connection configuration, use the graphical configuration component. See the [Graphical Configuration](#) section for details.

Order Tracking

The router automatically tracks which adapter was used to register each order. When receiving order updates (state changes, trades), the system routes them through the same adapter:

```
// The order will be registered through the adapter bound to the portfolio
var order = new Order
{
    Security = security,
    Portfolio = portfolio,
    Side = Sides.Buy,
    Price = price,
    Volume = volume,
```

```
};  
  
connector.RegisterOrder(order);  
  
// Cancellation will go through the same adapter automatically  
connector.CancelOrder(order);
```

See Also

- [Connectors](#)
- [Graphical Configuration](#)
- [Position Management](#)

Order books

Order Book

Description

The order book (a.k.a. market depth) is information about the current buy and sell orders for a specific security, organized by price levels. In StockSharp, the order book provides data on demand and supply, allowing for real-time market analysis.

Structure

The [Order Book](#) contains two lists of orders:

- Buy orders, sorted by descending price - [Bids](#).
- Sell orders, sorted by ascending price - [Asks](#).

Each order includes a price and volume.

Usage

Order book data is used for:

- Identifying price levels with maximum order volumes, which may indicate potential support or resistance levels.
- Assessing market liquidity for a security.
- Developing trading strategies based on analysis of changes in the order book.

Data Retrieval

In StockSharp, subscribing to order book data and receiving updates is done through the corresponding [API methods](#).

Subscriptions

Subscriptions

Subscribing to the Order Book

To subscribe to the order book in StockSharp, you need to perform the following steps:

1. Subscribe to the event for receiving order books [Connector.OrderBookReceived](#) and handle the [IOrderBookMessage](#) interface objects:

```
// event handler
private void OnOrderBookReceived(Subscription subscription, IOrderBookMessage orderBook)
{
    // Here you can process the order book data, for example, display it on the screen or use it in
    // your trading strategy
    Console.WriteLine($"Received order book for {orderBook.SecurityId}. Best buy price:
    {orderBook.GetBestBid()?.Price}, Best sell price: {orderBook.GetBestAsk()?.Price}");
}

// subscribing to the event
connector.OrderBookReceived += OnOrderBookReceived;
```

It's important to subscribe to the [Connector.OrderBookReceived](#) event **before** sending a subscription request for the order book. This ensures that you won't miss any data if order books start arriving very quickly after the subscription request is sent.

2. Send a subscription request using the [Connector.Subscribe](#) method:

```
var security = GetSecurity(); // Get the Security object you want to subscribe to

// subscribe to the order book
var subscription = new Subscription(DataType.MarketDepth, security);
connector.Subscribe(subscription);
```

Unsubscribing from the Order Book

To unsubscribe from the order book, call the [Connector.UnSubscribe](#) method:

```
connector.UnSubscribe(subscription);
```

Clarification on Receiving Order Books

When working with the [Connector.OrderBookReceived](#) event, it's important to understand that the order books coming through this event are already compiled and ready for use. This means that regardless of the method of data transmission by the source - whether differential data (only changes in the order book) or full snapshots of the order book - the StockSharp platform processes this data in such a way that the trader receives a complete and updated order book.

The platform automatically integrates changes into the order book, updating its contents to the current state before calling the [Connector.OrderBookReceived](#) event. This simplifies working with data, as traders do not need to independently process differential data or compile the order book from consecutive snapshots. Thus, you can be confident that the data received in the event handler reflects the latest state of the order book at the time of the event.

This significantly simplifies the development of trading strategies and market analysis, as traders can focus directly on the logic of their strategies, without spending time on the technical aspects of compiling and

processing order book data.

Usage Example

Examples of using the order book are available in the *Samples/01_Basic/02_MarketDepths* project on [GitHub](#) or in the StockSharp API archive, which can be obtained through the [Installer](#). These examples provide practical illustrations of connecting to a trading system, subscribing to a filtered order book, and processing the received data, which can serve as a good starting point for developing your own trading strategies.

See Also

[Subscriptions](#)

Examples

Examples with the Order Book

Getting the Best Prices

To get the best prices from the order book, it's important to focus on the first elements of the lists for buy orders ([Bids](#)) and sell orders ([Asks](#)), as these represent the most favorable available prices for transactions:

```
var bestBid = orderBook.Bids.FirstOrDefault();
var bestAsk = orderBook.Asks.FirstOrDefault();

if (bestBid != null)
{
    Console.WriteLine($"Best buy price: {bestBid.Price}");
}

if (bestAsk != null)
{
    Console.WriteLine($"Best sell price: {bestAsk.Price}");
}
```

Or use the ready-made extension methods [GetBestBid](#) and [GetBestAsk](#):

```
var bestBid = orderBook.GetBestBid();
var bestAsk = orderBook.GetBestAsk();

if (bestBid != null)
{
    Console.WriteLine($"Best buy price: {bestBid.Price}, volume: {bestBid.Volume}");
}
else
{
    Console.WriteLine("No best buy orders.");
}

if (bestAsk != null)
{
    Console.WriteLine($"Best sell price: {bestAsk.Price}, volume: {bestAsk.Volume}");
}
else
{
    Console.WriteLine("No best sell orders.");
}
```

Analyzing Order Book Depth

To analyze the depth of the order book, you can iterate through the items in the [Bids](#) and [Asks](#) lists, starting from the beginning of the list. This provides an overview of the distribution of orders at different price levels and helps identify potential support and resistance levels:

```
foreach (var bid in orderBook.Bids)
{
    Console.WriteLine($"Buy price: {bid.Price}, volume: {bid.Volume}");
}

foreach (var ask in orderBook.Asks)
{
    Console.WriteLine($"Sell price: {ask.Price}, volume: {ask.Volume}");
}
```


Searching for Volumes in the Order Book

An algorithm for searching for significant volumes in the order book helps identify levels where large orders accumulate. This can indicate the interest of major players and serve as an additional signal when making trading decisions.

Algorithm:

1. Determine a volume threshold that will be considered significant.
2. Iterate through the orders in the [Bids](#) and [Asks](#) lists, comparing the volume of each order against the threshold value.
3. Record the price levels where orders with volume above the threshold were found.

```
double significantVolumeThreshold = 10000; // Example of a threshold value

Console.WriteLine("Significant volumes in the order book:");

foreach (var bid in orderBook.Bids)
{
    if (bid.Volume >= significantVolumeThreshold)
    {
        Console.WriteLine($"Buy: Price {bid.Price}, volume {bid.Volume}");
    }
}

foreach (var ask in orderBook.Asks)
{
    if (ask.Volume >= significantVolumeThreshold)
    {
        Console.WriteLine($"Sell: Price {ask.Price}, volume {ask.Volume}");
    }
}
```

This algorithm helps highlight levels with significant volumes, which may play a key role in market price movements.

Filtered order book

Filtered Order Book

A filtered order book is a specialized tool in StockSharp that allows traders and automated strategies to operate in the market, excluding their own orders from consideration. This is critically important when using multiple strategies in parallel to prevent situations where one strategy starts "trading" with another, not realizing that the volume in the order book comes from another market participant or is the result of actions by another strategy running in parallel.

Advantages of the Filtered Order Book

- **Avoidance of self-trading:** Strategies will not execute orders against themselves or each other when run in parallel.
- **Purity of analysis:** Allows strategies to analyze market conditions based solely on external orders, without distortions caused by their own orders.
- **Execution efficiency:** Helps improve the quality of order execution by minimizing the impact on the market price by one's own orders.

Subscription Example

The approach to working with the filtered order book uses the same method as [subscribing to a regular order book](#), but with a different [DataType](#) value. Below is an example illustrating the subscription to a filtered order book for a specific instrument:

1. **Subscribing to the order book update event:** [Connector.OrderBookReceived](#) to receive order book updates. This event is used for both the regular and the filtered order book.

When processing the event, check the [Subscription.DataType](#) in the subscription object associated with the event. If [Subscription.DataType](#) is [DataType](#), it indicates that the received order book is filtered:

```
connector.OrderBookReceived += (sender, subscription, orderBook) =>
{
    if (subscription.DataType == DataType.FilteredMarketDepth)
    {
        // Handling logic for the filtered order book
        Console.WriteLine($"Received filtered order book for {orderBook.SecurityId}.");
    }
};
```

2. **Sending the subscription:** Form a [Subscription](#) object and send it to the connector:

```
var subscription = new Subscription(DataType.FilteredMarketDepth, security);
connector.Subscribe(subscription);

// or like this
//var subscription = connector.SubscribeFilteredMarketDepth(security);
```

Conclusion

Using the filtered order book in StockSharp provides traders and strategy developers with a flexible tool for market analysis, allowing them to avoid unwanted self-interaction between simultaneously running strategies and simplifying decision-making based on market order data.

Sparse Order Book

A sparse order book is a presentation of the order book that displays all possible price levels, including those without any active orders at the moment. This approach allows traders to visually assess the "gaps" between orders, i.e., price levels where there are no buy or sell orders, giving an insight into potential resistance or support levels.

Why Use a Sparse Order Book

Using a sparse order book has several advantages:

1. **Visualization of gaps:** It's easier to identify price levels with missing orders, which can indicate potential entry or exit points.
2. **Liquidity analysis:** A clear representation of order distribution helps assess the instrument's liquidity at different price levels.
3. **Strategic planning:** Understanding the structure of the order book allows for more accurate trading operations planning, considering potential "voids" in liquidity.

Creating a Sparse Order Book

To work with a grouped order book, you first need to set up reception through [subscriptions](#), and then call the extension method [Sparse](#). The method takes the following parameters:

- **priceRange** - the price difference up to which levels need to be expanded.
- **priceStep** - the trading instrument's price step. It is used in case priceRange has a lower precision on price levels than priceStep, and it is necessary to round the obtained prices to the instrument's price step.

```
// It is assumed that orderBook is an IOrderBookMessage object obtained from StockSharp
var sparseDepth = orderBook.Sparse(priceRange, priceStep);

// Now, sparseDepth contains a representation of the original order book,
// where all possible price levels are considered, including those without any orders.
```

In this example, [Sparse](#) is used to create a sparse order book, which allows for displaying all price levels, even those without any active orders. This can be useful for analyzing potential "empty" levels that may serve as support or resistance levels.

Grouped

Grouped Order Book

In addition to the [sparse order book](#), it can be useful to use a grouped order book, where orders are aggregated over broader price ranges to simplify analysis and identify general trends in demand and supply.

Advantages of a grouped order book:

- **Simplified analysis:** Aggregating order data simplifies the perception of the overall market picture.
- **Trend identification:** It's easier to identify key price levels where the majority of orders are concentrated.

Implementation of a grouped order book:

To work with a grouped order book, it's necessary to first set up reception through [subscriptions](#), and then call the extension method [Group](#):

```
// Grouping order book data with a price aggregation step, for example, 0.5 units of price
var groupedDepth = orderBook.Group(0.5);

// groupedDepth now contains an order book in which orders are grouped
// by price levels with the specified aggregation step.
```

The [Group](#) method allows for the aggregation of orders in the book over larger price levels, simplifying visual market analysis and helping to identify the main levels of demand and supply without the need to analyze every individual price change.

Display

Display

See

[Market Depth Control](#)

Strategies

Strategies in StockSharp

Introduction

StockSharp provides a powerful infrastructure for creating, testing, and running trading strategies. The foundation for developing algorithmic trading strategies is the base class [Strategy](#), which provides a set of standard functions and abstractions for working with market data, executing trading operations, and analyzing results.

Navigation

Strategy Basics

- [Market Data Subscriptions in Strategies](#) - Detailed guide on using market data subscriptions in strategies. Explains creating and configuring subscriptions, managing their lifecycle, and monitoring their state.
- [Indicators in Strategies](#) - Information about working with technical analysis indicators in strategies. Covers adding indicators to a strategy, controlling their formation, and using them in trading logic.
- [Trading Operations in Strategies](#) - Guide to performing trading operations in strategies. Describes methods for creating and sending orders, closing positions, and monitoring their status.
- [Position Protection](#) - Description of mechanisms for protecting open positions using Take Profit and Stop Loss. Examines local and server approaches to position protection.
- [Strategy Parameters](#) - Guide to working with strategy parameters through [StrategyParam<T>](#). Describes how to create configurable parameters, set their display in GUI, and use them in optimization.
- [Logging in Strategies](#) - Guide to using the logging mechanism in strategies for tracking and debugging algorithm performance.

Advanced Features

- [Strategy Platform Compatibility](#) - Recommendations for creating strategies compatible with various StockSharp platforms: [Designer](#), [Shell](#), [Runner](#), and cloud testing.
- [High-Level APIs in Strategies](#) - Description of high-level methods to simplify working with subscriptions, indicators, charts, and position protection. Explains how to write cleaner code by focusing on trading logic.
- [Working with Charts in Strategies](#) - Guide to visualizing strategy data on a chart. Explains how to access the chart, create areas, add elements, and render data.
- [Saving and Loading Settings](#) - Description of the mechanism for saving and loading strategy settings through [Strategy.Save](#) and [Strategy.Load](#) methods.
- [State Loading](#) - Guide to loading previously executed orders and trades into a strategy, for example, when restarting a strategy during a trading session.
- [Price Rounding](#) - Guide to correctly rounding prices in strategies using the [ShrinkPrice](#) method.
- [Unit Type](#) - Description of the [Unit](#) data type to simplify arithmetic operations on quantities like percentages, points, or pips.
- [Event Model](#) - Explanation of the strategy event model based on [IMarketRule](#). Covers creating rules to

react to market events, combining conditions, and managing rule lifecycles.

Getting Started with Strategy Development

To begin developing your own strategy, it is recommended to:

1. Familiarize yourself with the basics of working with strategies to understand the general principles of strategies in StockSharp.
2. Study the [Market Data Subscriptions in Strategies](#) section to understand the mechanism of receiving and processing market data.
3. Review the [Indicators in Strategies](#) section to understand working with technical analysis indicators.
4. Explore the [Trading Operations in Strategies](#) section to understand trading operation mechanisms.
5. Examine the [Strategy Parameters](#) section to learn about strategy configuration mechanisms.
6. Get acquainted with the [High-Level APIs in Strategies](#) section to simplify strategy code using built-in high-level functions.

Strategy Testing

StockSharp provides various methods for testing strategies:

- **Historical Data Testing** - Allows evaluating strategy effectiveness on historical data.
- **Parameter Optimization** - Helps find optimal strategy parameter values.
- **Virtual Account Testing** - Allows checking strategy performance in real-time mode without risking real funds.

Detailed descriptions of testing methods and strategy performance evaluation can be found in the [Testing](#) section.

Subscriptions

Market Data Subscriptions in Strategies

In StockSharp, strategies use a subscription mechanism to receive market data. This approach is the primary and preferred method of obtaining data in trading strategies.

Subscription Basics

Subscriptions in strategies are based on the general [StockSharp subscription mechanism](#). They provide a centralized and unified way of obtaining all types of market data.

Creating a Subscription in a Strategy

In the [OnStarted2](#) method of the strategy, you can create and start a subscription for the required data:

```
protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    // Creating a subscription for 5-minute candles directly through DataType
    var subscription = new Subscription(
        DataType.TimeFrame(TimeSpan.FromMinutes(5)),
        Security);

    // If additional parameters are required, you can configure the subscription
    subscription.From = DateTimeOffset.Now.Subtract(TimeSpan.FromDays(7));

    // Creating a rule to process incoming candles
    Connector
        .WhenCandlesFinished(subscription)
        .Do(ProcessCandle)
        .Apply(this);

    // Starting the subscription
    Connector.Subscribe(subscription);
}
```

In this example, a subscription is created for 5-minute candles using a convenient constructor that accepts `DataType` and `Security`. If necessary, you can additionally configure subscription parameters, such as the history period.

Advantages of Subscriptions in Strategies

Using subscriptions in strategies has several advantages compared to direct subscription to [Strategy.Connector](#) events:

1. **Isolation** - each subscription works independently, allowing different types of data to be received for different instruments without mutual interference. This also protects the strategy from receiving data intended for other strategies running in parallel. With direct subscription to connector events, you would have to additionally filter data to exclude information from other strategies.
2. **State Management** - subscriptions have clear states ([SubscriptionStates](#)), which allow precise determination of whether historical data is currently being received or the subscription has already transitioned to online mode.
3. **Automatic Strategy State Control** - the strategy automatically tracks the state of all its subscriptions and transitions to online mode ([IsOnline](#)) only when all subscriptions have gone online.
4. **Code Uniformity** - subscriptions use a unified approach, independent of the type of requested data.

5. **Integration with Rules** - subscriptions easily integrate with strategy [event model](#) through rules.
6. **Automatic Subscription Management** - when the strategy stops, all its subscriptions are automatically canceled, freeing up resources.
7. **Historical Data Support** - ability to load historical data before transitioning to real-time data.

Monitoring Subscription States

The strategy automatically tracks the state of all subscriptions to control its operating mode:

```
private void CheckRefreshOnlineState()
{
    bool nowOnline = ProcessState == ProcessStates.Started;

    if (nowOnline)
        nowOnline = _subscriptions.CachedKeys
            .Where(s => !s.SubscriptionMessage.IsHistoryOnly())
            .All(s => s.State == SubscriptionStates.Online);

    // Update strategy's IsOnline state
    IsOnline = nowOnline;
}
```

The [Strategy.IsOnline](#) property will be true only when all strategy subscriptions have transitioned to the [SubscriptionStates.Online](#) state. This allows the strategy to understand the moment when it is working with current market data.

Types of Subscriptions

In strategies, you can use subscriptions to various types of market data:

```
// Subscription to candles
var candleSubscription = new Subscription(
    DataType.TimeFrame(TimeSpan.FromMinutes(1)),
    Security);

// Subscription to market depth
var depthSubscription = new Subscription(
    DataType.MarketDepth,
    Security);

// Subscription to tick trades
var tickSubscription = new Subscription(
    DataType.Ticks,
    Security);

// Subscription to Level1 (best bid/ask and other basic information)
var level1Subscription = new Subscription(
    DataType.Level1,
    Security);
```

Processing Subscription Data Through Rules

To process data coming through a subscription, it is recommended to use [rules](#):

```
// Subscription to candles
var subscription = new Subscription(DataType.TimeFrame(TimeSpan.FromMinutes(5)), Security);

// Creating a rule for processing incoming candles
Connector
    .WhenCandlesFinished(subscription) // Rule activation when a completed candle is received
    .Do(ProcessCandle)                // Call processing method
```

```
.Apply(this); // Apply rule to strategy

// Start subscription
Connector.Subscribe(subscription);
```

In the example above, a rule is created that will call the `ProcessCandle` method when each completed candle is received.

Requesting Historical Data

The strategy automatically sets the history load period through the [Strategy.HistorySize](#) property:

```
// Set history load period to 30 days
strategy.HistorySize = TimeSpan.FromDays(30);
```

When creating a subscription, the strategy automatically sets the `From` parameter to load history if it was not explicitly specified.

Canceling Subscriptions

Subscriptions can be manually canceled by calling the [UnSubscribe](#) method:

```
// Cancel subscription
Connector.UnSubscribe(subscription);
```

When stopping the strategy, if the [UnsubscribeOnStop](#) parameter is set to `true` (default), all subscriptions will be automatically canceled.

See Also

- [Market Data Subscriptions](#)
- [Event Model](#)
- [Strategy Platform Compatibility](#)

Indicators

Indicators in Strategy

In StockSharp, the [Strategy](#) class provides a special mechanism for working with indicators, which allows you to control their formation state and determine when the strategy is ready to work.

Indicators Property

The [Strategy.Indicators](#) property is a collection of indicators used in the strategy. This collection is designed to automatically track the state of indicator formation (warm-up).

```
// Accessing the indicators collection
INotifyList<IIIndicator> indicators = strategy.Indicators;
```

IsFormed Property

By default, the implementation of the [Strategy.IsFormed](#) property checks if all indicators in the [Indicators](#) collection are formed:

```
// Standard implementation in the Strategy class
public virtual bool IsFormed => _indicators.AllFormed;
```

A strategy is considered "warmed up" and ready to work when all indicators in the collection are formed (their [IIIndicator.IsFormed](#) property returns true).

Adding Indicators to the Collection

To correctly determine when the strategy is ready, you need to add the indicators you use to the [Indicators](#) collection:

```
protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    // Creating indicators
    _shortSma = new SimpleMovingAverage { Length = ShortSmaLength };
    _longSma = new SimpleMovingAverage { Length = LongSmaLength };

    // Adding indicators to the collection
    Indicators.Add(_shortSma);
    Indicators.Add(_longSma);

    // ...
}
```

Which Indicators to Add

You should only add **independent indicators** to the [Indicators](#) collection. This is an important rule that will help avoid unnecessary waiting and correctly determine when the strategy is ready.

Rules for Adding Indicators:

1. **Independent Indicators** - add indicators that directly process market data (candles, ticks, etc.):

```
// Independent indicators
```

```
var sma = new SimpleMovingAverage { Length = 20 };
var rsi = new RelativeStrengthIndex { Length = 14 };

Indicators.Add(sma);
Indicators.Add(rsi);
```

2. **Indicator Chains** - when using an indicator chain (where the output of one is the input for another), add only the **first indicator in the chain** to the collection:

```
// Indicator chain
var sma = new SimpleMovingAverage { Length = 20 };
var stdev = new StandardDeviation { Length = 20 };
var bollingerBands = new BollingerBands
{
    SmaIndicator = sma,
    DeviationIndicator = stdev
};

// Add only the first indicator in the chain
Indicators.Add(sma);
// DO NOT add indicators dependent on other indicators
// Indicators.Add(stdev); - incorrect
// Indicators.Add(bollingerBands); - incorrect
```

3. **Combined Indicators** - for indicators that use multiple independent indicators (e.g., MACD), add all of them:

```
var fastEma = new ExponentialMovingAverage { Length = 12 };
var slowEma = new ExponentialMovingAverage { Length = 26 };
var signalEma = new ExponentialMovingAverage { Length = 9 };
var macd = new MovingAverageConvergenceDivergence
{
    FastEma = fastEma,
    SlowEma = slowEma,
    SignalEma = signalEma
};

// Add base indicators
Indicators.Add(fastEma);
Indicators.Add(slowEma);
```

Usage Examples

Basic Example with Two Moving Averages

```
public class SmaStrategy : Strategy
{
    private SimpleMovingAverage _longSma;
    private SimpleMovingAverage _shortSma;

    // ...

    protected override void OnStarted2(DateTime time)
    {
        base.OnStarted2(time);

        _longSma = new SimpleMovingAverage { Length = LongSmaLength };
        _shortSma = new SimpleMovingAverage { Length = ShortSmaLength };

        // Add indicators to the collection to track their state
        Indicators.Add(_longSma);
        Indicators.Add(_shortSma);

        // ...
    }
}
```

```

}

private void ProcessCandle(ICandleMessage candle)
{
    // Process indicators
    var longValue = _longSma.Process(candle);
    var shortValue = _shortSma.Process(candle);

    // Check if the strategy is ready before executing trading logic
    if (!IsFormed)
        return;

    // Trading logic
    // ...
}
}

```

Example Using IsFormedAndOnline

To check if the strategy is ready for trading, the [IsFormedAndOnlineAndAllowTrading](#) method is often used, which combines checking indicator formation, online status, and trading permission:

```

private void ProcessCandle(ICandleMessage candle)
{
    // Process indicators
    var longValue = _longSma.Process(candle);
    var shortValue = _shortSma.Process(candle);

    // Comprehensive check of strategy readiness
    if (!IsFormedAndOnlineAndAllowTrading())
        return;

    // Trading logic
    // ...
}

```

Optimizing Indicator Usage

In more complex strategies, it's important to properly organize work with indicators:

```

public class ComplexStrategy : Strategy
{
    private SimpleMovingAverage _sma;
    private RelativeStrengthIndex _rsi;
    private BollingerBands _bollinger;
    private StandardDeviation _stdev;

    protected override void OnStarted2(DateTime time)
    {
        base.OnStarted2(time);

        // Create indicators
        _sma = new SimpleMovingAverage { Length = 20 };
        _rsi = new RelativeStrengthIndex { Length = 14 };

        _stdev = new StandardDeviation { Length = 20 };
        _bollinger = new BollingerBands
        {
            SmaIndicator = _sma,
            DeviationIndicator = _stdev
        };

        // Add only independent indicators
        Indicators.Add(_sma);
        Indicators.Add(_rsi);
    }
}

```

```
// Do not add _stdev and _bollinger as they depend on _sma

// ...
}

// ...
}
```

Advanced Features

You can override the [IsFormed](#) property in your strategy if the standard behavior is not sufficient:

```
public override bool IsFormed
{
    get
    {
        // Standard indicator check
        if (!base.IsFormed)
            return false;

        // Additional strategy readiness conditions
        return _customCondition && _additionalCheck;
    }
}
```

See also

- [List of Indicators](#)
- [Custom Indicator](#)
- [Strategy Compatibility with Platforms](#)

Trading Operations in Strategies

In StockSharp, the [Strategy](#) class provides various methods for working with orders, making it convenient to implement trading strategies.

Order Placement Methods

There are several ways to place orders in StockSharp strategies:

1. Using High-Level Methods

The simplest way is to use built-in methods that create and register an order in a single call:

```
// Buy at market price
BuyMarket(volume);

// Sell at market price
SellMarket(volume);

// Buy at limit price
BuyLimit(price, volume);

// Sell at limit price
SellLimit(price, volume);

// Close the current position at market price
ClosePosition();
```

These methods provide maximum simplicity and code readability. They automatically:

- Create an order object with the specified parameters
- Fill in the necessary fields (instrument, portfolio, etc.)
- Register the order in the trading system

2. Using CreateOrder + RegisterOrder

A more flexible approach is to separate the creation and registration of orders:

```
// Create an order object
var order = CreateOrder(Sides.Buy, price, volume);

// Additional order settings
order.Comment = "My special order";
order.TimeInForce = TimeInForce.MatchOrCancel;

// Register the order
RegisterOrder(order);
```

The [CreateOrder](#) method creates an initialized order object that can be further customized before registration.

3. Direct Creation and Registration of an Order

For maximum control, you can create an order object directly and register it:

```
// Create an order object directly
var order = new Order
{
    Security = Security,
    Portfolio = Portfolio,
    Side = Sides.Buy,
    Type = OrderTypes.Limit,
    Price = price,
    Volume = volume,
    Comment = "Custom order"
};

// Register the order
RegisterOrder(order);
```

For more details on working with orders, see the [Orders](#) section.

Handling Order Events

After registering an order, it's important to track its status. In a strategy, you can:

1. Use Event Handlers

```
// Subscribe to the order received event
OrderReceived += OnOrderReceived;

// Subscribe to the order registration failure event
OrderRegisterFailed += OnOrderRegisterFailed;

private void OnOrderReceived(Order order)
{
    if (order.State == OrderStates.Done)
    {
        // Order executed - perform corresponding logic
    }
}

private void OnOrderRegisterFailed(OrderFail fail)
{
    // Handle order registration error
    LogError($"Order registration error: {fail.Error}");
}
```

2. Use Rules for Orders

A more powerful approach is to use [rules](#) for orders:

```
// Create an order
var order = BuyLimit(price, volume);

// Create a rule that will trigger when the order is executed
order
    .WhenMatched(this)
    .Do(() => {
        // Actions after order execution
        LogInfo($"Order {order.TransactionId} executed");

        // For example, place a stop order
        var stopOrder = SellLimit(price * 0.95, volume);
    })
    .Apply(this);

// Rule for handling registration error
order
```



```

.WhenRegisterFailed(this)
.Do(fail => {
    LogError($"Order registration error: {fail.Error}");
    // Possibly retry with different parameters
})
.Apply(this);

```

Detailed examples of using rules with orders can be found in the [Order Rule Examples](#) section.

Position Management

The strategy also provides methods for position management:

```

// Get current position
decimal currentPosition = Position;

// Close current position
ClosePosition();

// Protect position with stop-loss and take-profit
StartProtection(
    takeProfit: new Unit(50, UnitTypes.Absolute), // take-profit
    stopLoss: new Unit(20, UnitTypes.Absolute),   // stop-loss
    isStopTrailing: true,                         // trailing stop
    useMarketOrders: true                        // use market orders
);

```

Strategy State Before Trading

Before executing trading operations, it's important to ensure that the strategy is in the correct state. StockSharp provides several properties and methods to check the readiness of the strategy:

IsFormed Property

The [IsFormed](#) property indicates whether all indicators used in the strategy are formed (warmed up). By default, it checks that all indicators added to the [Indicators](#) collection are in the state [IIndicator.IsFormed](#) = true.

More about working with indicators in a strategy can be found in the [Indicators in Strategy](#) section.

IsOnline Property

The [IsOnline](#) property shows whether the strategy is in real-time mode. It becomes true only when the strategy is started and all its market data subscriptions have transitioned to the [SubscriptionStates.Online](#) state.

More details about market data subscriptions in strategies can be found in the [Market Data Subscriptions in Strategies](#) section.

TradingMode Property

The [TradingMode](#) property defines the trading mode for the strategy. Possible values:

- [StrategyTradingModes.Full](#) - all trading operations are allowed (default mode)
- [StrategyTradingModes.Disabled](#) - trading is completely disabled
- [StrategyTradingModes.CancelOrdersOnly](#) - only order cancellation is allowed
- [StrategyTradingModes.ReducePositionOnly](#) - only position reduction operations are allowed

This property can be configured through strategy parameters:

```
public SmaStrategy()
{
    _tradingMode = Param(nameof(TradingMode), StrategyTradingModes.Full)
        .SetDisplay("Trading Mode", "Allowed trading operations", "Basic settings");
}
```

Helper Methods for State Checking

For convenient checking of the strategy's readiness to trade, StockSharp provides helper methods:

- [IsFormedAndOnline\(\)](#) - checks that the strategy is in the state `IsFormed = true` and `IsOnline = true`
- [IsFormedAndOnlineAndAllowTrading\(StrategyTradingModes\)](#) - checks that the strategy is formed, is in online mode, and has the necessary trading permissions

The `IsFormedAndOnlineAndAllowTrading` method accepts an optional parameter required of type [StrategyTradingModes](#):

```
public bool IsFormedAndOnlineAndAllowTrading(StrategyTradingModes required =
    StrategyTradingModes.Full)
```

This parameter allows you to specify the minimum level of trading permissions required for a specific operation:

1. **StrategyTradingModes.Full** (default value) - returns true only if the strategy is in full trading mode (`TradingMode = StrategyTradingModes.Full`). Used for operations that can increase a position.
2. **StrategyTradingModes.ReducePositionOnly** - returns true if the strategy is in full trading mode or in position reduction mode only. Used for position closing or partial closing operations.
3. **StrategyTradingModes.CancelOrdersOnly** - returns true with any active trading mode (except `Disabled`). Used for order cancellation operations.

This allows you to selectively permit or prohibit various trading operations depending on the current trading mode:

```
// For placing a new order that increases a position, full trading mode is required
if (IsFormedAndOnlineAndAllowTrading(StrategyTradingModes.Full))
{
    // We can place any orders
    RegisterOrder(CreateOrder(Sides.Buy, price, volume));
}
// For closing a position, the position reduction mode is sufficient
else if (IsFormedAndOnlineAndAllowTrading(StrategyTradingModes.ReducePositionOnly) && Position != 0)
{
    // We can only close the position
    ClosePosition();
}
// For cancelling active orders, the order cancellation mode is sufficient
else if (IsFormedAndOnlineAndAllowTrading(StrategyTradingModes.CancelOrdersOnly))
{
    // We can only cancel orders
    CancelActiveOrders();
}
```

Thus, this method allows you to implement a secure access control mechanism for trading functions, where more critical operations (such as opening new positions) require a higher level of permissions, and less critical ones (cancelling orders) are performed even in a limited trading mode.

It's good practice to use these methods before performing trading operations:

```

private void ProcessCandle(ICandleMessage candle)
{
    // Check if the strategy is formed and in online mode,
    // and if trading is allowed
    if (!IsFormedAndOnlineAndAllowTrading())
        return;

    // Trading logic
    // ...
}

```

Trading Operations Example

Below is an example demonstrating different ways of placing orders in a strategy and handling their execution:

```

protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    // Subscribe to candles
    var subscription = new Subscription(
        DataType.TimeFrame(TimeSpan.FromMinutes(5)),
        Security);

    // Create a rule for processing candles
    Connector
        .WhenCandlesFinished(subscription)
        .Do(ProcessCandle)
        .Apply(this);

    Connector.Subscribe(subscription);
}

private void ProcessCandle(ICandleMessage candle)
{
    // Check if the strategy is ready to trade
    if (!this.IsFormedAndOnlineAndAllowTrading())
        return;

    // Example trading logic based on closing price
    if (candle.ClosePrice > _previousClose * 1.01)
    {
        // Option 1: Using a high-level method
        var order = BuyLimit(candle.ClosePrice, Volume);

        // Create a rule for handling order execution
        order
            .WhenMatched(this)
            .Do(() => {
                // When the order is executed, set stop-loss and take-profit
                StartProtection(
                    takeProfit: new Unit(50, UnitTypes.Absolute),
                    stopLoss: new Unit(20, UnitTypes.Absolute)
                );
            })
            .Apply(this);
    }
    else if (candle.ClosePrice < _previousClose * 0.99)
    {
        // Option 2: Separate creation and registration
        var order = CreateOrder(Sides.Sell, candle.ClosePrice, Volume);
        RegisterOrder(order);

        // Alternative way of handling through the event
        OrderReceived += (o) => {
            if (o == order && o.State == OrderStates.Done)

```

```
{  
    // Actions after execution  
}  
};  
}  
  
_previousClose = candle.ClosePrice;  
}
```

See Also

- [Orders](#)
- [Order Rules](#)
- [Event Model](#)
- [Position Protection](#)

Position Protection

Introduction

This modification of the SMA strategy implements a mechanism for protecting open positions using a local protective controller. This approach allows for flexible risk management and automatic closing of positions when certain conditions are met.

Key Components of Position Protection

Protective Controller

The strategy uses two key objects for position protection:

```
// Declaration of protective controllers
private readonly ProtectiveController _protectiveController = new();
private IProtectivePositionController _posController;

// This code initializes the main protective controller and creates a placeholder for
// a specific position controller. ProtectiveController manages all positions,
// while IProtectivePositionController is responsible for a specific position.
```

- `_protectiveController`: The main controller managing protection for all positions.
- `_posController`: Controller for a specific position.

Protection Initialization

When opening a new position or modifying an existing one, the protective controller is initialized:

```
// Initialization of the protective controller for a new position
this.WhenOwnTradeReceived()
    .Do(t =>
    {
        // ... (other code)

        if (TakeValue.IsSet() || StopValue.IsSet())
        {
            _posController ??= _protectiveController.GetController(
                security.ToSecurityId(),
                portfolio.Name,
                new LocalProtectiveBehaviourFactory(security.PriceStep, security.Decimals),
                TakeValue, StopValue, true, default, default, true);
        }

        var info = _posController?.Update(t.Trade.Price, t.GetPosition());

        if (info is not null)
            ActiveProtection(info.Value);
    })
    .Apply(this);

// This code creates and initializes a protective controller for a new position
// upon receiving information about a new trade. It also updates the information
// about the position in the controller and activates protection if necessary.
```

This creates a controller for a specific position with given take-profit and stop-loss parameters.

Updating Position Information

```
var info = _posController?.Update(t.Trade.Price, t.GetPosition());  
  
if (info is not null)  
    ActiveProtection(info.Value);
```

This allows the controller to track the current state of the position and adjust protective orders as necessary.

Checking Activation Conditions for Protection

In the method processing new data (e.g., when receiving a new candle), the conditions for activating protective orders are checked:

```
// Checking protection activation conditions in the ProcessCandle method  
var info = _posController?.TryActivate(candle.ClosePrice, CurrentTime);  
  
if (info is not null)  
    ActiveProtection(info.Value);  
  
// This code checks if a protective order needs to be activated based on  
// the current price (in this case, the candle's closing price) and time.  
// If the conditions are met, the ActiveProtection method is called.
```

Here, the candle's closing price is used as the current price, but it can be any relevant price value (e.g., the price of the last trade or the current spread in the order book).

Activating a Protective Order

If the conditions for activating a protective order are met, the corresponding logic is triggered:

```
// Method for activating a protective order  
private void ActiveProtection((bool isTake, Sides side, decimal price, decimal volume,  
OrderCondition condition) info)  
{  
    // sending a protective (position-closing) order as a regular order  
    RegisterOrder(this.CreateOrder(info.side, info.price, info.volume));  
}  
  
// This method creates and registers an order to close the position  
// based on the information received from the protective controller.
```

This method creates and registers an order to close the position according to the parameters returned by the protective controller.

Comparison with Server-Side Stop Orders

Advantages of Server-Side Stop Orders

1. Stop orders (stop loss and take profit) are sent directly to the broker.
2. The broker independently monitors the achievement of stop conditions.
3. When a stop is triggered, the broker automatically places a market or limit order.

Advantages of the Local Approach

1. **Flexibility:** Ability to implement complex protection logic unavailable in standard server-side stops.
2. **Confidentiality:** Information about stop levels is not transmitted to the broker, which can be important

in some markets.

3. **Reaction Speed:** Potentially faster reaction to changing market conditions.
4. **Adaptability:** Ability to dynamically adjust protection levels based on market data or strategy logic.
5. **Independence from broker/exchange implementation:** The local approach works the same regardless of whether the broker or exchange supports all necessary types of protective orders.
6. **Testing on historical data:** Ability to fully test the strategy with position protection on historical data, which is impossible with server-side stops.

Disadvantages of the Local Approach

1. **Dependence on trading terminal functionality:** If the terminal is disconnected, the protection will not work.
2. **System load:** Requires constant calculations on the client side.
3. **Delays:** Possible delays in placing an order after protection conditions are triggered.

Disadvantages of Server-Side Stop Orders

1. **Dependence on broker/exchange implementation:** Not all brokers or exchanges support all types of protective orders, which may limit strategy functionality.
2. **Inability to fully test on historical data:** Server-side stops cannot be accurately modeled when testing on historical data, making it difficult to assess the real effectiveness of the strategy.
3. **Limited flexibility:** Usually only basic types of stop orders are available, limiting the possibilities for implementing complex protective mechanisms.

Conclusion

Using a local protective controller in the SMA strategy allows for effective risk management of open positions. This approach provides flexibility in setting protection parameters and quick reaction to changes in market situations, which is critical for successful trading.

Parameters

Strategy Parameters

For strategy configuration and optimization, StockSharp provides a special class [StrategyParam<T>](#). Strategy parameters allow you to modify trading algorithm settings without changing the code, which is particularly convenient when switching between testing and live trading modes. Additionally, these parameters are used during optimization to automatically iterate through values and find optimal strategy settings.

Unlike regular C# properties, parameters created with this class are automatically displayed in visual settings (for example, in Designer) and can be used for strategy optimization.

Creating Strategy Parameters

Parameters are created in the strategy constructor using the [Strategy.Param](#) method:

```
public class SmaStrategy : Strategy
{
    private readonly StrategyParam<int> _longSmaLength;

    public int LongSmaLength
    {
        get => _longSmaLength.Value;
        set => _longSmaLength.Value = value;
    }

    public SmaStrategy()
    {
        _longSmaLength = Param(nameof(LongSmaLength), 80)
            .SetGreaterThanZero()
            .SetDisplay("Long SMA length", string.Empty, "Base settings");
    }
}
```

In this example, a parameter `LongSmaLength` is created with an initial value of 80, a validator is set to ensure the value is greater than zero, and display settings are configured for the user interface.

Parameter Configuration Methods

The [StrategyParam<T>](#) class provides several methods for parameter configuration:

SetDisplay

The [StrategyParam<T>.SetDisplay](#) method sets the display name, description, and category of the parameter:

```
_longSmaLength = Param(nameof(LongSmaLength), 80)
    .SetDisplay("Long SMA length", "Period of the long moving average", "Base settings");
```

SetValidator

The [StrategyParam<T>.SetValidator](#) method sets a validator to check the parameter value. StockSharp provides a range of predefined validators that can be used for the most common tasks:

```
// Check that the number is greater than zero
_longSmaLength = Param(nameof(LongSmaLength), 80)
    .SetValidator(new IntGreaterThanZeroAttribute());
```



```
// Check that the number is not negative
_volume = Param(nameof(Volume), 1)
    .SetValidator(new DecimalNotNegativeAttribute());

// Check for value range
_percentage = Param(nameof(Percentage), 50)
    .SetValidator(new RangeAttribute(0, 100));

// Check for required value
_security = Param<Security>(nameof(Security))
    .SetValidator(new RequiredAttribute());
```

For convenience, [StrategyParam<T>](#) has built-in methods for the most common validators:

```
// Check that the number is greater than zero
_longSmaLength = Param(nameof(LongSmaLength), 80).SetGreaterThanZero();

// Check that the number is not negative
_volume = Param(nameof(Volume), 1).SetNotNegative();

// Check that the value is NULL or not negative
_interval = Param<TimeSpan?>(nameof(Interval)).SetNullOrNotNegative();

// Set value range
_percentage = Param(nameof(Percentage), 50).SetRange(0, 100);
```

If the built-in validators are not sufficient, you can create your own by inheriting from [ValidationAttribute](#):

```
public class EvenNumberAttribute : ValidationAttribute
{
    public EvenNumberAttribute()
        : base("Value must be an even number.")
    {
    }

    public override bool IsValid(object value)
    {
        if (value is int intValue)
            return intValue % 2 == 0;

        return false;
    }
}

// Using custom validator
_barCount = Param(nameof(BarCount), 10)
    .SetValidator(new EvenNumberAttribute());
```

SetHidden

The [StrategyParam<T>.SetHidden](#) method hides the parameter in the property editor:

```
_systemParam = Param(nameof(SystemParam), "value")
    .SetHidden(true);
```

SetBasic

The [StrategyParam<T>.SetBasic](#) method marks the parameter as basic, which affects its display in the user interface. Basic parameters are displayed in the simplified property editor mode:

```
_longSmaLength = Param(nameof(LongSmaLength), 80)
```

```
.SetBasic(true);
```

[image: strategy parameters basic advanced]

SetReadOnly

The [StrategyParam<T>.SetReadOnly](#) method makes the parameter read-only:

```
_calculatedParam = Param(nameof(CalculatedParam), 0)
    .SetReadOnly(true);
```

SetCanOptimize and SetOptimize

The methods [StrategyParam<T>.SetCanOptimize](#) and [StrategyParam<T>.SetOptimize](#) specify whether the parameter can be used for optimization, and set the range of values for optimization:

```
_longSmaLength = Param(nameof(LongSmaLength), 80)
    .SetCanOptimize(true)
    .SetOptimize(10, 200, 10);
```

In the example above, the parameter will be optimized in the range from 10 to 200 with a step of 10.

Using Parameters in Strategy

Strategy parameters are used like regular properties:

```
protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    _shortSma = new SimpleMovingAverage { Length = ShortSmaLength };
    _longSma = new SimpleMovingAverage { Length = LongSmaLength };

    // ...
}
```

Saving and Loading Parameters

Parameter values are automatically saved and loaded in the base [Strategy](#) class. When overriding the [Strategy.Save](#) and [Strategy.Load](#) methods, you must call the base class methods:

```
public override void Save(SettingsStorage settings)
{
    base.Save(settings);

    // Additional saving logic...
}

public override void Load(SettingsStorage settings)
{
    base.Load(settings);

    // Additional loading logic...
}
```

Example: Strategy with Multiple Parameters

Below is an example of a strategy with multiple parameters:

```

public class SmaStrategy : Strategy
{
    private readonly StrategyParam<DataType> _series;
    private readonly StrategyParam<int> _longSmaLength;
    private readonly StrategyParam<int> _shortSmaLength;

    public DataType Series
    {
        get => _series.Value;
        set => _series.Value = value;
    }

    public int LongSmaLength
    {
        get => _longSmaLength.Value;
        set => _longSmaLength.Value = value;
    }

    public int ShortSmaLength
    {
        get => _shortSmaLength.Value;
        set => _shortSmaLength.Value = value;
    }

    public SmaStrategy()
    {
        base.Name = "SMA strategy";

        Param("TypeId", GetType().GetTypeName(false)).SetHidden();
        _longSmaLength = Param(nameof(LongSmaLength), 80)
            .SetGreaterThanZero()
            .SetDisplay("Long SMA length", string.Empty, "Base settings")
            .SetCanOptimize(true)
            .SetOptimize(20, 200, 10);

        _shortSmaLength = Param(nameof(ShortSmaLength), 30)
            .SetGreaterThanZero()
            .SetDisplay("Short SMA length", string.Empty, "Base settings")
            .SetCanOptimize(true)
            .SetOptimize(5, 50, 5);

        _series = Param(nameof(Series), DataType.TimeFrame(TimeSpan.FromMinutes(15)))
            .SetDisplay("Series", string.Empty, "Base settings");
    }

    // ...
}

```

In this example, we created a strategy based on crossing two moving averages with three configurable parameters:

- Series - data type and timeframe
- LongSmaLength - period of the long moving average
- ShortSmaLength - period of the short moving average

For the two numeric parameters, we configured optimization capabilities with specified ranges.

See Also

[Saving and Loading Settings](#)

Logging

Logging in Strategy

In StockSharp, the [Strategy](#) class inherits from [BaseLogReceiver](#), which allows you to use built-in tools to log all actions and events occurring during the operation of a trading strategy.

Logging Levels

StockSharp supports the following logging levels (listed in order of increasing importance):

1. Verbose - the most detailed logging level for tracing
2. Debug - messages for debugging
3. Info - regular informational messages
4. Warning - warnings about potential problems
5. Error - error messages

Logging Methods in Strategy

The strategy provides the following methods for writing messages to the log:

LogVerbose

The [LogVerbose](#) method is designed to record detailed messages for tracing:

```
protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    LogVerbose("Strategy started with parameters: Long SMA={0}, Short SMA={1}", LongSmaLength,
ShortSmaLength);

    // ...
}
```

LogDebug

The [LogDebug](#) method is used for debug messages:

```
private void ProcessCandle(ICandleMessage candle)
{
    LogDebug("Processing candle: {0}, Open={1}, Close={2}, High={3}, Low={4}, Volume={5}",
candle.OpenTime, candle.OpenPrice, candle.ClosePrice, candle.HighPrice, candle.LowPrice,
candle.TotalVolume);

    // ...
}
```

LogInfo

The [LogInfo](#) method is used for regular informational messages:

```
private void CalculateSignal(decimal shortSma, decimal longSma)
{
    bool isShortGreaterThanLong = shortSma > longSma;
```

```

LogInfo("Signal: {0}, Short SMA={1}, Long SMA={2}",
    isShortGreaterThanLong ? "Buy" : "Sell", shortSma, longSma);

// ...
}

```

LogWarning

The [LogWarning](#) method is used to record warnings:

```

public void RegisterOrder(Order order)
{
    if (order.Volume <= 0)
    {
        LogWarning("Attempt to register an order with invalid volume: {0}", order.Volume);
        return;
    }

    // ...
}

```

LogError

The [LogError](#) method is used to record error messages:

```

try
{
    // Some actions
}
catch (Exception ex)
{
    LogError("Error while performing operation: {0}", ex.Message);
    Stop();
}

```

There is also an overload [LogError](#) that directly accepts an exception:

```

try
{
    // Some actions
}
catch (Exception ex)
{
    LogError(ex);
    Stop();
}

```

Configuring the Logging Level

The [Strategy](#) class contains a [LogLevel](#) property that determines which messages will be written to the log:

```

// Set the logging level for the strategy
strategy.LogLevel = LogLevels.Info;

```

With the selected logging level, only messages of that level and higher levels will be recorded. For example, if `LogLevels.Info` is set, `Verbose` and `Debug` messages will be ignored.

LogLevel Parameter

For convenient logging level configuration in the strategy constructor, you can add a parameter:

```
public class SmaStrategy : Strategy
{
    private readonly StrategyParam<LogLevels> _logLevel;

    public SmaStrategy()
    {
        _logLevel = Param(nameof(LogLevel), LogLevels.Info)
            .SetDisplay("Logging Level", "Level of log message detail", "Logging Settings");
    }

    public override LogLevels LogLevel
    {
        get => _logLevel.Value;
        set => _logLevel.Value = value;
    }

    // ...
}
```

Examples of Use in a Real Strategy

Logging Strategy Start and Stop

```
protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    LogInfo("Strategy {0} started at {1}. Instrument: {2}, Portfolio: {3}",
        Name, time, Security?.Code, Portfolio?.Name);

    // ...
}

protected override void OnStopped()
{
    LogInfo("Strategy {0} stopped. Position: {1}, P&L: {2}",
        Name, Position, PnL);

    base.OnStopped();
}
```

Logging Trades

```
protected override void OnOwnTradeReceived(MyTrade trade)
{
    LogInfo("{0} {1} {2} at price {3}. Volume: {4}",
        trade.Order.Direction == Sides.Buy ? "Bought" : "Sold",
        trade.Order.Security.Code,
        trade.Order.Type,
        trade.Trade.Price,
        trade.Trade.Volume);

    base.OnOwnTradeReceived(trade);
}
```

Logging Order Registration Errors

```
protected override void OnOrderRegisterFailed(OrderFail fail, bool calcRisk)
{
    LogError("Order registration error {0}: {1}",
```

```
fail.Order.TransactionId, fail.Error.Message);  
  
base.OnOrderRegisterFailed(fail, calcRisk);  
}
```

Viewing Logs

Messages written to the strategy log can be viewed:

1. In the [Designer](#) program on the "Logs" panel
2. In log files, if [FileLogListener](#) is configured
3. In the user interface through [LogControl](#), if [GuiLogListener](#) is used

See also

[Logging LogControl Component](#)

Platform Compatibility

Strategy Compatibility with StockSharp Platforms

When developing trading strategies in StockSharp, it's important to consider their compatibility with various platforms: [Designer](#), [Shell](#), [Runner](#), and [cloud backtesting](#). By following the recommendations below, you'll create a strategy that works correctly in all environments.

Strategy Constructor Parameters

Avoid Parameters in the Constructor

To ensure compatibility with StockSharp platforms, especially with cloud backtesting, **you should not add parameters to the strategy constructor**:

```
// Correct: constructor without parameters
public class SmaStrategy : Strategy
{
    public SmaStrategy()
    {
        // Parameters initialization
    }
}

// Incorrect: constructor with parameters
public class SmaStrategy : Strategy
{
    public SmaStrategy(int longLength, int shortLength) // Don't use this approach
    {
        // ...
    }
}
```

StockSharp platforms create strategy instances using a parameterless constructor. If your strategy requires a constructor with parameters, it won't be correctly initialized.

Using StrategyParam Instead of Regular Properties

Benefits of StrategyParam

Instead of creating regular C# properties and then overriding the Save and Load methods, use [StrategyParam<T>](#) for all customizable parameters:

```
// Correct: using StrategyParam
private readonly StrategyParam<int> _longSmaLength;

public int LongSmaLength
{
    get => _longSmaLength.Value;
    set => _longSmaLength.Value = value;
}

public SmaStrategy()
{
    _longSmaLength = Param(nameof(LongSmaLength), 80)
        .SetDisplay("Long SMA length", string.Empty, "Base settings");
}

// Incorrect: using regular properties
private int _longSmaLength = 80; // Don't use this approach
```



```

public int LongSmaLength
{
    get => _longSmaLength;
    set => _longSmaLength = value;
}

```

Parameters created through [StrategyParam<T>](#) will automatically:

- Be displayed in the platform user interfaces
- Be saved and loaded without overriding the Save and Load methods
- Be used in optimization
- Be correctly serialized when sent to cloud backtesting

Working with the User Interface

Use Abstractions Instead of Direct UI Access

Instead of directly accessing user interface elements, use the abstractions provided by StockSharp:

```

// Correct approach: using IChart
protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    // Get the chart provided by the runtime environment
    _chart = GetChart();

    if (_chart != null)
    {
        // Chart is available (e.g., in Designer or Shell)
        InitChart();
    }
    else
    {
        // Chart is unavailable (e.g., in Runner or cloud backtesting)
        // Strategy continues to work without visualization
    }
}

private void InitChart()
{
    // Configure chart through the abstract interface
    _chart.ClearAreas();
    var area = _chart.AddArea();
    _chartCandleElement = area.AddCandles();
    // ...
}

```

The [Strategy.GetChart\(\)](#) method returns an [IChart](#) interface if a chart is available in the current runtime environment. If the strategy is running in console [Runner](#) or cloud backtesting, where there's no graphical interface, the method will return null.

The [IChart](#) interface provides methods for working with charts:

- [AddArea](#) - to add an area to the chart
- [RemoveArea](#) - to remove an area
- [AddElement](#) - to add an element to the chart
- [RemoveElement](#) - to remove an element

- [Reset](#) - to reset element values

Check Chart Availability

Always check chart availability before using it:

```
private void DrawCandlesAndIndicators(ICandleMessage candle, IIndicatorValue longSma,
IIndicatorValue shortSma)
{
    if (_chart == null) return; // Important check

    var data = _chart.CreateData();
    data.Group(candle.OpenTime)
        .Add(_chartCandleElement, candle)
        .Add(_longSmaIndicatorElement, longSma)
        .Add(_shortSmaIndicatorElement, shortSma);
    _chart.Draw(data);
}
```

Threads and Synchronization

Avoid Creating Additional Threads

In StockSharp, **you don't need to create additional threads** for data processing. All events (market data, transactions) come in a single thread:

```
// Correct: using standard event handlers
private void ProcessCandle(ICandleMessage candle)
{
    // Process candle in the main thread
    var longSmaIsFormedPrev = _longSma.IsFormed;
    var ls = _longSma.Process(candle);
    var ss = _shortSma.Process(candle);

    // ...
}

// Incorrect: creating additional threads
private void ProcessCandle(ICandleMessage candle)
{
    // DON'T do this
    Task.Run(() => {
        var longSmaIsFormedPrev = _longSma.IsFormed;
        // ...
    });
}
```

Avoid Synchronization Objects

Since all events are processed in a single thread, **there's no need to use synchronization objects**:

```
// Correct: regular processing without synchronization
private void ProcessCandle(ICandleMessage candle)
{
    var ls = _longSma.Process(candle);
    var ss = _shortSma.Process(candle);
    // ...
}

// Incorrect: unnecessary synchronization
private readonly object _syncLock = new object(); // Not needed

private void ProcessCandle(ICandleMessage candle)
```

```

{
    lock (_syncLock) // Not needed
    {
        var ls = _longSma.Process(candle);
        // ...
    }
}

```

External Resources

Use StockSharp Infrastructure

Instead of directly accessing external resources (files, databases, network), use the capabilities provided by StockSharp platforms:

```

// Correct: using built-in mechanisms for data saving
protected override void OnStopped()
{
    // Data is automatically saved through strategy parameters
    base.OnStopped();
}

// Incorrect: direct access to external resources
protected override void OnStopped()
{
    // DON'T do this
    File.WriteAllText("results.txt", $"PnL: {PnL}");

    // or this
    using (var connection = new SqlConnection("..."))
    {
        // ...
    }

    base.OnStopped();
}

```

Data Storage

To save strategy results, use:

- [Strategy parameters](#) for settings and configuration
- Built-in storage mechanisms in [Designer](#) and [Shell](#)
- [Statistics](#) for collecting trading metrics

Save and Load Methods

The [Strategy.Save](#) and [Strategy.Load](#) methods are specifically designed for saving additional strategy data that are not settings or parameters. This is the ideal place to save data needed to restore the strategy state:

```

public override void Save(SettingsStorage settings)
{
    base.Save(settings); // First save strategy parameters

    // Then save custom data
    settings.SetValue("CustomState", _customState);
    settings.SetValue("LastSignalTime", _lastSignalTime);
}

public override void Load(SettingsStorage settings)
{

```

```

base.Load(settings); // First load strategy parameters

// Then load custom data
if (settings.Contains("CustomState"))
    _customState = settings.GetValue<string>("CustomState");

if (settings.Contains("LastSignalTime"))
    _lastSignalTime = settings.GetValue<DateTimeOffset>("LastSignalTime");
}

```

However, the main configurable parameters should still be implemented through [StrategyParam<T>](#), as described above, as this will ensure their automatic display in the user interface.

Market Data Subscription

Use Rules Instead of Direct Subscription

For market data processing, it's recommended to use the [Event Model](#) and rules:

```

protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    _shortSma = new SimpleMovingAverage { Length = ShortSmaLength };
    _longSma = new SimpleMovingAverage { Length = LongSmaLength };

    Indicators.Add(_shortSma);
    Indicators.Add(_longSma);

    var subscription = new Subscription(Series, Security);

    // Correct: using rules for data processing
    Connector
        .WhenCandlesFinished(subscription)
        .Do(ProcessCandle)
        .Apply(this);

    Connector.Subscribe(subscription);
}

```

Rules have several important advantages over regular event handlers:

1. **Automatic unsubscription** - rules automatically unsubscribe from events when the strategy stops or when they're no longer needed. You don't need to manually manage subscriptions.
2. **High-level API** - rules provide a more understandable and convenient interface than standard event handlers. For example, `WhenCandlesFinished` is much clearer than subscribing to the `CandleReceived` event with subsequent checking of the candle state.
3. **Condition combining** - rules can be combined using operators like `And`, `Or`, and others, creating complex activation conditions:

```

// Example of combining rules
Security
    .WhenNewTrade()
    .And(Portfolio.WhenMoneyChanged())
    .Do(() => {
        // Code that executes only when there's a new trade
        // AND the portfolio balance changes
    })
    .Apply(this);

```

4. **Lifecycle management** - rules can be made one-time (`Once()`), have cancellation conditions set

(Until()), add delayed actions, etc.

Compatible Strategy Example

Below is an example of a strategy that follows all recommendations and will work correctly on all StockSharp platforms:

```
public class SmaStrategy : Strategy
{
    private readonly StrategyParam<DataType> _series;
    private readonly StrategyParam<int> _longSmaLength;
    private readonly StrategyParam<int> _shortSmaLength;

    public DataType Series
    {
        get => _series.Value;
        set => _series.Value = value;
    }

    public int LongSmaLength
    {
        get => _longSmaLength.Value;
        set => _longSmaLength.Value = value;
    }

    public int ShortSmaLength
    {
        get => _shortSmaLength.Value;
        set => _shortSmaLength.Value = value;
    }

    private SimpleMovingAverage _longSma;
    private SimpleMovingAverage _shortSma;
    private IChart _chart;
    private IChartCandleElement _chartCandleElement;
    private IChartIndicatorElement _longSmaIndicatorElement;
    private IChartIndicatorElement _shortSmaIndicatorElement;

    public SmaStrategy()
    {
        _longSmaLength = Param(nameof(LongSmaLength), 80)
            .SetDisplay("Long SMA length", string.Empty, "Base settings")
            .SetCanOptimize(true);

        _shortSmaLength = Param(nameof(ShortSmaLength), 30)
            .SetDisplay("Short SMA length", string.Empty, "Base settings")
            .SetCanOptimize(true);

        _series = Param(nameof(Series), DataType.TimeFrame(TimeSpan.FromMinutes(15)))
            .SetDisplay("Series", string.Empty, "Base settings");
    }

    protected override void OnStarted2(DateTime time)
    {
        base.OnStarted2(time);

        _longSma = new SimpleMovingAverage { Length = LongSmaLength };
        _shortSma = new SimpleMovingAverage { Length = ShortSmaLength };

        Indicators.Add(_shortSma);
        Indicators.Add(_longSma);

        // Initialize chart if available
        _chart = GetChart();
        if (_chart != null)
            InitChart();

        var subscription = new Subscription(Series, Security);
```

```

Connector
    .WhenCandlesFinished(subscription)
    .Do(ProcessCandle)
    .Apply(this);

Connector.Subscribe(subscription);
}

private void InitChart()
{
    _chart.ClearAreas();
    var area = _chart.AddArea();

    _chartCandleElement = area.AddCandles();

    _longSmaIndicatorElement = area.AddIndicator(_longSma);
    _longSmaIndicatorElement.Color = System.Drawing.Color.Brown;
    _longSmaIndicatorElement.DrawStyle = DrawStyles.Line;

    _shortSmaIndicatorElement = area.AddIndicator(_shortSma);
    _shortSmaIndicatorElement.Color = System.Drawing.Color.Blue;
    _shortSmaIndicatorElement.DrawStyle = DrawStyles.Line;
}

private void ProcessCandle(ICandleMessage candle)
{
    var ls = _longSma.Process(candle);
    var ss = _shortSma.Process(candle);

    // Draw on chart if available
    if (_chart != null)
    {
        var data = _chart.CreateData();
        data.Group(candle.OpenTime)
            .Add(_chartCandleElement, candle)
            .Add(_longSmaIndicatorElement, ls)
            .Add(_shortSmaIndicatorElement, ss);
        _chart.Draw(data);
    }

    if (!_longSma.IsFormed)
        return;

    var isShortLessCurrent = _shortSma.GetCurrentValue() < _longSma.GetCurrentValue();
    var isShortLessPrev = _shortSma.GetValue(1) < _longSma.GetValue(1);

    if (isShortLessCurrent == isShortLessPrev)
        return;

    // Trading logic
    var volume = Volume + Math.Abs(Position);

    if (isShortLessCurrent)
        SellMarket(volume);
    else
        BuyMarket(volume);
}
}

```

See also

- [Strategy Parameters](#)
- [Event Model](#)
- [Strategy Logging](#)

High-Level APIs

High-Level API in Strategies

StockSharp provides a set of high-level APIs to simplify working with common tasks in trading strategies. These interfaces allow writing cleaner code that focuses on trading logic rather than technical details.

Simplified Subscription Management

High-level methods for working with subscriptions hide the complexities of managing subscription lifecycle and data processing.

SubscribeCandles Method

Instead of manually creating a subscription and setting up event handlers, you can use the [SubscribeCandles](#) method:

```
// Create and configure a candle subscription in a single line
var subscription = SubscribeCandles(CandleType);
```

This method returns an object of type [ISubscriptionHandler<ICandleMessage>](#), which provides a convenient interface for further subscription configuration.

Automatic Binding of Indicators with Subscription

The high-level API makes it easy to bind indicators to a data subscription:

```
var longSma = new SMA { Length = Long };
var shortSma = new SMA { Length = Short };

subscription
    // Bind indicators to candle subscription
    .Bind(longSma, shortSma, OnProcess)
    // Start processing
    .Start();
```

Automatic Addition of Indicators to Strategy.Indicators Collection

It's important to note that when using the [Bind](#) method to link indicators with a subscription, you **do not need** to additionally add these indicators to the [Strategy.Indicators](#) collection, as is typically done in the traditional approach (described in the [indicators documentation](#)). The system automatically:

1. Adds indicators to the [Indicators](#) collection
2. Tracks the formation state of indicators
3. Updates the strategy's [IsFormed](#) state

This significantly simplifies code and reduces the likelihood of errors.

If it is necessary to receive indicator values even when some of them have no data yet ([IIndicatorValue.IsEmpty](#) is true), use the [BindWithEmpty](#) method. In this case the handler arguments must be of type [decimal?](#). You can also use [BindEx](#) to inspect the raw [IIndicatorValue](#) objects directly.

Using BindEx to Work with Raw Indicator Values

If an indicator returns non-standard values (not just numbers), you can use the [BindEx](#) method, which provides access to the original [IIndicatorValue](#) object:

```

subscription
    .BindEx(indicator, OnProcessWithRawValue)
    .Start();

// Handler receives the original IIndicatorValue
private void OnProcessWithRawValue(ICandleMessage candle, IIndicatorValue value)
{
    // Access to IIndicatorValue properties
    if (value.IsFinal)
    {
        // For indicators returning boolean values
        var boolValue = value.GetValue<bool>();

        // Or other data types specific to a particular indicator
        // ...
    }
}

```

The [BindEx](#) method is particularly useful in the following cases:

- Working with indicators that return boolean values (e.g., [Fractals](#))
- Accessing additional properties of the indicator value type (e.g., the [IsFinal](#) flag)
- Working with indicators that return structured data

Working with Complex Indicators (IComplexIndicator)

For complex indicators that contain multiple internal indicators (e.g., [BollingerBands](#), [MACD](#)), the API provides special overloads of the Bind and BindEx methods:

```

// Create a complex indicator
var bollinger = new BollingerBands
{
    Length = 20,
    Deviation = 2
};

// Bind the complex indicator to a subscription
subscription
    .BindEx(bollinger, OnProcessBollinger)
    .Start();

// Handler receives the BollingerBandsValue instance
private void OnProcessBollinger(ICandleMessage candle, IIndicatorValue value)
{
    var typed = (BollingerBandsValue)value;

    // Use Bollinger band values
    if (candle.ClosePrice >= typed.UpBand && Position >= 0)
        SellMarket(Volume + Math.Abs(Position));
    else if (candle.ClosePrice <= typed.LowBand && Position <= 0)
        BuyMarket(Volume + Math.Abs(Position));
}

```

For more flexible work, you can use [BindEx](#) with direct access to the complex indicator value:

```

subscription.BindEx(bollinger, (candle, indicatorValue) =>
{
    var typed = (BollingerBandsValue)indicatorValue;

    if (candle.ClosePrice >= typed.UpBand && Position >= 0)
        SellMarket(Volume + Math.Abs(Position));
    else if (candle.ClosePrice <= typed.LowBand && Position <= 0)
        BuyMarket(Volume + Math.Abs(Position));
});

```


The [BindEx](#) method for complex indicators automatically:

1. Processes input data through the complex indicator
2. Passes the resulting `IIndicatorValue` to the specified handler

Cast the value to the indicator's dedicated **value type** to work with its individual fields.

The Bind Method establishes a connection between subscription data and indicators. When a new candle is received:

1. The candle is automatically sent for processing to the indicators
2. Processing results are passed to the specified handler (in the example, the `OnProcess` method)
3. All synchronization and state management code is hidden from the developer

The handler receives ready-to-use values as simple decimal types. The method is called only when all bound indicators return data:

```
private void OnProcess(ICandleMessage candle, decimal longValue, decimal shortValue)
{
    // Work directly with ready-made indicator values
    var isShortLessThenLong = shortValue < longValue;

    // Trading logic uses clean numeric values
    // without the need to extract them from IIndicatorValue
    // ...
}
```

This significantly simplifies code and makes it more readable, as the developer doesn't need to:

- Manually handle the event of receiving a candle
- Pass data to indicators manually
- Extract values from indicator results

Simplified Chart Management

Automatic Visualization

The high-level API provides simple methods for binding subscriptions and indicators to chart elements:

```
var area = CreateChartArea();

// area can be null when running without GUI
if (area != null)
{
    // Automatic binding of candles to chart area
    DrawCandles(area, subscription);

    // Drawing indicators with color customization
    DrawIndicator(area, shortSma, System.Drawing.Color.Coral);
    DrawIndicator(area, longSma);

    // Drawing own trades
    DrawOwnTrades(area);

    // Drawing orders
    DrawOrders(area);
}
```

DrawCandles Method

The [DrawCandles](#) method automatically links a candle subscription to a chart candle display element:

```
// Create a chart element for displaying candles
IChartCandleElement candles = DrawCandles(area, subscription);

// Additional element parameters can be configured
candles.DrawOpenClose = true; // Display open/close lines
candles.DrawHigh = true;     // Display highs
candles.DrawLow = true;      // Display lows
```

The method returns a [IChartCandleElement](#) chart element that can be further customized.

DrawIndicator Method

The [DrawIndicator](#) method creates and configures a chart element for displaying indicator values:

```
// Simple addition of an indicator to the chart with default color
IChartIndicatorElement smaElem = DrawIndicator(area, sma);

// Adding an indicator with a specified primary color
IChartIndicatorElement rsiFast = DrawIndicator(area, rsi, System.Drawing.Color.Red);

// Adding an indicator with specified primary and secondary colors
IChartIndicatorElement bollingerElem = DrawIndicator(
    area,
    bollinger,
    System.Drawing.Color.Blue,    // Primary color
    System.Drawing.Color.Gray    // Secondary color (for the second line)
);

// Additional element configuration
smaElem.DrawStyle = DrawStyles.Line;           // Drawing style: line
rsiFast.DrawStyle = DrawStyles.Dot;            // Drawing style: dots
bollingerElem.DrawStyle = DrawStyles.Dashdot;  // Drawing style: dash-dot
```

The method returns a [IChartIndicatorElement](#) chart element that can be customized. For indicators with multiple values (e.g., [BollingerBands](#)), the primary color is applied to the first value, and the secondary color to the second.

DrawOwnTrades Method

The [DrawOwnTrades](#) method creates an element for displaying strategy's own trades on the chart:

```
// Create an element for displaying trades
IChartTradeElement trades = DrawOwnTrades(area);

// Element configuration
trades.BuyColor = System.Drawing.Color.Green; // Color for buy trades
trades.SellColor = System.Drawing.Color.Red;  // Color for sell trades
trades.FullTitle = "My Strategy Trades";      // Element title
```

This method automatically sets up the display of all trades executed by the strategy. Trades are displayed on the chart as markers at the points where they were executed, taking into account the trade side (buy/sell).

DrawOrders Method

The [DrawOrders](#) method creates an element for displaying orders on the chart:

```
// Create an element for displaying orders
IChartOrderElement orders = DrawOrders(area);

// Element configuration
```

```
orders.BuyPendingColor = System.Drawing.Color.DarkGreen; // Color for active buy orders
orders.SellPendingColor = System.Drawing.Color.DarkRed; // Color for active sell orders
orders.BuyColor = System.Drawing.Color.Green; // Color for executed buy orders
orders.SellColor = System.Drawing.Color.Red; // Color for executed sell orders
orders.CancelColor = System.Drawing.Color.Gray; // Color for canceled orders
```

This method automatically sets up the display of all orders placed by the strategy. Orders are displayed as markers at their price levels with different color coding for different order states.

CreateChartArea Method

The [CreateChartArea](#) method creates a new area on the strategy chart:

```
// Create the first area for candles and indicators
var mainArea = CreateChartArea();
DrawCandles(mainArea, subscription);
DrawIndicator(mainArea, sma);

// Create a second area for separate indicators (e.g., RSI)
var secondArea = CreateChartArea();
DrawIndicator(secondArea, rsi);
```

Dividing the chart into areas allows for a more visual display of different data types. For example, indicators with a value range different from price (RSI, stochastic, etc.) are better displayed in separate areas.

Advantages of high-level visualization methods:

- No need to manually create ChartDrawData objects
- No need to manage data grouping by time
- No need to call chart.Draw() to update the chart
- Automatic data synchronization between subscriptions and chart elements
- Simplified management of graphical element appearance

The system automatically updates the chart when new data is received, allowing the developer to avoid focusing on technical visualization details.

Position Protection

StartProtection Method

To protect open positions, StockSharp provides the high-level [StartProtection](#) method:

```
// Start position protection with Take Profit and Stop Loss levels
StartProtection(TakeValue, StopValue);
```

This method automatically sets up protection for all open positions:

- Tracks price changes
- Automatically creates orders to close positions when Take Profit or Stop Loss levels are reached
- Supports various types of measurement units (absolute values, percentages, points)
- Can use trailing stop for adaptive position protection

Example with additional parameters:

```
// Start protection with trailing stop and market orders
```

```

StartProtection(
    takeProfit: new Unit(50, UnitTypes.Absolute), // Take Profit
    stopLoss: new Unit(2, UnitTypes.Percent),     // Stop Loss in percentage
    isStopTrailing: true,                         // Enable trailing stop
    useMarketOrders: true                        // Use market orders
);

```

Advantages of High-Level API

The high-level API in StockSharp strategies provides the following advantages:

1. **Code Volume Reduction** - performing common tasks requires fewer lines of code
2. **Separation of Responsibilities** - trading logic is separated from technical details of data processing and visualization
3. **Improved Readability** - code becomes more understandable and expressive, focused on business logic
4. **Reduced Error Probability** - many typical errors are eliminated through automation of routine tasks
5. **Working with Clean Data Types** - instead of working with complex objects, you can operate with simple data types (e.g., decimal)

Example Strategy Using High-Level API

Below is a complete example of a strategy demonstrating the use of the high-level API:

```

public class SmaStrategy : Strategy
{
    private bool? _isShortLessThenLong;

    public SmaStrategy()
    {
        _candleType = Param(nameof(CandleType), DataType.TimeFrame(TimeSpan.FromMinutes(1)));
        _long = Param(nameof(Long), 80);
        _short = Param(nameof(Short), 30);
        _takeValue = Param(nameof(TakeValue), new Unit(50, UnitTypes.Absolute));
        _stopValue = Param(nameof(StopValue), new Unit(2, UnitTypes.Percent));
    }

    private readonly StrategyParam<DataType> _candleType;
    public DataType CandleType
    {
        get => _candleType.Value;
        set => _candleType.Value = value;
    }

    private readonly StrategyParam<int> _long;
    public int Long
    {
        get => _long.Value;
        set => _long.Value = value;
    }

    private readonly StrategyParam<int> _short;
    public int Short
    {
        get => _short.Value;
        set => _short.Value = value;
    }

    private readonly StrategyParam<Unit> _takeValue;
    public Unit TakeValue
    {
        get => _takeValue.Value;
    }
}

```

```

    set => _takeValue.Value = value;
}

private readonly StrategyParam<Unit> _stopValue;
public Unit StopValue
{
    get => _stopValue.Value;
    set => _stopValue.Value = value;
}

protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    // Create indicators
    var longSma = new SMA { Length = Long };
    var shortSma = new SMA { Length = Short };

    // Create a candle subscription and bind to indicators
    var subscription = SubscribeCandles(CandleType);
    subscription
        .Bind(longSma, shortSma, OnProcess)
        .Start();

    // Configure visualization
    var area = CreateChartArea();
    if (area != null)
    {
        DrawCandles(area, subscription);
        DrawIndicator(area, shortSma, System.Drawing.Color.Coral);
        DrawIndicator(area, longSma);
        DrawOwnTrades(area);
    }

    // Start position protection
    StartProtection(TakeValue, StopValue);
}

private void OnProcess(ICandleMessage candle, decimal longValue, decimal shortValue)
{
    // Process only finished candles
    if (candle.State != CandleStates.Finished)
        return;

    // Trading logic based on indicator crossover
    var isShortLessThenLong = shortValue < longValue;

    if (_isShortLessThenLong == null)
    {
        _isShortLessThenLong = isShortLessThenLong;
    }
    else if (_isShortLessThenLong != isShortLessThenLong)
    {
        // Crossover occurred
        var direction = isShortLessThenLong ? Sides.Sell : Sides.Buy;
        var volume = Position == 0 ? Volume : Position.Abs().Min(Volume) * 2;
        var priceStep = GetSecurity().PriceStep ?? 1;
        var price = candle.ClosePrice + (direction == Sides.Buy ? priceStep : -priceStep);

        // Place an order
        if (direction == Sides.Buy)
            BuyLimit(price, volume);
        else
            SellLimit(price, volume);

        // Save current indicator position
        _isShortLessThenLong = isShortLessThenLong;
    }
}

```

```
}
```

Conclusion

The high-level API in StockSharp significantly simplifies the development of trading strategies, allowing developers to focus on trading logic rather than technical details. It is especially useful for typical use cases where fine-tuning of data processing or visualization is not required.

Combined with the strategy parameter system, event model, and position protection mechanisms, the high-level API makes StockSharp a powerful and convenient tool for algorithmic trading, suitable for both beginners and experienced developers.

Working with Charts

Working with Charts in Strategies

In StockSharp, the [Strategy](#) class provides a convenient interface for visualizing trading activity on a chart. In this article, we'll look at how to access a chart from a strategy, create areas (ChartArea), add various elements (candles, indicators, trades), and render data.

Accessing the Chart

GetChart Method

To access the chart from a strategy, use the [Strategy.GetChart\(\)](#) method:

```
protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    // Obtaining the chart
    _chart = GetChart();

    // Checking chart availability
    if (_chart != null)
    {
        // Initializing the chart
        InitializeChart();
    }
    else
    {
        // Chart is unavailable, for example, when running in console mode
        LogInfo("Chart is unavailable. Visualization disabled.");
    }
}
```

The [GetChart\(\)](#) method returns an [IChart](#) interface that provides access to chart functions. It's important to check the result for null, as the chart may be unavailable, for example, when running a strategy in console mode or cloud testing.

SetChart Method

In some cases, the chart may be set from outside. For this, use the [Strategy.SetChart](#) method:

```
// Setting the chart from an external source
public void ConfigureVisualization(IChart chart)
{
    SetChart(chart);

    if (chart != null)
    {
        InitializeChart();
    }
}
```

Creating Chart Areas

After obtaining access to the chart, you can create one or more areas to display various data. Use the [CreateChartArea](#) method:

```
private void InitializeChart()
{
    // Creating the main area for candles and indicators
    _mainArea = CreateChartArea();

    // Creating an additional area for volume
    _volumeArea = CreateChartArea();

    // Configuring areas and adding elements
    ConfigureChartElements();
}
```

You can also use the [IChart.AddArea](#) method directly:

```
private void InitializeChart()
{
    // Clear existing areas if necessary
    foreach (var area in _chart.Areas.ToArray())
        _chart.RemoveArea(area);

    // Create the main area for candles and indicators
    _mainArea = _chart.AddArea();

    // Create an additional area for volume
    _volumeArea = _chart.AddArea();

    // Configure areas and add elements
    ConfigureChartElements();
}
```

Adding Elements to the Chart

After creating chart areas, you can add various elements to display data. StockSharp supports different types of elements such as candles, indicators, trades, and orders.

Adding Candles

To display candles, use the [AddCandles](#) method of the chart area:

```
private void ConfigureChartElements()
{
    // Adding a candle element to the main area
    _candleElement = _mainArea.AddCandles();

    // Configuring candle display
    _candleElement.DrawStyle = ChartCandleDrawStyles.CandleStick; // Japanese candles
    _candleElement.AntiAliasing = true; // Smoothing
    _candleElement.UpFillColor = Color.Green; // Rising candle body color
    _candleElement.DownFillColor = Color.Red; // Falling candle body color
    _candleElement.UpBorderColor = Color.DarkGreen; // Rising candle border color
    _candleElement.DownBorderColor = Color.DarkRed; // Falling candle border color
    _candleElement.StrokeThickness = 1; // Line thickness
    _candleElement.ShowAxisMarker = true; // Show Y-axis marker
}
```

The [IChartCandleElement](#) interface provides many properties for configuring candle display:

- **DrawStyle** - candle display style:
 - **CandleStick** - Japanese candles
 - **Ohlc** - bars
 - **LineOpen/LineHigh/LineLow/LineClose** - lines for respective prices
 - **BoxVolume** - volume boxes

- **ClusterProfile** - cluster profile
- **Area** - area
- **PnF** - point and figure chart
- **Color settings:**
 - **UpFillColor/DownFillColor** - rising/falling candle body color
 - **UpBorderColor/DownBorderColor** - rising/falling candle border color
 - **LineColor** - line color for line-type charts
 - **AreaColor** - area color for Area type
- **Other settings:**
 - **StrokeThickness** - line thickness
 - **AntiAliasing** - smoothing
 - **ShowAxisMarker** - show Y-axis marker

Adding Indicators

To display indicators, use the [DrawIndicator](#) method:

```
// Creating indicators
_sma = new SimpleMovingAverage { Length = SmaLength };
_bollinger = new BollingerBands
{
    Length = BollingerLength,
    Deviation = BollingerDeviation
};

// Adding indicators to the strategy collection
Indicators.Add(_sma);
Indicators.Add(_bollinger);

// Visualizing indicators
_smaElement = DrawIndicator(_mainArea, _sma, Color.Blue);
_bollingerUpperElement = DrawIndicator(_mainArea, _bollinger, Color.Purple);
_bollingerLowerElement = DrawIndicator(_mainArea, _bollinger, Color.Purple);
_bollingerMiddleElement = DrawIndicator(_mainArea, _bollinger, Color.Gray);
```

The [DrawIndicator](#) method automatically creates an indicator element and adds it to the specified chart area. You can specify a color and additional color for display.

You can also add an indicator element directly through the chart area's [AddIndicator](#) method:

```
// Adding SMA directly through the chart area
var smaElement = _mainArea.AddIndicator(_sma);
smaElement.Color = Color.Blue;
smaElement.StrokeThickness = 2;
smaElement.DrawStyle = DrawStyles.Line;
smaElement.AntiAliasing = true;
smaElement.ShowAxisMarker = true;
smaElement.AutoAssignYAxis = true; // Automatically assign Y-axis
```

The [IChartIndicatorElement](#) interface provides the following properties for configuration:

- **Color** - main indicator color
- **AdditionalColor** - additional color (for indicators with two lines)
- **StrokeThickness** - line thickness
- **AntiAliasing** - smoothing

- **DrawStyle** - drawing style (line, points, histogram, etc.)
- **ShowAxisMarker** - show Y-axis marker
- **AutoAssignYAxis** - automatically assign Y-axis

Adding Trades

To display trades, use the [DrawOwnTrades](#) method:

```
// Adding an element to display trades
_tradesElement = DrawOwnTrades(_mainArea);

// Configuring trade display
_tradesElement.BuyBrush = Color.Green; // Buy color
_tradesElement.SellBrush = Color.Red;  // Sell color
_tradesElement.PointSize = 10;        // Point size
```

Adding Orders

To display orders, use the [DrawOrders](#) method:

```
// Adding an element to display orders
_ordersElement = DrawOrders(_mainArea);

// Configuring order display
_ordersElement.ActiveBrush = Color.Blue;    // Active orders color
_ordersElement.CanceledBrush = Color.Gray;  // Canceled orders color
_ordersElement.DoneBrush = Color.Green;     // Completed orders color
_ordersElement.ErrorColor = Color.Red;      // Error color
_ordersElement.PointSize = 8;               // Point size
```

The [IChartOrderElement](#) interface provides the following properties for configuration:

- **ActiveBrush** - color of active orders
- **CanceledBrush** - color of canceled orders
- **DoneBrush** - color of completed orders
- **ErrorColor** - error color
- **ErrorStrokeColor** - error border color
- **Filter** - order display filter

Drawing Data on the Chart

After configuring all chart elements, you can proceed to draw data. Different methods are used depending on the type of data.

Drawing Candles and Indicators

The most efficient way to draw data is to use the [IChart.Draw](#) method with an [IChartDrawData](#) object:

```
private void ProcessCandle(ICandleMessage candle)
{
    // Processing candle in indicators
    var smaValue = _sma.Process(candle);
    var bollingerValue = _bollinger.Process(candle);

    // If chart is unavailable, skip drawing
    if (_chart == null)
```

```

    return;

    // Create data for drawing
    var drawData = _chart.CreateData();

    // Group data by candle time
    var group = drawData.Group(candle.OpenTime);

    // Add candle
    group.Add(_candleElement,
        candle.DataType,
        candle.SecurityId,
        candle.OpenPrice,
        candle.HighPrice,
        candle.LowPrice,
        candle.ClosePrice,
        candle.PriceLevels,
        candle.State);

    // Add indicator values
    group.Add(_smaElement, smaValue);

    if (bollingerValue != null)
    {
        group.Add(_bollingerUpperElement, bollingerValue);
        group.Add(_bollingerMiddleElement, bollingerValue);
        group.Add(_bollingerLowerElement, bollingerValue);
    }

    // Draw data on the chart
    _chart.Draw(drawData);
}

```

The [IChart.CreateData](#) method creates an [IChartDrawData](#) object used to group and add data for different chart elements. Data grouping is done by timestamp using the [Group](#) method.

For adding data of different types, various overloads of the [Add](#) method of the [IChartDrawDataItem](#) object are used.

Drawing Trades and Orders

For drawing trades and orders, an automatic mechanism is usually used that is triggered when new trades are received or orders change. However, if manual drawing is required, you can use the following code:

```

// Drawing a trade
var tradeDrawData = _chart.CreateData();
var tradeGroup = tradeDrawData.Group(trade.Time);
tradeGroup.Add(_tradesElement, trade.Id, trade.StringId, trade.Side, trade.Price, trade.Volume);
_chart.Draw(tradeDrawData);

// Drawing an order
var orderDrawData = _chart.CreateData();
var orderGroup = orderDrawData.Group(order.Time);
orderGroup.Add(_ordersElement, order.Id, order.StringId, order.Side, order.Price, order.Volume);
_chart.Draw(orderDrawData);

```

Full Example of Chart Rendering in a Strategy

Below is a complete example of a strategy with chart setup and rendering:

```

public class SmaStrategy : Strategy
{
    private readonly StrategyParam<int> _smaLength;
    private readonly StrategyParam<int> _bollingerLength;
}

```

```

private readonly StrategyParam<decimal> _bollingerDeviation;

private SimpleMovingAverage _sma;
private BollingerBands _bollinger;

private IChart _chart;
private IChartArea _mainArea;
private IChartArea _volumeArea;

private IChartCandleElement _candleElement;
private IChartIndicatorElement _smaElement;
private IChartIndicatorElement _bollingerUpperElement;
private IChartIndicatorElement _bollingerMiddleElement;
private IChartIndicatorElement _bollingerLowerElement;
private IChartOrderElement _ordersElement;
private IChartTradeElement _tradesElement;

public SmaStrategy()
{
    _smaLength = Param(nameof(SmaLength), 20);
    _bollingerLength = Param(nameof(BollingerLength), 20);
    _bollingerDeviation = Param(nameof(BollingerDeviation), 2m);
}

public int SmaLength
{
    get => _smaLength.Value;
    set => _smaLength.Value = value;
}

public int BollingerLength
{
    get => _bollingerLength.Value;
    set => _bollingerLength.Value = value;
}

public decimal BollingerDeviation
{
    get => _bollingerDeviation.Value;
    set => _bollingerDeviation.Value = value;
}

protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    // Creating indicators
    _sma = new SimpleMovingAverage { Length = SmaLength };
    _bollinger = new BollingerBands
    {
        Length = BollingerLength,
        Deviation = BollingerDeviation
    };

    // Adding indicators to strategy collection
    Indicators.Add(_sma);
    Indicators.Add(_bollinger);

    // Getting the chart
    _chart = GetChart();

    // Initializing the chart if available
    if (_chart != null)
    {
        InitializeChart();
    }

    // Subscribing to candles
    var subscription = new Subscription(

```

```

        DataType.TimeFrame(TimeSpan.FromMinutes(5)),
        Security);

subscription
    .WhenCandlesFinished(this)
    .Do(ProcessCandle)
    .Apply(this);

Subscribe(subscription);
}

private void InitializeChart()
{
    // Clear existing areas
    foreach (var area in _chart.Areas.ToArray())
        _chart.RemoveArea(area);

    // Create the main area for candles and indicators
    _mainArea = _chart.AddArea();

    // Create an additional area for volume
    _volumeArea = _chart.AddArea();

    // Configure chart elements
    ConfigureChartElements();
}

private void ConfigureChartElements()
{
    // Adding an element for displaying candles
    _candleElement = _mainArea.AddCandles();
    _candleElement.DrawStyle = ChartCandleDrawStyles.CandleStick;
    _candleElement.AntiAliasing = true;
    _candleElement.UpFillColor = Color.Green;
    _candleElement.DownFillColor = Color.Red;
    _candleElement.UpBorderColor = Color.DarkGreen;
    _candleElement.DownBorderColor = Color.DarkRed;
    _candleElement.StrokeThickness = 1;
    _candleElement.ShowAxisMarker = true;

    // Adding elements for indicators
    _smaElement = _mainArea.AddIndicator(_sma);
    _smaElement.Color = Color.Blue;
    _smaElement.StrokeThickness = 2;

    _bollingerUpperElement = _mainArea.AddIndicator(_bollinger);
    _bollingerUpperElement.Color = Color.Purple;
    _bollingerUpperElement.StrokeThickness = 1;

    _bollingerMiddleElement = _mainArea.AddIndicator(_bollinger);
    _bollingerMiddleElement.Color = Color.Gray;
    _bollingerMiddleElement.StrokeThickness = 1;

    _bollingerLowerElement = _mainArea.AddIndicator(_bollinger);
    _bollingerLowerElement.Color = Color.Purple;
    _bollingerLowerElement.StrokeThickness = 1;

    // Adding elements for orders and trades
    _ordersElement = DrawOrders(_mainArea);
    _tradesElement = DrawOwnTrades(_mainArea);
}

private void ProcessCandle(ICandleMessage candle)
{
    // Processing candle with indicators
    var smaValue = _sma.Process(candle);
    var bollingerValue = _bollinger.Process(candle);

    // If chart is unavailable, skip drawing

```

```

if (_chart == null)
    return;

// Drawing data on the chart
var drawData = _chart.CreateData();
var group = drawData.Group(candle.OpenTime);

// Adding candle
group.Add(_candleElement,
    candle.DataType,
    candle.SecurityId,
    candle.OpenPrice,
    candle.HighPrice,
    candle.LowPrice,
    candle.ClosePrice,
    candle.PriceLevels,
    candle.State);

// Adding indicator values
group.Add(_smaElement, smaValue);

if (bollingerValue != null)
{
    group.Add(_bollingerUpperElement, bollingerValue);
    group.Add(_bollingerMiddleElement, bollingerValue);
    group.Add(_bollingerLowerElement, bollingerValue);
}

// Drawing data on the chart
_chart.Draw(drawData);

// Trading logic
if (!IsFormed)
    return;

// ... implementation of trading logic ...
}
}

```

Conclusion

Using charts in StockSharp strategies allows visualizing trading activity, which significantly simplifies the development, debugging, and monitoring of trading strategies. The [Strategy](#) class provides many methods for working with charts, allowing easy addition of various elements and rendering data.

When developing a strategy with a graphical interface, always consider that the chart may be unavailable, for example, when running in console mode or cloud testing. Therefore, it's important to check the result of the [GetChart\(\)](#) method for null and provide an alternative scenario for the strategy to work without visualization.

See Also

- [Indicators in Strategy](#)
- [Trading Operations in Strategies](#)

Strategy Statistics

Overview

The StockSharp platform provides a comprehensive system for statistical analysis of trading strategies, which helps traders evaluate effectiveness, optimize parameters, and make informed decisions. The statistics system collects and processes data from various aspects of trading, including orders, trades, positions, and profit/loss indicators.

Purpose and Benefits

Statistical analysis in trading strategies serves several important functions:

1. **Performance Measurement:** Quantitative assessment of your strategy's success using metrics such as net profit, maximum drawdown, and recovery factor.
2. **Risk Management:** Understanding your strategy's risk profile through metrics such as maximum drawdown percentage and position size statistics.
3. **Optimization:** Finding optimal strategy parameters by comparing statistical indicators across different parameter sets.
4. **Trade Quality Analysis:** Analyzing trade distribution, ratio of profitable to losing trades, average profit per trade.
5. **Operational Metrics:** Tracking operational metrics such as latency statistics and order error rates to identify execution issues.

Available Statistical Indicators

The [IStatisticManager](#) interface in StockSharp provides access to numerous statistical parameters organized into several categories:

Profit and Loss Statistics

- Net Profit
- Net Profit (%)
- Maximum Profit
- Maximum Drawdown
- Maximum Drawdown (%)
- Maximum Relative Drawdown
- Recovery Factor

Trade Statistics

- Number of Profitable Trades
- Number of Losing Trades
- Total Number of Trades
- Average Profit per Trade

- Average Profitable Trade
- Average Losing Trade
- Number of Trades per Month/Day

Position Statistics

- Maximum Long Position
- Maximum Short Position

Order Statistics

- Number of Orders
- Number of Order Errors
- Maximum/Minimum Registration Delay
- Maximum/Minimum Cancellation Delay

Integration with the Strategy Class

The [Strategy](#) class automatically collects and calculates statistics during execution. The statistics manager is available through the `StatisticManager` property, which implements the [IStatisticManager](#) interface.

Key statistical values are also directly represented as properties of the Strategy class:

- PnL: Profit and loss value
- Commission: Total paid commission
- Slippage: Total slippage
- Latency: Average order operation latency

Visualization

StockSharp provides a special graphical component for visualizing strategy statistics called `StatisticParameterGrid`, which is available in the `StockSharp.Xaml` namespace. This grid displays all statistical parameters in a user-friendly format.

For more information about the graphical component, see the documentation on [Statistics](#).

Usage Example

Here's an example of working with strategy statistics in your code:

```
// Create a strategy
var strategy = new SmaStrategy
{
    // Configure strategy parameters
    Security = security,
    Portfolio = portfolio,
    Volume = 1,
    // Set SMA parameters
    LongSma = 200,
    ShortSma = 50,
};

// Connect the strategy to a chart for visualization
var chart = new ChartPanel();
strategy.SetChart(chart);
```



```

// Access the statistics manager
var statisticManager = strategy.StatisticManager;

// Display strategy statistics in the user interface
// Assuming you have a StatisticParameterGrid defined in XAML as 'StatisticsGrid'
StatisticsGrid.Parameters.Clear();
StatisticsGrid.Parameters.AddRange(statisticManager.Parameters);

// Start the strategy
strategy.Start();

// When you need to react to changes in statistics
strategy.PnLChanged += () =>
{
    Console.WriteLine($"Current PnL: {strategy.PnL}");

    // You can also access individual statistical parameters
    var netProfit = statisticManager.Parameters
        .OfType<NetProfitParameter>()
        .FirstOrDefault();

    if (netProfit != null)
    {
        Console.WriteLine($"Net Profit: {netProfit.Value}");
    }
};

// For tracking position statistics
strategy.PositionChanged += () =>
{
    Console.WriteLine($"Current Position: {strategy.Position}");
};

```

Custom Statistics

You can also create your own statistical parameters by implementing the appropriate interfaces:

- [IStatisticParameter](#): Base interface for all statistical parameters
- [IPnLStatisticParameter](#): For parameters related to profit/loss
- [ITradeStatisticParameter](#): For parameters related to trades
- [IPositionStatisticParameter](#): For parameters related to positions
- [IOrderStatisticParameter](#): For parameters related to orders

Here's a simple example of a custom statistical parameter:

```

[Display(
    ResourceType = typeof(LocalizedStrings),
    Name = "My Custom Indicator",
    Description = "Description of my custom indicator",
    GroupName = "Custom Parameters",
    Order = 1000
)]
public class MyCustomParameter : BasePnLStatisticParameter<decimal>
{
    public MyCustomParameter()
        : base(StatisticParameterTypes.Custom)
    {
    }

    public override void Add(DateTimeOffset marketTime, decimal pnl, decimal? commission)
    {
        // Custom calculation logic
    }
}

```

```
    Value = /* your custom calculation */;
}
}

// Then add it to your strategy's StatisticManager
strategy.StatisticManager.Parameters.Add(new MyCustomParameter());
```

Conclusion

The statistical analysis system in StockSharp provides traders with powerful tools for evaluating and optimizing their trading strategies. By using these statistics, you can gain valuable insights into your strategy's performance, identify areas for improvement, and make data-driven decisions to enhance your trading results.

Settings saving and loading

Saving and Loading Settings

In StockSharp, the mechanism for saving and loading strategy settings is implemented through the [Strategy.Save](#) and [Strategy.Load](#) methods.

Automatic Parameter Handling

For most cases, **there is no need** to override the Save and Load methods, as the base [Strategy](#) class automatically saves and loads strategy parameters created using [StrategyParam<T>](#).

The recommended approach is to use the strategy parameter mechanism described in detail in the [Strategy Parameters](#) section. With this approach, all parameters are automatically saved and loaded:

```
public class SmaStrategy : Strategy
{
    private readonly StrategyParam<int> _longSmaLength;

    public int LongSmaLength
    {
        get => _longSmaLength.Value;
        set => _longSmaLength.Value = value;
    }

    public SmaStrategy()
    {
        _longSmaLength = Param(nameof(LongSmaLength), 80)
            .SetDisplay("Long SMA length", string.Empty, "Base settings");
    }
}
```

Overriding for Special Cases

Overriding the Save and Load methods is only required in special cases when you need to save or load data that is not part of the standard set of strategy parameters. For example, to save internal state, non-standard data structures, or cached values.

If you override these methods, **you must call the base class methods**:

```
public override void Save(SettingsStorage settings)
{
    // First call the base method to save standard parameters
    base.Save(settings);

    // Then add your specific saving logic
    settings.SetValue("CustomState", _customState);
}

public override void Load(SettingsStorage settings)
{
    // First call the base method to load standard parameters
    base.Load(settings);

    // Then add your specific loading logic
    if (settings.Contains("CustomState"))
        _customState = settings.GetValue<string>("CustomState");
}
```

Saving and Loading from a File

To save settings to a file or load from a file, you can use the serialization and deserialization implemented in StockSharp:

```
// Save settings to a file
var settingsStorage = new SettingsStorage();
strategy.Save(settingsStorage);
new JsonSerializer<SettingsStorage>().Serialize(settingsStorage, "strategy.json");

// Load settings from a file
var newStrategy = new SmaStrategy();
if (File.Exists("strategy.json"))
{
    var loadedSettings = new JsonSerializer<SettingsStorage>().Deserialize("strategy.json");
    newStrategy.Load(loadedSettings);
}
```

Recommendations

1. When possible, use [StrategyParam<T>](#) for all configurable strategy parameters.
2. Override the Save and Load methods only when you need to save/load non-standard data.
3. Always call the base methods `base.Save()` and `base.Load()` when overriding.
4. Use StockSharp's standard serialization tools to save settings to a file or load from a file.

See also

[Strategy Parameters](#)

Orders and trades loading

Loading Orders and Trades

When starting a strategy, it may be necessary to load previously executed orders and trades (for example, when a robot was restarted during a trading session or when orders and trades are carried over overnight). To do this, you need to:

1. Load the transaction IDs of orders saved earlier (for example, from a file).
2. Subscribe to the `OrderReceived` event to record new transaction IDs for future sessions.
3. Override the `CanAttach` method to associate previously placed orders with the strategy.
4. After the orders are attached to the strategy, all trades executed on them will be loaded automatically.

The following example shows how to load all trades into a strategy:

Loading Previously Executed Orders and Trades into a Strategy

1. When the strategy starts, load saved transaction numbers and subscribe to `OrderReceived` to store new ones:

```
private HashSet<long> _transactions;

protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    _transactions = File.Exists($"orders_{Name}.txt")
        ? File.ReadAllLines($"orders_{Name}.txt").Select(l => l.To<long>()).ToHashSet()
        : new HashSet<long>();

    OrderReceived += order =>
    {
        File.AppendAllLines($"orders_{Name}.txt", new[] { order.TransactionId.ToString() });
        _transactions.Add(order.TransactionId);
    };
}
```

2. Override `CanAttach` so the strategy can recognize its orders after a restart:

```
protected override bool CanAttach(Order order)
{
    return _transactions.Contains(order.TransactionId);
}
```

3. After the orders are loaded into the strategy, all trades executed on them will also be loaded automatically.

Round price

Price Rounding

Introduction

The [ShrinkPrice](#) method in StockSharp is an essential tool for correctly rounding prices according to market requirements. This ensures that submitted orders comply with exchange or broker rules.

Purpose

The primary goal of [ShrinkPrice](#) is to round prices to allowable values, considering:

1. The instrument's price step ([Security.PriceStep](#))
2. The number of decimal places ([Security.Decimals](#))

Importance of Usage

Using [ShrinkPrice](#) is crucial for:

- Preventing order rejection by the exchange or broker due to incorrect prices
- Ensuring accuracy in calculations and trading operations
- Complying with the rules and restrictions of specific markets or instruments

Operation Principle

1. If [Security.PriceStep](#) is set:
 - The price is rounded to the nearest value multiple of the price step.
2. If [Security.Decimals](#) is set:
 - The price is rounded to the specified number of decimal places.
3. If both parameters are set:
 - More stringent rounding is applied (usually to the price step).

Example Usage

```
// Create a Security object with specified parameters
var security = new Security
{
    PriceStep = 0.01m, // Price step of 0.01
    Decimals = 2       // Two decimal places
};

// Examples of using ShrinkPrice

// Example 1: Rounding to the price step
decimal price1 = 10.234m;
decimal shrunkPrice1 = price1.ShrinkPrice(security);
Console.WriteLine($"Original price: {price1}, After ShrinkPrice: {shrunkPrice1}");
// Output: Original price: 10.234, After ShrinkPrice: 10.23

// Example 2: Rounding a price that already matches the step
decimal price2 = 10.22m;
decimal shrunkPrice2 = price2.ShrinkPrice(security);
```

```
Console.WriteLine($"Original price: {price2}, After ShrinkPrice: {shrunkPrice2}");
// Output: Original price: 10.22, After ShrinkPrice: 10.22

// Example 3: Rounding a price with more decimal places
decimal price3 = 10.2345678m;
decimal shrunkPrice3 = price3.ShrinkPrice(security);
Console.WriteLine($"Original price: {price3}, After ShrinkPrice: {shrunkPrice3}");
// Output: Original price: 10.2345678, After ShrinkPrice: 10.23

// Example 4: Using ShrinkPrice when creating an order
var order = new Order
{
    Security = security,
    Price = 10.237m.ShrinkPrice(security) // Round the price before creating the order
};
Console.WriteLine($"Order price: {order.Price}");
// Output: Order price: 10.24
```

Application

[ShrinkPrice](#) should be used before submitting any orders or performing calculations that require precise price compliance with market conditions.

Conclusion

Proper use of [ShrinkPrice](#) helps avoid errors when placing orders and ensures that trading algorithms operate correctly according to market requirements.

Unit type

Unit Type

To simplify working with arithmetic operations on values such as percentages and absolute values, you can use the [Unit](#) data type. It allows transparent operations with addition, subtraction, multiplication, and division. [Unit](#) can be converted to [Decimal](#) (except when the value is a percentage, i.e., the type [Unit.Type](#) is set to [UnitTypes.Percent](#)) and back (in this case, a value with an absolute value is always created, i.e., the type [Unit.Type](#) is set to [UnitTypes.Absolute](#)).

Using Unit

- You can create a [Unit](#) using special constructors or use a shorter notation with the help of [UnitHelper](#):

```
// creating an absolute value
var absolute = new Unit(30);

// creating a percentage value
var percent = 30.0.Percents();
```

- [Unit](#) has formatted output. Therefore, converting values to a string depending on the type [UnitTypes](#):

```
Console.WriteLine("absolute = " + absolute);
Console.WriteLine("percent = " + percent);
```

will output the following lines:

```
absolute = 30
percent = 30%
```

- Arithmetic operations on [Unit](#) are performed in the same way as on regular numbers:

```
// addition of values
Console.WriteLine("absolute + percent = " + (absolute + percent));

// multiplication of values
Console.WriteLine("absolute * percent = " + (absolute * percent));

// subtraction of values
Console.WriteLine("absolute - percent = " + (absolute - percent));

// division of values
Console.WriteLine("absolute / percent = " + (absolute / percent));
```

- The result of arithmetic operations on [Unit](#) becomes a [Unit](#) itself, the type of which equals the type of the first operand. For example, if you add an absolute value and a percentage, the result will be in an absolute value:

```
// addition of absolute value and percentage
var resultAbsolutePercents = absolute + percent;
// and converting from absolute value to decimal
var resultAbsolutePercentsDecimal = (decimal)resultAbsolutePercents;
Console.WriteLine("absolute + percent = " + resultAbsolutePercents);
Console.WriteLine("(decimal)(absolute + percent) = " + resultAbsolutePercentsDecimal);
```

The output of such operations will be as follows:

```
absolute + percent = 39
(decimal)(absolute + percent) = 39
```


- You can also use familiar comparison operators to compare values:

```
if (absolute > percent)
    Console.WriteLine("Absolute value is greater than percentage");

if (absolute == percent)
    Console.WriteLine("Values are equal");
```

- To make a value positive, you can use the `Abs()` method:

```
var negative = new Unit(-10);
var positive = negative.Abs(); // 10
```

- To multiply a value by a specific factor, you can use the `Times()` method:

```
var multiplied = absolute.Times(5); // 150
```

Limitations

- Values of type `UnitTypes.Percent` cannot be converted to other types
- Values of type `UnitTypes.Limit` cannot be used in arithmetic operations
- Comparison between values of different types is not always possible

The `Unit` type provides a consistent way to work with various kinds of values in trading applications, making it easier to express calculations related to price and volume.

Quoting algorithm

Quoting Algorithm

Overview

A quoting algorithm is a mechanism that allows automatically placing and updating orders in the market with the aim of achieving the best execution price. Instead of placing aggressive market orders, quoting uses limit orders, which helps minimize slippage and reduce trading costs.

Main Components

StockSharp provides two key components for quoting:

1. [QuotingProcessor](#) — the main processor that analyzes market data and order status to recommend quoting actions.
2. [IQuotingBehavior](#) — an interface that defines quoting behavior, including calculating the optimal price and determining the need to update orders.

Examples of Strategies with Quoting

The documentation presents examples of strategies that demonstrate the use of the quoting mechanism:

- **MqStrategy** — an example of a strategy that uses the quoting mechanism to manage a position in the market.
- **MqSpreadStrategy** — an example of a strategy that creates a spread in the market by simultaneously placing buy and sell quotes.
- **StairsCountertrendStrategy** — an example of a countertrend strategy using quoting for more precise market entry.

These examples help to understand how to integrate the quoting mechanism into your own trading algorithms.

Quoting Behaviors

StockSharp supports various quoting behaviors:

- [MarketQuotingBehavior](#) — quoting based on market price with customizable offset and type.
- [BestByPriceQuotingBehavior](#) — quoting based on the best price with customizable offset.
- [LimitQuotingBehavior](#) — quoting at a fixed price.
- [BestByVolumeQuotingBehavior](#) — quoting based on the best price by volume.
- [LevelQuotingBehavior](#) — quoting based on a specified level in the order book.
- [LastTradeQuotingBehavior](#) — quoting based on the price of the last trade.
- [TheorPriceQuotingBehavior](#) — quoting options based on theoretical price.
- [VolatilityQuotingBehavior](#) — quoting options based on volatility.

Use in Your Own Strategy

Step 1: Create a Quoting Behavior

```
// Create a behavior for market quoting
var behavior = new MarketQuotingBehavior(
    new Unit(0.01m), // Price offset from the best quote
    new Unit(0.1m, UnitTypes.Percent), // Minimum deviation for quote update
    MarketPriceTypes.Following // Market price type for quoting
);
```

Step 2: Create and Initialize the Processor

```
// Create a quoting processor
_quotingProcessor = new QuotingProcessor(
    behavior,
    Security, // Instrument
    Portfolio, // Portfolio
    Sides.Buy, // Quoting direction
    Volume, // Quoting volume
    Volume, // Maximum order volume
    TimeSpan.Zero, // No timeout
    this, // Strategy implements ISubscriptionProvider
    this, // Strategy implements IMarketRuleContainer
    this, // Strategy implements ITransactionProvider
    this, // Strategy implements ITimeProvider
    this, // Strategy implements IMarketDataProvider
    IsFormedAndOnlineAndAllowTrading, // Check trading permission
    true, // Use order book prices
    true // Use last trade price if order book is empty
)
{
    Parent = this
};
```

Step 3: Subscribe to Processor Events

```
// Subscribe to processor events for logging and handling
_quotingProcessor.OrderRegistered += order =>
    this.AddInfoLog($"Order {order.TransactionId} registered at price {order.Price}");

_quotingProcessor.OrderFailed += fail =>
    this.AddInfoLog($"Order failed: {fail.Error.Message}");

_quotingProcessor.OwnTrade += trade =>
    this.AddInfoLog($"Trade executed: {trade.Trade.Volume} at {trade.Trade.Price}");

_quotingProcessor.Finished += isOk => {
    this.AddInfoLog($"Quoting finished with success: {isOk}");
    _quotingProcessor?.Dispose();
    _quotingProcessor = null;
};
```

Step 4: Start the Processor

```
// Start the processor
_quotingProcessor.Start();
```

Step 5: Release Resources

Don't forget to clean up processor resources when stopping the strategy:

```
protected override void OnStopped()
{
    // Release resources of the current processor
}
```

```
_quotingProcessor?.Dispose();  
_quotingProcessor = null;  
  
base.OnStopped();  
}
```

Advantages of Usage

1. **Reduced Slippage** — quoting helps to get a better execution price compared to market orders.
2. **Configuration Flexibility** — various quoting behaviors for different market situations.
3. **Automatic Updates** — the processor automatically tracks market changes and updates orders when necessary.
4. **Full Control** — ability to configure quoting parameters, including price offset, minimum deviation, and market price type.
5. **Risk Management** — ability to set a timeout to limit execution time.

Applied Use Cases

- **Market Making** — creating liquidity in the market by placing two-sided quotes.
- **Algorithmic Trading** — improving execution price in automated strategies.
- **Countertrend Trading** — more precise market entry against the trend.
- **Arbitrage** — simultaneous quoting on multiple markets to take advantage of price discrepancies.

Implementing the quoting mechanism in your strategy can significantly improve order execution and increase its efficiency.

Working Securities

Description

The `GetWorkingSecurities()` method in the base class `Strategy` is used to obtain a list of instruments and data types that the strategy uses in its operation. This method plays an important role when working with the [Designer](#).

Purpose

The main purpose of the method is to provide the Designer with information about which instruments and data types are necessary for the strategy to work. This allows the Designer to:

1. Check for the availability of necessary historical data in the storage before starting testing
2. Automatically load the required data when available
3. Correctly set up subscriptions when launching the strategy

Implementation

In the base class `Strategy`, the method returns an empty collection. For correct work with the Designer, it is recommended to override it in your strategy:

```
public override IEnumerable<(Security sec, DataType dt)> GetWorkingSecurities()
{
    // Return a list of pairs (instrument, data type) used by the strategy
    return new[]
    {
        (Security, CandleType),
        // Other instrument-data type pairs if the strategy uses multiple
    };
}
```

Importance of Overriding the Method

If the `GetWorkingSecurities()` method is not overridden in your strategy:

- The Designer will not be able to automatically check for the necessary data
- If the required historical data is missing in the storage, the Designer will not issue warnings
- The strategy may be launched for testing, but no results will be shown
- The user will not receive any information about the reason for the absence of results

Usage Example

```
public class MySmaStrategy : Strategy
{
    private readonly StrategyParam<DataType> _candleType;

    public DataType CandleType
    {
        get => _candleType.Value;
        set => _candleType.Value = value;
    }
}
```

```

}

public MySmaStrategy()
{
    _candleType = Param(nameof(CandleType), DataType.TimeFrame(TimeSpan.FromMinutes(1)));
}

// Override the method for correct work with the Designer
public override IEnumerable<(Security sec, DataType dt)> GetWorkingSecurities()
{
    return new[] { (Security, CandleType) };
}

// The rest of the strategy code...
}

```

Conclusion

Although the `GetWorkingSecurities()` method is not mandatory for implementing the basic functionality of a strategy, overriding it is strongly recommended for correct work with the StockSharp Designer. This helps avoid situations where a strategy is launched for testing but does not show any results due to the absence of necessary historical data.

Strategy Overview

Strategies

Overview

The Strategy class is the base class for creating trading strategies in StockSharp. It provides a complete set of tools for subscribing to market data, managing orders and positions, calculating statistics, and generating reports.

Key capabilities of the Strategy class:

- Subscribing to candles, order books, ticks, and other market data
- Placing, modifying, and canceling orders
- Target position management
- PnL, commission, and statistics calculation
- Risk management
- Timer and rule system
- Alerts
- Report generation

Child strategies functionality (ChildStrategies) has been declared obsolete and is no longer supported. The ChildStrategies property is marked with the `[Obsolete("Child strategies no longer supported.")]` attribute. If your code uses child strategies, it is recommended to refactor -- run each strategy as an independent instance.

Documentation Sections

- [Target Position Management](#) -- declarative position size management via `SetTargetPosition`
- [Trading Modes](#) -- restricting trading activity via `StrategyTradingModes`
- [Alert System](#) -- sending notifications (popup, sound, log, Telegram)
- [Timer System](#) -- periodic action execution
- [Risk Management](#) -- risk management rules
- [High-Level Subscriptions](#) -- simplified market data subscriptions
- [Strategy Reports](#) -- generating trade result reports
- [Advanced Features](#) -- order comments, schedules, risk-free rate, indicator source

Minimal Strategy

```
public class MyStrategy : Strategy
{
    private readonly StrategyParam<DataType> _candleType;

    public DataType CandleType
    {
        get => _candleType.Value;
        set => _candleType.Value = value;
    }
}
```

```

    }

    public MyStrategy()
    {
        _candleType = Param(nameof(CandleType), TimeSpan.FromMinutes(5).TimeFrame());
    }

    protected override void OnStarted2(DateTime time)
    {
        base.OnStarted2(time);

        var subscription = SubscribeCandles(CandleType);

        subscription
            .Bind(ProcessCandle)
            .Start();
    }

    private void ProcessCandle(ICandleMessage candle)
    {
        if (!IsFormedAndOnlineAndAllowTrading())
            return;

        // Trading logic
    }
}

```

Strategy Lifecycle

1. **Creation** -- constructor, declaring parameters via `Param<T>`.
2. **Configuration** -- setting `Security`, `Portfolio`, `Connector`, and parameters.
3. **Start** -- calling `Start()`, transition to `ProcessStates.Started` state, invocation of `OnStarted2(DateTime)`.
4. **Running** -- processing market data, placing orders.
5. **Stop** -- calling `Stop()`, transition through `ProcessStates.Stopping` to `ProcessStates.Stopped`, invocation of `OnStopped()`.

Target Position Management

Target Position Management

Overview

The target position management system allows a strategy to declaratively specify the desired position size, while the platform automatically places the necessary orders to reach that level. Instead of manually calculating the volume and direction of a trade, you simply call `SetTargetPosition(10)` -- and the manager will determine whether to buy or sell, and in what volume.

The key component is the `PositionTargetManager` class, which automatically:

- Calculates the difference between the current and target position
- Determines the direction and volume of the order
- Handles order execution, cancellation, and errors
- Supports retry attempts on failures

Strategy Methods

SetTargetPosition

Sets the target position. Two calling variants are available:

```
// For the strategy's main security and portfolio
SetTargetPosition(decimal target);

// For an arbitrary security and portfolio
SetTargetPosition(Security security, Portfolio portfolio, decimal target);
```

When target is greater than the current position, the manager will place a buy order. When less -- a sell order. If the position already equals the target (taking into account `PositionTolerance`), no action is taken.

CancelTargetPosition

Cancels a previously set target position and stops all related active orders:

```
// For the strategy's main security and portfolio
CancelTargetPosition();

// For an arbitrary security and portfolio
CancelTargetPosition(Security security, Portfolio portfolio);
```

GetTargetPosition

Returns the current target position value, or null if no target is set:

```
decimal? target = GetTargetPosition();
decimal? target = GetTargetPosition(security, portfolio);
```

TargetPositionManager Property

The `TargetPositionManager` property provides direct access to the `PositionTargetManager` object for

fine-tuning:

```
// Maximum number of retry attempts on order error (default is 3)
TargetPositionManager.MaxRetries = 5;

// Tolerance for determining whether the target position is reached
TargetPositionManager.PositionTolerance = 0.01m;

// Order type (default is Market)
TargetPositionManager.OrderType = OrderTypes.Market;
```

The manager generates the following events:

- TargetReached -- target position has been reached
- Error -- an error occurred during order execution
- OrderRegistered -- the manager has registered an order

TargetAlgoFactory Property

The TargetAlgoFactory property allows setting a factory for position change algorithms. By default, MarketOrderAlgo is used, which creates market orders:

```
// Use a custom algorithm instead of market orders
TargetAlgoFactory = (side, volume) => new MyCustomAlgo(side, volume);
```

Usage Example

```
public class TargetPositionStrategy : Strategy
{
    private readonly StrategyParam<DataType> _candleType;

    public DataType CandleType
    {
        get => _candleType.Value;
        set => _candleType.Value = value;
    }

    public TargetPositionStrategy()
    {
        _candleType = Param(nameof(CandleType), TimeSpan.FromMinutes(5).TimeFrame());
    }

    protected override void OnStarted2(DateTime time)
    {
        base.OnStarted2(time);

        // Configure the target position manager
        TargetPositionManager.MaxRetries = 5;
        TargetPositionManager.TargetReached += (sec, pf) =>
        {
            this.AddInfoLog("Target position reached: {0}, {1}", sec, pf);
        };

        var subscription = SubscribeCandles(CandleType);

        subscription
            .Bind(ProcessCandle)
            .Start();
    }

    private void ProcessCandle(ICandleMessage candle)
    {

```

```

    if (!IsFormedAndOnlineAndAllowTrading())
        return;

    if (candle.OpenPrice < candle.ClosePrice)
    {
        // Bullish candle -- set target position for buying
        SetTargetPosition(Volume);
    }
    else if (candle.OpenPrice > candle.ClosePrice)
    {
        // Bearish candle -- set target position for selling
        SetTargetPosition(-Volume);
    }
}
}

```

In this example, the strategy does not deal with manual volume and direction calculations. It simply declares the desired position size, and `PositionTargetManager` handles all the order placement work.

Alert System

Overview

Strategies in StockSharp have a built-in alert system that allows sending notifications of various types: popup windows, sound signals, log entries, and Telegram messages. Alerts are useful for informing the trader about important events -- position entries, level breaches, errors, and other trading signals.

During backtesting, alerts other than the Log type are automatically skipped to avoid interference during testing.

Alert Types

The `AlertNotifications` enumeration defines the available types:

Type	Description
Sound	Sound signal
Popup	Popup window
Log	Log file entry
Telegram	Telegram message

Methods

Alert

Base method for sending an alert with a specified type, caption, and message:

```
// With caption and message
Alert(AlertNotifications type, string caption, string message);

// With automatic caption (uses the strategy name)
Alert(AlertNotifications type, string message);
```

AlertPopup

Sends a popup notification. The caption is the strategy name:

```
AlertPopup(string message);
```

AlertSound

Sends a sound notification:

```
AlertSound(string message);
```

AlertLog

Sends a notification to the log. This type works during backtesting as well:

```
AlertLog(string message);
```

Configuring the Alert Service

For alerts to work, the `IAAlertNotificationService` service must be registered in the strategy's environment. This is done via the extension method:

```
strategy.SetAlertService(alertService);
```

The current service can be retrieved via:

```
var service = strategy.GetAlertService();
```

In graphical applications (Designer, terminal), the service is usually registered automatically.

Usage Example

```
public class AlertStrategy : Strategy
{
    private readonly StrategyParam<DataType> _candleType;
    private readonly StrategyParam<decimal> _priceLevel;

    public DataType CandleType
    {
        get => _candleType.Value;
        set => _candleType.Value = value;
    }

    public decimal PriceLevel
    {
        get => _priceLevel.Value;
        set => _priceLevel.Value = value;
    }

    public AlertStrategy()
    {
        _candleType = Param(nameof(CandleType), TimeSpan.FromMinutes(5).TimeFrame());
        _priceLevel = Param(nameof(PriceLevel), 100m);
    }

    protected override void OnStarted2(DateTime time)
    {
        base.OnStarted2(time);

        var subscription = SubscribeCandles(CandleType);

        subscription
            .Bind(ProcessCandle)
            .Start();

        // Alert about strategy start
        AlertLog("Strategy started, tracked level: " + PriceLevel);
    }

    private void ProcessCandle(ICandleMessage candle)
    {
        if (!IsFormedAndOnlineAndAllowTrading())
            return;

        // Price crossed the level from below upward
        if (candle.OpenPrice < PriceLevel && candle.ClosePrice >= PriceLevel)
        {
            AlertPopup("Price crossed level " + PriceLevel + " upward!");
        }
    }
}
```

```

        AlertSound("Level breakout!");
        BuyMarket();
    }

    // Price crossed the level from above downward
    if (candle.OpenPrice > PriceLevel && candle.ClosePrice <= PriceLevel)
    {
        Alert(AlertNotifications.Telegram, "Trading signal",
            "Price broke level " + PriceLevel + " downward");
        SellMarket();
    }
}

```

In this example, the strategy uses different alert types for different situations: `AlertPopup` and `AlertSound` for immediately capturing the trader's attention, and `Alert` with the `Telegram` type for remote notification.

Timer System

Overview

Strategies in StockSharp support a built-in timer system that allows executing actions at specified time intervals. Timers are based on the connector's `WhenIntervalElapsed` mechanism and work correctly both in live trading and during backtesting (using the emulator's virtual time).

Timers are convenient for:

- Periodic market condition checks
- Portfolio rebalancing at fixed intervals
- Closing positions by time
- Strategy state monitoring

ITimerHandler Interface

The timer is managed through the `ITimerHandler` interface, which provides the following members:

Member	Description
<code>Start()</code>	Starts the timer. Returns <code>ITimerHandler</code> for method chaining
<code>Stop()</code>	Stops the timer. Returns <code>ITimerHandler</code> for method chaining
<code>Interval</code>	Timer trigger interval (read and write)
<code>IsStarted</code>	Indicates whether the timer is running
<code>Dispose()</code>	Releases resources and stops the timer

Timer Creation Methods

CreateTimer

Creates a timer but does not start it. Starting is done by a separate `Start()` call:

```
ITimerHandler CreateTimer(TimeSpan interval, Action callback);
```

StartTimer

Creates and immediately starts a timer. Equivalent to `CreateTimer(...).Start()`:

```
ITimerHandler StartTimer(TimeSpan interval, Action callback);
```

The interval must be a positive value (`> TimeSpan.Zero`), otherwise an exception will be thrown.

Timer Management

A timer can be stopped and restarted:

```

var timer = CreateTimer(TimeSpan.FromMinutes(1), MyCallback);

// Start
timer.Start();

// Stop
timer.Stop();

// Change interval
timer.Interval = TimeSpan.FromMinutes(5);

// Start again
timer.Start();

// Release resources
timer.Dispose();

```

Usage Example

```

public class TimerStrategy : Strategy
{
    private ITimerHandler _checkTimer;
    private ITimerHandler _closeTimer;

    private readonly StrategyParam<TimeSpan> _checkInterval;
    private readonly StrategyParam<TimeSpan> _maxHoldTime;

    public TimeSpan CheckInterval
    {
        get => _checkInterval.Value;
        set => _checkInterval.Value = value;
    }

    public TimeSpan MaxHoldTime
    {
        get => _maxHoldTime.Value;
        set => _maxHoldTime.Value = value;
    }

    public TimerStrategy()
    {
        _checkInterval = Param(nameof(CheckInterval), TimeSpan.FromMinutes(1));
        _maxHoldTime = Param(nameof(MaxHoldTime), TimeSpan.FromHours(4));
    }

    protected override void OnStarted2(DateTime time)
    {
        base.OnStarted2(time);

        // Timer for periodic market condition checks
        _checkTimer = StartTimer(CheckInterval, OnCheckTimer);

        // Timer for forced position closing (created but not started)
        _closeTimer = CreateTimer(MaxHoldTime, OnCloseTimer);

        var subscription = SubscribeCandles(TimeSpan.FromMinutes(5));

        subscription
            .Bind(ProcessCandle)
            .Start();
    }

    private void OnCheckTimer()
    {
        if (!IsFormedAndOnlineAndAllowTrading())
            return;
    }
}

```



```

        this.AddInfoLog("Checking market conditions at {0}", CurrentTime);

        // Periodic position state check
        if (Position != 0)
        {
            this.AddInfoLog("Current position: {0}", Position);
        }
    }

    private void OnCloseTimer()
    {
        if (Position != 0)
        {
            this.AddInfoLog("Position hold time expired, closing");
            ClosePosition();

            // Stop the close timer after it fires
            _closeTimer.Stop();
        }
    }

    private void ProcessCandle(ICandleMessage candle)
    {
        if (!IsFormedAndOnlineAndAllowTrading())
            return;

        if (candle.OpenPrice < candle.ClosePrice && Position == 0)
        {
            BuyMarket();

            // Start the forced close timer
            _closeTimer.Start();
        }
    }
}

```

In this example, two timers are used: one for periodic state checking (StartTimer -- created and immediately started), and another for limiting the position hold time (CreateTimer -- created but started only upon entering a position).

High-Level Subscriptions

High-Level Subscriptions

Overview

The Strategy class provides a set of high-level market data subscription methods: `SubscribeCandles`, `SubscribeTicks`, `SubscribeLevel1`, and `SubscribeOrderBook`. These methods return an `ISubscriptionHandler<T>` object that allows binding data handlers and indicators in a convenient fluent style.

Unlike manually creating a `Subscription` object and calling `Subscribe()`, the high-level methods:

- Automatically create a subscription with the correct parameters
- Provide a typed `ISubscriptionHandler<T>` for binding handlers
- Integrate with the indicator system via `Bind` methods
- Support automatic chart rendering
- Properly manage the subscription lifecycle

Subscription Methods

SubscribeCandles

Subscribes to candles. Accepts a timeframe or `DataType`:

```
// Subscribe by timeframe
ISubscriptionHandler<ICandleMessage> SubscribeCandles(
    TimeSpan tf,
    bool isFinishedOnly = true,
    Security security = default);

// Subscribe by DataType (supports all candle types)
ISubscriptionHandler<ICandleMessage> SubscribeCandles(
    DataType dt,
    bool isFinishedOnly = true,
    Security security = default);

// Subscribe with a ready-made Subscription object
ISubscriptionHandler<ICandleMessage> SubscribeCandles(Subscription subscription);
```

The `isFinishedOnly` parameter is true by default -- the handler receives only completed candles.

SubscribeTicks

Subscribes to tick trades:

```
ISubscriptionHandler<ITickTradeMessage> SubscribeTicks(Security security = null);
ISubscriptionHandler<ITickTradeMessage> SubscribeTicks(Subscription subscription);
```

SubscribeLevel1

Subscribes to Level1 data (best bid/ask prices, last trade, and other fields):

```
ISubscriptionHandler<Level1ChangeMessage> SubscribeLevel1(Security security = null);
```

```
ISubscriptionHandler<Level1ChangeMessage> SubscribeLevel1(Subscription subscription);
```

SubscribeOrderBook

Subscribes to the order book:

```
ISubscriptionHandler<IOrderBookMessage> SubscribeOrderBook(Security security = null);  
ISubscriptionHandler<IOrderBookMessage> SubscribeOrderBook(Subscription subscription);
```

If the security parameter is not specified, the strategy's Security is used.

ISubscriptionHandler Interface

The ISubscriptionHandler<T> object provides the following methods:

Start / Stop

Starting and stopping the subscription:

```
handler.Start();    // calls Subscribe  
handler.Stop();     // calls UnSubscribe
```

Bind (without indicators)

Binding a simple data handler:

```
handler.Bind(Action<T> callback);
```

Bind (with indicators)

Binding a handler with one or more indicators. The indicator automatically processes incoming data, and the handler receives the already-calculated value:

```
// One indicator -- decimal value  
handler.Bind(IIIndicator indicator, Action<T, decimal> callback);  
  
// Two indicators  
handler.Bind(IIIndicator ind1, IIIndicator ind2, Action<T, decimal, decimal> callback);  
  
// Up to eight indicators  
handler.Bind(ind1, ind2, ind3, ..., callback);  
  
// Array of indicators  
handler.Bind(IIIndicator[] indicators, Action<T, decimal[]> callback);
```

The handler with Bind is called only when all indicators have returned a non-empty value.

BindWithEmpty

Similar to Bind, but the handler is called even if the indicator returned an empty value. Values are represented as decimal?:

```
handler.BindWithEmpty(IIIndicator indicator, Action<T, decimal?> callback);
```

BindEx

Provides access to the full IIIndicatorValue object instead of the extracted decimal:

```
handler.BindEx(IIndicator indicator, Action<T, IIndicatorValue> callback, bool allowEmpty = false);
```

Example: Strategy with Indicators

```
public class SmaStrategy : Strategy
{
    private readonly StrategyParam<int> _shortPeriod;
    private readonly StrategyParam<int> _longPeriod;
    private readonly StrategyParam<DataType> _candleType;

    public int ShortPeriod
    {
        get => _shortPeriod.Value;
        set => _shortPeriod.Value = value;
    }

    public int LongPeriod
    {
        get => _longPeriod.Value;
        set => _longPeriod.Value = value;
    }

    public DataType CandleType
    {
        get => _candleType.Value;
        set => _candleType.Value = value;
    }

    public SmaStrategy()
    {
        _shortPeriod = Param(nameof(ShortPeriod), 10);
        _longPeriod = Param(nameof(LongPeriod), 20);
        _candleType = Param(nameof(CandleType), TimeSpan.FromMinutes(5).TimeFrame());
    }

    protected override void OnStarted2(DateTime time)
    {
        base.OnStarted2(time);

        var shortSma = new SimpleMovingAverage { Length = ShortPeriod };
        var longSma = new SimpleMovingAverage { Length = LongPeriod };

        var subscription = SubscribeCandles(CandleType);

        // Binding two indicators -- handler is called
        // when both indicators are formed
        subscription
            .Bind(shortSma, longSma, (candle, shortValue, longValue) =>
            {
                if (!IsFormedAndOnlineAndAllowTrading())
                    return;

                if (shortValue > longValue && Position <= 0)
                    BuyMarket(Volume + Math.Abs(Position));
                else if (shortValue < longValue && Position >= 0)
                    SellMarket(Volume + Math.Abs(Position));
            })
            .Start();

        // Chart setup
        var area = CreateChartArea();
        if (area != null)
        {
            DrawCandles(area, subscription);
            DrawIndicator(area, shortSma);
            DrawIndicator(area, longSma);
            DrawOwnTrades(area);
        }
    }
}
```

```

    }
}

```

Example: Tick Subscription

```

protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    SubscribeTicks()
        .Bind(tick =>
        {
            if (!IsFormedAndOnlineAndAllowTrading())
                return;

            this.AddInfoLog("Tick: price={0}, volume={1}", tick.Price, tick.Volume);
        })
        .Start();
}

```

Example: Order Book Subscription

```

protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    SubscribeOrderBook()
        .Bind(book =>
        {
            var bestBid = book.GetBestBid();
            var bestAsk = book.GetBestAsk();

            if (bestBid != null && bestAsk != null)
            {
                var spread = bestAsk.Price - bestBid.Price;
                this.AddInfoLog("Spread: {0}", spread);
            }
        })
        .Start();
}

```

Differences from Manual Subscription Creation

Aspect	Manual subscription	High-level method
Creation	<code>new Subscription(DataType, Security)</code>	<code>SubscribeCandles(tf)</code>
Data handling	Subscribing to connector events	<code>Bind(callback)</code>
Indicators	<code>Manual indicator.Process()</code> call	Automatically via <code>Bind(indicator, callback)</code>
Indicator registration	Manual addition to Indicators	Automatically on Bind
Chart rendering	Manual integration with IChart	<code>DrawCandles, DrawIndicator</code>

High-level methods are recommended for use in most strategies, as they significantly reduce the amount of code and decrease the likelihood of errors.

Risk Management

Risk Management

Overview

Every strategy in StockSharp has a built-in RiskManager that allows automatic control of trading risks. When a rule is triggered, the manager can close positions, cancel active orders, or completely stop the strategy's trading.

The risk manager processes all trading messages passing through the strategy, and when rule conditions are met, it automatically performs the specified action without any additional code in the strategy logic.

Strategy Properties

RiskManager

The IRiskManager object that manages the list of rules. It is created automatically when the strategy is initialized:

```
// Access the risk manager
IRiskManager manager = strategy.RiskManager;
```

RiskRules

A convenient property for reading and setting the list of rules:

```
// Read current rules
IEnumerable<IRiskRule> rules = strategy.RiskRules;

// Set new rules
strategy.RiskRules = new IRiskRule[]
{
    new RiskPnLRule { PnL = -1000m, Action = RiskActions.StopTrading },
    new RiskPositionSizeRule { Position = 100m, Action = RiskActions.ClosePositions },
};
```

Trigger Actions

The RiskActions enumeration defines the possible actions:

Action	Description
ClosePositions	Close all open positions with a market order
StopTrading	Block the strategy's trading
CancelOrders	Cancel all active orders

When a rule is triggered, the strategy logs a warning with the rule name, its parameters, and the action performed.

Available Rules

RiskPnLRule -- profit/loss control

Triggers when the specified P&L level is reached. A positive value controls profit, a negative value controls loss:

```
// Stop trading on a loss exceeding 5000
new RiskPnLRule
{
    PnL = -5000m,
    Action = RiskActions.StopTrading
}
```

RiskPositionSizeRule -- position size control

Triggers when the specified position size is reached:

```
// Cancel orders when position >= 100
new RiskPositionSizeRule
{
    Position = 100m,
    Action = RiskActions.CancelOrders
}
```

RiskOrderFreqRule -- order frequency control

Triggers when the number of orders within the specified interval exceeds the limit:

```
// Stop on more than 50 orders per minute
new RiskOrderFreqRule
{
    Count = 50,
    Interval = TimeSpan.FromMinutes(1),
    Action = RiskActions.StopTrading
}
```

RiskOrderVolumeRule -- order volume control

Triggers when the volume of a single order is exceeded.

RiskOrderPriceRule -- order price control

Triggers when the order price goes beyond the established boundaries.

RiskTradeVolumeRule -- trade volume control

Triggers when the volume of a single trade is exceeded.

RiskTradePriceRule -- trade price control

Triggers when the trade price goes beyond the established boundaries.

RiskTradeFreqRule -- trade frequency control

Triggers when the number of trades within the interval is exceeded.

RiskPositionTimeRule -- position hold time control

Triggers when a position has been held longer than the established time.

RiskCommissionRule -- commission control

Triggers when the total commission is exceeded.

RiskSlippageRule -- slippage control

Triggers when the slippage level is exceeded.

RiskErrorRule -- error control

Triggers when a certain number of errors have accumulated.

RiskOrderErrorRule -- order registration error control

Triggers when order registration errors have accumulated.

Usage Example

```
public class RiskAwareStrategy : Strategy
{
    private readonly StrategyParam<DataType> _candleType;

    public DataType CandleType
    {
        get => _candleType.Value;
        set => _candleType.Value = value;
    }

    public RiskAwareStrategy()
    {
        _candleType = Param(nameof(CandleType), TimeSpan.FromMinutes(5).TimeFrame());
    }

    protected override void OnStarted2(DateTime time)
    {
        base.OnStarted2(time);

        // Configure risk management rules
        RiskRules = new IRiskRule[]
        {
            // Close positions on a loss exceeding 10000
            new RiskPnLRule
            {
                PnL = -10000m,
                Action = RiskActions.ClosePositions
            },

            // Stop trading when position exceeds 500 contracts
            new RiskPositionSizeRule
            {
                Position = 500m,
                Action = RiskActions.StopTrading
            },

            // Cancel orders when frequency exceeds 100 per minute
            new RiskOrderFreqRule
            {
                Count = 100,
                Interval = TimeSpan.FromMinutes(1),
                Action = RiskActions.CancelOrders
            },
        };

        var subscription = SubscribeCandles(CandleType);

        subscription
            .Bind(ProcessCandle)
            .Start();
    }
}
```



```

private void ProcessCandle(ICandleMessage candle)
{
    if (!IsFormedAndOnlineAndAllowTrading())
        return;

    // Trading logic
    if (candle.OpenPrice < candle.ClosePrice && Position <= 0)
        BuyMarket(Volume + Math.Abs(Position));
    else if (candle.OpenPrice > candle.ClosePrice && Position >= 0)
        SellMarket(Volume + Math.Abs(Position));
}
}

```

In this example, risk management rules are set when the strategy starts. When a loss of 10000 is reached, positions will be automatically closed. When the position size exceeds 500 contracts, trading will be blocked. When orders are placed too frequently, active orders will be canceled. All these checks are performed automatically without any additional code in the ProcessCandle method.

Reporting

Strategy Reports

Overview

StockSharp provides a report generation system for strategy trading results. The system is built on two key components:

- **IReportSource** -- an interface describing the data source for the report (strategy parameters, orders, trades, positions, statistics).
- **IReportGenerator** -- a report generator interface supporting various formats (CSV, JSON, XML, Excel).

The Strategy class implements the IReportSource interface, so a strategy can be passed directly to the report generator.

IReportSource Interface

The IReportSource interface provides all the data needed to generate a report:

Property	Type	Description
Name	string	Strategy name
TotalWorkingTime	TimeSpan	Total working time
Commission	decimal?	Total commission
Position	decimal	Current position
PnL	decimal	Total profit/loss
Slippage	decimal?	Total slippage
Latency	TimeSpan?	Total latency
Parameters	IEnumerable<(string, object)>	Strategy parameters
StatisticParameters	IEnumerable<(string, object)>	Statistical parameters
Orders	IEnumerable<ReportOrder>	Orders
OwnTrades	IEnumerable<ReportTrade>	Own trades
Positions	IEnumerable<ReportPosition>	Position round-trips

Before reading data, the Prepare() method is called to synchronize the source's internal state.

ReportSource Class

ReportSource is a standalone implementation of IReportSource, not tied to the Strategy class. It allows manually constructing the data source for a report:

```
var source = new ReportSource();
source.Name = "My strategy";
source.PnL = 15000m;
```

```

source.TotalWorkingTime = TimeSpan.FromHours(8);

source.AddParameter("Timeframe", "5 minutes");
source.AddStatisticParameter("Sharpe Ratio", 1.85);

source.AddOrder(new ReportOrder(
    Id: 123,
    TransactionId: 456,
    SecurityId: securityId,
    Side: Sides.Buy,
    Time: DateTime.UtcNow,
    Price: 100m,
    State: OrderStates.Done,
    Balance: 0,
    Volume: 10,
    Type: OrderTypes.Limit
));

```

Data Aggregation

With a large number of orders and trades, ReportSource automatically aggregates data to reduce the report size:

```

// Automatic aggregation threshold (default is 10000)
source.MaxOrdersBeforeAggregation = 5000;
source.MaxTradesBeforeAggregation = 5000;

// Grouping interval (default is 1 hour)
source.AggregationInterval = TimeSpan.FromMinutes(30);

// Manual aggregation
source.AggregateOrders(TimeSpan.FromHours(1));
source.AggregateTrades(TimeSpan.FromHours(1));

```

During aggregation, orders and trades are grouped by time interval, security, and direction. Volumes are summed, and prices are calculated as weighted averages.

PositionLifecycleTracker

PositionLifecycleTracker tracks the lifecycle of positions and generates round-trips -- records of position opening and closing.

A round-trip is recorded when:

- A position is fully closed (the value becomes zero)
- A position reversal occurs (sign change)

In the Strategy class, the tracker is automatically integrated: completed round-trips are added to ReportSource via the RoundTripClosed event.

```

var tracker = new PositionLifecycleTracker();

// Event on round-trip close
tracker.RoundTripClosed += roundTrip =>
{
    Console.WriteLine($"Position closed: {roundTrip.SecurityId}, " +
        $"Open: {roundTrip.OpenTime} at {roundTrip.OpenPrice}, " +
        $"Close: {roundTrip.CloseTime} at {roundTrip.ClosePrice}, " +
        $"Max volume: {roundTrip.MaxPosition}");
};

// Process position update
tracker.ProcessPosition(position);

```

```
// Access round-trip history
IReadOnlyList<ReportPosition> history = tracker.History;
```

Report Generators

The following generators are available:

Generator	Format	Description
CsvReportGenerator	CSV	Text format with delimiters
JsonReportGenerator	JSON	Structured JSON
XmlReportGenerator	XML	XML format
ExcelReportGenerator	Excel	Excel format (requires IExcelWorkerProvider)

All generators inherit from BaseReportGenerator and support configuration of included sections:

```
var generator = new CsvReportGenerator();

// Configure report sections
generator.IncludeOrders = true;
generator.IncludeTrades = true;
generator.IncludePositions = true;
generator.Encoding = Encoding.UTF8;
```

Generating a Report from a Strategy

Since Strategy implements IReportSource, a report can be generated directly:

```
// The strategy itself is the data source
var generator = new JsonReportGenerator();

using var stream = File.Create("report.json");
await generator.Generate(strategy, stream, CancellationToken.None);
```

For a separate data source:

```
var source = new ReportSource();
source.Name = strategy.Name;
source.PnL = strategy.PnL;
source.TotalWorkingTime = strategy.TotalWorkingTime;

// Add positions from the tracker
source.AddPositions(tracker.History);

var generator = new CsvReportGenerator();
using var stream = File.Create("report.csv");
await generator.Generate(source, stream, CancellationToken.None);
```

Example: Strategy with Report Generation on Stop

```
public class ReportingStrategy : Strategy
{
    private readonly StrategyParam<DataType> _candleType;

    public DataType CandleType
```

```

{
    get => _candleType.Value;
    set => _candleType.Value = value;
}

public ReportingStrategy()
{
    _candleType = Param(nameof(CandleType), TimeSpan.FromMinutes(5).TimeFrame());
}

protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    var subscription = SubscribeCandles(CandleType);

    subscription
        .Bind(ProcessCandle)
        .Start();
}

private void ProcessCandle(ICandleMessage candle)
{
    if (!IsFormedAndOnlineAndAllowTrading())
        return;

    // Trading logic...
}

protected override void OnStopped()
{
    // Generate report when the strategy stops
    var generator = new CsvReportGenerator();

    using var stream = File.Create($"report_{Name}_{DateTime.Now:yyyyMMdd_HH:mm:ss}.csv");
    generator.Generate(this, stream, CancellationToken.None).AsTask().Wait();

    base.OnStopped();
}
}

```

In this example, the strategy automatically creates a CSV report when it stops. The report includes strategy parameters, statistics, orders, trades, and position round-trips.

Trading Modes

Strategy Trading Modes

Overview

The `TradingMode` property allows restricting a strategy's trading activity without stopping it completely. This is useful for risk management -- for example, prohibiting opening new positions while only allowing closing existing ones, or completely blocking order submission.

The mode is set using the `StrategyTradingModes` enumeration and can be changed while the strategy is running.

StrategyTradingModes Enumeration

Value	Description
Full	Full trading access. No restrictions on orders. Default value.
Disabled	Trading is completely prohibited. All order placement attempts will be rejected.
CancelOrdersOnly	Only order cancellation is allowed. New orders and modification of existing ones are prohibited.
ReducePositionOnly	Only orders that reduce the current position are allowed. Opening new positions and increasing existing ones are prohibited.
LongOnly	Only long positions are allowed. Selling is only allowed to close an existing long position (sell volume cannot exceed the current position). Opening short positions is prohibited.

Setting the Mode

```
// When creating the strategy
var strategy = new MyStrategy();
strategy.TradingMode = StrategyTradingModes.ReducePositionOnly;

// Dynamic change during operation
strategy.TradingMode = StrategyTradingModes.Disabled;
```

Mode Checking Logic

When attempting to register an order, the strategy checks the current mode:

- **Disabled** -- the order is rejected with the reason "trading is prohibited".
- **ReducePositionOnly** -- the order is rejected if the current position is zero, if the order direction matches the position direction, or if the order volume exceeds the absolute value of the position.
- **LongOnly** -- a sell order is rejected if the current position is non-positive or if the sell volume exceeds the current position.
- **Full** -- no restrictions.

- **CancelOrdersOnly** -- only order cancellation is allowed.

IsFormedAndOnlineAndAllowTrading Method

The `IsFormedAndOnlineAndAllowTrading` extension method checks that the strategy is formed (`IsFormed`), is in an online state (`IsOnline`), and the trading mode allows the required action:

```
// Check permission for full trading (default)
if (!IsFormedAndOnlineAndAllowTrading())
    return;

// Check permission for order cancellation only
if (!IsFormedAndOnlineAndAllowTrading(StrategyTradingModes.CancelOrdersOnly))
    CancelActiveOrders();

// Check permission for position reduction
if (!IsFormedAndOnlineAndAllowTrading(StrategyTradingModes.ReducePositionOnly))
    return;
```

Permission logic when called with a required parameter:

Current TradingMode \ required	Full	CancelOrdersOnly	ReducePositionOnly
Full	yes	yes	yes
Disabled	no	no	no
CancelOrdersOnly	no	yes	no
ReducePositionOnly	no	yes	yes
LongOnly	no	yes	yes

Usage Example

```
public class TradingModeStrategy : Strategy
{
    private readonly StrategyParam<DataType> _candleType;

    public DataType CandleType
    {
        get => _candleType.Value;
        set => _candleType.Value = value;
    }

    public TradingModeStrategy()
    {
        _candleType = Param(nameof(CandleType), TimeSpan.FromMinutes(5).TimeFrame());
    }

    protected override void OnStarted2(DateTime time)
    {
        base.OnStarted2(time);

        var subscription = SubscribeCandles(CandleType);

        subscription
            .Bind(ProcessCandle)
            .Start();
    }

    private void ProcessCandle(ICandleMessage candle)
    {
    }
```

```

    // Check that the strategy is ready for full trading
    if (!IsFormedAndOnlineAndAllowTrading())
        return;

    if (candle.ClosePrice > candle.OpenPrice)
    {
        BuyMarket(Volume);
    }
    else if (candle.ClosePrice < candle.OpenPrice)
    {
        SellMarket(Volume);
    }
}

// Start the strategy with a restriction -- long positions only
var strategy = new TradingModeStrategy();
strategy.TradingMode = StrategyTradingModes.LongOnly;
strategy.Start();

// Later -- switch to position closing mode
strategy.TradingMode = StrategyTradingModes.ReducePositionOnly;

// Full trading block
strategy.TradingMode = StrategyTradingModes.Disabled;

```

In this example, the strategy initially operates in LongOnly mode, which allows only purchases and closing of long positions. When market conditions change, the mode can be switched to ReducePositionOnly for gradual position closing, and then to Disabled for a complete halt of trading activity.

Advanced Features

Advanced Strategy Features

Overview

The `Strategy` class provides a number of additional properties for fine-tuning behavior: automatic order commenting, trading schedule, risk-free rate for statistics, data source for indicators, and historical period management.

CommentMode -- Order Comments

The `CommentMode` property controls automatic population of the `Order.Comment` field for all orders submitted by the strategy. This allows identifying which strategy created an order, which is especially useful when running multiple strategies on the same account simultaneously.

StrategyCommentModes Enumeration

Value	Description
Disabled	Comment is not populated automatically. Default value.
Id	The comment is set to <code>Strategy.Id</code> (unique GUID identifier).
Name	The comment is set to <code>Strategy.Name</code> (strategy name).

Example

```
public class CommentStrategy : Strategy
{
    protected override void OnStarted2(DateTime time)
    {
        base.OnStarted2(time);

        // All orders will be tagged with the strategy name
        CommentMode = StrategyCommentModes.Name;

        // Or with the identifier for exact binding
        // CommentMode = StrategyCommentModes.Id;
    }
}
```

With the `Name` value and a strategy name of "SMA Crossover", every order will receive the comment "SMA Crossover", allowing you to filter this strategy's orders in the trade journal.

WorkingTime -- Working Schedule

The `WorkingTime` property sets the schedule during which the strategy is active. Outside of the specified time intervals, the strategy can automatically restrict its activity.

```
public class ScheduledStrategy : Strategy
{
    protected override void OnStarted2(DateTime time)
    {

```

```

base.OnStarted2(time);

// Configure working time
WorkingTime = new WorkingTime
{
    Periods = new List<WorkingTimePeriod>
    {
        new WorkingTimePeriod
        {
            Till = DateTime.MaxValue,
            Times = new List<Range<TimeSpan>>
            {
                // Trade from 10:00 to 18:00
                new Range<TimeSpan>(
                    TimeSpan.FromHours(10),
                    TimeSpan.FromHours(18))
            }
        }
    }
};
}

```

The `TotalWorkingTime` property (read-only) shows the total working time of the strategy since it started. It is automatically calculated on strategy stop and restart.

RiskFreeRate -- Risk-Free Rate

The `RiskFreeRate` property sets the annual risk-free rate used in statistical calculations -- primarily the Sharpe ratio and the Sortino ratio.

```

var strategy = new MyStrategy();

// Risk-free rate of 5% per annum
strategy.RiskFreeRate = 0.05m;

```

The value is automatically passed to all statistic parameters implementing `IRiskFreeRateStatisticParameter` when the strategy's statistics manager is initialized.

IndicatorSource -- Indicator Data Source

The `IndicatorSource` property sets the default value for the `IIndicator.Source` property for all strategy indicators that do not have an explicitly specified source. It defines which `Level1Fields` field to use as indicator input data.

```

var strategy = new MyStrategy();

// All indicators will use the last trade price by default
strategy.IndicatorSource = Level1Fields.LastTradePrice;

// Or the average price
// strategy.IndicatorSource = Level1Fields.AveragePrice;

```

If the property is `null` (the default value), indicators use their own data source.

HistoryCalculated -- Computed Historical Period

The virtual property `HistoryCalculated` allows a strategy to programmatically determine the required historical data period for indicator warm-up. It returns `TimeSpan?` -- the duration of the historical period, or `null` if no period is specified.

```

public class SmaCrossStrategy : Strategy
{
    private readonly StrategyParam<int> _longPeriod;

    public int LongPeriod
    {
        get => _longPeriod.Value;
        set => _longPeriod.Value = value;
    }

    public SmaCrossStrategy()
    {
        _longPeriod = Param(nameof(LongPeriod), 50);
    }

    // Automatic calculation of the required historical period
    protected override TimeSpan? HistoryCalculated
        => TimeSpan.FromDays(LongPeriod * 2);
}

```

HistoryCalculated is the code-computed version of the HistorySize property. The difference is that HistorySize is set by the user as a strategy parameter, while HistoryCalculated is computed programmatically based on strategy parameters (for example, indicator periods).

Example: Strategy with All Advanced Settings

```

public class AdvancedStrategy : Strategy
{
    private readonly StrategyParam<DataType> _candleType;
    private readonly StrategyParam<int> _smaPeriod;

    public DataType CandleType
    {
        get => _candleType.Value;
        set => _candleType.Value = value;
    }

    public int SmaPeriod
    {
        get => _smaPeriod.Value;
        set => _smaPeriod.Value = value;
    }

    public AdvancedStrategy()
    {
        _candleType = Param(nameof(CandleType), TimeSpan.FromMinutes(5).TimeFrame());
        _smaPeriod = Param(nameof(SmaPeriod), 20);
    }

    // Automatic historical period calculation
    protected override TimeSpan? HistoryCalculated
        => TimeSpan.FromDays(SmaPeriod * 2);

    protected override void OnStarted2(DateTime time)
    {
        base.OnStarted2(time);

        // Order comments -- strategy name
        CommentMode = StrategyCommentModes.Name;

        // Risk-free rate for Sharpe calculation
        RiskFreeRate = 0.05m;

        // Data source for indicators
        IndicatorSource = Level1Fields.LastTradePrice;
    }
}

```

```

        var subscription = SubscribeCandles(CandleType);

        subscription
            .Bind(ProcessCandle)
            .Start();
    }

    private void ProcessCandle(ICandleMessage candle)
    {
        if (!IsFormedAndOnlineAndAllowTrading())
            return;

        // Trading logic...
    }
}

```

In this example, the strategy uses all the described features: it automatically comments orders, sets the risk-free rate for statistics, establishes the data source for indicators, and calculates the required historical period.

Statistics Reference

Statistics Reference

The [StatisticManager](#) manages a collection of [IStatisticParameter](#) instances. Each parameter tracks a specific metric during strategy execution. All available parameters are created using the [StatisticParameterRegistry](#).

For a general overview of working with strategy statistics, see the [Statistics](#) section.

Interfaces

The statistics system is built on a hierarchy of interfaces. Each interface defines the data source for calculating the parameter:

Interface	Description
IStatisticParameter	Base interface: properties Name, Type, Value, DisplayName, Description, Category, Order; method Reset()
IPnLStatisticParameter	Parameters based on profit/loss: method Add(marketTime, pnl, commission)
ITradeStatisticParameter	Parameters based on trades: method Add(PnLInfo)
IOrderStatisticParameter	Parameters based on orders: methods New(order), Changed(order), RegisterFailed(fail), CancelFailed(fail)
IPositionStatisticParameter	Parameters based on positions: method Add(marketTime, position)
IRiskFreeRateStatisticParameter	Parameters with risk-free rate: property RiskFreeRate
IBeginValueStatisticParameter	Parameters with initial value: property BeginValue

Profit and Loss (P&L) Parameters

All parameters in this group implement the [IPnLStatisticParameter](#) interface and inherit from [BasePnLStatisticParameter](#). They receive data on each update of the strategy's P&L value.

Class	Description	Value Type
NetProfitParameter	Net profit for the entire period. Set equal to the current P&L value	decimal
NetProfitPercentParameter	Net profit as a percentage. Requires setting BeginValue (initial capital). Formula: $pnl * 100 / \text{BeginValue}$	decimal
MaxProfitParameter	Maximum profit (highest P&L value over the entire period)	decimal

MaxProfitPercentParameter	Maximum profit as a percentage. Requires BeginValue. Formula: $\text{MaxProfit} * 100 / \text{BeginValue}$	decimal
MaxProfitDateParameter	Date when maximum profit was reached	DateTime
MaxDrawdownParameter	Maximum absolute drawdown. Difference between the peak and trough of the equity curve	decimal
MaxDrawdownPercentParameter	Maximum drawdown as a percentage. Formula: $\text{MaxDrawdown} * 100 / \text{MaxEquity}$	decimal
MaxDrawdownDateParameter	Date of maximum drawdown	DateTime
MaxRelativeDrawdownParameter	Maximum relative drawdown. Calculated as the ratio of drawdown to peak equity value	decimal
ReturnParameter	Relative return for the entire period. Maximum growth from trough to current value in relative terms	decimal
CommissionParameter	Total commission paid. Accumulates all commission values	decimal
AverageDrawdownParameter	Average drawdown. Arithmetic mean of all completed and current drawdowns	decimal
RecoveryFactorParameter	Recovery factor. Formula: $\text{NetProfit} / \text{MaxDrawdown}$	decimal
SharpeRatioParameter	Sharpe ratio. Formula: $(\text{annualized return} - \text{risk-free rate}) / \text{annualized standard deviation}$	decimal
SortinoRatioParameter	Sortino ratio. Similar to Sharpe, but only accounts for downside deviations	decimal
CalmarRatioParameter	Calmar ratio. Formula: $\text{NetProfit} / \text{MaxDrawdown}$	decimal
SterlingRatioParameter	Sterling ratio. Formula: $\text{NetProfit} / \text{AverageDrawdown}$	decimal

Risk Coefficients

The [SharpeRatioParameter](#) and [SortinoRatioParameter](#) inherit from the base class [RiskAdjustedRatioParameter](#) and implement the [IRiskFreeRateStatisticParameter](#) interface.

They support the following settings:

- **RiskFreeRate** -- annual risk-free rate (e.g., 0.03m = 3%)

- **Period** -- return calculation period (default `TimeSpan.FromDays(1)`)

The [CalmarRatioParameter](#) and [SterlingRatioParameter](#) depend on other parameters ([NetProfitParameter](#), [MaxDrawdownParameter](#), [AverageDrawdownParameter](#)) and are automatically linked when created via [StatisticParameterRegistry](#).

Trade Parameters

All parameters in this group implement the [ITradeStatisticParameter](#) interface. They receive data through the [PnLInfo](#) object for each trade executed.

Class	Description	Value Type
TradeCountParameter	Total number of trades (only trades with <code>ClosedVolume > 0</code> are counted)	int
WinningTradesParameter	Number of profitable trades (<code>ClosedVolume > 0</code> and <code>PnL > 0</code>)	int
LosingTradesParameter	Number of losing trades (<code>ClosedVolume > 0</code> and <code>PnL < 0</code>)	int
RoundtripCountParameter	Number of completed round-trips (closing trades with <code>ClosedVolume > 0</code>)	int
AverageTradeProfitParameter	Average profit per trade. Formula: $\text{SumPnL} / \text{Count}$	decimal
AverageWinTradeParameter	Average profit of profitable trades. Only trades with <code>PnL > 0</code> are considered	decimal
AverageLossTradeParameter	Average loss of losing trades. Only trades with <code>PnL < 0</code> are considered	decimal
ProfitFactorParameter	Profit factor. Formula: $\text{GrossProfit} / \text{GrossLoss}$	decimal
ExpectancyParameter	Mathematical expectancy. Formula: $P(\text{win}) * \text{AvgWin} + P(\text{loss}) * \text{AvgLoss}$	decimal
PerMonthTradeParameter	Average number of trades per month	decimal
PerDayTradeParameter	Average number of trades per day	decimal
GrossProfitParameter	Gross profit. Sum of P&L of all profitable trades (<code>PnL > 0</code>)	decimal
GrossLossParameter	Gross loss. Sum of P&L of all losing trades (<code>PnL < 0</code> , value is negative)	decimal

Position Parameters

Parameters in this group implement the [IPositionStatisticParameter](#) interface. They receive data on each position change.

Class	Description	Value Type
MaxLongPositionParameter	Maximum long position. Highest positive position value	decimal
MaxShortPositionParameter	Maximum short position. Highest absolute negative position value	decimal

Order Parameters

All parameters in this group implement the [IOrderStatisticParameter](#) interface and inherit from [BaseOrderStatisticParameter](#). They receive data on order registration, changes, and errors.

Class	Description	Value Type
OrderCountParameter	Total number of registered orders	int
OrderRegisterErrorCountParameter	Number of order registration errors	int
OrderInsufficientFundErrorCountParameter	Number of "insufficient funds" errors (type InsufficientFundException)	int
OrderCancelErrorCountParameter	Number of order cancellation errors	int

Latency Parameters

Latency parameters also implement [IOrderStatisticParameter](#), but track the time characteristics of order processing.

Class	Description	Value Type
MaxLatencyRegistrationParameter	Maximum order registration latency (Order.LatencyRegistration property)	TimeSpan
MinLatencyRegistrationParameter	Minimum order registration latency	TimeSpan
MaxLatencyCancellationParameter	Maximum order cancellation latency (Order.LatencyCancellation property)	TimeSpan
MinLatencyCancellationParameter	Minimum order cancellation latency	TimeSpan

Usage

Accessing Strategy Statistics

```
var strategy = new MyStrategy();
```



```
// Access statistics after execution
foreach (var param in strategy.StatisticManager.Parameters)
{
    Console.WriteLine($"{param.DisplayName}: {param.Value}");
}
```

Retrieving a Specific Parameter

```
// Get the net profit value
var netProfit = strategy.StatisticManager.Parameters
    .OfType<NetProfitParameter>()
    .First();

Console.WriteLine($"Net profit: {netProfit.Value}");
```

Configuring the Risk-Free Rate for Coefficients

The Sharpe and Sortino ratios require setting the risk-free rate for correct calculation:

```
// Set a 3% risk-free rate for all coefficients
foreach (var param in strategy.StatisticManager.Parameters
    .OfType<IRiskFreeRateStatisticParameter>())
{
    param.RiskFreeRate = 0.03m;
}
```

Configuring Initial Capital for Percentage Parameters

The [NetProfitPercentParameter](#) and [MaxProfitPercentParameter](#) parameters require setting the initial capital value:

```
// Set initial capital for percentage calculations
foreach (var param in strategy.StatisticManager.Parameters
    .OfType<IBeginValueStatisticParameter>())
{
    param.BeginValue = 1_000_000m; // 1,000,000
}
```

Resetting Statistics

```
// Reset all statistic parameters
strategy.StatisticManager.Reset();
```

Saving and Loading State

All parameters support serialization via the `IPersistable` interface:

```
// Saving
var storage = new SettingsStorage();
strategy.StatisticManager.Save(storage);

// Loading
strategy.StatisticManager.Load(storage);
```

See Also

- [Strategy Statistics](#)

- [Statistics Graphical Component](#)

Advanced Optimization

Overview

StockSharp provides advanced tools for fine-tuning the strategy optimization process. This section describes advanced components: genetic algorithm configuration via [GeneticSettings](#), custom fitness formula compilation via [FitnessFormulaProvider](#), and progress monitoring via [OptimizationProgressTracker](#).

GeneticSettings

The [GeneticSettings](#) class manages all aspects of the genetic optimization algorithm. It is located in the `StockSharp.Algo.Strategies.Optimization` namespace.

Population and Generation Parameters

Property	Type	Default	Description
Population	int	8	Initial population size
PopulationMax	int	16	Maximum population size
GenerationsMax	int	20	Maximum number of generations
GenerationsStagnation	int	5	Stop on stagnation (generations without improvement)

Probability Parameters

Property	Type	Default	Description
MutationProbability	decimal	0.1	Mutation probability (0-1)
CrossoverProbability	decimal	0.8	Crossover probability (0-1)

Genetic Operators

Property	Default	Description
Reinsertion	ElitistReinsertion	Generation replacement strategy
Mutation	UniformMutation	Mutation operator
Crossover	OnePointCrossover	Crossover operator
Selection	TournamentSelection	Selection operator

FitnessFormulaProvider

The [FitnessFormulaProvider](#) class compiles C# string expressions into `Func<Strategy, decimal>` functions used for evaluating strategies during optimization.

Compile Method

```
var fitnessProvider = new FitnessFormulaProvider();  
Func<Strategy, decimal> fitness = fitnessProvider.Compile("PnL / (MaxDD + 1)");
```

Available Variables

The following variables are available in fitness function formulas, corresponding to strategy statistical indicators:

Variable	Description
PnL	Net profit
WinTrades	Winning trades
LosTrades	Losing trades
TCount	Total trades
RTrip	Round-trips
AvgTPnL	Average profit per trade
AvgWTrades	Average winning trade
AvgLTrades	Average losing trade
MaxLong	Max long position
MaxShort	Max short position
MaxPnL	Max profit
MaxDD	Max drawdown
MaxRelDD	Max relative drawdown
Ret	Return
Recovery	Recovery factor
MaxLatReg	Max registration latency
MaxLatCan	Max cancellation latency
MinLatReg	Min registration latency
MinLatCan	Min cancellation latency
OrdCount	Order count
OrdRegErrCount	Registration errors
OrdCancelErrCount	Cancellation errors
OrdFundErrCount	Insufficient funds errors

Variables can be combined in arbitrary mathematical expressions: "PnL - MaxDD * 2", "Recovery * WinTrades", "PnL / (MaxDD + 1)".

OptimizationProgressTracker

The [OptimizationProgressTracker](#) class provides convenient monitoring of the optimization process.

Properties

Property	Type	Description
TotalIterations	int	Total number of iterations
CompletedIterations	int	Number of completed iterations
TotalProgress	double	Overall progress (0-100)
StartedAt	DateTimeOffset	Optimization start time
Elapsed	TimeSpan	Elapsed time
Remaining	TimeSpan	Estimated remaining time

Methods

Method	Description
IterationCompleted()	Marks one iteration as completed
Reset(totalIterations)	Resets the tracker for a new run

Optimizer Classes

StockSharp provides two optimizers that inherit from BaseOptimizer:

- [BruteForceOptimizer](#) -- exhaustive enumeration of all parameter combinations. Suitable for small parameter spaces.
- [GeneticOptimizer](#) -- genetic algorithm. Efficient with a large number of parameters, automatically converges to the optimal solution.

Usage Example

```
var geneticSettings = new GeneticSettings
{
    Population = 16,
    PopulationMax = 32,
    GenerationsMax = 50,
    GenerationsStagnation = 10,
    MutationProbability = 0.15m,
    CrossoverProbability = 0.75m,
};

// Custom fitness function formula
var fitnessProvider = new FitnessFormulaProvider();
var fitness = fitnessProvider.Compile("PnL / (MaxDD + 1)");

// Create the genetic optimizer
var optimizer = new GeneticOptimizer(
    new CollectionSecurityProvider(new[] { security }),
    new CollectionPortfolioProvider(new[] { portfolio }),
    storageRegistry,
    Paths.FileSystem);

optimizer.Settings.Population = geneticSettings.Population;
optimizer.Settings.PopulationMax = geneticSettings.PopulationMax;
optimizer.Settings.GenerationsMax = geneticSettings.GenerationsMax;
optimizer.Settings.GenerationsStagnation = geneticSettings.GenerationsStagnation;
optimizer.Settings.MutationProbability = geneticSettings.MutationProbability;
optimizer.Settings.CrossoverProbability = geneticSettings.CrossoverProbability;
```

```
// Progress monitoring
var tracker = new OptimizationProgressTracker();
tracker.Reset(totalIterations: 100);

optimizer.SingleProgressChanged += (strategy, parameters, progress) =>
{
    if (progress == 100)
    {
        tracker.IterationCompleted();
        Console.WriteLine($"Progress: {tracker.TotalProgress:F1}%, " +
            $"Remaining: {tracker.Remaining:hh\\:mm\\:ss}");
    }
};
```

See Also

- [Strategy Optimization](#)
- [Strategy Statistics](#)
- [Genetic Optimization in Designer](#)

Quoting Reference

Quoting Reference

Overview

This section contains a complete reference of the quoting system architecture and components in StockSharp. All available quoting behaviors, action types, and interfaces are described. For a basic introduction, see [Quoting Algorithm](#).

Architecture

QuotingStrategy (Deprecated)

The [QuotingStrategy](#) class is deprecated. It is recommended to use [QuotingProcessor](#) instead.

Main parameters of the deprecated class:

- `QuotingSide` -- quoting direction (Buy/Sell)
- `QuotingVolume` -- quoting volume
- `TimeOut` -- execution timeout
- `UseBidAsk` -- use order book prices
- `UseLastTradePrice` -- use last trade price

QuotingEngine

[QuotingEngine](#) -- the functional core of the quoting system. It calculates remaining volume and timeouts, returning action recommendations without side effects.

QuotingBehaviorAlgo

[QuotingBehaviorAlgo](#) -- implementation of the `IPositionModifyAlgo` interface for algorithmic position management.

Main methods:

Method	Description
<code>UpdateMarketData(time, price, volume)</code>	Update market data
<code>UpdateOrderBook(depth)</code>	Update order book
<code>GetNextAction()</code>	Get the next quoting action

Supports VWAP and TWAP modes.

QuotingProcessor

[QuotingProcessor](#) -- the main processor for executing quoting within a strategy. Manages the order lifecycle: placement, modification, cancellation.

QuotingAction Types

The processor returns actions of type [QuotingAction](#):

Type	Description
None(reason)	No action required
PlaceOrder(price, volume)	Place a new order
ModifyOrder(price, volume)	Modify an existing order
CancelOrder()	Cancel the order
Finish(success, reason)	Finish quoting

Quoting Behaviors

StockSharp provides 10 types of quoting behaviors, each implementing the [IQuotingBehavior](#) interface:

#	Class	Description	Key Parameters
1	BestByPriceQuotingBehavior	Best price from order book	BestPriceOffset
2	LastTradeQuotingBehavior	Last trade price	BestPriceOffset
3	LimitQuotingBehavior	Fixed limit price	LimitPrice
4	MarketQuotingBehavior	Market price with offset	PriceOffset, PriceType (Following/Opposite/Middle)
5	VolatilityQuotingBehavior	Option volatility (Black-Scholes)	IVRange, Model
6	TheorPriceQuotingBehavior	Option theoretical price	TheorPriceOffset
7	BestByVolumeQuotingBehavior	Cumulative volume in order book	VolumeExchange
8	LevelQuotingBehavior	Order book depth level	Level (Range<int>), OwnLevel
9	VWAPQuotingBehavior	VWAP -- volume-weighted average	BestPriceOffset
10	TWAPQuotingBehavior	TWAP -- time-weighted average	TimeInterval, PriceBufferSize (default 10)

IQuotingBehavior Interface

The [IQuotingBehavior](#) interface defines two key methods:

CalculateBestPrice

Calculates the best price for placing an order:

```
decimal? CalculateBestPrice(
    Security security,
    IMarketDataProvider provider,
    Sides quotingDirection,
    decimal? bestBid,
```



```
decimal? bestAsk,
decimal? lastTradePrice,
decimal? lastTradeVolume,
IEnumerable<QuoteChange> bids,
IEnumerable<QuoteChange> asks);
```

NeedQuoting

Determines whether the current order needs to be updated:

```
decimal? NeedQuoting(
    Security security,
    IMarketDataProvider provider,
    DateTimeOffset currentTime,
    decimal? currentPrice,
    decimal? currentVolume,
    decimal? newVolume,
    decimal? bestPrice);
```

Returns null if no update is needed, or the new price for placing the order.

Usage Examples

Market Quoting with Offset

```
var behavior = new MarketQuotingBehavior
{
    PriceOffset = new Unit(2, UnitTypes.Absolute),
    PriceType = MarketPriceTypes.Following,
    BestPriceOffset = new Unit(0.5m),
};
```

VWAP Quoting

```
var behavior = new VWAPQuotingBehavior
{
    BestPriceOffset = new Unit(1),
};
```

Option Volatility Quoting

```
var behavior = new VolatilityQuotingBehavior
{
    IVRange = new Range<decimal>(0.2m, 0.3m),
    Model = new BlackScholes(option, connector),
};
```

Full Example with Processor

```
// Choose the quoting behavior
var behavior = new BestByPriceQuotingBehavior
{
    BestPriceOffset = new Unit(0.01m),
};

// Create the processor
var processor = new QuotingProcessor(
    behavior,
    Security,
```

```

Portfolio,
Sides.Buy,
volume: 10,
maxOrderVolume: 10,
timeout: TimeSpan.FromMinutes(5),
subscriptionProvider: this,
ruleContainer: this,
transactionProvider: this,
timeProvider: this,
marketDataProvider: this,
isTradingAllowed: IsFormedAndOnlineAndAllowTrading,
useBidAsk: true,
useLastTradePrice: true)
{
    Parent = this,
};

processor.Finished += isOk =>
{
    this.AddInfoLog($"Quoting finished: {isOk}");
    processor?.Dispose();
};

processor.Start();

```

See Also

- [Quoting Algorithm](#)
- [Quoting Strategy](#)
- [Volatility Quoting](#)

Event model

Event Model

The event model is built using [IMarketRule](#). [IMarketRule](#) can be used both within and outside [strategies](#).

In [API](#), there are several predefined conditions and actions for [IMarketRule](#) tailored for common scenarios. These are added as extension methods in [MarketRuleHelper](#).

- [Using rules](#)
- [Rule synchronization and suspension](#)
- [Mutually exclusive rules](#)
- [Creating custom rules](#)

Rules using

Using Rules

Creating Rules

- **Creating a rule for order registration condition:**

```
private void btnBuy_Click(object sender, RoutedEventArgs e)
{
    var order = new Order
    {
        Portfolio = Portfolio.SelectedPortfolio,
        Price = _instr1.BestAsk.Price,
        Security = _instr1,
        Volume = 1,
        Direction = Sides.Buy,
    };
    order
        .WhenRegistered(Connector)
        .Do(() => Connector.AddInfoLog("Order successfully registered"))
        .Once()
        .Apply(this);

    // order registration
    Connector.RegisterOrder(order);
}
```

Now, when the event fires (the order is registered on the exchange), the action specified through the [IMarketRule.Do\(System.Action action\)](#) method will be called.

At the end of rule formation, the [MarketRuleHelper.Apply\(StockSharp.Algo.IMarketRule rule\)](#) method is called. Until this method is called for the rule, it is inactive (the handler in [IMarketRule.Do\(System.Action action\)](#) will not be called).

- **Creating rules within a strategy:**

```
class FirstStrategy : Strategy
{
    protected override void OnStarted2(DateTime time)
    {
        // Subscription to candles
        var candleSubscription = new Subscription(TimeSpan.FromMinutes(5).TimeFrame(), Security);
        this
            .WhenCandlesStarted(candleSubscription)
            .Do(ProcessCandle)
            .Apply(this);

        // Subscription to tick trades
        var tickSubscription = new Subscription(DataType.Ticks, Security);
        tickSubscription
            .WhenTickTradeReceived(this)
            .Do(ProcessTick)
            .Apply(this);

        // Send subscription requests
        Subscribe(candleSubscription);
        Subscribe(tickSubscription);

        base.OnStarted2(time);
    }

    // Methods for event processing
    private void ProcessCandle(ICandleMessage candle) { /* ... */ }
}
```

```
private void ProcessTick(ITickTradeMessage tick) { /* ... */ }
}
```

- **Removing unnecessary rules.**

[IMarketRule](#) has [IMarketRule.Token](#) - a token of the rule with which it is associated. For example, for the rule `[WhenCanceled](xref:StockSharp.Algo.MarketRuleHelper.WhenCanceled(StockSharp.BusinessEntities.Order,StockSharp.BusinessEntities.ISubscriptionProvider))`, the token will be the order.

When a rule for successful order cancellation has been triggered, it's better to remove all other rules related to this order:

```
var order = this.CreateOrder(direction, (decimal)Security.GetCurrentPrice(direction), Volume);
var ruleCanceled = order.WhenCanceled(Connector);
ruleCanceled
    .Do(() =>
    {
        this.AddInfoLog("Order successfully canceled");
        // removing all rules associated with order
        Rules.RemoveRulesByToken(ruleCanceled, (IMarketRule)ruleCanceled.Token);
    })
    .Once()
    .Apply(this);
order
    .WhenRegistered(Connector)
    .Do(() => this.AddInfoLog("Order successfully registered"))
    .Once()
    .Apply(this);
order
    .WhenRegisterFailed(Connector)
    .Do(() => this.AddInfoLog("Order not accepted by the exchange"))
    .Once()
    .Apply(this);
order
    .WhenMatched(Connector)
    .Do(() => this.AddInfoLog("Order fully executed"))
    .Once()
    .Apply(this);
// order registration
RegisterOrder(order);
```

- **Combining rules with condition [MarketRuleHelper.Or\(StockSharp.Algo.IMarketRule rule, StockSharp.Algo.IMarketRule\[\] rules\)](#) / [MarketRuleHelper.And\(StockSharp.Algo.IMarketRule rule, StockSharp.Algo.IMarketRule\[\] rules\)](#).**

When time expires **OR** a candle closes:

```
// Create a subscription to candles
var subscription = new Subscription(TimeSpan.FromMinutes(5).TimeFrame(), Security);
var timeInterval = TimeSpan.FromMilliseconds(5000);

Connector
    .WhenIntervalElapsed(timeInterval)
    .Or(this.WhenCandlesStarted(subscription))
    .Do(() => this.AddInfoLog("Candle closed or time expired"))
    .Once()
    .Apply(this);

// Send subscription request
Subscribe(subscription);
```

Or this format:

```
// Create a subscription to candles
var subscription = new Subscription(TimeSpan.FromMinutes(5).TimeFrame(), Security);
var timeInterval = TimeSpan.FromMilliseconds(5000);
```

```

MarketRuleHelper
    .Or(new IMarketRule[] {
        Connector.WhenIntervalElapsed(timeInterval),
        this.WhenCandlesStarted(subscription)
    })
    .Do(() => this.AddInfoLog("Candle closed or time expired"))
    .Once()
    .Apply(this);

// Send subscription request
Subscribe(subscription);

```

When the last trade price is above 135000 **AND** below 140000:

```

// Create a subscription to tick trades
var subscription = new Subscription(DataType.Ticks, Security);
var priceMore = new Unit(135000m, UnitTypes.Limit);
var priceLess = new Unit(140000m, UnitTypes.Limit);

MarketRuleHelper
    .And(new IMarketRule[] {
        subscription.WhenLastTradePriceMore(this, 135000m),
        subscription.WhenLastTradePriceLess(this, 140000m)
    })
    .Do(() => this.AddInfoLog($"Last trade price is in the range from {priceMore} to {priceLess}"))
    .Apply(this);

// Send subscription request
Subscribe(subscription);

```

[!TIP] The handler in `IMarketRule.Do(System.Action action)` will be called after the last rule added through `MarketRuleHelper.And(StockSharp.Algo.IMarketRule rule, StockSharp.Algo.IMarketRule[] rules)` is triggered.

- Rule operation periodicity - `IMarketRule.Until(System.Func<System.Boolean> canFinish)`:

```

bool flag = false;

// Create a subscription to tick trades
var subscription = new Subscription(DataType.Ticks, Security);

subscription
    .WhenTickTradeReceived(this)
    .Do((tick) =>
    {
        if(condition) flag = true;
    })
    .Until(() => flag)
    .Apply(this);

// Send subscription request
Subscribe(subscription);

```

Examples of Using Rules

Rules on Candles

```

// Create a subscription to 5-minute candles
var subscription = new Subscription(TimeSpan.FromMinutes(5).TimeFrame(), Security);

// Variable for counting candles
var i = 0;
var diff = "10%".ToUnit();

```

```
// Rule that activates when a new candle starts
this.WhenCandlesStarted(subscription)
  .Do((candle) =>
  {
    i++;

    // Nested rule: check when total volume exceeds threshold
    this
      .WhenTotalVolumeMore(candle, diff)
      .Do((candle1) =>
      {
        LogInfo($"Rule WhenCandlesStarted and WhenTotalVolumeMore candle={candle1}");
        LogInfo($"Rule WhenCandlesStarted and WhenTotalVolumeMore i={i}");
      })
      .Once().Apply(this);

  }).Apply(this);

// Send subscription request
Subscribe(subscription);
```

Rules on Order Books (Market Depth)

```
// Subscription to order book data
var mdSub = new Subscription(DataType.MarketDepth, Security);

// Method 1: Creating a rule in a chain
mdSub.WhenOrderBookReceived(this).Do((depth) =>
{
  LogInfo($"Rule WhenOrderBookReceived #1 BestBid={depth.GetBestBid()},
BestAsk={depth.GetBestAsk()}");
}).Once().Apply(this);

// Method 2: First create a rule variable
var whenMarketDepthChanged = mdSub.WhenOrderBookReceived(this);

whenMarketDepthChanged.Do((depth) =>
{
  LogInfo($"Rule WhenOrderBookReceived #2 BestBid={depth.GetBestBid()},
BestAsk={depth.GetBestAsk()}");
}).Once().Apply(this);

// Rule within a rule
mdSub.WhenOrderBookReceived(this).Do((depth) =>
{
  LogInfo($"Rule WhenOrderBookReceived #3 BestBid={depth.GetBestBid()},
BestAsk={depth.GetBestAsk()}");

  // Rule without specifying Once()
  mdSub.WhenOrderBookReceived(this).Do((depth1) =>
  {
    LogInfo($"Rule WhenOrderBookReceived #4 BestBid={depth1.GetBestBid()},
BestAsk={depth1.GetBestAsk()}");
  }).Apply(this);
}).Once().Apply(this);

// Send subscription request
Subscribe(mdSub);
```

Rules with Completion Condition

```
// Subscription to order book data
var mdSub = new Subscription(DataType.MarketDepth, Security);

// Counter
```

```

var i = 0;

// Create a rule that processes order books until i reaches 10
mdSub.WhenOrderBookReceived(this).Do(depth =>
{
    i++;
    LogInfo($"Rule WhenOrderBookReceived BestBid={depth.GetBestBid()}, BestAsk={depth.GetBestAsk()}");
    LogInfo($"Rule WhenOrderBookReceived i={i}");
})
.Until(() => i >= 10)
.Apply(this);

// Send subscription request
Subscribe(mdSub);

```

Rules on Orders

```

// Subscription to tick trades
var sub = new Subscription(DataType.Ticks, Security);

// When we receive the first tick, we'll create an order
sub.WhenTickTradeReceived(this).Do(() =>
{
    var order = CreateOrder(Sides.Buy, default, 1);

    var ruleReg = order.WhenRegistered(this);
    var ruleRegFailed = order.WhenRegisterFailed(this);

    ruleReg
        .Do(() => LogInfo("Order #1 registered"))
        .Once()
        .Apply(this)
        .Exclusive(ruleRegFailed); // Rules are mutually exclusive

    ruleRegFailed
        .Do(() => LogInfo("Order #1 not registered"))
        .Once()
        .Apply(this)
        .Exclusive(ruleReg); // Rules are mutually exclusive

    RegisterOrder(order);
}).Once().Apply(this);

// Send subscription request
Subscribe(sub);

```

Rules on Price Changes

```

// Subscription to tick trades
var sub = new Subscription(DataType.Ticks, Security);

// Rule activates on the first tick and creates another rule
sub.WhenTickTradeReceived(this).Do(t =>
{
    // Create a rule that activates when the price moves 2 points in any direction
    sub
        .WhenLastTradePriceMore(this, t.Price + 2)
        .Or(sub.WhenLastTradePriceLess(this, t.Price - 2))
        .Do(t =>
        {
            LogInfo($"Rule WhenLastTradePriceMore or WhenLastTradePriceLess triggered: tick={t}");
        })
        .Apply(this);
})
.Once() // call this rule only once

```



```
.Apply(this);
```

```
// Send subscription request  
Subscribe(sub);
```

Rules suspension

Rules suspension

Sometimes you want to set multiple rules in suspension mode (prevent them from triggering until code initialization will be finished). To do this, the [MarketRuleHelper.SuspendRules\(System.Action action\)](#) method is used.

Using rule suspension

- Out of the strategy:

```
MarketRuleHelper.SuspendRules(() =>
{
    order
        .WhenRegistered(Connector)
        .Do(() => Connector.AddInfoLog("The order was successfully registered."))
        .Once()
        .Apply(this);

    order
        .WhenCanceled(Connector)
        .Do(() => Connector.AddInfoLog("The order was successfully cancelled."))
        .Once()
        .Apply(this);
});
```

- Inside the strategy:

```
class FirstStrategy : Strategy
{
    ...

    this.SuspendRules(() =>
    {
        _connector
            .WhenCandlesFinished(_series)
            .Do(FinishCandle)
            .Apply(this);
        Security
            .WhenNewTrade(Connector)
            .Do(NewTrade)
            .Apply(this);
    });
    ...
}
```

Mutually exclusive rules

Mutually exclusive rules

Mutually exclusive rules are rules which are removed after the activation of one of these rules. To do this, the [MarketRuleHelper.Exclusive](#)([StockSharp.Algo.IMarketRule](#) rule1, [StockSharp.Algo.IMarketRule](#) rule2) method is called, and the rule is passed to it. This rule will be removed after its activation.

For example, two rules are registered: one on the successful order registration and another on the unsuccessful order registration. One of them must be removed in case of activation of another:

```
var order = this.CreateOrder(direction, (decimal) Security.GetCurrentPrice(direction), Volume);
var ruleReg = order.WhenRegistered();
var ruleRegFailed = order.WhenRegisterFailed();
ruleReg
    .Do(() => this.AddInfoLog("The order was successfully registered. "))
    .Once()
    .Apply(this)
    .Exclusive(ruleRegFailed);
ruleRegFailed
    .Do(() => this.AddInfoLog("The order was not accepted by broker. "))
    .Once()
    .Apply(this)
    .Exclusive(ruleReg);
// registering the order
RegisterOrder(order);
```

Also mutually exclusive rules can be created through the adding to [IMarketRule.ExclusiveRules](#):

```
var order = this.CreateOrder(direction, (decimal) Security.GetCurrentPrice(direction), Volume);
var ruleReg = order.WhenRegistered(Connector);
var ruleRegFailed = order.WhenRegisterFailed(Connector);
ruleReg.ExclusiveRules.Add(ruleRegFailed);
ruleRegFailed.ExclusiveRules.Add(ruleReg);
ruleReg
    .Do(() => this.AddInfoLog("The order was successfully registered. "))
    .Once()
    .Apply(this);
ruleRegFailed
    .Do(() => this.AddInfoLog("The order was not accepted by broker. "))
    .Once()
    .Apply(this);
// registering the order
RegisterOrder(order);
```

Create own rule

Create own rule

If you need to create your own unique rule (on any event, which is not provided as standard), you must create the derived [MarketRule<TToken,TArg>](#) class, which will work with a prerequisite. Below is the [MarketRuleHelper.WhenMoneyMore](#)([StockSharp.BusinessEntities.Portfolio](#) portfolio, [StockSharp.BusinessEntities.IPortfolioProvider](#) provider, [StockSharp.Messages.Unit](#) money) method implementation:

```
private sealed class PortfolioRule : MarketRule<Portfolio, Portfolio>
{
    private readonly Func<Portfolio, bool> _changed;
    private readonly Portfolio _portfolio;
    private readonly IConnector _connector;
    public PortfolioRule(Portfolio portfolio, IConnector connector, Func<Portfolio, bool> changed) :
    base(portfolio)
    {
        if (portfolio == null)
            throw new ArgumentNullException("portfolio");
        if (changed == null)
            throw new ArgumentNullException("changed");
        _changed = changed;
        _portfolio = portfolio;
        _connector = connector;
        _connector.PortfolioChanged += OnPortfolioChanged;
    }
    private void OnPortfolioChanged(Portfolio portfolio)
    {
        if ((portfolio==_portfolio) && _changed(_portfolio))
            Activate(_portfolio);
    }
    protected override void DisposeManaged()
    {
        _connector.PortfolioChanged -= OnPortfolioChanged;
        base.DisposeManaged();
    }
}

public static MarketRule<Portfolio, Portfolio> WhenMoneyMore(this Portfolio portfolio, Unit money)
{
    if (portfolio == null)
        throw new ArgumentNullException("portfolio");
    if (money == null)
        throw new ArgumentNullException("money");
    var finishMoney = money.Type == UnitTypes.Limit ? money : portfolio.CurrentValue + money;
    return new PortfolioRule(portfolio, pf => pf.CurrentValue > finishMoney)
    {
        Name = "Money increase of portfolio {0} above {1}".Put(portfolio, finishMoney)
    };
}
```

The *PortfolioRule* rule subscribes to the [IPortfolioProvider.PortfolioChanged](#) event, and as soon as it is called, then the condition is checked to exceed the current level of money in the portfolio above a specified limit. If the condition returns **true**, then the rule is activated through the [MarketRule<TToken,TArg>.Activate](#) method.

Candle rule

Rule for a Single Candle

Overview

SimpleCandleRulesStrategy is a strategy that demonstrates the use of rules for candles in StockSharp. It tracks candle volumes and logs information when certain conditions are met.

Main Components

```
// Main components
public class SimpleCandleRulesStrategy : Strategy
{
}
```

OnStarted Method

Called when the strategy starts:

- Initializes a subscription to 5-minute candles
- Establishes rules for processing candles

```
// OnStarted method
protected override void OnStarted2(DateTime time)
{
    var subscription = new Subscription(TimeSpan.FromMinutes(5).TimeFrame(), Security)
    {
        // ready-to-use candles much faster than compression on fly mode
        // turn off compression to boost optimizer (!!! make sure you have candles)

        //MarketData =
        //{
        //    BuildMode = MarketDataBuildModes.Build,
        //    BuildFrom = Data.Type.Ticks,
        //}
    };
    Subscribe(subscription);

    var i = 0;
    var diff = "10%".ToUnit();

    this.WhenCandlesStarted(subscription)
        .Do((candle) =>
        {
            i++;

            this
                .WhenTotalVolumeMore(candle, diff)
                .Do((candle1) =>
                {
                    LogInfo($"The rule WhenCandlesStarted and WhenTotalVolumeMore candle={candle1}");
                    LogInfo($"The rule WhenCandlesStarted and WhenTotalVolumeMore i={i}");
                })
                .Once().Apply(this);

        }).Apply(this);

    base.OnStarted2(time);
}
```

Logic

- The strategy subscribes to 5-minute candles
- When each candle begins forming, a rule is established
- The rule triggers when the total volume of the candle exceeds 10% (using a percentage value)
- When the rule is triggered, information about the candle and counter is added to the log
- After the first trigger, the rule stops working thanks to the `Once()` method

Features

- Demonstrates the use of the `WhenCandlesStarted` and `WhenTotalVolumeMore` rules
- Uses the candle subscription mechanism
- Shows an example of creating a percentage value through `"10%".ToUnit()`
- Shows an example of logging information in a strategy using the `LogInfo` method
- Contains commented code for configuring candle building from ticks

Ticks rule

Rules for Tick Trades

Overview

SimpleTradeRulesStrategy is a strategy that demonstrates the use of combined rules for analyzing trade prices in StockSharp. It subscribes to trades and creates a rule that triggers under certain price conditions.

Main Components

```
// Main components
public class SimpleTradeRulesStrategy : Strategy
{
}
```

OnStarted Method

Called when the strategy starts:

- Creates a subscription to ticks
- Creates a combined rule for analyzing trade prices

```
// OnStarted method
protected override void OnStarted2(DateTime time)
{
    var sub = new Subscription(DataType.Ticks, Security);

    sub.WhenTickTradeReceived(this).Do(t =>
    {
        sub
            .WhenLastTradePriceMore(this, t.Price + 2)
            .Or(sub.WhenLastTradePriceLess(this, t.Price - 2))
            .Do(t =>
            {
                LogInfo($"The rule WhenLastTradePriceMore Or WhenLastTradePriceLess tick={t}");
            })
            .Apply(this);
    })
    .Once() // call this rule only once
    .Apply(this);

    // Sending request for subscribe to market data.
    Subscribe(sub);

    base.OnStarted2(time);
}
```

Logic

- When the first tick is received, a combined rule is created
- It is based on the price of the received tick: creates a rule that triggers when the price changes by +/- 2
- The rule triggers when the price of the last trade becomes more than current + 2 or less than current - 2

- When the rule triggers, information about the tick is added to the log
- The outer rule triggers only once (`Once()`)

Features

- Demonstrates creating combined rules using `Or()`
- Uses `WhenLastTradePriceMore` and `WhenLastTradePriceLess` for price analysis
- Shows an example of logging information about trades using the `LogInfo` method
- Illustrates the use of `Once()` to limit rule triggering
- Passes the tick parameter to the event handler (unlike the example in the documentation)

Market depth and ticks rule

Rules for Order Books and Trades

Overview

SimpleRulesStrategy is a strategy that demonstrates various ways to create and apply rules in StockSharp. It subscribes to trades and order books, and then establishes various rules for processing the received data.

Main Components

```
// Main components
public class SimpleRulesStrategy : Strategy
{
}
```

OnStarted Method

Called when the strategy starts:

- Creates subscriptions to trades and order books
- Demonstrates various ways to create and apply rules

```
// OnStarted method
protected override void OnStarted2(DateTime time)
{
    var tickSub = new Subscription(DataType.Ticks, Security);
    var mdSub = new Subscription(DataType.MarketDepth, Security);

    //-----Create a rule. Method №1-----
    mdSub.WhenOrderBookReceived(this).Do((depth) =>
    {
        LogInfo($"The rule WhenOrderBookReceived №1 BestBid={depth.GetBestBid()},
        BestAsk={depth.GetBestAsk()}");
    }).Once().Apply(this);

    //-----Create a rule. Method №2-----
    var whenMarketDepthChanged = mdSub.WhenOrderBookReceived(this);

    whenMarketDepthChanged.Do((depth) =>
    {
        LogInfo($"The rule WhenOrderBookReceived №2 BestBid={depth.GetBestBid()},
        BestAsk={depth.GetBestAsk()}");
    }).Once().Apply(this);

    //-----Rule inside rule-----
    mdSub.WhenOrderBookReceived(this).Do((depth) =>
    {
        LogInfo($"The rule WhenOrderBookReceived №3 BestBid={depth.GetBestBid()},
        BestAsk={depth.GetBestAsk()}");
    });

    //-----not a Once rule-----
    mdSub.WhenOrderBookReceived(this).Do((depth1) =>
    {
        LogInfo($"The rule WhenOrderBookReceived №4 BestBid={depth1.GetBestBid()},
        BestAsk={depth1.GetBestAsk()}");
    }).Apply(this);
    }).Once().Apply(this);
```

```
// Sending requests for subscribe to market data.  
Subscribe(tickSub);  
Subscribe(mdSub);  
  
base.OnStarted2(time);  
}
```

Logic

Method #1: Creating a Rule

- Creates a rule that triggers when an order book is received
- Logs the best bid and ask
- The rule triggers only once (`Once()`)

Method #2: Creating a Rule

- Demonstrates an alternative way to create a rule
- Functionally identical to Method #1

Rule Inside a Rule

- Creates a rule that triggers when an order book is received
- Inside this rule, another rule is created
- The outer rule triggers once, the inner rule - every time an order book is received

Features

- Demonstrates various ways to create and apply rules in StockSharp
- Uses subscription to trades and order books
- Shows an example of logging information in a strategy using the `LogInfo` method
- Illustrates the use of `Once()` to limit rule triggering

Order rule

Rule for Orders

Overview

SimpleOrderRulesStrategy is a strategy that demonstrates the use of rules for processing events related to orders in StockSharp. It subscribes to trades and creates rules for processing order registration events.

Main Components

```
// Main components
public class SimpleOrderRulesStrategy : Strategy
{
}
```

OnStarted Method

Called when the strategy starts:

- Creates a subscription to ticks
- Creates two sets of rules for processing order registration events

```
// OnStarted method
protected override void OnStarted2(DateTime time)
{
    var sub = new Subscription(DataType.Ticks, Security);

    sub.WhenTickTradeReceived(this).Do(() =>
    {
        var order = CreateOrder(Sides.Buy, default, 1);

        var ruleReg = order.WhenRegistered(this);
        var ruleRegFailed = order.WhenRegisterFailed(this);

        ruleReg
            .Do(() => LogInfo("Order №1 Registered"))
            .Once()
            .Apply(this)
            .Exclusive(ruleRegFailed);

        ruleRegFailed
            .Do(() => LogInfo("Order №1 RegisterFailed"))
            .Once()
            .Apply(this)
            .Exclusive(ruleReg);

        RegisterOrder(order);
    }).Once().Apply(this);

    sub.WhenTickTradeReceived(this).Do(() =>
    {
        var order = CreateOrder(Sides.Buy, default, 10000000);

        var ruleReg = order.WhenRegistered(this);
        var ruleRegFailed = order.WhenRegisterFailed(this);

        ruleReg
            .Do(() => LogInfo("Order №2 Registered"))
            .Once()
```

```

        .Apply(this)
        .Exclusive(ruleRegFailed);

ruleRegFailed
    .Do(() => LogInfo("Order №2 RegisterFailed"))
    .Once()
    .Apply(this)
    .Exclusive(ruleReg);

RegisterOrder(order);
}).Once().Apply(this);

// Sending request for subscribe to market data.
Subscribe(sub);

base.OnStarted2(time);
}

```

Logic

First Set of Rules

- When a tick is received, creates an order to buy 1 unit
- The order is created using the CreateOrder method, specifying direction, price (default = market), and volume
- Sets rules for processing successful registration and registration errors
- The rules are mutually exclusive and trigger only once

Second Set of Rules

- When the next tick is received, creates an order to buy 10,000,000 units
- Similarly sets rules for processing successful registration and registration errors
- The rules are also mutually exclusive and trigger only once

Features

- Demonstrates creating rules for processing order registration events
- Uses the mechanism of mutually exclusive rules (Exclusive)
- Shows an example of logging information about order events using the LogInfo method
- Illustrates the use of Once() to limit rule triggering
- Creates orders with different volumes to demonstrate various scenarios (successful registration and registration error)

Rule until

Until Rule

Overview

SimpleRulesUntilStrategy is a strategy that demonstrates the use of a rule with a termination condition (Until) in StockSharp. It subscribes to trades and order books, and then establishes a rule that executes until a certain condition is met.

Main Components

```
// Main components
public class SimpleRulesUntilStrategy : Strategy
{
}
```

OnStarted Method

Called when the strategy starts:

- Creates subscriptions to ticks and order books
- Creates a rule that executes when order book data is received until a certain condition is met

```
// OnStarted method
protected override void OnStarted2(DateTime time)
{
    var tickSub = new Subscription(DataType.Ticks, Security);
    var mdSub = new Subscription(DataType.MarketDepth, Security);

    var i = 0;
    mdSub.WhenOrderBookReceived(this).Do(depth =>
    {
        i++;
        LogInfo($"The rule WhenOrderBookReceived BestBid={depth.GetBestBid()},
BestAsk={depth.GetBestAsk()}");
        LogInfo($"The rule WhenOrderBookReceived i={i}");
    })
    .Until(() => i >= 10)
    .Apply(this);

    // Sending requests for subscribe to market data.
    Subscribe(tickSub);
    Subscribe(mdSub);

    base.OnStarted2(time);
}
```

Logic

- When launched, the strategy creates subscriptions to ticks and order books
- A rule is created that triggers each time order book data is received
- When the rule triggers:
 - The counter `i` is incremented
 - Information about the best bid and ask prices is added to the log

- The current value of the counter `i` is added to the log
- The rule executes until the value of the counter `i` reaches or exceeds 10
- After the condition is met, the rule automatically stops working

Features

- Demonstrates the use of the `Until()` method to limit rule execution
- Uses subscription to trades and order books
- Shows an example of logging information about the order book and counter state using the `LogInfo` method
- Illustrates how to limit the number of rule executions based on a specific condition

Samples

Other Strategy Examples

Hundreds of strategy samples written in C# and Python are maintained in the [AlgoTrading API folder](#). This link appears first on the page. All of these strategies are also compatible with the [Designer](#).

Additional examples can be found in the [Samples](#) folder.

Arbitrage Strategy

Arbitrage Strategy

Overview

ArbitrageStrategy is an arbitrage strategy between a futures contract and its underlying asset. It tracks spreads between instruments and opens positions when arbitrage opportunities arise.

Main Components

The strategy inherits from [Strategy](#) and uses parameters for configuration:

```
public class ArbitrageStrategy : Strategy
{
    private enum ArbitrageState
    {
        Contango,          // Futures price is higher than the underlying asset
        Backwardation,     // Underlying asset price is higher than the futures
        None,              // No position
        OrderRegistration  // In the process of registering orders
    }

    // Strategy parameters
    private readonly StrategyParam<Security> _futureSecurity;
    private readonly StrategyParam<Security> _stockSecurity;
    private readonly StrategyParam<Portfolio> _futurePortfolio;
    private readonly StrategyParam<Portfolio> _stockPortfolio;
    private readonly StrategyParam<decimal> _stockMultiplier;
    private readonly StrategyParam<decimal> _futureVolume;
    private readonly StrategyParam<decimal> _stockVolume;
    private readonly StrategyParam<decimal> _profitToExit;
    private readonly StrategyParam<decimal> _spreadToGenerateSignal;
}
```

Strategy Parameters

The strategy allows customizing the following parameters:

- **FutureSecurity** - futures instrument
- **StockSecurity** - underlying asset instrument
- **FuturePortfolio** - portfolio for futures trading
- **StockPortfolio** - portfolio for underlying asset trading
- **StockMultiplier** - multiplier for the underlying asset (e.g., lot size)
- **FutureVolume** - volume for futures trading
- **StockVolume** - volume for underlying asset trading
- **ProfitToExit** - profit threshold for position exit
- **SpreadToGenerateSignal** - spread threshold for entry signal generation

Strategy Initialization

In the [OnStarted2](#) method, parameters are validated, subscriptions to order books and own trades are created:


```

protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    if (FutureSecurity == null)
        throw new InvalidOperationException("Future security is not specified.");

    if (StockSecurity == null)
        throw new InvalidOperationException("Stock security is not specified.");

    if (FuturePortfolio == null)
        throw new InvalidOperationException("Future portfolio is not specified.");

    if (StockPortfolio == null)
        throw new InvalidOperationException("Stock portfolio is not specified.");

    _futId = FutureSecurity.ToSecurityId();
    _stockId = StockSecurity.ToSecurityId();

    // Subscription to order book updates for both instruments
    var futureDepthSubscription = new Subscription(DataType.MarketDepth, FutureSecurity);
    var stockDepthSubscription = new Subscription(DataType.MarketDepth, StockSecurity);

    futureDepthSubscription.WhenOrderBookReceived(this).Do(ProcessMarketDepth).Apply(this);
    stockDepthSubscription.WhenOrderBookReceived(this).Do(ProcessMarketDepth).Apply(this);

    // Subscription to own trades to track execution prices
    this
        .WhenOwnTradeReceived()
        .Do(OnOwnTradeReceived)
        .Apply(this);

    // Sending requests for market data subscription
    Subscribe(futureDepthSubscription);
    Subscribe(stockDepthSubscription);
}

```

Processing Market Data

The `ProcessMarketDepth` method is called when an order book is updated and implements the main logic:

```

private void ProcessMarketDepth(IOrderBookMessage depth)
{
    // Update the last order book for each instrument
    if (depth.SecurityId == _futId)
        _lastFut = depth;
    else if (depth.SecurityId == _stockId)
        _lastSt = depth;

    // Wait for data for both instruments
    if (_lastFut is null || _lastSt is null)
        return;

    // Calculate volume-weighted average prices for specific volumes
    _futBid = GetAveragePrice(_lastFut, Sides.Sell, FutureVolume);
    _futAsk = GetAveragePrice(_lastFut, Sides.Buy, FutureVolume);
    _stBid = GetAveragePrice(_lastSt, Sides.Sell, StockVolume) * StockMultiplier;
    _stAsk = GetAveragePrice(_lastSt, Sides.Buy, StockVolume) * StockMultiplier;

    // Validate prices
    if (_futBid == 0 || _futAsk == 0 || _stBid == 0 || _stAsk == 0)
        return;

    // Calculate spreads
    var contangoSpread = _futBid - _stAsk;           // Futures price > underlying asset price
}

```

```

var backwardationSpread = _stBid - _futAck;    // Underlying asset price > futures price

decimal spread;
ArbitrageState arbitrageSignal;

// Determine the best arbitrage opportunity
if (backwardationSpread > contangoSpread)
{
    arbitrageSignal = ArbitrageState.Backwardation;
    spread = backwardationSpread;
}
else
{
    arbitrageSignal = ArbitrageState.Contango;
    spread = contangoSpread;
}

// Log current state and spreads
LogInfo($"Current state {_currentState}, enter spread = {_enterSpread}");
LogInfo($"{{ArbitrageState.Backwardation}} spread = {backwardationSpread}");
LogInfo($"{{ArbitrageState.Contango}} spread = {contangoSpread}");
LogInfo($"Entry from spread:{{SpreadToGenerateSignal}}. Exit from profit:{{ProfitToExit}}");

// Recalculate profit based on current market conditions
if (_currentState != ArbitrageState.None && _currentState != ArbitrageState.OrderRegistration)
{
    CalculateProfit();
    LogInfo($"Profit: {_profit}");
}

// Process signals based on current state and market conditions
ProcessSignals(arbitrageSignal, spread);
}

```

Trading Logic

Signal processing and entry/exit decision making are implemented in the ProcessSignals method:

```

private void ProcessSignals(ArbitrageState arbitrageSignal, decimal spread)
{
    // Enter a new position when there's no open position and spread exceeds threshold
    if (_currentState == ArbitrageState.None && spread > SpreadToGenerateSignal)
    {
        _currentState = ArbitrageState.OrderRegistration;

        if (arbitrageSignal == ArbitrageState.Backwardation)
        {
            ExecuteBackwardation();
        }
        else
        {
            ExecuteContango();
        }
    }
    // Exit from Backwardation position when profit threshold is reached
    else if (_currentState == ArbitrageState.Backwardation && _profit >= ProfitToExit)
    {
        _currentState = ArbitrageState.OrderRegistration;
        CloseBackwardationPosition();
    }
    // Exit from Contango position when profit threshold is reached
    else if (_currentState == ArbitrageState.Contango && _profit >= ProfitToExit)
    {
        _currentState = ArbitrageState.OrderRegistration;
        CloseContangoPosition();
    }
}

```

Profit Calculation

The CalculateProfit method calculates current profit based on entry prices and current prices:

```
private void CalculateProfit()
{
    switch (_currentState)
    {
        case ArbitrageState.Backwardation:
            // Buy futures, sell underlying asset - profit when futures price rises and underlying asset
            // price falls
            _profit = (_stockExitPrice * StockMultiplier - _stAsk) + (_futBid - _futureBuyPrice);
            break;

        case ArbitrageState.Contango:
            // Sell futures, buy underlying asset - profit when futures price falls and underlying asset
            // price rises
            _profit = (_futureExitPrice - _futAck) + (_stBid - _stockBuyPrice * StockMultiplier);
            break;

        default:
            _profit = 0;
            break;
    }
}
```

Order Generation

For executing arbitrage strategies, methods for generating orders are used:

```
private (Order buy, Order sell) GenerateOrdersBackwardation()
{
    var futureBuy = CreateOrder(Sides.Buy, FutureVolume);
    futureBuy.Portfolio = FuturePortfolio;
    futureBuy.Security = FutureSecurity;
    futureBuy.Type = OrderTypes.Market;

    var stockSell = CreateOrder(Sides.Sell, StockVolume);
    stockSell.Portfolio = StockPortfolio;
    stockSell.Security = StockSecurity;
    stockSell.Type = OrderTypes.Market;

    return (futureBuy, stockSell);
}

private (Order sell, Order buy) GenerateOrdersContango()
{
    var futureSell = CreateOrder(Sides.Sell, FutureVolume);
    futureSell.Portfolio = FuturePortfolio;
    futureSell.Security = FutureSecurity;
    futureSell.Type = OrderTypes.Market;

    var stockBuy = CreateOrder(Sides.Buy, StockVolume);
    stockBuy.Portfolio = StockPortfolio;
    stockBuy.Security = StockSecurity;
    stockBuy.Type = OrderTypes.Market;

    return (futureSell, stockBuy);
}
```

Features

- The strategy supports working with two different instruments and two portfolios
- Market orders are used for fast execution

- Rules (IMarketRule) are used to track order execution
- Volume-weighted average price is calculated based on volume for more accurate prices
- Arbitrage logic considers both direct (contango) and reverse (backwardation) spreads
- Supports automatic profit calculation and exit when target threshold is reached

Classic Bollinger strategy

Classic Bollinger Strategy

Overview

BollingerStrategyClassicStrategy is a strategy based on the [BollingerBands](#) indicator. It opens positions when the price reaches the upper or lower boundary of the Bollinger Bands.

Main Components

The strategy inherits from [Strategy](#) and uses parameters for configuration:

```
public class BollingerStrategyClassicStrategy : Strategy
{
    private readonly StrategyParam<int> _bollingerLength;
    private readonly StrategyParam<decimal> _bollingerDeviation;
    private readonly StrategyParam<DataType> _candleType;

    private BollingerBands _bollingerBands;
}
```

Strategy Parameters

The strategy allows customizing the following parameters:

- **BollingerLength** - Bollinger Bands indicator period (default 20)
- **BollingerDeviation** - standard deviation multiplier (default 2.0)
- **CandleType** - candle type to work with (default 5-minute)

All parameters are available for optimization with specified value ranges.

Strategy Initialization

In the [OnStarted2](#) method, the Bollinger Bands indicator is created, candle subscription is set up, and visualization is prepared:

```
protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    // Create indicator
    _bollingerBands = new BollingerBands
    {
        Length = BollingerLength,
        Width = BollingerDeviation
    };

    // Create subscription and bind indicator
    var subscription = SubscribeCandles(CandleType);
    subscription
        .BindEx(_bollingerBands, ProcessCandle)
        .Start();

    // Set up visualization on the chart
    var area = CreateChartArea();
    if (area != null)
    {

```

```

    DrawCandles(area, subscription);
    DrawIndicator(area, _bollingerBands, System.Drawing.Color.Purple);
    DrawOwnTrades(area);
}
}

```

Processing Candles

The ProcessCandle method is called for each completed candle and implements the trading logic:

```

private void ProcessCandle(ICandleMessage candle, IIndicatorValue bollingerValue)
{
    // Skip incomplete candles
    if (candle.State != CandleStates.Finished)
        return;

    // Check if the strategy is ready for trading
    if (!IsFormedAndOnlineAndAllowTrading())
        return;

    var typed = (BollingerBandsValue)bollingerValue;

    // Trading logic:
    // Sell when price reaches or exceeds the upper band
    if (candle.ClosePrice >= typed.UpBand && Position >= 0)
    {
        SellMarket(Volume + Math.Abs(Position));
    }
    // Buy when price reaches or falls below the lower band
    else if (candle.ClosePrice <= typed.LowBand && Position <= 0)
    {
        BuyMarket(Volume + Math.Abs(Position));
    }
}

```

Trading Logic

- **Sell signal:** candle close price reaches or exceeds the upper Bollinger Band when there is no short position
- **Buy signal:** candle close price reaches or falls below the lower Bollinger Band when there is no long position
- Position volume increases by the current position amount with each new trade

Features

- The strategy automatically determines instruments to work with via the GetWorkingSecurities() method
- The strategy only works with completed candles
- The indicator and trades are visualized on a chart when a graphic area is available
- Parameter optimization is supported to find optimal strategy settings

Bollinger low band

Bollinger Strategy with Focus on Lower Band

Overview

`BollingerStrategyLowBandStrategy` is a strategy based on the [BollingerBands](#) indicator. It opens a short position when the price reaches the lower boundary of the Bollinger Bands and closes it when the price reaches the middle line.

Main Components

The strategy inherits from [Strategy](#) and uses parameters for configuration:

```
public class BollingerStrategyLowBandStrategy : Strategy
{
    private readonly StrategyParam<int> _bollingerLength;
    private readonly StrategyParam<decimal> _bollingerDeviation;
    private readonly StrategyParam<DataType> _candleType;

    private BollingerBands _bollingerBands;
}
```

Strategy Parameters

The strategy allows customizing the following parameters:

- **BollingerLength** - Bollinger Bands indicator period (default 20)
- **BollingerDeviation** - standard deviation multiplier (default 2.0)
- **CandleType** - candle type to work with (default 5-minute)

All parameters are available for optimization with specified value ranges.

Strategy Initialization

In the [OnStarted2](#) method, the Bollinger Bands indicator is created, candle subscription is set up, and visualization is prepared:

```
protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    // Create indicator
    _bollingerBands = new BollingerBands
    {
        Length = BollingerLength,
        Width = BollingerDeviation
    };

    // Create subscription and bind indicator
    var subscription = SubscribeCandles(CandleType);
    subscription
        .BindEx(_bollingerBands, ProcessCandle)
        .Start();

    // Set up visualization on the chart
    var area = CreateChartArea();
}
```

```

if (area != null)
{
    DrawCandles(area, subscription);
    DrawIndicator(area, _bollingerBands, System.Drawing.Color.Purple);
    DrawOwnTrades(area);
}
}

```

Processing Candles

The ProcessCandle method is called for each completed candle and implements the trading logic:

```

private void ProcessCandle(ICandleMessage candle, IIndicatorValue bollingerValue)
{
    // Skip incomplete candles
    if (candle.State != CandleStates.Finished)
        return;

    // Check if the strategy is ready for trading
    if (!IsFormedAndOnlineAndAllowTrading())
        return;

    var typed = (BollingerBandsValue)bollingerValue;

    // Trading logic:
    // Sell when price touches the lower band (only when no position exists)
    if (candle.ClosePrice <= typed.LowBand && Position == 0)
    {
        SellMarket(Volume);
    }
    // Buy to close the position when price reaches the middle line (only with a short position)
    else if (candle.ClosePrice >= typed.MiddleBand && Position < 0)
    {
        BuyMarket(Math.Abs(Position));
    }
}

```

Trading Logic

- **Sell signal:** candle close price reaches or falls below the lower Bollinger Band when there is no open position
- **Buy signal** (closing short position): candle close price reaches or exceeds the middle Bollinger Band line when there is a short position
- Position volume is fixed when opening and equals the entire current position when closing

Features

- The strategy automatically determines instruments to work with via the GetWorkingSecurities() method
- The strategy only works with completed candles
- The strategy uses only the lower band and middle line of the Bollinger Bands indicator
- Only short positions are opened
- The indicator and trades are visualized on a chart when a graphic area is available
- Parameter optimization is supported to find optimal strategy settings

Bollinger up band

Bollinger Strategy with Focus on Upper Band

Overview

`BollingerStrategyUpBandStrategy` is a strategy based on the [BollingerBands](#) indicator. It opens a long position when the price reaches the upper boundary of the Bollinger Bands and closes it when the price reaches the middle line.

Main Components

The strategy inherits from [Strategy](#) and uses parameters for configuration:

```
public class BollingerStrategyUpBandStrategy : Strategy
{
    private readonly StrategyParam<int> _bollingerLength;
    private readonly StrategyParam<decimal> _bollingerDeviation;
    private readonly StrategyParam<DataType> _candleType;

    private BollingerBands _bollingerBands;
}
```

Strategy Parameters

The strategy allows customizing the following parameters:

- **BollingerLength** - Bollinger Bands indicator period (default 20)
- **BollingerDeviation** - standard deviation multiplier (default 2.0)
- **CandleType** - candle type to work with (default 5-minute)

All parameters are available for optimization with specified value ranges.

Strategy Initialization

In the [OnStarted2](#) method, the Bollinger Bands indicator is created, candle subscription is set up, and visualization is prepared:

```
protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    // Create indicator
    _bollingerBands = new BollingerBands
    {
        Length = BollingerLength,
        Width = BollingerDeviation
    };

    // Create subscription and bind indicator
    var subscription = SubscribeCandles(CandleType);
    subscription
        .BindEx(_bollingerBands, ProcessCandle)
        .Start();

    // Set up visualization on the chart
    var area = CreateChartArea();
}
```

```

if (area != null)
{
    DrawCandles(area, subscription);
    DrawIndicator(area, _bollingerBands, System.Drawing.Color.Purple);
    DrawOwnTrades(area);
}
}

```

Processing Candles

The ProcessCandle method is called for each completed candle and implements the trading logic:

```

private void ProcessCandle(ICandleMessage candle, IIndicatorValue bollingerValue)
{
    // Skip incomplete candles
    if (candle.State != CandleStates.Finished)
        return;

    // Check if the strategy is ready for trading
    if (!IsFormedAndOnlineAndAllowTrading())
        return;

    var typed = (BollingerBandsValue)bollingerValue;

    // Trading logic:
    // Buy when price touches the upper band (only when no position exists)
    if (candle.ClosePrice >= typed.UpBand && Position == 0)
    {
        BuyMarket(Volume);
    }
    // Sell to close the position when price reaches the middle line (only with a long position)
    else if (candle.ClosePrice <= typed.MiddleBand && Position > 0)
    {
        SellMarket(Math.Abs(Position));
    }
}

```

Trading Logic

- **Buy signal:** candle close price reaches or exceeds the upper Bollinger Band when there is no open position
- **Sell signal** (closing long position): candle close price reaches or falls below the middle Bollinger Band line when there is a long position
- Position volume is fixed when opening and equals the entire current position when closing

Features

- The strategy automatically determines instruments to work with via the GetWorkingSecurities() method
- The strategy only works with completed candles
- The strategy uses only the upper band and middle line of the Bollinger Bands indicator
- Only long positions are opened
- The indicator and trades are visualized on a chart when a graphic area is available
- Parameter optimization is supported to find optimal strategy settings

Moving Averages with Martingale

Moving Averages with Martingale

Overview

SmaStrategyMartingaleStrategy is a trading strategy based on the crossover of two simple moving averages ([SimpleMovingAverage](#)) with martingale elements. The strategy uses long and short SMAs to determine entry and exit signals, increasing position size with each new trade.

Main Components

```
public class SmaStrategyMartingaleStrategy : Strategy
{
    private readonly StrategyParam<int> _longSmaLength;
    private readonly StrategyParam<int> _shortSmaLength;
    private readonly StrategyParam<DataType> _candleType;

    // Variables to store previous indicator values
    private decimal _prevLongValue;
    private decimal _prevShortValue;
    private bool _isFirstValue = true;
}
```

Strategy Parameters

The strategy allows customizing the following parameters:

- **LongSmaLength** - long moving average period (default 80)
- **ShortSmaLength** - short moving average period (default 30)
- **CandleType** - candle type to work with (default 5-minute)

All parameters are available for optimization with specified value ranges.

Strategy Initialization

In the [OnStarted2](#) method, SMA indicators are created, candle subscription is set up, and visualization is prepared:

```
protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    // Create indicators
    var longSma = new SimpleMovingAverage { Length = LongSmaLength };
    var shortSma = new SimpleMovingAverage { Length = ShortSmaLength };

    // Add indicators to the strategy collection for automatic IsFormed tracking
    Indicators.Add(longSma);
    Indicators.Add(shortSma);

    // Create subscription and bind indicators
    var subscription = SubscribeCandles(CandleType);
    subscription
        .Bind(longSma, shortSma, ProcessCandle)
        .Start();
}
```

```
// Set up visualization on the chart
var area = CreateChartArea();
if (area != null)
{
    DrawCandles(area, subscription);
    DrawIndicator(area, longSma, System.Drawing.Color.Blue);
    DrawIndicator(area, shortSma, System.Drawing.Color.Red);
    DrawOwnTrades(area);
}
}
```

Processing Candles

The ProcessCandle method is called for each completed candle and implements the trading logic:

```
private void ProcessCandle(ICandleMessage candle, decimal longValue, decimal shortValue)
{
    // Skip incomplete candles
    if (candle.State != CandleStates.Finished)
        return;

    // Check if the strategy is ready for trading
    if (!IsFormedAndOnlineAndAllowTrading())
        return;

    // For the first value, only save data without generating signals
    if (_isFirstValue)
    {
        _prevLongValue = longValue;
        _prevShortValue = shortValue;
        _isFirstValue = false;
        return;
    }

    // Get current and previous comparison of indicator values
    var isShortLessThenLongCurrent = shortValue < longValue;
    var isShortLessThenLongPrevious = _prevShortValue < _prevLongValue;

    // Save current values as previous for the next candle
    _prevLongValue = longValue;
    _prevShortValue = shortValue;

    // Check for crossover (signal)
    if (isShortLessThenLongPrevious == isShortLessThenLongCurrent)
        return;

    // Cancel active orders before placing new ones
    CancelActiveOrders();

    // Determine trade direction
    var direction = isShortLessThenLongCurrent ? Sides.Sell : Sides.Buy;

    // Calculate position size (increase position with each trade - martingale approach)
    var volume = Volume + Math.Abs(Position);

    // Create and register an order with the appropriate price
    var price = Security.ShrinkPrice(shortValue);
    RegisterOrder(CreateOrder(direction, price, volume));
}
```

Trading Logic

- **Buy signal:** short SMA crosses the long SMA from below
- **Sell signal:** short SMA crosses the long SMA from above

- Position size increases by the current position amount with each new trade (martingale element)
- Order price is set to the current short SMA value, rounded to the instrument's tick size

Features

- The strategy automatically determines instruments to work with via the `GetWorkingSecurities()` method
- The strategy only works with completed candles
- The strategy tracks indicator crossovers by comparing the current and previous relationship between SMAs
- All active orders are canceled before placing new ones
- Martingale principle is implemented - increasing position size with each new trade
- Indicators and trades are visualized on the chart when a graphic area is available
- Parameter optimization is supported to find optimal strategy settings

Trend Strategy

One Candle Trend Strategy

Overview

OneCandleTrendStrategy is a simple trend strategy that makes decisions based on the analysis of a single candle.

Main Components

```
public class OneCandleTrendStrategy : Strategy
{
    private readonly StrategyParam<DataType> _candleType;
}
```

Strategy Parameters

The strategy allows customizing the following parameters:

- **CandleType** - candle type to work with (default 5-minute)

Strategy Initialization

In the [OnStarted2](#) method, candle subscription is created and visualization is prepared:

```
protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    // Create subscription
    var subscription = SubscribeCandles(CandleType);

    subscription
        .Bind(ProcessCandle)
        .Start();

    // Set up visualization on the chart
    var area = CreateChartArea();
    if (area != null)
    {
        DrawCandles(area, subscription);
        DrawOwnTrades(area);
    }
}
```

Processing Candles

The ProcessCandle method is called for each completed candle and implements the trading logic:

```
private void ProcessCandle(ICandleMessage candle)
{
    // Check if the candle is finished
    if (candle.State != CandleStates.Finished)
        return;

    // Check if the strategy is ready for trading
    if (!IsFormedAndOnlineAndAllowTrading())
```

```

return;

// Trend strategy: buy on bullish candle, sell on bearish candle
if (candle.OpenPrice < candle.ClosePrice && Position <= 0)
{
    // Bullish candle - buy
    BuyMarket(Volume + Math.Abs(Position));
}
else if (candle.OpenPrice > candle.ClosePrice && Position >= 0)
{
    // Bearish candle - sell
    SellMarket(Volume + Math.Abs(Position));
}
}

```

Trading Logic

- **Buy signal:** bullish candle (close price above open price) when there is no long position
- **Sell signal:** bearish candle (close price below open price) when there is no short position
- Position volume increases by the current position amount with each new trade

Features

- The strategy automatically determines instruments to work with via the `GetWorkingSecurities()` method
- The strategy only works with completed candles
- The strategy uses market orders for position entry
- The strategy applies a simple trend detection logic based on a single candle
- Candles and trades are visualized on the chart when a graphic area is available

Stairs Trend Strategy

Stairs Trend Strategy

Overview

StairsTrendStrategy is a trading strategy based on the analysis of consecutive candles to determine a trend. The strategy opens positions when a sustained trend of a specific length forms.

Main Components

```
public class StairsTrendStrategy : Strategy
{
    private readonly StrategyParam<int> _lengthParam;
    private readonly StrategyParam<DataType> _candleType;

    private int _bullLength;
    private int _bearLength;
}
```

Strategy Parameters

The strategy allows customizing the following parameters:

- **Length** - number of consecutive candles in one direction to identify a trend (default 3)
- **CandleType** - candle type to work with (default 5-minute)

The Length parameter is available for optimization in the range from 2 to 10 with a step of 1.

Strategy Initialization

In the [OnStarted2](#) method, counters are reset, candle subscription is created, and visualization is prepared:

```
protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    // Reset counters
    _bullLength = 0;
    _bearLength = 0;

    // Create subscription
    var subscription = SubscribeCandles(CandleType);

    subscription
        .Bind(ProcessCandle)
        .Start();

    // Set up visualization on the chart
    var area = CreateChartArea();
    if (area != null)
    {
        DrawCandles(area, subscription);
        DrawOwnTrades(area);
    }
}
```

Processing Candles

The ProcessCandle method is called for each completed candle and implements the trading logic:

```
private void ProcessCandle(ICandleMessage candle)
{
    // Check if the candle is finished
    if (candle.State != CandleStates.Finished)
        return;

    // Check if the strategy is ready for trading
    if (!IsFormedAndOnlineAndAllowTrading())
        return;

    // Update counters based on candle direction
    if (candle.OpenPrice < candle.ClosePrice)
    {
        // Bullish candle
        _bullLength++;
        _bearLength = 0;
    }
    else if (candle.OpenPrice > candle.ClosePrice)
    {
        // Bearish candle
        _bullLength = 0;
        _bearLength++;
    }

    // Trend strategy:
    // Buy after Length consecutive bullish candles
    if (_bullLength >= Length && Position <= 0)
    {
        BuyMarket(Volume + Math.Abs(Position));
    }
    // Sell after Length consecutive bearish candles
    else if (_bearLength >= Length && Position >= 0)
    {
        SellMarket(Volume + Math.Abs(Position));
    }
}
```

Trading Logic

- **Buy signal:** Length consecutive bullish candles (close price above open price) when there is no long position
- **Sell signal:** Length consecutive bearish candles (close price below open price) when there is no short position
- Position volume increases by the current position amount with each new trade

Features

- The strategy automatically determines instruments to work with via the GetWorkingSecurities() method
- The strategy only works with completed candles
- The strategy uses market orders for position entry
- The strategy applies a simple trend detection logic based on a sequence of candles
- Candle counters are reset when a candle in the opposite direction appears
- Candles and trades are visualized on the chart when a graphic area is available
- The sequence length optimization is supported to find optimal strategy settings

Countertrend Strategy

One Candle Countertrend Strategy

Overview

OneCandleCountertrendStrategy is a simple countertrend strategy that makes decisions based on the analysis of a single candle.

Main Components

```
public class OneCandleCountertrendStrategy : Strategy
{
    private readonly StrategyParam<DataType> _candleType;
}
```

Strategy Parameters

The strategy allows customizing the following parameters:

- **CandleType** - candle type to work with (default 5-minute)

Strategy Initialization

In the [OnStarted2](#) method, candle subscription is created and visualization is prepared:

```
protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    // Create subscription
    var subscription = SubscribeCandles(CandleType);

    subscription
        .Bind(ProcessCandle)
        .Start();

    // Set up visualization on the chart
    var area = CreateChartArea();
    if (area != null)
    {
        DrawCandles(area, subscription);
        DrawOwnTrades(area);
    }
}
```

Processing Candles

The ProcessCandle method is called for each completed candle and implements the trading logic:

```
private void ProcessCandle(ICandleMessage candle)
{
    // Check if the candle is finished
    if (candle.State != CandleStates.Finished)
        return;

    // Check if the strategy is ready for trading
    if (!IsFormedAndOnlineAndAllowTrading())
```

```

return;

// Countertrend strategy: buy on bearish candle, sell on bullish candle
if (candle.OpenPrice < candle.ClosePrice && Position >= 0)
{
    // Bullish candle - sell
    SellMarket(Volume + Math.Abs(Position));
}
else if (candle.OpenPrice > candle.ClosePrice && Position <= 0)
{
    // Bearish candle - buy
    BuyMarket(Volume + Math.Abs(Position));
}
}

```

Trading Logic

- **Sell signal:** bullish candle (close price above open price) when there is no short position
- **Buy signal:** bearish candle (close price below open price) when there is no long position
- Position volume increases by the current position amount with each new trade

Features

- The strategy automatically determines instruments to work with via the `GetWorkingSecurities()` method
- The strategy only works with completed candles
- The strategy uses market orders for position entry
- The strategy applies a simple countertrend detection logic based on a single candle
- Candles and trades are visualized on the chart when a graphic area is available

Stairs Countertrend Strategy

Stairs Countertrend Strategy

Overview

StairsCountertrendStrategy is a countertrend trading strategy that opens positions against an established trend of a specific length.

Main Components

```
public class StairsCountertrendStrategy : Strategy
{
    private readonly StrategyParam<int> _length;
    private readonly StrategyParam<DataType> _candleType;

    private int _bullLength;
    private int _bearLength;
}
```

Strategy Parameters

The strategy allows customizing the following parameters:

- **Length** - number of consecutive candles in one direction to identify a trend (default 3)
- **CandleType** - candle type to work with (default 5-minute)

The Length parameter is available for optimization in the range from 2 to 10 with a step of 1.

Strategy Initialization

In the [OnStarted2](#) method, counters are reset, candle subscription is created, and visualization is prepared:

```
protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    // Reset counters
    _bullLength = 0;
    _bearLength = 0;

    // Create subscription
    var subscription = SubscribeCandles(CandleType);

    subscription
        .Bind(ProcessCandle)
        .Start();

    // Set up visualization on the chart
    var area = CreateChartArea();
    if (area != null)
    {
        DrawCandles(area, subscription);
        DrawOwnTrades(area);
    }
}
```

Processing Candles

The ProcessCandle method is called for each completed candle and implements the trading logic:

```
private void ProcessCandle(ICandleMessage candle)
{
    // Check if the candle is finished
    if (candle.State != CandleStates.Finished)
        return;

    // Check if the strategy is ready for trading
    if (!IsFormedAndOnlineAndAllowTrading())
        return;

    // Update counters based on candle direction
    if (candle.OpenPrice < candle.ClosePrice)
    {
        // Bullish candle
        _bullLength++;
        _bearLength = 0;
    }
    else if (candle.OpenPrice > candle.ClosePrice)
    {
        // Bearish candle
        _bullLength = 0;
        _bearLength++;
    }

    // Countertrend strategy:
    // Sell after Length consecutive bullish candles
    if (_bullLength >= Length && Position >= 0)
    {
        SellMarket(Volume + Math.Abs(Position));
    }
    // Buy after Length consecutive bearish candles
    else if (_bearLength >= Length && Position <= 0)
    {
        BuyMarket(Volume + Math.Abs(Position));
    }
}
```

Trading Logic

- **Sell signal:** Length consecutive bullish candles (close price above open price) when there is no short position
- **Buy signal:** Length consecutive bearish candles (close price below open price) when there is no long position
- Position volume increases by the current position amount with each new trade

Features

- The strategy automatically determines instruments to work with via the `GetWorkingSecurities()` method
- The strategy only works with completed candles
- The strategy uses market orders for position entry
- The strategy applies a countertrend approach, opening positions against the established trend
- Candle counters are reset when a candle in the opposite direction appears
- Candles and trades are visualized on the chart when a graphic area is available
- The sequence length optimization is supported to find optimal strategy settings

Countertrend Strategy with Quoting

Countertrend Strategy with Quoting

Overview

StairsCountertrendStrategy is a countertrend trading strategy that opens positions against an established trend of a specific length, using a quoting mechanism for more precise market entry.

Main Components

```
public class StairsCountertrendStrategy : Strategy
{
    private readonly StrategyParam<DataType> _candleDataType;
    private readonly StrategyParam<int> _length;
    private QuotingProcessor _quotingProcessor;

    private int _bullLength;
    private int _bearLength;
}
```

Strategy Parameters

The strategy allows customizing the following parameters:

- **CandleDataType** - candle type to work with (default 1-minute)
- **Length** - number of consecutive candles in one direction to identify a trend (default 5)

The Length parameter is available for optimization in the range from 2 to 10 with a step of 1.

Strategy Initialization

In the [OnStarted2](#) method, counters are reset, candle subscription is created, and visualization is prepared:

```
protected override void OnStarted2(DateTime time)
{
    // Reset counters at start
    _bullLength = 0;
    _bearLength = 0;

    // Create candle subscription
    var subscription = SubscribeCandles(CandleDataType);

    subscription
        .Bind(ProcessCandle)
        .Start();

    // Set up visualization on the chart
    var area = CreateChartArea();
    if (area != null)
    {
        DrawCandles(area, subscription);
        DrawOwnTrades(area);
    }

    base.OnStarted2(time);
}
```

Processing Candles

The ProcessCandle method is called for each completed candle and implements the logic for trend detection and quoting processor management:

```
private void ProcessCandle(ICandleMessage candle)
{
    if (candle.State != CandleStates.Finished)
        return;

    // Identify bullish or bearish candle
    if (candle.OpenPrice < candle.ClosePrice)
    {
        _bullLength++;
        _bearLength = 0;

        this.AddInfoLog($"Bullish candle detected. Streak: {_bullLength}");
    }
    else if (candle.OpenPrice > candle.ClosePrice)
    {
        _bullLength = 0;
        _bearLength++;

        this.AddInfoLog($"Bearish candle detected. Streak: {_bearLength}");
    }

    // Stop existing processor when direction change is needed
    if (_quotingProcessor != null)
    {
        // Check if processor needs to be cleared (trend change or position)
        var shouldClearProcessor = false;

        // Need to sell if bullish trend and no short position
        if (_bullLength >= Length && Position >= 0)
            shouldClearProcessor = true;
        // Need to buy if bearish trend and no long position
        else if (_bearLength >= Length && Position <= 0)
            shouldClearProcessor = true;

        if (shouldClearProcessor)
        {
            _quotingProcessor?.Dispose();
            _quotingProcessor = null;
        }
    }

    // Create new quoting processor when needed
    if (_quotingProcessor == null && IsFormedAndOnlineAndAllowTrading())
    {
        if (_bullLength >= Length && Position >= 0)
        {
            // Bullish trend - open short position
            CreateQuotingProcessor(Sides.Sell);
            this.AddInfoLog($"Starting SELL quoting after {_bullLength} bullish candles");
        }
        else if (_bearLength >= Length && Position <= 0)
        {
            // Bearish trend - open long position
            CreateQuotingProcessor(Sides.Buy);
            this.AddInfoLog($"Starting BUY quoting after {_bearLength} bearish candles");
        }
    }
}
```

Creating Quoting Processor

The CreateQuotingProcessor method creates a quoting processor with the specified direction:

```

private void CreateQuotingProcessor(Sides side)
{
    // Create behavior for market quoting
    var behavior = new MarketQuotingBehavior(
        0, // No price offset
        new Unit(0.1m, UnitTypes.Percent), // Use 0.1% as minimum deviation
        MarketPriceTypes.Following // Follow market price
    );

    // Create quoting processor
    _quotingProcessor = new(
        behavior,
        Security,
        Portfolio,
        side,
        Volume, // Quoting volume
        Volume, // Maximum order volume
        TimeSpan.Zero, // No timeout
        this, // Strategy implements ISubscriptionProvider
        this, // Strategy implements IMarketRuleContainer
        this, // Strategy implements ITransactionProvider
        this, // Strategy implements ITimeProvider
        this, // Strategy implements IMarketDataProvider
        IsFormedAndOnlineAndAllowTrading, // Check trading permission
        true, // Use order book prices
        true // Use last trade price if order book is empty
    )
    {
        Parent = this
    };

    // Subscribe to processor events
    _quotingProcessor.OrderRegistered += order =>
        this.AddInfoLog($"Order {order.TransactionId} registered at price {order.Price}");

    _quotingProcessor.OrderFailed += fail =>
        this.AddInfoLog($"Order failed: {fail.Error.Message}");

    _quotingProcessor.OwnTrade += trade =>
        this.AddInfoLog($"Trade executed: {trade.Trade.Volume} at {trade.Trade.Price}");

    _quotingProcessor.Finished += isOk =>
    {
        _quotingProcessor?.Dispose();
        _quotingProcessor = null;
    };

    // Initialize processor
    _quotingProcessor.Start();
}

```

Trading Logic

- **Sell signal:** Length consecutive bullish candles (close price above open price) when there is no short position
- **Buy signal:** Length consecutive bearish candles (close price below open price) when there is no long position
- Quoting processor is used for market entry, following the market price

Features

- The strategy automatically determines instruments to work with via the `GetWorkingSecurities()` method

- The strategy only works with completed candles
- Quoting is used instead of market orders for more efficient market entry
- The strategy applies a countertrend approach, opening positions against the established trend
- Detailed logging of main events for debugging is implemented
- The quoting processor is automatically cleared when the trend direction changes or when targets are reached
- Candles and trades visualization on the chart is supported
- The sequence length parameter optimization is implemented for strategy configuration

Quoting Strategy

Quoting Strategy

Overview

MqStrategy is a strategy that uses a quoting mechanism to manage market positions. It creates a quoting processor based on the current position, allowing it to adaptively respond to market condition changes.

Main Components

```
public class MqStrategy : Strategy
{
    private readonly StrategyParam<MarketPriceTypes> _priceType;
    private readonly StrategyParam<Unit> _priceOffset;
    private readonly StrategyParam<Unit> _bestPriceOffset;

    private QuotingProcessor _quotingProcessor;
}
```

Strategy Parameters

The strategy allows customizing the following parameters:

- **PriceType** - market price type for quoting (default Following)
- **PriceOffset** - price offset from the market price
- **BestPriceOffset** - minimum deviation for quote update (default 0.1%)

Strategy Initialization

In the [OnStarted2](#) method, the strategy subscribes to market time changes:

```
protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    // Subscribe to market time changes for quote updates
    Connector.CurrentTimeChanged += Connector_CurrentTimeChanged;
    Connector_CurrentTimeChanged(default);
}
```

Managing the Quoting Processor

The Connector_CurrentTimeChanged method is called when the market time changes and manages the creation and update of the quoting processor:

```
private void Connector_CurrentTimeChanged(TimeSpan obj)
{
    // Create a new processor only if the current one is stopped or doesn't exist
    if (_quotingProcessor != null && _quotingProcessor.LeftVolume > 0)
        return;

    // Release resources of the old processor if it exists
    _quotingProcessor?.Dispose();
    _quotingProcessor = null;
}
```

```

// Determine quoting side based on current position
var side = Position <= 0 ? Sides.Buy : Sides.Sell;

// Create new quoting behavior
var behavior = new MarketQuotingBehavior(
    PriceOffset,
    BestPriceOffset,
    PriceType
);

// Calculate quoting volume
var quotingVolume = Volume + Math.Abs(Position);

// Create and initialize the processor
_quotingProcessor = new QuotingProcessor(
    behavior,
    Security,
    Portfolio,
    side,
    quotingVolume,
    Volume, // Maximum order volume
    TimeSpan.Zero, // No timeout
    this, // Strategy implements ISubscriptionProvider
    this, // Strategy implements IMarketRuleContainer
    this, // Strategy implements ITransactionProvider
    this, // Strategy implements ITimeProvider
    this, // Strategy implements IMarketDataProvider
    IsFormedAndOnlineAndAllowTrading, // Check trading permission
    true, // Use order book prices
    true // Use last trade price if the order book is empty
)
{
    Parent = this
};

// Subscribe to processor events for logging
_quotingProcessor.OrderRegistered += order =>
    this.AddInfoLog($"Order {order.TransactionId} registered at price {order.Price}");

_quotingProcessor.OrderFailed += fail =>
    this.AddInfoLog($"Order failed: {fail.ErrorMessage}");

_quotingProcessor.OwnTrade += trade =>
    this.AddInfoLog($"Trade executed: {trade.Trade.Volume} at {trade.Trade.Price}");

_quotingProcessor.Finished += isOk => {
    this.AddInfoLog($"Quoting finished with success: {isOk}");
    _quotingProcessor?.Dispose();
    _quotingProcessor = null;
};

// Start the processor
_quotingProcessor.Start();
}

```

Resource Release

In the [OnStopped](#) method, the strategy releases resources:

```

protected override void OnStopped()
{
    // Unsubscribe to prevent memory leaks
    Connector.CurrentTimeChanged -= Connector_CurrentTimeChanged;

    // Release resources of the current processor if it exists
    _quotingProcessor?.Dispose();
}

```

```
_quotingProcessor = null;  
  
base.OnStopped();  
}
```

Trading Logic

- The strategy responds to market time changes
- Quoting direction is determined based on the current position:
 - If position ≤ 0 , a Buy quote is created
 - If position > 0 , a Sell quote is created
- Quoting volume is calculated as the base volume plus the absolute value of the current position
- [QuotingProcessor](#) with [MarketQuotingBehavior](#) is used for quoting

Features

- Uses modern quoting processor instead of legacy quoting strategies
- Adaptively responds to position changes by changing quoting direction
- Supports configuration of various quoting parameters (price type, offset, minimum deviation)
- Includes detailed logging of quoting processor events
- Properly manages resources when stopping the strategy and creating new processors
- Supports working with different types of market prices (Following, Best, Opposite, etc.)

Spread Quoting Strategy

Spread Quoting Strategy

Overview

MqSpreadStrategy is a strategy that creates a spread in the market by simultaneously placing quotes for buying and selling. It uses two quoting processors to manage orders on both sides of the market.

Main Components

```
public class MqSpreadStrategy : Strategy
{
    private readonly StrategyParam<MarketPriceTypes> _priceType;
    private readonly StrategyParam<Unit> _priceOffset;
    private readonly StrategyParam<Unit> _bestPriceOffset;

    private QuotingProcessor _buyProcessor;
    private QuotingProcessor _sellProcessor;
}
```

Strategy Parameters

The strategy allows customizing the following parameters:

- **PriceType** - market price type for quoting (default Following)
- **PriceOffset** - price offset from the market price
- **BestPriceOffset** - minimum deviation for quote update (default 0.1%)

Strategy Initialization

In the [OnStarted2](#) method, the strategy subscribes to market time changes:

```
protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    // Subscribe to market time changes for quote updates
    Connector.CurrentTimeChanged += Connector_CurrentTimeChanged;
    Connector_CurrentTimeChanged(new TimeSpan());
}
```

Managing Quoting Processors

The `Connector_CurrentTimeChanged` method is called when the market time changes and manages the creation and update of quoting processors:

```
private void Connector_CurrentTimeChanged(TimeSpan obj)
{
    // Create new processors only with zero position and if current ones are stopped
    if (Position != 0)
        return;

    if (_buyProcessor != null && _buyProcessor.LeftVolume > 0)
        return;
}
```

```

if (_sellProcessor != null && _sellProcessor.LeftVolume > 0)
    return;

// Release resources of existing processors
_buyProcessor?.Dispose();
_buyProcessor = null;

_sellProcessor?.Dispose();
_sellProcessor = null;

// Create behaviors for market quoting
var buyBehavior = new MarketQuotingBehavior(
    PriceOffset,
    BestPriceOffset,
    PriceType
);

var sellBehavior = new MarketQuotingBehavior(
    PriceOffset,
    BestPriceOffset,
    PriceType
);

// Create processor for buying
_buyProcessor = new QuotingProcessor(
    buyBehavior,
    Security,
    Portfolio,
    Sides.Buy,
    Volume,
    Volume, // Maximum order volume
    TimeSpan.Zero, // No timeout
    this, // Strategy implements ISubscriptionProvider
    this, // Strategy implements IMarketRuleContainer
    this, // Strategy implements ITransactionProvider
    this, // Strategy implements ITimeProvider
    this, // Strategy implements IMarketDataProvider
    IsFormedAndOnlineAndAllowTrading, // Check trading permission
    true, // Use order book prices
    true // Use last trade price if the order book is empty
)
{
    Parent = this
};

// Create processor for selling
_sellProcessor = new QuotingProcessor(
    sellBehavior,
    Security,
    Portfolio,
    Sides.Sell,
    Volume,
    Volume, // Maximum order volume
    TimeSpan.Zero, // No timeout
    this, // Strategy implements ISubscriptionProvider
    this, // Strategy implements IMarketRuleContainer
    this, // Strategy implements ITransactionProvider
    this, // Strategy implements ITimeProvider
    this, // Strategy implements IMarketDataProvider
    IsFormedAndOnlineAndAllowTrading, // Check trading permission
    true, // Use order book prices
    true // Use last trade price if the order book is empty
)
{
    Parent = this
};

// Log creation of new quoting processors

```

```

this.AddInfoLog($"Created buy/sell spread at {CurrentTime}");

// Subscribe to buy processor events for logging
_buyProcessor.OrderRegistered += order =>
    this.AddInfoLog($"Buy order {order.TransactionId} registered at price {order.Price}");

_buyProcessor.OrderFailed += fail =>
    this.AddInfoLog($"Buy order failed: {fail.Error.Message}");

_buyProcessor.OwnTrade += trade =>
    this.AddInfoLog($"Buy trade executed: {trade.Trade.Volume} at {trade.Trade.Price}");

_buyProcessor.Finished += isOk => {
    this.AddInfoLog($"Buy quoting finished with success: {isOk}");
    _buyProcessor?.Dispose();
    _buyProcessor = null;
};

// Subscribe to sell processor events for logging
_sellProcessor.OrderRegistered += order =>
    this.AddInfoLog($"Sell order {order.TransactionId} registered at price {order.Price}");

_sellProcessor.OrderFailed += fail =>
    this.AddInfoLog($"Sell order failed: {fail.Error.Message}");

_sellProcessor.OwnTrade += trade =>
    this.AddInfoLog($"Sell trade executed: {trade.Trade.Volume} at {trade.Trade.Price}");

_sellProcessor.Finished += isOk => {
    this.AddInfoLog($"Sell quoting finished with success: {isOk}");
    _sellProcessor?.Dispose();
    _sellProcessor = null;
};

// Start both processors
_buyProcessor.Start();
_sellProcessor.Start();
}

```

Resource Release

In the [OnStopped](#) method, the strategy releases resources:

```

protected override void OnStopped()
{
    // Unsubscribe to prevent memory leaks
    Connector.CurrentTimeChanged -= Connector_CurrentTimeChanged;

    // Release processor resources
    _buyProcessor?.Dispose();
    _buyProcessor = null;

    _sellProcessor?.Dispose();
    _sellProcessor = null;

    base.OnStopped();
}

```

Trading Logic

- The strategy responds to market time changes
- With zero position and stopped processors, two new processors are created:
 - Buy processor (Buy)
 - Sell processor (Sell)

- Both processors are configured with the same volume and use the same quoting settings
- Processors create a spread in the market by simultaneously placing buy and sell orders

Features

- Uses modern quoting processor [QuotingProcessor](#) with [MarketQuotingBehavior](#)
- Creates a spread in the market by simultaneously placing buy and sell orders
- Works only with zero position, preventing accumulation of unwanted risk
- Supports configuration of various quoting parameters (price type, offset, minimum deviation)
- Includes detailed logging of quoting processor events
- Properly manages resources when stopping the strategy and creating new processors
- Supports working with different types of market prices (Following, Best, Opposite, etc.)

Pairs Trading

Pairs Trading Strategy

Overview

PairsTradingStrategy is a pairs trading strategy based on statistical arbitrage between two related instruments. It tracks the spread between the prices of two assets and opens positions when the spread deviates significantly from the mean, expecting a reversion to the mean.

Main Components

The strategy inherits from [Strategy](#) and uses parameters for configuration:

```
public class PairsTradingStrategy : Strategy
{
    private readonly StrategyParam<int> _spreadLength;
    private readonly StrategyParam<decimal> _entryThreshold;
    private readonly StrategyParam<decimal> _exitThreshold;
    private readonly StrategyParam<DataType> _candleType;

    // Latest prices for each instrument
    private decimal? _lastPrice1;
    private decimal? _lastPrice2;
}
```

Strategy Parameters

The strategy allows customizing the following parameters:

- **SpreadLength** - period for calculating the spread mean and standard deviation (default 20)
- **EntryThreshold** - Z-Score threshold for entering a position (default 2.0)
- **ExitThreshold** - Z-Score threshold for exiting a position (default 0.5)
- **CandleType** - candle type to work with (default 5-minute)

All parameters are available for optimization with specified value ranges.

Strategy Initialization

In the [OnStarted2](#) method, indicators are created and candle subscriptions are set up for two instruments:

```
protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    // Get two instruments for pairs trading
    var securities = GetWorkingSecurities().ToArray();
    if (securities.Length < 2)
        throw new InvalidOperationException("Необходимо указать 2 инструмента.");

    var sec1 = securities[0].sec;
    var sec2 = securities[1].sec;

    // Indicators for calculating the spread mean and standard deviation
    var sma = new SimpleMovingAverage { Length = SpreadLength };
    var stdDev = new StandardDeviation { Length = SpreadLength };
```

```

_lastPrice1 = null;
_lastPrice2 = null;

// Subscribe to candles of the first instrument
SubscribeCandles(CandleType, security: sec1)
.Bind(c =>
{
    if (c.State != CandleStates.Finished)
        return;

    _lastPrice1 = c.ClosePrice;
})
.Start();

// Subscribe to candles of the second instrument with spread processing
SubscribeCandles(CandleType, security: sec2)
.Bind(c =>
{
    if (c.State != CandleStates.Finished)
        return;

    _lastPrice2 = c.ClosePrice;

    if (_lastPrice1 == null)
        return;

    ProcessSpread(_lastPrice1.Value, _lastPrice2.Value, sma, stdDev);
})
.Start();
}

```

Spread Processing

The ProcessSpread method calculates the Z-Score of the spread and generates trading signals:

```

private void ProcessSpread(decimal price1, decimal price2,
    SimpleMovingAverage sma, StandardDeviation stdDev)
{
    // Calculate spread as the price difference
    var spread = price1 - price2;

    // Process indicators
    var smaValue = sma.Process(new DecimalIndicatorValue(sma, spread));
    var devValue = stdDev.Process(new DecimalIndicatorValue(stdDev, spread));

    if (!sma.IsFormed || !stdDev.IsFormed)
        return;

    if (!IsFormedAndOnlineAndAllowTrading())
        return;

    var mean = smaValue.ToDecimal();
    var dev = devValue.ToDecimal();

    if (dev == 0)
        return;

    // Calculate Z-Score: spread deviation from the mean in standard deviation units
    var zScore = (spread - mean) / dev;

    // Spread too high: sell the first instrument, buy the second
    if (zScore > EntryThreshold && Position >= 0)
    {
        SellMarket(Volume + Math.Abs(Position));
    }
    // Spread too low: buy the first instrument, sell the second
    else if (zScore < -EntryThreshold && Position <= 0)

```

```

{
    BuyMarket(Volume + Math.Abs(Position));
}
// Reversion to the mean: close position
else if (Math.Abs(zScore) < ExitThreshold && Position != 0)
{
    ClosePosition();
}
}

```

Trading Logic

- **Sell signal:** the spread Z-Score exceeds the entry threshold (default 2.0) when there is no short position
- **Buy signal:** the spread Z-Score falls below the negative entry threshold (default -2.0) when there is no long position
- **Close position:** the absolute Z-Score value drops below the exit threshold (default 0.5), signaling that the spread is reverting to the mean

Features

- The strategy works with two instruments obtained via the `GetWorkingSecurities()` method
- The spread is calculated as the difference of closing prices of candles from two instruments
- Z-Score is used to normalize the spread deviation from the mean
- The strategy implements the classic mean reversion concept
- The strategy only works with completed candles
- Parameter optimization is supported to find optimal strategy settings

Multi-Timeframe Strategy

Multi-Timeframe Strategy

Overview

MultiTimeframeStrategy is a strategy that uses two timeframes for making trading decisions. Hourly candles determine the trend direction via moving average crossovers, while 5-minute candles with the [RelativeStrengthIndex](#) indicator are used for precise entry in the trend direction.

Main Components

The strategy inherits from [Strategy](#) and uses parameters for configuration:

```
public class MultiTimeframeStrategy : Strategy
{
    private readonly StrategyParam<int> _fastSmaLength;
    private readonly StrategyParam<int> _slowSmaLength;
    private readonly StrategyParam<int> _rsiLength;
    private readonly StrategyParam<decimal> _takeProfit;
    private readonly StrategyParam<decimal> _stopLoss;

    // Trend direction on the higher timeframe
    private Sides? _hourlyTrend;
}
```

Strategy Parameters

The strategy allows customizing the following parameters:

- **FastSmaLength** - fast moving average period for the hourly chart (default 10)
- **SlowSmaLength** - slow moving average period for the hourly chart (default 30)
- **RsiLength** - RSI period for the 5-minute chart (default 14)
- **TakeProfit** - take-profit size in percent (default 2)
- **StopLoss** - stop-loss size in percent (default 1)

All parameters are available for optimization with specified value ranges.

Strategy Initialization

In the [OnStarted2](#) method, indicators are created and candle subscriptions are set up for two timeframes:

```
protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    var fastSma = new SimpleMovingAverage { Length = FastSmaLength };
    var slowSma = new SimpleMovingAverage { Length = SlowSmaLength };
    var rsi = new RelativeStrengthIndex { Length = RsiLength };

    _hourlyTrend = null;

    // Hourly candles for trend detection (SMA crossover)
    SubscribeCandles(TimeSpan.FromHours(1))
        .Bind(fastSma, slowSma, ProcessHourlyCandle)
        .Start();
}
```

```
// 5-minute candles for precise entry (RSI)
SubscribeCandles(TimeSpan.FromMinutes(5))
    .Bind(rsi, ProcessEntryCandle)
    .Start();

// Set up position protection (take-profit and stop-loss)
StartProtection(
    new Unit(TakeProfit, UnitTypes.Percent),
    new Unit(StopLoss, UnitTypes.Percent)
);

// Set up visualization on the chart
var area = CreateChartArea();
if (area != null)
{
    DrawIndicator(area, fastSma, System.Drawing.Color.Blue);
    DrawIndicator(area, slowSma, System.Drawing.Color.Red);
    DrawOwnTrades(area);
}
}
```

Processing Hourly Candles

The `ProcessHourlyCandle` method determines the trend direction on the higher timeframe:

```
private void ProcessHourlyCandle(ICandleMessage candle, decimal fastValue, decimal slowValue)
{
    if (candle.State != CandleStates.Finished)
        return;

    // Determine trend by moving average crossover
    _hourlyTrend = fastValue > slowValue ? Sides.Buy : Sides.Sell;
}
```

Processing 5-Minute Candles

The `ProcessEntryCandle` method implements position entry based on the RSI signal in the trend direction:

```
private void ProcessEntryCandle(ICandleMessage candle, decimal rsiValue)
{
    if (candle.State != CandleStates.Finished)
        return;

    if (_hourlyTrend == null || !IsFormedAndOnlineAndAllowTrading())
        return;

    // Buy: uptrend and RSI in oversold zone
    if (_hourlyTrend == Sides.Buy && rsiValue < 30 && Position <= 0)
    {
        BuyMarket(Volume + Math.Abs(Position));
    }
    // Sell: downtrend and RSI in overbought zone
    else if (_hourlyTrend == Sides.Sell && rsiValue > 70 && Position >= 0)
    {
        SellMarket(Volume + Math.Abs(Position));
    }
}
```

Trading Logic

- **Trend detection:** fast SMA above slow SMA on the hourly chart indicates an uptrend, below indicates a

downtrend

- **Buy signal:** uptrend on the hourly chart and RSI < 30 on the 5-minute chart when there is no long position
- **Sell signal:** downtrend on the hourly chart and RSI > 70 on the 5-minute chart when there is no short position
- **Position protection:** automatic take-profit and stop-loss via StartProtection

Features

- The strategy uses two timeframes: hourly for trend and 5-minute for entry
- Position entry is only performed in the direction of the higher timeframe trend
- RSI is used as a filter to find optimal entry points (oversold/overbought)
- Positions are automatically protected with stop-loss and take-profit
- The strategy only works with completed candles
- Indicators and trades are visualized on the chart when a graphic area is available
- Parameter optimization is supported to find optimal strategy settings

Commissions

Commission System

[S#](#) implements a flexible commission calculation system through the [CommissionManager](#). The manager accepts order and trade messages and calculates commissions based on configured rules.

ICommissionManager Interface

The [ICommissionManager](#) interface defines the base contract:

- **Rules** — a collection of [ICommissionRule](#) rules for commission calculation.
- **Commission** — the total accumulated commission amount (decimal).
- **Reset()** — resets the state of the manager and all rules.
- **Process(Message)** — processes a message; returns the commission for the given message or null.

ICommissionRule Interface

Each rule implements [ICommissionRule](#):

- **Title** — the rule title.
- **Value** — the commission value ([Unit](#)), can be absolute or percentage-based.
- **Process(ExecutionMessage)** — calculates the commission for a specific message.

The base class [CommissionRule](#) contains a helper method `GetValue(price, volume)`:

- For **absolute** values, it returns `Value` as-is.
- For **percentage** values, it computes $(\text{price} * \text{volume} * \text{Value}) / 100$.

Rule Types

Order Rules

Class	Description
CommissionOrderRule	Commission per order (based on order price and volume).
CommissionOrderVolumeRule	Commission based on order volume. For absolute values: $\text{Value} * \text{volume}$.
CommissionOrderCountRule	Commission for every N orders. The <code>Count</code> property sets the threshold.

Trade Rules

Class	Description
CommissionTradeRule	Commission per trade (based on trade price and volume).
CommissionTradeVolumeRule	Commission based on trade volume.

CommissionTradePriceRule	Commission: price * volume * Value.
CommissionTradeCountRule	Commission for every N trades. The Count property sets the threshold.
CommissionTurnOverRule	Commission for each turnover threshold. The TurnOver property sets the threshold.

Filter Rules

Class	Description
CommissionSecurityIdRule	Commission only for a specific instrument. Security property.
CommissionBoardCodeRule	Commission only for a specific exchange board. Board property.
CommissionSecurityTypeRule	Commission only for a specific instrument type. SecurityType property.

Integration via Adapter

The [CommissionMessageAdapter](#) class wraps an inner adapter and automatically calculates commissions on incoming and outgoing [ExecutionMessage](#). If a message does not have the Commission field set, the adapter populates it from the manager.

Integration with Strategy

The strategy ([Strategy](#)) exposes the Commission property, through which you can track the accumulated commission.

Usage Example

```
var manager = new CommissionManager();

// Fixed commission of 1.5 per trade
manager.Rules.Add(new CommissionTradeRule { Value = 1.5m });

// 0.1% of turnover for futures
manager.Rules.Add(new CommissionSecurityTypeRule
{
    SecurityType = SecurityTypes.Future,
    Value = new Unit(0.1m, UnitTypes.Percent)
});

// Commission of 50 for every 100 orders
manager.Rules.Add(new CommissionOrderCountRule
{
    Count = 100,
    Value = 50m
});

// Commission of 10 for every 1,000,000 in turnover
manager.Rules.Add(new CommissionTurnOverRule
{
    TurnOver = 1_000_000m,
    Value = 10m
});

// Processing a message
decimal? commission = manager.Process(executionMsg);
```



```
if (commission != null)
{
    Console.WriteLine($"Commission for message: {commission.Value}");
}

// Total accumulated commission
Console.WriteLine($"Total commission: {manager.Commission}");
```

Resetting State

The `Reset()` method resets the total commission to zero and calls `Reset()` on each rule, which clears internal counters (order count, current turnover, etc.):

```
manager.Reset();
```

Profit and Loss (PnL)

Profit and Loss Management

[S#](#) implements profit and loss (PnL) calculation through the [PnLManager](#). The manager processes a stream of messages (trades, market data) and computes realized and unrealized profit.

IPnLManager Interface

The [IPnLManager](#) interface defines the base contract:

- **RealizedPnL** — realized profit/loss (decimal). Accumulated when positions are closed.
- **UnrealizedPnL** — unrealized profit/loss (decimal). Recalculated based on current market prices.
- **Reset()** — resets the state of the manager.
- **UpdateSecurity(Level1ChangeMessage)** — updates instrument parameters (price step, step price, lot multiplier).
- **ProcessMessage(Message, ICollection<PortfolioPnLManager>)** — processes a message; returns [PnLInfo](#) when a position is closed, otherwise null.

Architecture

The PnL system has a three-level hierarchy:

```
PnLManager
├── PortfolioPnLManager (by portfolio name)
│   └── PnLQueue (by SecurityId)
```

- [PnLManager](#) — top level, manages a dictionary of portfolio managers.
- [PortfolioPnLManager](#) — PnL manager for a specific portfolio, manages queues by instrument.
- [PnLQueue](#) — FIFO queue for matching trades on a single instrument.

PnLQueue — Calculation Queue

[PnLQueue](#) is responsible for matching opening and closing trades:

- **PriceStep** — instrument price step.
- **StepPrice** — price step cost (for futures).
- **Leverage** — leverage.
- **LotMultiplier** — lot multiplier.

The profit multiplier is calculated using the formula:

```
Multiplier = (StepPrice / PriceStep) * Leverage * LotMultiplier
```

For regular stocks (where StepPrice is not set), the multiplier equals $1 * \text{Leverage} * \text{LotMultiplier}$.

PnLInfo — Trade Processing Result

The [PnLInfo](#) class contains the result of closing a position:

- **ServerTime** — trade time.

- **ClosedVolume** — volume of the closed position.
- **PnL** — realized profit from this trade.

For example, if the position was +2 and a trade for -5 contracts arrived, then `ClosedVolume` = 2 (2 contracts from the position were closed).

Configuring Data Sources

[PnlManager](#) allows you to select market data sources for unrealized profit calculation:

Property	Default	Description
UseTick	true	Use tick trades.
UseOrderBook	false	Use order book (best bid/ask).
UseLevel1	false	Use Level1 data.
UseOrderLog	false	Use order log.
UseCandles	true	Use candles (close price).

Integration via Adapter

The [PnlMessageAdapter](#) class wraps an inner adapter and automatically processes all messages for PnL calculation.

Integration with Strategy

The strategy ([Strategy](#)) provides:

- `PnlManager` property — the manager instance.
- `PnL` property — total profit (`RealizedPnL` + `UnrealizedPnL`).
- `PnLChanged` event — notification of profit changes.
- `PnLReceived2` event — notification when new PnL data is received.

Usage Example

```
var pnlManager = new PnlManager
{
    UseTick = true,
    UseOrderBook = true,
    UseCandles = true
};

// Processing messages
var info = pnlManager.ProcessMessage(executionMsg);
if (info != null)
{
    Console.WriteLine($"Closed: {info.ClosedVolume}, PnL: {info.PnL}");
}

// Total profit/loss
var realizedPnL = pnlManager.RealizedPnL;
var unrealizedPnL = pnlManager.UnrealizedPnL;
var totalPnL = realizedPnL + unrealizedPnL;

Console.WriteLine($"Realized PnL: {realizedPnL}");
Console.WriteLine($"Unrealized PnL: {unrealizedPnL}");
Console.WriteLine($"Total PnL: {totalPnL}");
```

Resetting State

The `Reset()` method clears all portfolio managers, calculation queues, and resets realized PnL to zero:

```
pn1Manager.Reset();
```

Slippage

Slippage Measurement

[S#](#) calculates slippage through the [SlippageManager](#). Slippage is the difference between the expected execution price of an order and the actual trade price.

ISlippageManager Interface

The [ISlippageManager](#) interface defines the base contract:

- **Slippage** — total accumulated slippage (decimal).
- **Reset()** — resets the state of the manager.
- **ProcessMessage(Message)** — processes a message; returns the slippage for the given execution or null.

How It Works

The slippage manager works in three stages:

1. Updating Market Prices

When a [Level1ChangeMessage](#) or [QuoteChangeMessage](#) is received, the manager saves the best bid and ask prices for each instrument.

2. Saving the Planned Price

When an [OrderRegisterMessage](#) is received, the manager saves the "planned price" — the best market price at the time of order registration:

- For a buy (Buy), the best ask is used.
- For a sell (Sell), the best bid is used.

3. Calculating Slippage

When an [ExecutionMessage](#) with a trade is received, the manager calculates slippage:

- For a buy: $(\text{TradePrice} - \text{PlannedPrice}) * \text{TradeVolume}$
- For a sell: $(\text{PlannedPrice} - \text{TradePrice}) * \text{TradeVolume}$

A positive value means unfavorable slippage (price worse than expected), while a negative value means favorable slippage (price better than expected).

State: ISlippageManagerState

The [ISlippageManagerState](#) interface stores the manager's internal state:

- Best bid/ask prices for each instrument ([SecurityId](#)).
- Planned prices and directions for each transaction ([TransactionId](#)).
- Total accumulated slippage.

The default implementation is [SlippageManagerState](#).

Settings

Property	Default	Description
CalculateNegative	true	Account for favorable slippage. If false, negative values are replaced with zero.

Integration via Adapter

The [SlippageMessageAdapter](#) class wraps an inner adapter and automatically calculates slippage for all trades.

Integration with Strategy

The strategy ([Strategy](#)) exposes the `Slippage` property for tracking overall slippage.

Usage Example

```
// Creating a manager with a state store
var manager = new SlippageManager(new SlippageManagerState());

// Only account for unfavorable slippage
manager.CalculateNegative = false;

// Processing market data (updating best prices)
manager.ProcessMessage(level1Msg);
manager.ProcessMessage(quoteChangeMsg);

// Processing order registration (saving the planned price)
manager.ProcessMessage(orderRegisterMsg);

// Processing a trade (calculating slippage)
decimal? slippage = manager.ProcessMessage(executionMsg);
if (slippage != null)
{
    Console.WriteLine($"Slippage: {slippage.Value}");
}

// Total accumulated slippage
Console.WriteLine($"Total slippage: {manager.Slippage}");
```

Resetting State

The `Reset()` method completely clears the internal state: best prices, planned prices, and accumulated slippage:

```
manager.Reset();
```

Latency

Latency Measurement

[S#](#) measures order registration and cancellation latency through the [LatencyManager](#). The manager determines how much time passes between sending an order and receiving confirmation from the exchange.

ILatencyManager Interface

The [ILatencyManager](#) interface defines the base contract:

- **LatencyRegistration** — total registration latency across all orders (TimeSpan).
- **LatencyCancellation** — total cancellation latency across all orders (TimeSpan).
- **Reset()** — resets the state of the manager.
- **ProcessMessage(Message)** — processes a message; returns the latency for the given operation or null.

How It Works

The latency manager operates on a "request-response" principle:

1. Order Registration

When an [OrderRegisterMessage](#) is received, the manager saves the pair (TransactionId, LocalTime) — the moment the order was sent.

2. Order Cancellation

When an [OrderCancelMessage](#) is received, the manager saves the pair (TransactionId, LocalTime) — the moment the cancellation was sent.

3. Order Replacement

When an [OrderReplaceMessage](#) is received, the manager registers both a cancellation (of the old order) and a registration (of the new order) simultaneously.

4. Confirmation

When an [ExecutionMessage](#) with order information (not in Pending state and not Failed) is received, the manager calculates the latency:

```
Latency = ExecutionMessage.LocalTime - StoredLocalTime
```

The result is added to LatencyRegistration or LatencyCancellation depending on the operation type.

State: ILatencyManagerState

The [ILatencyManagerState](#) interface stores the manager's internal state:

- Pending registrations: AddRegistration(transactionId, localTime) / TryGetAndRemoveRegistration(transactionId, out localTime)
- Pending cancellations: AddCancellation(transactionId, localTime) /

```
TryGetAndRemoveCancellation(transactionId, out localTime)
```

- Accumulated latencies: `LatencyRegistration`, `LatencyCancellation`
- Addition methods: `AddLatencyRegistration(TimeSpan)`, `AddLatencyCancellation(TimeSpan)`

The default implementation is [LatencyManagerState](#).

Error Handling

If an order fails (`OrderState == Failed`), the latency is not counted — the record is simply removed from the state store.

Integration via Adapter

The [LatencyMessageAdapter](#) class wraps an inner adapter and automatically measures latency for all order operations.

Integration with Strategy

The strategy ([Strategy](#)) exposes the `Latency` property for tracking latency.

Usage Example

```
// Creating a manager with a state store
var manager = new LatencyManager(new LatencyManagerState());

// Processing order registration (saving the send time)
manager.ProcessMessage(orderRegisterMsg);

// Processing confirmation (calculating latency)
TimeSpan? latency = manager.ProcessMessage(executionMsg);
if (latency != null)
{
    Console.WriteLine($"Latency: {latency.Value.TotalMilliseconds} ms");
}

// Total latencies
Console.WriteLine($"Registration latency: {manager.LatencyRegistration.TotalMilliseconds} ms");
Console.WriteLine($"Cancellation latency: {manager.LatencyCancellation.TotalMilliseconds} ms");
```

Resetting State

The `Reset()` method clears all pending records and resets accumulated latencies to zero:

```
manager.Reset();
```


Logging

Logging

For the monitoring of trading algorithms written in [S#](#), you can use the special [LogManager](#) class. This class receives the [LogMessage](#) messages from the [LogManager.Sources](#) through the [ILogSource.Log](#) event and passes them to the [LogManager.Listeners](#) listeners. Therefore, the algorithm code will be able to pass debug information (for example, about errors occurred during the operation, or additional information about mathematical calculations) and [LogManager](#) will decide how to display this information for the operator.

Normally the [S#](#) contains the following implementations of the [ILogListener](#), whose choice affects where messages received from strategies will be passed:

1. [FileLogListener](#) - writes messages into a text file. It is recommended to use for already created algorithm, and to use the logs in cases of force majeure.
2. [ConsoleLogListener](#) - outputs messages to the console window (if the algorithm does not have a window, it will be automatically created). It is recommended to use for debugging and testing the algorithm
3. [DebugLogListener](#) - outputs messages to the debug window. This window can be viewed through special programs such as [DebugView](#). It is recommended to use for debugging and testing the algorithm.
4. [EmailLogListener](#) - sends messages to the specified email address. It is recommended to use if the algorithm is located at not controlled computer (at hoster's server).
5. [GuiLogListener](#) - displays messages through the special [LogControl](#) window. Able to work in two modes: when all messages output in a single window, and when a separate window is created for each [ILogSource](#). It is recommended to use if the algorithm has a graphical interface.

[LogListener](#) can be configured to filter messages through the [LogListener.Filters](#) property. For example, through the filters you can specify what type of messages should be processed. This is particularly useful when the [EmailLogListener](#) is used, in order that, for example, to send e-mail in emergency situations only (trade algorithm error) rather than on each debug message.

Next Steps

[Strategy logging](#)

[IConnector logging](#)

[Other logs sources](#)

[Visual monitoring](#)

[ILogListener creating](#)

Strategy logging

Strategy logging

The [Strategy](#) class implements the [ILogSource](#) interface. Therefore, the strategies can be passed to the [LogManager.Sources](#), and all of its messages will automatically get to the [LogManager.Listeners](#).

Prerequisites

[Trading strategies](#)

Logging to a test file

1. First, you need to create the special manager:

```
var logManager = new LogManager();
```

2. Then you need to create a file logger, passing to it the name of the file and to add it to the [LogManager.Listeners](#):

```
var fileListener = new FileLogListener("{0}_{1:00}_{2:00}.txt".Put(DateTime.Now.Year,
DateTime.Now.Month, DateTime.Now.Day));
logManager.Listeners.Add(fileListener);
```

3. For logging messages, you need to add a strategy to the [LogManager.Sources](#):

```
logManager.Sources.Add(lkohSmaStrategy);
```

4. After adding strategy to the logging manager all of its messages will be recorded to file.

Sound playback

1. Creating a logger and passing the name of the sound file to it:

```
var soundListener = new SoundLogListener("error.mp3");

logManager.Listeners.Add(soundListener);
logManager.Sources.Add(lkohSmaStrategy);
```

2. Setting the filter so sound plays only when the message type is [LogLevels.Error](#):

```
soundListener.Filters.Add(msg => msg.Level == LogLevels.Error);
```

Email sending

1. Create the logger and pass it the parameters for the sent letters:

```
var emailListener = new EmailLogListener("from@stocksharp.com", "to@stocksharp.com");
logManager.Listeners.Add(emailListener);
logManager.Sources.Add(lkohSmaStrategy);
```

2. Setting the filter on the sending of messages of [LogLevels.Error](#) and [LogLevels.Warning](#) types:

```
emailListener.Filters.Add(msg => msg.Level == LogLevels.Error);
emailListener.Filters.Add(msg => msg.Level == LogLevels.Warning);
```

Logging in to the LogWindow

1. Creating the [GuiLogListener](#) logger:

```
// each strategy will have their own windows
var guiListener = new GuiLogListener();
logManager.Listeners.Add(guiListener);
logManager.Sources.Add(lkohSmaStrategy);
```

2. Here is the log window when the strategy is working: *[image: strategylogging]*

Recommended content

[Visual logging components](#)

IConnector logging

IConnector logging

The [IConnector](#) objects similar [strategies](#) also implement the [ILogSource](#) interface. Therefore, from the [IConnector](#) object you can receive messages through the [LogManager](#) by all means that are available for strategies.

IConnector logging example

1. First, you need to create a log manager:

```
...
private readonly Connector _connector = new Connector();
private readonly LogManager _logManager = new LogManager();
...
```

2. Then you need to create a file logger, and to add it to the [LogManager.Listeners](#):

```
_logManager.Listeners.Add(new FileLogListener());
```

3. The final step is to add the [Connector](#) to [LogManager.Sources](#):

```
_logManager.Sources.Add(_connector);
```

4. As a result, the program will display messages as shown below after the start:

```
18:43:15 | Info | RithmicTrader
18:43:15 | Debug | RithmicTrader | ReadPortfolios()
18:43:15 | Debug | RithmicTrader | OnProcessPortfolios()
18:43:15 | Debug | RithmicTrader | 41469|15152,43|15530,8|
18:43:15 | Debug | RithmicTrader | ReadSecurities()
18:43:15 | Debug | RithmicTrader | OnProcessSecurities
18:43:15 | Debug | RithmicTrader | 291|ESU5|
```

Other logs sources

Other logs sources

In previous topics, objects embedded in the [S#](#) classes were sources of logs. The [S#](#) provides possibilities for cases when the source of the logs is your own class, or when the source does not have to be associated with a particular class but serves the entire application. For the first case you must implement in your class the [ILogSource](#) interface or inherit from the [BaseLogReceiver](#). In the second situation, you can use the [TraceSource](#), which uses the .NET tracing system. How to do this is shown in the *Samples/08_Misc/01_Logging* sample.

Logging sample

1. Create a custom class that inherits from the [BaseLogReceiver](#).

```
private class TestSource : BaseLogReceiver
{
}
```

2. Create the [LogManager](#) and declare a variable of the user class.

```
private readonly LogManager _logManager = new LogManager();
private readonly TestSource _testSource;
```

3. Add log sources.

```
_logManager.Sources.Add(_testSource = new TestSource());
_logManager.Sources.Add(new Ecng.Logging.TraceSource());
```

4. Add log listeners.

```
// log messages will be displayed in GUI component
_logManager.Listeners.Add(new GuiLogListener(Monitor));
// also writing in files
_logManager.Listeners.Add(new FileLogListener
{
    FileName = "logs",
});
```

5. Add messages of the custom class logging. The level of logging is chosen randomly.

```
var level = RandomGen.GetEnum<LogLevels>();
switch (level)
{
    case LogLevels.Inherit:
    case LogLevels.Debug:
    case LogLevels.Info:
    case LogLevels.Off:
        _testSource.AddInfoLog("{0} (source)!!!".Put(level));
        break;
    case LogLevels.Warning:
        _testSource.AddWarningLog("Warning (source)!!!");
        break;
    case LogLevels.Error:
        _testSource.AddErrorLog("Error (source)!!!");
        break;
    default:
        throw new ArgumentOutOfRangeException();
}
```

6. Add tracing messages.

```
var level = RandomGen.GetEnum<LogLevels>();
switch (level)
{
    case LogLevels.Inherit:
    case LogLevels.Debug:
    case LogLevels.Info:
    case LogLevels.Off:
        Trace.TraceInformation("{0} (trace)!!!".Put(level));
        break;
    case LogLevels.Warning:
        Trace.TraceWarning("Warning (trace)!!!");
        break;
    case LogLevels.Error:
        Trace.TraceError("Error (trace)!!!");
        break;
    default:
        throw new ArgumentOutOfRangeException();
}
```

7. The result of the example work.*[image: sample logging]*

Visual monitoring

Visual monitoring

To simplify monitoring, you can use the special [Monitor](#) component. See also [Visual logging components](#).

[image: GUI LogControl]

This window allows you to display messages from all [ILogSource](#):

- strategies ([Strategy](#));
- connectors ([IConnector](#));
- own [ILogSource](#) implementations (for example, the main window in the algorithm).

The nesting of sources is shown in the form of a tree. Each parent node contains messages from all nested sources and so on down to the lowest level. For connectors this is also useful when using [BasketTrader](#). Similarly, the same nesting can be arranged for your own algorithm by implementing the [ILogSource.Parent](#) property.

Using Monitor

1. First, you need to create a window and add the component.
2. Then, the created window must be added to your [LogManager](#) through the [GuiLogListener](#):

```
_logManager.Listeners.Add(new GuiLogListener(monitor));
```

3. Thereafter all sources [LogManager.Sources](#) (strategies, connectors, etc.) will send messages to the [Monitor](#).

Recommended content

[Visual logging components](#)

Custom ILogListener

Custom ILogListener

If you want to create your own implementation of the [ILogListener](#) (for example, when you want to save messages in the database), the class [LogListener](#) can be inherited or the [ILogListener](#) interface can be implemented directly. The object of the [LogMessage](#) class is passed through the [ILogListener.WriteMessages\(System.Collections.Generic.ICollection<Ecng.Logging.LogMessage> messages\)](#) method. This class contains the information about the message source [LogMessage.Source](#) (for example, a strategy that has generated the message), the message type [LogMessage.Level](#) (information, warning or error), as well as the [LogMessage.Message](#) text itself. The following example shows the source code of [EmailLogListener](#):

```
/// <summary>
public class EmailLogListener : LogListener
{
    public EmailLogListener(string from, string to)
    {
        if (from.IsEmpty())
            throw new ArgumentNullException("from");
        if (to.IsEmpty())
            throw new ArgumentNullException("to");
        From = from;
        To = to;
    }
    public string From { get; private set; }
    public string To { get; private set; }
    protected override void OnWriteMessage(LogMessage message)
    {
        var email = new SmtpClient();
        email.Send(new MailMessage(From, To, message.Source.Name + " " + message.Level, message.Message));
    }
}
```


Options

Options

To work with the options [S#](#) provides special methods for calculation of “greeks”, synthetic positions, options quoting strategies for a given volatility and delta hedging.

Recommended content

[Greeks](#)

[Volatility trading](#)

[Delta hedging](#)

[Graphic components](#)

Greeks

The formula of [Black-Scholes model](#) is realized in the [S#](#) to calculate the basic “Greeks”: delta, gamma, vega, theta and rho. The [volatility trading](#) and [Delta-hedging](#) strategies are realized on the basis of this formula. Also [S#](#) allows you to calculate the option premium and [IV](#).

The following code shows the [BlackScholes](#) class methods to calculate the “Greeks”.

```
var bs = new BlackScholes(option, _connector, _connector);
DateTimeOffset currentTime = DateTimeOffset.Now;
decimal delta = bs.Delta(currentTime);
decimal gamma = bs.Gamma(currentTime);
decimal vega = bs.Vega(currentTime);
decimal theta = bs.Theta(currentTime);
decimal rho = bs.Rho(currentTime);
decimal iv = bs.ImpliedVolatility(currentTime, premium); // premium is premium of the option
contract
```

In addition, the installation package includes the OptionCalculator example, in which all the “Greeks” are calculated and visualized using the [OptionDesk](#) graphical component. See [Graphic components](#).

Volatility trading

Volatility trading

For option quoting, a special [VolatilityQuotingStrategy](#) strategy is implemented, which provides volume quoting within the specified range of volatility.

Quoting by volatility

1. The [S#](#) installation package includes the example SampleOptionQuoting, which quotes the selected strike within the specified range of volatility.
2. Creating a connection to the [OpenECry](#) and starting the export:

```
private void InitConnector()
{
    // subscribe on connection successfully event
    Connector.Connected += () =>
    {
        // update gui labels
        this.GuiAsync(() => ChangeConnectStatus(true));
    };
    // subscribe on disconnection event
    Connector.Disconnected += () =>
    {
        // update gui labels
        this.GuiAsync(() => ChangeConnectStatus(false));
    };
    // subscribe on connection error event
    Connector.ConnectionError += error => this.GuiAsync(() =>
    {
        // update gui labels
        ChangeConnectStatus(false);
        MessageBox.Show(this, error.ToString(), LocalizedStrings.ErrorConnection);
    });
    // fill underlying asset's list
    Connector.SecurityReceived += (sub, security) =>
    {
        if (security.Type == SecurityTypes.Future)
            _assets.Add(security);
    };
    Connector.Level1Received += (sub, security) =>
    {
        if (_model.UnderlyingAsset == security || _model.UnderlyingAsset.Id ==
security.UnderlyingSecurityId)
            _isDirty = true;
    };
    // subscribing on tick prices and updating asset price
    Connector.TickTradeReceived += (sub, trade) =>
    {
        if (_model.UnderlyingAsset == trade.Security || _model.UnderlyingAsset.Id ==
trade.Security.UnderlyingSecurityId)
            _isDirty = true;
    };
    Connector.NewPosition += position => this.GuiAsync(() =>
    {
        var asset = SelectedAsset;
        if (asset == null)
            return;
        var assetPos = position.Security == asset;
        var newPos = position.Security.UnderlyingSecurityId == asset.Id;
        if (!assetPos && !newPos)
            return;
        if (assetPos)
```

```

        PosChart.AssetPosition = position;
        if (newPos)
            PosChart.Positions.Add(position);
            RefreshChart();
    });
    Connector.PositionChanged += position => this.GuiAsync(() =>
    {
        if ((PosChart.AssetPosition != null && PosChart.AssetPosition == position) ||
        PosChart.Positions.Cache.Contains(position))
            RefreshChart();
    });
    try
    {
        if (File.Exists(_settingsFile))
            Connector.Load(new JsonSerializer<SettingsStorage>().Deserialize(_settingsFile));
    }
    catch
    {
    }
}
private void ConnectClick(object sender, RoutedEventArgs e)
{
    if (!_isConnected)
    {
        ConnectBtn.IsEnabled = false;
        _model.Clear();
        _model.MarketDataProvider = Connector;
        ClearSmiles();
        PosChart.Positions.Clear();
        PosChart.AssetPosition = null;
        PosChart.Refresh(1, 1, default(DateTimeOffset), default(DateTimeOffset));
        Portfolio.Portfolios = new PortfolioDataSource(Connector);
        PosChart.MarketDataProvider = Connector;
        PosChart.SecurityProvider = Connector;
        Connector.Connect();
    }
    else
        Connector.Disconnect();
}

```

3. Set up the [VolatilityQuotingStrategy](#) strategy (filling the range of volatility, as well as the creation of the order, wherethrough the required volume and quoting direction are specified):

```

private void StartClick(object sender, RoutedEventArgs e)
{
    var option = SelectedOption;
    // create DOM window
    var wnd = new QuotesWindow { Title = option.Name };
    wnd.Init(option);
    // create delta hedge strategy
    var hedge = new DeltaHedgeStrategy
    {
        Security = option.GetUnderlyingAsset(Connector),
        Portfolio = Portfolio.SelectedPortfolio,
        Connector = Connector,
    };
    // create option quoting for 20 contracts
    var quoting = new VolatilityQuotingStrategy(Sides.Buy, 20,
        new Range<decimal>(ImpliedVolatilityMin.Value ?? 0, ImpliedVolatilityMax.Value ?? 100))
    {
        // working size is 1 contract
        Volume = 1,
        Security = option,
        Portfolio = Portfolio.SelectedPortfolio,
        Connector = Connector,
    };
    // link quoting and hedging
    hedge.ChildStrategies.Add(quoting);
}

```

```

        // start hedging
        hedge.Start();
        wnd.Closed += (s1, e1) =>
        {
            // force close all strategies while the DOM was closed
            hedge.Stop();
        };
        // show DOM
        wnd.Show();
    }

```

4. Starting quoting:

```
hedge.Start();
```

5. For a visual presentation of the volatility the example shows how you can convert the standard order book with quotations to the order book of volatility through the use of the [DerivativesHelper.ImpliedVolatility](#)([StockSharp.Messages.IOrderBookMessage](#) depth, [StockSharp.BusinessEntities.ISecurityProvider](#) securityProvider, [StockSharp.BusinessEntities.IMarketDataProvider](#) dataProvider, [StockSharp.BusinessEntities.IExchangeInfoProvider](#) exchangeInfoProvider, [System.DateTimeOffset](#) currentTime, [System.Decimal](#) riskFree, [System.Decimal](#) dividend) method:

```

private void OnQuotesChanged()
{
    DepthCtrl.UpdateDepth(_depth.ImpliedVolatility(Connector, Connector, Connector.CurrentTime));
}

```

[image: sample quote iv]

6. Ending quoting and stopping the strategy:

```
hedge.Stop();
```

Recommended content

[Delta hedging](#)

Delta hedging

Delta hedging

If you want to protect positions by option strategies (for example, as [Volatility trading](#)) you can use the [DeltaHedgeStrategy](#) hedging strategy by delta.

Delta hedging

1. As a demonstration of how the [DeltaHedgeStrategy](#) works, the [SampleOptionQuoting](#) example is modified (for details see [Volatility trading](#)).
2. The [VolatilityQuotingStrategy](#) strategy does not start, but instead it is passed as a child strategy to the [DeltaHedgeStrategy](#).

```
// create delta hedge strategy
var hedge = new DeltaHedgeStrategy
{
    Security = option.GetUnderlyingAsset(Connector),
    Portfolio = Portfolio.SelectedPortfolio,
    Connector = Connector,
};
// create option quoting for 20 contracts
var quoting = new VolatilityQuotingStrategy(Sides.Buy, 20,
    new Range<decimal>(ImpliedVolatilityMin.Value ?? 0, ImpliedVolatilityMax.Value ?? 100))
{
    // working size is 1 contract
    Volume = 1,
    Security = option,
    Portfolio = Portfolio.SelectedPortfolio,
    Connector = Connector,
};
// link quoting and hedging
hedge.ChildStrategies.Add(quoting);
// start hedging
hedge.Start();
```

The [DeltaHedgeStrategy](#) takes strategies, working separately on their strike, as child strategies. Thus, [DeltaHedgeStrategy](#) controls the total position by all child option strategies.

3. Completing the delta hedging:

```
hedge.Stop();
```

Graphic components

Graphic components

See [Graphic components](#) section.

Testing

Backtesting/Emulation

Strategies written by using [Strategy](#) can be tested in three modes:

1. [Historical data](#) testing. With this kind of data both a market analysis to find patterns and the strategy parameters optimization may be carried out.
2. [Random data](#) testing. A convenient tool for initial strategy testing to identify errors in the algorithms. Or for automated tests that run on a schedule.
3. [Simulator](#) testing based on data received from a real connection to the trading system (for example, from [OpenECry](#)), but without the actual orders registration (execution is emulated based on incoming order books).

When using all three modes the maximum emphasis is focused on the fact that the strategy code, written by using [Strategy](#) is not changed when switching from real trading to testing and back. This is achieved through the implementation of the [IConnector](#) main interface, which is a gateway to the trading system. How the interface is used – this has been shown already in the [API](#) section. In testing mode not real trading system, but the emulation will act as the trading system (depending on the selected mode). Therefore, the strategy code will never know about it - whether it is trading with a real exchange, or it is the emulation.

Historical data

Historical Data

Testing on historical data allows both market analysis to find patterns and [strategy parameter optimization](#). The main work is performed by the [HistoryEmulationConnector](#) class, which retrieves data stored in a local repository through a special [API](#). Additional parameters are described in the [testing settings](#) section.

Testing can be performed using various types of market data:

- Tick trades ([ITickTradeMessage](#))
- Order books ([IOrderBookMessage](#))
- Candles of different timeframes
- [OrderLog](#)
- [Level1](#) (best bid and ask prices)
- Combinations of different data types

If there are no saved order books for the testing period, they can be generated based on trades using [MarketDepthGenerator](#) or reconstructed from the order log using [OrderLogMarketDepthBuilder](#).

Data for historical testing must be downloaded and saved in a special [S#](#) format in advance. This can be done manually using [Connectors](#) and [Storage API](#), or by configuring and running the special [Hydra](#) application.

Main stages of historical testing

1. Setting up the data storage

The first step is to create an [IStorageRegistry](#) object through which [HistoryEmulationConnector](#) will access historical data:

```
// storage for accessing historical data
var storageRegistry = new StorageRegistry
{
    // set path to directory with historical data
    DefaultDrive = new LocalMarketDataDrive(HistoryPath.Folder)
};
```

The [LocalMarketDataDrive](#) constructor takes the path to the root directory where history for **all instruments** is stored, not to a directory with a specific instrument. For example, if the HistoryData.zip archive was unpacked into the C:\R\RIZ2@FORTS\ directory, then you need to pass the path C:\ to [LocalMarketDataDrive](#). More details in the [API](#) section.

2. Creating instruments and portfolios

```
// create test instrument for testing
var security = new Security
{
    Id = SecId.Text, // ID of the instrument corresponds to the name of the folder with historical data
    Code = secCode,
    Board = board,
};

// test portfolio
var portfolio = new Portfolio
```

```
{
    Name = "test account",
    BeginValue = 1000000,
};
```

3. Creating the emulation connector

```
// create connector for emulation
var connector = new HistoryEmulationConnector(
    new[] { security },
    new[] { portfolio })
{
    EmulationAdapter =
    {
        Emulator =
        {
            Settings =
            {
                // match order if historical price touched our limit order price
                // By default it's turned off, price should go through the limit order price
                // (more strict testing mode)
                MatchOnTouch = false,

                // commission for trades
                CommissionRules = new ICommissionRule[]
                {
                    new CommissionPerTradeRule { Value = 0.01m },
                }
            }
        },
        UseExternalCandleSource = emulationInfo.UseCandle != null,
        CreateDepthFromOrdersLog = emulationInfo.UseOrderLog,
        CreateTradesFromOrdersLog = emulationInfo.UseOrderLog,
        HistoryMessageAdapter =
        {
            StorageRegistry = storageRegistry,
            // set testing range
            StartDate = startTime,
            StopDate = stopTime,
            OrderLogMarketDepthBuilders =
            {
                {
                    secId,
                    LocalizedStrings.ActiveLanguage == Languages.Russian
                    ? (IOrderLogMarketDepthBuilder)new PlazaOrderLogMarketDepthBuilder(secId)
                    : new ItchOrderLogMarketDepthBuilder(secId)
                }
            }
        },
        // set market time update interval
        MarketTimeChangedInterval = timeFrame,
    };
};
```

4. Subscribing to events and configuring data generation

When connecting, we set up receiving the necessary data depending on the testing parameters:

```
connector.SecurityReceived += (subscr, s) =>
{
    if (s != security)
        return;

    // fill Level1 values
    connector.EmulationAdapter.SendInMessage(level1Info);
};
```

```

// subscribe to necessary data depending on testing settings
if (emulationInfo.UseMarketDepth)
{
    connector.Subscribe(new(DataType.MarketDepth, security));

    // if we need to generate order books
    if (generateDepths || emulationInfo.UseCandle != null)
    {
        // if no historical order book data is available but required by the strategy,
        // use generator based on last prices
        connector.RegisterMarketDepth(new TrendMarketDepthGenerator(connector.GetSecurityId(security))
        {
            Interval = TimeSpan.FromSeconds(1), // order book refresh frequency - 1 sec
            MaxAsksDepth = maxDepth,
            MaxBidsDepth = maxDepth,
            UseTradeVolume = true,
            MaxVolume = maxVolume,
            MinSpreadStepCount = 2,
            MaxSpreadStepCount = 5,
            MaxPriceStepCount = 3
        });
    }
}

if (emulationInfo.UseOrderLog)
{
    connector.Subscribe(new(DataType.OrderLog, security));
}

if (emulationInfo.UseTicks)
{
    connector.Subscribe(new(DataType.Ticks, security));
}

if (emulationInfo.UseLevel1)
{
    connector.Subscribe(new(DataType.Level1, security));
}

// start strategy before emulation begins
strategy.Start();

// start loading historical data
connector.Start();
};

```

5. Creating and configuring the strategy

```

// create trading strategy based on moving averages with periods 80 and 10
var strategy = new SmaStrategy
{
    LongSma = 80,
    ShortSma = 10,
    Volume = 1,
    Portfolio = portfolio,
    Security = security,
    Connector = connector,
    LogLevel = DebugLogCheckBox.IsChecked == true ? LogLevels.Debug : LogLevels.Info,
    // default interval is 1 min, which is excessive for a range of several months
    UnrealizedPnLInterval = ((stopTime - startTime).Ticks / 1000).To<TimeSpan>()
};

// configure the type of data used to build candles
if (emulationInfo.UseCandle != null)
{
    strategy.CandleType = emulationInfo.UseCandle;
}

```

```

if (strategy.CandleType != TimeSpan.FromMinutes(1).TimeFrame())
{
    strategy.BuildFrom = TimeSpan.FromMinutes(1).TimeFrame();
}
}
else if (emulationInfo.UseTicks)
    strategy.BuildFrom = DataType.Ticks;
else if (emulationInfo.UseLevel1)
{
    strategy.BuildFrom = DataType.Level1;
    strategy.BuildField = emulationInfo.BuildField;
}
else if (emulationInfo.UseOrderLog)
    strategy.BuildFrom = DataType.OrderLog;
else if (emulationInfo.UseMarketDepth)
    strategy.BuildFrom = DataType.MarketDepth;

```

6. Visualizing results

To display testing results visually, we subscribe to P&L and position changes:

```

var pnlCurve = equity.CreateCurve(LocalizedStrings.PnL + " " + emulationInfo.StrategyName,
    Colors.Green, Colors.Red, DrawStyles.Area);
var realizedPnlCurve = equity.CreateCurve(LocalizedStrings.PnLRealized + " " +
    emulationInfo.StrategyName, Colors.Black, DrawStyles.Line);
var unrealizedPnlCurve = equity.CreateCurve(LocalizedStrings.PnLUnreal + " " +
    emulationInfo.StrategyName, Colors.DarkGray, DrawStyles.Line);
var commissionCurve = equity.CreateCurve(LocalizedStrings.Commission + " " +
    emulationInfo.StrategyName, Colors.Red, DrawStyles.DashedLine);

strategy.PnLReceived2 += (s, pf, t, r, u, c) =>
{
    var data = equity.CreateData();

    data
        .Group(t)
        .Add(pnlCurve, r - (c ?? 0))
        .Add(realizedPnlCurve, r)
        .Add(unrealizedPnlCurve, u ?? 0)
        .Add(commissionCurve, c ?? 0);

    equity.Draw(data);
};

var posItems = pos.CreateCurve(emulationInfo.StrategyName, emulationInfo.CurveColor,
    DrawStyles.Line);

strategy.PositionReceived += (s, p) =>
{
    var data = pos.CreateData();

    data
        .Group(p.LocalTime)
        .Add(posItems, p.CurrentValue);

    pos.Draw(data);
};

// subscribe to progress updates
connector.ProgressChanged += steps => this.GuiAsync(() => progressBar.Value = steps);

```

7. Starting the test

```

// start emulation

```

```
connector.Connect();
```

Modern implementation of historical testing

In the latest versions of [S#](#), the historical testing example has been significantly modernized and now allows testing strategies using various types of market data:

- Ticks (trades)
- Order books
- Candles of different timeframes
- Order log
- Level1 data (best prices)
- Combinations of different data types

A separate tab with charts and statistics is created for each data type:

```
// create testing modes
_settings = new[]
{
    (
        TicksCheckBox,
        TicksProgress,
        TicksParameterGrid,
        // ticks
        new EmulationInfo
        {
            UseTicks = true,
            CurveColor = Colors.DarkGreen,
            StrategyName = LocalizedStrings.Ticks
        },
        TicksChart,
        TicksEquity,
        TicksPosition
    ),

    (
        TicksAndDepthsCheckBox,
        TicksAndDepthsProgress,
        TicksAndDepthsParameterGrid,
        // ticks + order books
        new EmulationInfo
        {
            UseTicks = true,
            UseMarketDepth = true,
            CurveColor = Colors.Red,
            StrategyName = LocalizedStrings.TicksAndDepths
        },
        TicksAndDepthsChart,
        TicksAndDepthsEquity,
        TicksAndDepthsPosition
    ),

    // other combinations of data types
};
```

This approach allows for visual comparison of strategy performance when using different data sources.

Improved SMA Strategy

The Moving Average (SMA) strategy has been redesigned and now uses a more modern approach to data

subscription and candle processing:

```
protected override void OnStarted2(DateTime time)
{
    base.OnStarted2(time);

    // create subscription to candles of the required type
    var dt = CandleTimeFrame is null
        ? CandleType
        : DataType.Create(CandleType.MessageType, CandleTimeFrame);

    var subscription = new Subscription(dt, Security)
    {
        MarketData =
        {
            IsFinishedOnly = true,
            BuildFrom = BuildFrom,
            BuildMode = BuildFrom is null ? MarketDataBuildModes.LoadAndBuild : MarketDataBuildModes.Build,
            BuildField = BuildField,
        }
    };

    // create indicators
    var longSma = new SMA { Length = LongSma };
    var shortSma = new SMA { Length = ShortSma };

    // subscribe to candles and bind them to indicators
    SubscribeCandles(subscription)
        .Bind(longSma, shortSma, OnProcess)
        .Start();

    // configure display on the chart
    var area = CreateChartArea();

    if (area != null)
    {
        DrawCandles(area, subscription);
        DrawIndicator(area, shortSma, System.Drawing.Color.Coral);
        DrawIndicator(area, longSma);
        DrawOwnTrades(area);
    }

    // configure position protection
    StartProtection(TakeValue, StopValue);
}
```

Candle processing and trading decisions are now separated into a dedicated method:

```
private void OnProcess(ICandleMessage candle, decimal longValue, decimal shortValue)
{
    LogInfo(LocalizedStrings.SmaNewCandleLog, candle.OpenTime, candle.OpenPrice, candle.HighPrice,
        candle.LowPrice, candle.ClosePrice, candle.TotalVolume, candle.SecurityId);

    // check if the candle is completed
    if (candle.State != CandleStates.Finished)
        return;

    // analyze indicator crossover
    var isShortLessThenLong = shortValue < longValue;

    if (_isShortLessThenLong == null)
    {
        _isShortLessThenLong = isShortLessThenLong;
    }
    else if (_isShortLessThenLong != isShortLessThenLong) // crossover occurred
    {
        // if short is less than long - sell, otherwise buy
        var direction = isShortLessThenLong ? Sides.Sell : Sides.Buy;
    }
}
```

```
// calculate volume for opening position or reversal
var volume = Position == 0 ? Volume : Position.Abs().Min(Volume) * 2;

// use the candle's close price
var price = candle.ClosePrice;

if (direction == Sides.Buy)
    BuyLimit(price, volume);
else
    SellLimit(price, volume);

_isShortLessThanLong = isShortLessThanLong;
}
}
```

Additional testing settings

Extended settings for testing are available in [S#](#), including:

- Generation of order books with specified parameters
- Commission settings
- Price slippage settings
- Execution delay emulation

These settings are described in more detail in the [Testing Settings](#) section.

Optimization

To optimize the [backtesting](#) process it is possible to use parallel computation based on multiple threads. On computers with multiple cores or processors this will reduce the overall testing time due to concurrent execution of multiple operations.

Using multiple threads increases the memory consumption (approximately as many times as threads created, if each thread uses its time range in the history). Therefore, the use of parallel testing without sufficient memory, will not allow a significant increase in performance, and even reduce it.

Example of moving averages strategy testing in several threads

1. The SampleHistoryTesting example, described in the [Historical data](#) testing section, is taken as the basis. This example is a modified conventional testing for testing with optimization through fitting of the optimum value of the moving averages lengths: *[image: sample history test parallel]*
2. Creating a few settings of the moving averages lengths (the first value is responsible for the length of the longest, the second for the length of the shortest, and the third for the equity curve color):

```
var periods = new[]
{
    new Tuple<int, int, Color>(80, 10, Colors.DarkGreen),
    new Tuple<int, int, Color>(70, 8, Colors.Red),
    new Tuple<int, int, Color>(60, 6, Colors.DarkBlue)
};
```

3. Creating an instance of the storage, the instrument, the message to set the values of Level1 and the portfolio:

```
// storage to historical data
var storageRegistry = new StorageRegistry
{
    // set historical path
    DefaultDrive = new LocalMarketDataDrive(HistoryPath.Folder)
};
var timeframe = TimeSpan.FromMinutes(5);
// create test security
var security = new Security
{
    Id = "ESM2@NYSE", // sec id has the same name as folder with historical data
    Code = "ESM2",
    Name = "ES-12.12",
    Board = ExchangeBoard.Nyse,
};
var startTime = new DateTime(2012, 10, 1);
var stopTime = new DateTime(2012, 10, 31);
var level1Info = new Level1ChangeMessage
{
    SecurityId = security.ToSecurityId(),
    ServerTime = startTime,
}
.TryAdd(Level1Fields.PriceStep, 10m)
.TryAdd(Level1Fields.StepPrice, 6m)
.TryAdd(Level1Fields.MinPrice, 10m)
.TryAdd(Level1Fields.MaxPrice, 1000000m)
.TryAdd(Level1Fields.MarginBuy, 10000m)
.TryAdd(Level1Fields.MarginSell, 10000m);
// test portfolio
var portfolio = new Portfolio
```



```
{
    Name = "test account",
    BeginValue = 1000000,
};
```

4. Creating a unified connector [BatchEmulation](#), which will contain all created in the next step [HistoryEmulationConnector](#):

```
// create backtesting connector
var batchEmulation = new BatchEmulation(new[] { security }, new[] { portfolio }, storageRegistry)
{
    EmulationSettings =
    {
        MarketTimeChangedInterval = timeFrame,
        StartTime = startTime,
        StopTime = stopTime,
        // count of parallel testing strategies
        BatchSize = periods.Length,
    }
};
```

5. Next, the subscription to the unified gateway and the connector events is performed, testing parameters are configured, as well as the strategy is created for each period.

```
// handle historical time for update ProgressBar
batchEmulation.ProgressChanged += (curr, total) => this.GuiAsync(() => TestingProcess.Value = total);
batchEmulation.StateChanged += (oldState, newState) =>
{
    if (batchEmulation.State != EmulationStates.Stopped)
        return;
    this.GuiAsync(() =>
    {
        if (batchEmulation.IsFinished)
        {
            TestingProcess.Value = TestingProcess.Maximum;
            MessageBox.Show(this, LocalizedStrings.Str3024.Put(DateTime.Now - _startEmulationTime));
        }
        else
            MessageBox.Show(this, LocalizedStrings.cancelled);
    });
};
// get emulation connector
var connector = batchEmulation.EmulationConnector;
logManager.Sources.Add(connector);
connector.SecurityReceived += (sub, s) =>
{
    if (s != security)
        return;
    // fill level1 values
    connector.SendInMessage(level1Info);
    connector.MarketDataAdapter.SendInMessage(new GeneratorMessage
    {
        IsSubscribe = true,
        Generator = new RandomWalkTradeGenerator(new SecurityId { SecurityCode = security.Code })
        {
            Interval = TimeSpan.FromSeconds(1),
            MaxVolume = maxVolume,
            MaxPriceStepCount = 3,
            GenerateOriginSide = true,
            MinVolume = minVolume,
            RandomArrayLength = 99,
        }
    });
};
TestingProcess.Maximum = 100;
TestingProcess.Value = 0;
```

```

_startEmulationTime = DateTime.Now;
var strategies = periods
    .Select(period =>
    {
        ...

        // create strategy based SMA
        var strategy = new SmaStrategy(series, new SimpleMovingAverage { Length = period.Item1 }, new
SimpleMovingAverage { Length = period.Item2 })
        {
            Volume = 1,
            Security = security,
            Portfolio = portfolio,
            Connector = connector,
            // by default interval is 1 min,
            // it is excessively for time range with several months
            UnrealizedPnLInterval = ((stopTime - startTime).Ticks / 1000).To<TimeSpan>()
        };
        ...
        var curveItems = Curve.CreateCurve(LocalizedStrings.Str3026Params.Put(period.Item1,
period.Item2), period.Item3, DrawStyles.Line);
        strategy.PnLChanged += () =>
        {
            var data = new EquityData
            {
                Time = strategy.CurrentTime,
                Value = strategy.PnL,
            };
            this.GuiAsync(() => curveItems.Add(data));
        };
        Stat.AddStrategies(new[] { strategy });
        return strategy;
    });

```

6. The testing starts:

```

// start emulation
batchEmulation.Start(strategies, periods.Length);

```

Random data

Random data

Testing on random data is a special kind of testing. It is not intended to search for the optimal parameters. Instead, such testing can detect errors in the code by exposing the trading algorithm to various exchange scenarios.

As a rule, only a certain set of scenarios is used in the algorithm development. Therefore, when there is a special situation, the algorithm may not react right or throw exceptions. For example, the following situations may occur:

- The strategy works with the [TimeFrameCandleMessage](#) candles and expects at each iteration, that there will always be a candle for the requested period of time. A period began when there was not a single trade, and a candle was not created. As a result, in the absence of proper handling, the [NullReferenceException](#) will be thrown and the strategy will stop.
- The strategy works with non-liquid instrument and uses [IOrderBookMessage](#). The strategy expects that the order book is always filled. At some time, the order book is half filled (for example, there are bids, but no offers). If the strategy does not expect this situation, then either it will register an order incorrectly, or it will throw an exception and the strategy will stop.
- The strategy calculates the price levels. The code is written in such a way that the strategy awaits when levels set in advance will be broken through. If the levels are calculated and are set not correctly, then they will never be broken through, or always only one of them will be broken through. As a result, either the strategy will not perform any trades, or these trades will lose money.

For these and for many other scenarios of the exchange work, which is impossible to predict in advance, the [S#](#) provides testing on random data, which will be able to generate the maximum number of conditions at the short range due to its uniform uniqueness.

Testing on random data of moving averages strategy

1. The SampleRandomEmulation example ([../Samples/Testing/SampleRandomEmulation](#)) is almost identical to the SampleHistoryTesting example (its description can be found in the [testing on historical data](#) section) due to the use of the unified [HistoryEmulationConnector](#) class. But, unlike the [testing on historical data](#), testing on random data the market data are not loaded and are generated "on the fly". Therefore, two random data generators are added to the example: for the order book and for the tick trades. In SampleHistoryTesting only one generator is used - for the order book, as there is no history stored.

```
_connector.MarketDataAdapter.SendInMessage(new GeneratorMessage
{
    IsSubscribe = true,
    Generator = new RandomWalkTradeGenerator(new SecurityId { SecurityCode = security.Code })
    {
        Interval = TimeSpan.FromSeconds(1),
        MaxVolume = maxVolume,
        MaxPriceStepCount = 3,
        GenerateOriginSide = true,
        MinVolume = minVolume,
        RandomArrayLength = 99,
    }
});
_connector.SubscribeMarketDepth(new TrendMarketDepthGenerator(_connector.GetSecurityId(security))
{ GenerateDepthOnEachTrade = false });
```

2. The result of the example work is as follows: *[image: sampleemulationtest]*

Real-Time Market Data Testing

Real-time market data testing involves trading with an actual connection to the exchange ("live" quotes), but without placing real orders on the exchange. All registered orders are intercepted, and their execution is emulated based on market order books. Such testing can be useful, for example, when developing a trading simulator, or when checking a trading algorithm over a short period with real quotes.

To emulate trading with real data, you need to use [RealTimeEmulationTrader<TAdapter>](#), which acts as a "wrapper" for a specific trading system connector ([Binance](#), [Tinkoff](#), etc.).

Creating an Emulation Connector

To create an emulation connector, first create a regular connector to receive market data, and then create an emulation connector based on it:

```
// Create a regular connector for receiving market data
private readonly Connector _realConnector = new();

// Create an emulation connector
_emuConnector = new RealTimeEmulationTrader<IMessageAdapter>(_realConnector.Adapter, _realConnector,
_emuPf, false);

// Configure emulation parameters
var settings = _emuConnector.EmulationAdapter.Emulator.Settings;
settings.TimeZone = TimeHelper.Est;
settings.ConvertTime = true;
```

For trade emulation, you need to use a special portfolio:

```
private readonly Portfolio _emuPf = Portfolio.CreateSimulator();
```

Subscribing to Events

Like a regular connector, the emulation connector generates events when receiving market data and executing transactions:

```
// Subscribe to connector events
_emuConnector.Connected += () =>
{
    // update gui labels
    this.GuiAsync(() => { ChangeConnectStatus(true); });
};

_emuConnector.Disconnected += () =>
{
    // update gui labels
    this.GuiAsync(() => { ChangeConnectStatus(false); });
};

_emuConnector.ConnectionError += error => this.GuiAsync(() =>
{
    // update gui labels
    ChangeConnectStatus(false);
    MessageBox.Show(this, error.ToString(), LocalizedStrings.ErrorConnection);
});

_emuConnector.OrderBookReceived += OnDepth;
```

```

_emuConnector.PositionReceived += (sub, p) => PortfolioGrid.Positions.TryAdd(p);
_emuConnector.OwnTradeReceived += (s, t) => TradeGrid.Trades.TryAdd(t);
_emuConnector.OrderReceived += (s, o) =>
{
    if (!_firstTimeOrders.Add(o))
        return;

    _bufferOrders.Add(o);
    OrderGrid.Orders.Add(o);
};

// Subscribe to order registration errors
_emuConnector.OrderRegisterFailReceived += (s, f) => OrderGrid.AddRegistrationFail(f);

_emuConnector.CandleReceived += (s, candle) =>
{
    if (s == _candlesSubscription)
        _buffer.Add(candle);
};

```

Subscribing to Market Data

To work with market data, you need to subscribe to the appropriate data types:

```

// Subscribe to order books, ticks, and Level1 for the emulation connector
_emuConnector.Subscribe(new(DataType.MarketDepth, security));
_emuConnector.Subscribe(new(DataType.Ticks, security));
_emuConnector.Subscribe(new(DataType.Level1, security));

// Subscribe to order books for the real connector (needed for emulation)
_realConnector.Subscribe(new(DataType.MarketDepth, security));

// Subscribe to candles
_candlesSubscription = new(CandleDataTypeEdit.DataType, security)
{
    From = DateTimeOffset.UtcNow - TimeSpan.FromDays(10),
};
_emuConnector.Subscribe(_candlesSubscription);

```

Order Registration and Management

Orders are registered through the emulation connector similar to a regular connector:

```

// Order registration
_emuConnector.RegisterOrder(order);

// Order cancellation
_emuConnector.CancelOrder(order);

// Order replacement
_emuConnector.ReRegisterOrder(order, newPrice, order.Balance);

```

Configuring Emulation Parameters

You can use the [MarketEmulatorSettings](#) property to configure emulation parameters:

```

var settings = _emuConnector.EmulationAdapter.Emulator.Settings;

// Set timezone
settings.TimeZone = TimeHelper.Est;

// Convert time

```

```
settings.ConvertTime = true;

// Match orders on price touch
settings.MatchOnTouch = false;

// Emulate order execution latency
settings.Latency = TimeSpan.FromMilliseconds(100);
```

Interface Example

The SampleRealTimeEmulation example demonstrates the ability to simultaneously display data from both the real connector and the emulation connector:

[image: sample realtime emulation]

The application interface contains the following elements:

- Charts for displaying candles and orders
- Order and own trade tables
- Real market and emulation order books
- Controls for creating and canceling orders

Advantages and Limitations

Real-time market data testing has the following advantages:

- Using real market data without financial risks
- Testing algorithms in conditions very close to real trading
- Ability to compare results with the real market in real time

Limitations:

- Testing speed is limited by the rate of real data
- Inability to test on historical periods
- Dependence on the quality and completeness of the received market data

Extended settings

Testing Settings

This section describes the main settings of [HistoryEmulationConnector](#) for testing trading strategies.

Basic Emulator Settings

- [MarketTimeChangedInterval](#) - interval for time change event arrival. If trade generators are used, trades will be generated with this frequency. The default is 1 minute.
- [MarketEmulatorSettings.Latency](#) - minimum delay value for submitted orders. The default is `TimeSpan.Zero`, which means instant acceptance of submitted orders by the exchange.
- [MarketEmulatorSettings.MatchOnTouch](#) - satisfy orders if the price "touches" the level (this assumption is sometimes too "optimistic" and should be turned off for realistic testing). If disabled, limit orders will be fulfilled only if the price "passes through them" by at least 1 step. This option works in all modes except order log mode. It's disabled by default.

Market Data Subscriptions

For correct strategy testing, it's necessary to set up subscriptions to the required market data types. Even if the strategy is tested on candles, for correct trade emulation, a subscription to tick trades is required:

```
// Create a subscription to tick trades
var tickSubscription = new Subscription(DataType.Ticks, security);
_connector.Subscribe(tickSubscription);
```

If the strategy requires order book data:

```
// Create a subscription to order books
var depthSubscription = new Subscription(DataType.MarketDepth, security);
_connector.Subscribe(depthSubscription);
```

Order Book Generation for Testing

If there are no historical order books, but they are needed for strategy testing, you can enable order book generation:

```
// Create an order book generator with trend behavior
var mdGenerator = new TrendMarketDepthGenerator(security.ToSecurityId());

// Send a subscription message to the generator
_connector.MarketDataAdapter.SendInMessage(new GeneratorMessage
{
    IsSubscribe = true,
    Generator = mdGenerator
});
```

Order Book Generator Settings

- Order book update interval ([MarketDataGenerator.Interval](#)) - updates cannot be more frequent than the arrival of tick trades, as order books are generated before each trade:

```
mdGenerator.Interval = TimeSpan.FromSeconds(1);
```

- Order book depth ([MarketDepthGenerator.MaxBidsDepth](#) and [MarketDepthGenerator.MaxAsksDepth](#))
- the deeper, the slower the testing:

```
mdGenerator.MaxAsksDepth = 1;
mdGenerator.MaxBidsDepth = 1;
```

- To get a realistic level volume in the order book, you can use the [MarketDepthGenerator.UseTradeVolume](#) option, which takes the volumes for the best quotes from the trade volume being generated:

```
mdGenerator.UseTradeVolume = true;
```

- Level volume range ([MarketDataGenerator.MinVolume](#) and [MarketDataGenerator.MaxVolume](#)):

```
mdGenerator.MinVolume = 1;
mdGenerator.MaxVolume = 1;
```

- Spread settings - the minimum generated spread equals [Security.PriceStep](#). It is recommended not to generate a spread between the best quotes greater than 5 price steps, so that when generating from candles, the spread doesn't become too wide:

```
mdGenerator.MinSpreadStepCount = 1;
mdGenerator.MaxSpreadStepCount = 5;
```

Comprehensive Testing Configuration Example

```
// Create a historical connection
var connector = new HistoryEmulationConnector();

// Configure basic parameters
connector.MarketTimeChangedInterval = TimeSpan.FromSeconds(10);
connector.EmulationAdapter.Emulator.Settings.Latency = TimeSpan.FromMilliseconds(100);
connector.EmulationAdapter.Emulator.Settings.MatchOnTouch = false;

// Load historical data
var storage = new StorageRegistry();
var security = new Security { Id = "AAPL", PriceStep = 0.01m };

// Create a subscription to candles
var candleSubscription = new Subscription(
    DataType.TimeFrame(TimeSpan.FromMinutes(5)),
    security)
{
    MarketData =
    {
        From = DateTime.Today.Subtract(TimeSpan.FromDays(30)),
        To = DateTime.Today,
        BuildMode = MarketDataBuildModes.Load
    }
};
connector.Subscribe(candleSubscription);

// Create a subscription to ticks for correct emulation
var tickSubscription = new Subscription(DataType.Ticks, security);
connector.Subscribe(tickSubscription);

// Configure order book generation
var mdGenerator = new TrendMarketDepthGenerator(security.ToSecurityId())
{
    Interval = TimeSpan.FromSeconds(1),
    MaxAsksDepth = 5,
    MaxBidsDepth = 5,
```



```

    UseTradeVolume = true,
    MinVolume = 1,
    MaxVolume = 100,
    MinSpreadStepCount = 1,
    MaxSpreadStepCount = 5
};

connector.MarketDataAdapter.SendInMessage(new GeneratorMessage
{
    IsSubscribe = true,
    Generator = mdGenerator
});

// Subscribe to data reception
connector.CandleReceived += OnCandleReceived;
connector.TickTradeReceived += OnTickReceived;
connector.OrderBookReceived += OnOrderBookReceived;

// Start testing
connector.Connect();

```

Event Handling

```

private void OnCandleReceived(Subscription subscription, ICandleMessage candle)
{
    // Processing received candles
    Console.WriteLine($"Candle: {candle.OpenTime}, O:{candle.OpenPrice}, H:{candle.HighPrice}, L:
{candle.LowPrice}, C:{candle.ClosePrice}");
}

private void OnTickReceived(Subscription subscription, ITickTradeMessage tick)
{
    // Processing received ticks
    Console.WriteLine($"Tick: {tick.ServerTime}, Price: {tick.Price}, Volume: {tick.Volume}");
}

private void OnOrderBookReceived(Subscription subscription, IOrderBookMessage orderBook)
{
    // Using extension methods for IOrderBookMessage
    var bestBid = orderBook.GetBestBid();
    var bestAsk = orderBook.GetBestAsk();
    var spreadMiddle = orderBook.GetSpreadMiddle(Security.PriceStep);

    // Processing received order books
    Console.WriteLine($"Order Book: {orderBook.ServerTime}, Best Bid: {bestBid?.Price}, Best Ask:
{bestAsk?.Price}, Middle of Spread: {spreadMiddle}");

    // Getting price by order side
    var bidPrice = orderBook.GetPrice(Sides.Buy);
    var askPrice = orderBook.GetPrice(Sides.Sell);

    Console.WriteLine($"Bid Price: {bidPrice}, Ask Price: {askPrice}");
}

```

Working with Order Book Data

When working with order books as `IOrderBookMessage`, you can use the following extension methods:

```

// Get the best bid
var bestBid = orderBook.GetBestBid();

// Get the best ask
var bestAsk = orderBook.GetBestAsk();

// Get the middle of the spread

```

```
var spreadMiddle = orderBook.GetSpreadMiddle(Security.PriceStep);

// Get the price by order side
var price = orderBook.GetPrice(Sides.Buy); // or Sides.Sell, or null for the middle of the spread
```

When working with Level1 data, you can also get the middle of the spread:

```
// Get the middle of the spread from a Level1 message
var spreadMiddle = level1.GetSpreadMiddle(Security.PriceStep);
```

These extension methods simplify access to order book data and allow you to write cleaner and more understandable code when working with exchange quotes.

Data Export

Data Export

[S#](#) implements a market data export subsystem supporting various formats. All exporters inherit from the base class [BaseExporter](#) and support a unified asynchronous interface.

BaseExporter

The base abstract class [BaseExporter](#) defines the common contract for all exporters:

- **DataType** — the type of data being exported (ticks, candles, order book, etc.).
- **Encoding** — encoding (UTF-8 by default).
- **Export<T>(IEnumerable<T>, CancellationToken)** — the main export method. Returns `Task<(int count, DateTime? lastTime)>` — the number of exported records and the time of the last record.

The method automatically routes data to type-specific handlers for: [QuoteChangeMessage](#), [Level1ChangeMessage](#), [ExecutionMessage](#) (ticks, order log, transactions), [CandleMessage](#), [NewsMessage](#), [SecurityMessage](#), [PositionChangeMessage](#), [IndicatorValue](#), and [BoardStateMessage](#).

Exporter Types

1. TextExporter — CSV/Text Export

[TextExporter](#) exports data to text format using SmartFormat templates.

- **Constructor:** (DataType dataType, Stream stream, string template, string header)
- Templates use SmartFormat syntax, for example: `{ServerTime:default:yyyyMMdd};{TradePrice};{TradeVolume}`

```
await using var stream = File.Create("trades.csv");
var exporter = new TextExporter(DataType.Ticks, stream,
    "{ServerTime:default:yyyyMMdd};{TradePrice};{TradeVolume}",
    "Date;Price;Volume");

var (count, lastTime) = await exporter.Export(tickMessages, token);
```

2. JsonExporter — JSON Export

[JsonExporter](#) saves data in JSON format.

- **Constructor:** (DataType dataType, Stream stream)
- **Indent** — indented formatting (default true).

```
await using var stream = File.Create("candles.json");
var exporter = new JsonExporter(
    DataType.CandleTimeFrame(TimeSpan.FromMinutes(5)), stream);
await exporter.Export(candleMessages, token);
```

3. XmlExporter — XML Export

[XmlExporter](#) saves data in XML format.

- **Constructor:** (DataType dataType, Stream stream)

- **Indent** — indented formatting (default true).

```
await using var stream = File.Create("candles.xml");
var exporter = new XmlExporter(
    DataType.CandleTimeFrame(TimeSpan.FromMinutes(5)), stream);
await exporter.Export(candleMessages, token);
```

4. ExcelExporter — Excel Export

[ExcelExporter](#) exports data to Excel spreadsheets.

- **Constructor:** (IExcelWorkerProvider provider, DataType dataType, Stream stream, Action broken)
- Maximum row count: 1,048,576 (Excel format limitation).

```
await using var stream = File.Create("data.xlsx");
var exporter = new ExcelExporter(excelProvider, DataType.Ticks, stream,
    () => { /* handle interruption */ });
await exporter.Export(tickMessages, token);
```

5. DatabaseExporter — Database Export

[DatabaseExporter](#) saves data to a database via LinqToDB.

- **Constructor:** (IDatabaseProvider dbProvider, DataType dataType, DatabaseConnectionPair connection, decimal? priceStep, decimal? volumeStep)
- **BatchSize** — batch size for records (default 50).
- **CheckUnique** — check record uniqueness (default false).
- **DropExisting** — delete existing data before export (default false).

```
var exporter = new DatabaseExporter(
    dbProvider, DataType.Ticks, dbConnection)
{
    BatchSize = 100,
    CheckUnique = true
};
await exporter.Export(tickMessages, token);
```

6. StockSharpExporter — StockSharp Native Format

[StockSharpExporter](#) saves data in the internal StockSharp storage format.

- **Constructor:** (DataType dataType, IStorageRegistry storageRegistry, IMarketDataDrive drive, StorageFormats format)
- **BatchSize** — batch size for records (default 50).

```
var exporter = new StockSharpExporter(
    DataType.Ticks, storageRegistry, drive, StorageFormats.Binary);
await exporter.Export(tickMessages, token);
```

TemplateTxtRegistry — Text Template Registry

The [TemplateTxtRegistry](#) class contains predefined templates for exporting various data types via [TextExporter](#):

- **TemplateTxtTick** — template for tick data.

- **TemplateTxtDepth** — template for order books.
- **TemplateTxtCandle** — template for candles.
- **TemplateTxtLevel1** — template for Level1 data.
- **TemplateTxtOrderLog** — template for order log.
- **TemplateTxtTransaction** — template for transactions.
- **TemplateTxtSecurity** — template for instruments.
- **TemplateTxtNews** — template for news.

Templates can be customized or replaced as needed. The registry implements [IPersistable](#) and can be saved/loaded from settings.

See Also

[Data Import](#)

[Data Storage](#)

Data Import

Data Import

[S#](#) implements a market data import subsystem from CSV files. The main classes are located in the `StockSharp.Algo.Import` namespace.

CsvParser — Base Parser

The [CsvParser](#) class performs CSV file parsing and converts rows into [S#](#) messages.

- **Constructor:** (`DataType dataType, IEnumerable<FieldMapping> fields`)
- **ColumnSeparator** — column separator (default `" ; "`).
- **LineSeparator** — line separator (default `CRLF`).
- **SkipFromHeader** — number of rows to skip from the beginning of the file (default `0`).
- **IgnoreNonIdSecurities** — ignore rows with unrecognized instruments (default `true`).
- **Parse(Stream)** — parsing method, returns `IAsyncEnumerable<Message>`.

```
var fields = FieldMappingRegistry.CreateFields(DataType.Ticks);
var parser = new CsvParser(DataType.Ticks, fields)
{
    ColumnSeparator = ";",
    SkipFromHeader = 1
};

await using var stream = File.OpenRead("trades.csv");

await foreach (var msg in parser.Parse(stream))
{
    // process each message
}
```

CsvImporter — Import with Storage Saving

The [CsvImporter](#) class extends [CsvParser](#), adding the ability to automatically save data to a market data storage.

- **Constructor:** (`DataType dataType, IEnumerable<FieldMapping> fields, ISecurityStorage securityStorage, IExchangeInfoProvider exchangeInfoProvider, Func<SecurityId, IMarketDataStorage> getStorage`)
- **Import(Stream, Action<int> progress, CancellationToken)** — performs the import and returns `ValueTask<(int count, DateTime? lastTime)>`.
- **UpdateDuplicateSecurities** — whether to update duplicate instruments (default `false`).
- **SecurityUpdated** — event raised when an instrument is updated.

```
var fields = FieldMappingRegistry.CreateFields(DataType.Ticks);

var importer = new CsvImporter(
    DataType.Ticks,
    fields,
    securityStorage,
    exchangeInfoProvider,
    secId => storageRegistry.GetStorage(secId, DataType.Ticks))
```

```

{
    ColumnSeparator = ";",
    SkipFromHeader = 1
};

await using var stream = File.OpenRead("trades.csv");
var (count, lastTime) = await importer.Import(
    stream,
    p => Console.WriteLine($"Progress: {p}%"),
    token);

Console.WriteLine($"Imported {count} records, last: {lastTime}");

```

FieldMapping — Field Descriptions

The [FieldMapping](#) class describes the mapping between CSV file columns and message properties.

Main properties:

- **Name** — field name in the message.
- **DisplayName** — display name.
- **Type** — value type.
- **Order** — column index in the file (starting from 0).
- **IsRequired** — whether the field is required.
- **Format** — parsing format (e.g., date format).
- **DefaultValue** — default value.
- **ZeroAsNull** — whether to interpret zero values as null.

For custom value transformations, use [FieldMappingValue](#). For example, you can define a mapping of text values to enumerations:

```

var sideField = fields.First(f => f.Name == "Side");
sideField.Order = 3;
sideField.Values.Add(new FieldMappingValue
{
    ValueFrom = "B",
    ValueTo = Sides.Buy
});
sideField.Values.Add(new FieldMappingValue
{
    ValueFrom = "S",
    ValueTo = Sides.Sell
});

```

FieldMappingRegistry — Standard Fields Registry

The static class [FieldMappingRegistry](#) provides a method for creating a standard set of fields:

- **CreateFields(DataType)** — returns a list of [FieldMapping](#) for the specified data type.

Supported data types: ticks, candles, order books, Level1, order log, transactions, instruments, news, and positions.

ImportSettings — Import Settings

The [ImportSettings](#) class combines all import parameters into a single configuration object:

- **DataType** — the type of data being imported.
- **FileName** — file path.
- **Directory** — directory for file search.
- **FileMask** — file search mask (e.g., *.csv).
- **ColumnSeparator** — column separator.
- **SkipFromHeader** — number of rows to skip.
- **SelectedFields** — fields selected for import.
- **UpdateDuplicateSecurities** — whether to update duplicate instruments.

Helper methods:

- **GetFiles(IFileSystem)** — get a list of files matching the mask.
- **FillParser(CsvParser)** — populate parser settings.
- **FillImporter(CsvImporter)** — populate importer settings.

```
var settings = new ImportSettings
{
    DataType = DataType.Ticks,
    FileName = "trades.csv",
    ColumnSeparator = ";",
    SkipFromHeader = 1
};

var fields = FieldMappingRegistry.CreateFields(settings.DataType);
// configure column order
fields[0].Order = 0; // SecurityId
fields[1].Order = 1; // Date
fields[2].Order = 2; // Price
fields[3].Order = 3; // Volume

var importer = new CsvImporter(
    settings.DataType,
    fields,
    securityStorage,
    exchangeInfoProvider,
    secId => storageRegistry.GetStorage(secId, settings.DataType));

settings.FillImporter(importer);

await using var stream = File.OpenRead(settings.FileName);
var (count, lastTime) = await importer.Import(stream, p => { }, token);
```

See Also

[Data Export](#)

[Data Storage](#)

Analytics Scripts

Analytics Scripts

[S#](#) implements an analytics scripts subsystem that allows you to perform arbitrary analysis of market data with result visualization. The classes are located in the `StockSharp.Algo.Analytics` namespace.

IAAnalyticsScript — Main Interface

The [IAAnalyticsScript](#) interface defines a single method:

```
Task Run(
    ILogReceiver logs,
    IAnalyticsPanel panel,
    SecurityId[] securities,
    DateTime from,
    DateTime to,
    IStorageRegistry storage,
    IMarketDataDrive drive,
    StorageFormats format,
    DataType dataType,
    CancellationToken cancellationToken);
```

Parameters:

- **logs** — log receiver for outputting diagnostic messages.
- **panel** — panel for displaying analysis results.
- **securities** — array of instruments to analyze.
- **from** / **to** — time range.
- **storage** — market data storage registry.
- **drive** — data source.
- **format** — data storage format.
- **dataType** — type of data to analyze.
- **cancellationToken** — cancellation token.

IAAnalyticsPanel — Results Panel

The [IAAnalyticsPanel](#) interface provides methods for creating various visualizations:

- **CreateGrid(params string[] columns)** — creates a table [IAAnalyticsGrid](#) with the specified columns.
- **CreateChart<X, Y>()** — creates a two-dimensional chart [IAAnalyticsChart](#).
- **CreateChart<X, Y, Z>()** — creates a three-dimensional chart [IAAnalyticsChart](#).
- **DrawHeatmap(string[] xTitles, string[] yTitles, double[,] data)** — draws a heatmap.
- **Draw3D(string[] xTitles, string[] yTitles, data, xTitle, yTitle, zTitle)** — draws a 3D visualization.

IAAnalyticsChart — Charts

The [IAAnalyticsChart](#) interface provides a method for adding data series:

```
void Append(string title, IEnumerable<X> xValues, IEnumerable<Y> yValues,
    DrawStyles style, Color? color = null);
```

Available drawing styles ([DrawStyles](#)):

- **Line** — line chart.
- **DashedLine** — dashed line.
- **Histogram** — histogram.
- **Bubble** — bubble chart.

IAalyticsGrid — Tables

The [IAalyticsGrid](#) interface allows displaying tabular data:

- **SetSort(string column, bool ascending)** — set sorting by column.
- **SetRow(params object[] values)** — add a data row.

Built-in Scripts

The `StockSharp.Algo.Analytics.CSharp` package includes ready-made scripts:

- **IndicatorScript** — calculates and visualizes indicators on a chart.
- **ChartDrawScript** — demonstrates building various chart types.
- **PriceVolumeScript** — analyzes volume distribution across price levels.

Example: Custom Analytics Script

Below is an example script that loads candles for a list of instruments and displays close prices on a line chart:

```
public class MyAnalyticsScript : IAnalyticsScript
{
    public async Task Run(ILogReceiver logs, IAnalyticsPanel panel,
        SecurityId[] securities, DateTime from, DateTime to,
        IStorageRegistry storage, IMarketDataDrive drive,
        StorageFormats format, DataType dataType,
        CancellationToken cancellationToken)
    {
        // create a two-dimensional chart
        var chart = panel.CreateChart<DateTime, decimal>();

        foreach (var secId in securities)
        {
            // get candle storage
            var candleStorage = storage.GetCandleMessageStorage(
                secId, dataType, drive, format);

            // load data for the period
            var candles = await candleStorage
                .LoadAsync(from, to)
                .WithCancellation(cancellationToken)
                .ToArrayAsync(cancellationToken);

            if (candles.Length == 0)
            {
                logs.AddWarningLog($"No data for {secId}");
                continue;
            }
        }
    }
}
```

```

        // add series to the chart
        chart.Append(secId.ToString(),
            candles.Select(c => c.OpenTime.UtcDateTime),
            candles.Select(c => c.ClosePrice),
            DrawStyles.Line);

        logs.AddInfoLog($"{secId}: loaded {candles.Length} candles");
    }
}
}

```

Example: Volume Table

```

public class VolumeTableScript : IAnalyticsScript
{
    public async Task Run(ILogReceiver logs, IAnalyticsPanel panel,
        SecurityId[] securities, DateTime from, DateTime to,
        IStorageRegistry storage, IMarketDataDrive drive,
        StorageFormats format, DataType dataType,
        CancellationToken cancellationToken)
    {
        var grid = panel.CreateGrid("Instrument", "Total Candles",
            "Total Volume", "Average Volume");
        grid.SetSort("Total Volume", false);

        foreach (var secId in securities)
        {
            var candleStorage = storage.GetCandleMessageStorage(
                secId, dataType, drive, format);

            var candles = await candleStorage
                .LoadAsync(from, to)
                .WithCancellation(cancellationToken)
                .ToArrayAsync(cancellationToken);

            if (candles.Length == 0)
                continue;

            var totalVolume = candles.Sum(c => c.TotalVolume);
            var avgVolume = totalVolume / candles.Length;

            grid.SetRow(secId.ToString(), candles.Length,
                totalVolume, avgVolume);
        }
    }
}

```

See Also

[Indicators](#)

[Data Storage](#)

Graphical controls

Graphical user interface

Graphical components of S#

S# includes a large number of its own graphical components. The components are placed in the [StockSharp.Xaml](#), [StockSharp.Xaml.Charting](#) and [StockSharp.Xaml.Diagram](#) namespaces.

S# has a variety of controls for:

- search and select data (instruments, portfolios, addresses);
- the orders creation;
- display exchange and other information (trades, orders, transactions, order books, logs, etc.);
- charts plotting.

To access the S# graphical controls in the XAML code it is necessary to define the aliases for the corresponding namespace and to use these aliases in the XAML code. How to do this is shown in the following example:

```
<Window x:Class="SampleSmartSMA.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:loc="clr-namespace:StockSharp.Localization;assembly=StockSharp.Localization"
  xmlns:sx="clr-namespace:StockSharp.Xaml;assembly=StockSharp.Xaml"
  xmlns:charting="http://schemas.stocksharp.com/xaml"
  Title="{x:Static loc:LocalizedStrings.XamlStr570}" Height="700" Width="900">

  <Grid>
  </Grid>
</Window>
```

Charts

Charts

[S#](#) provides convenient components for charting. These components are gathered in the namespace [StockSharp.Xaml.Charting](#).

The key concept in the graphics library is the notion of a *chart*. A *chart* is a container for other elements that are used in constructing charts. There are several types of *charts* in [S#](#):

- [Chart](#) - A graphical component for displaying stock charts.
- [ChartPanel](#) - An advanced graphical component for displaying stock charts.
- [EquityCurveChart](#) - A graphical component for displaying equity curves.
- [BoxChart](#) - A chart representing volumes as a grid of numbers.
- [ClusterChart](#) - A chart displaying volumes as clusters with histograms.
- [OptionPositionChart](#) - A graphical component showing option positions and "Greeks" relative to the underlying asset. See [OptionPositionChart](#).

Additionally, [S#](#) includes two types of charts for volume analysis: [BoxChart](#) and [ClusterChart](#).

The following figure shows the main elements of the graphical component.

[image: Gui ChartElements]

Graphical Component Elements

IChart

[IChart](#) is the basic interface for all types of charts. It includes methods for adding and removing "child" elements, properties to customize the appearance and drawing methods of the component, as well as a method to draw the charts themselves. A *chart* can contain several areas ([IChartArea](#)) for plotting (see figure). [Chart](#) also includes an *OverView* preview area (see figure). In this area, you can select the chart viewing zone using sliders. Additionally, you can scroll and zoom the chart using the dragging of the [IChartArea](#), the X-axis, and the mouse wheel.

Key Properties and Methods of [IChart](#)

- [IChart.Areas](#) - List of [IChartArea](#) areas.
- [IThemeableChart.ChartTheme](#) - Theme of the component.
- [IChart.IndicatorTypes](#) - List of indicators that can be displayed on the chart.
- [IChart.CrossHair](#) - Enable/disable the display of crosshairs.
- [IChart.CrossHairAxisLabels](#) - Enable/disable the display of axis labels at crosshairs.
- [IChart.IsAutoRange](#) - Enable/disable automatic X-axis scaling.
- [IChart.IsAutoScroll](#) - Enable/disable auto-scrolling on the X-axis.
- [IChart.ShowLegend](#) - Enable/disable the display of the legend.
- [IChart.ShowOverview](#) - Enable/disable the display of the *OverView* preview area.
- [IChart.AddArea](#) - Add an [IChartArea](#).
- `[IChart.AddElement](xref:StockSharp.Charting.IChart.AddElement(StockSharp.Charting.IChartArea,`

StockSharp.Charting.IChartElement)) - Add a data series element. Has several overloads.

- [IChart.Reset](#) - "Reset" previously drawn values.
- [IChart.Draw](#) - Draw a value on the chart.
- [IChart.OrderCreationMode](#) - Order creation mode, when set allows creating orders from the chart. Default is off.

IChartArea

[IChartArea](#) - A chart plotting area, acts as a container for [IChartElement](#) (indicators, candles, etc.) that are rendered on the chart, and chart axes ([IChartAxis](#)).

Key Properties of [IChartArea](#)

- [IChartArea.Elements](#) - List of [IChartElement](#).
- [IChartArea.XAxes](#) - List of horizontal axes.
- [IChartArea.YAxes](#) - List of vertical axes.

IChartElement

All elements displayed on the chart must implement the [IChartElement](#) interface. In [S#](#), the following classes implement this interface:

- [ChartCandleElement](#) - An element for displaying candles.
- [ChartIndicatorElement](#) - An element for displaying indicators.
- [ChartOrderElement](#) - An element for displaying orders.
- [ChartTradeElement](#) - An element for displaying trades.

The classes of visual elements have several properties for adjusting the appearance of the chart. You can adjust colors, line thickness, and style of elements. For example, using the property [IChartCandleElement.DrawStyle](#), you can change the appearance of the candle (candle or bar). Using the property [ChartIndicatorElement.DrawStyle](#), you can set the style of the indicator line. To display the indicator as a histogram, use the value [DrawStyles.Histogram](#). The properties [ChartCandleElement.ShowAxisMarker](#) and [ChartIndicatorElement.ShowAxisMarker](#) allow turning on/off the display of markers (see figure) on the axes of the chart.

See also

- [Candlestick chart](#)
- [Chart panel](#)
- [Equity curve chart](#)
- [Box charts](#)
- [Clusters](#)

Candle chart

Candle Chart

[Chart](#) is a graphical component that allows building stock charts: candles, indicators, and displaying order and trade markers on charts.

Below is an example of building a chart using the [Chart](#) component. The example is based on Samples/02_Candles/01_Realtime, with some modifications.

[image: Gui ChartSample]

Example of building a chart using Chart

1. In XAML, we create a window and add the [Chart](#) graphical component to it. We assign the name **Chart** to the component. Note that when creating the window, you need to add the namespace `http://schemas.stocksharp.com/xaml`.

```
<Window x:Class="SampleCandles.ChartWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:charting="http://schemas.stocksharp.com/xaml"
        Title="ChartWindow" Height="300" Width="300">
    <charting:Chart x:Name="Chart" x:FieldModifier="public" />
</Window>
```

2. In the main window code, we declare variables for chart areas, chart elements, indicators, and subscriptions.

```
private readonly Dictionary<Subscription, ChartWindow> _chartWindows = new
Dictionary<Subscription, ChartWindow>();
private readonly Connector _connector = new Connector();
private readonly LogManager _logManager;
private ChartArea _candlesArea;
private ChartArea _indicatorsArea;
private ChartIndicatorElement _smaChartElement;
private ChartIndicatorElement _macdChartElement;
private ChartCandleElement _candlesElem;
private SimpleMovingAverage _sma;
private MovingAverageConvergenceDivergence _macd;
```

3. In the **Connect** button's **Click** event handler, along with subscribing to connector events and calling the [IConnector.Connect](#) method, we subscribe to the [Connector.CandleReceived](#) event. In this event handler, the chart will be drawn when a new candle is received.

```
private void ConnectClick(object sender, RoutedEventArgs e)
{
    _connector.CandleReceived += OnCandleReceived;

    // Subscribe to other necessary events
    _connector.Connected += () => this.GuiAsync(() => { /* Handle connection */ });
    _connector.Disconnected += () => this.GuiAsync(() => { /* Handle disconnection */ });

    // Connect to the trading system
    _connector.Connect();
}
```

4. In the **ShowChart** button handler, we create indicator objects, areas, and chart elements. We add elements to areas, and areas to the chart. We open the chart window and start a subscription to candles.

```

private void ShowChartClick(object sender, RoutedEventArgs e)
{
    var security = SelectedSecurity;

    // Create a subscription to candles
    var subscription = new Subscription(
        DataType.TimeFrame(TimeSpan.FromMinutes(5)),
        security)
    {
        MarketData =
        {
            // Request historical data for 30 days
            From = DateTime.Today.Subtract(TimeSpan.FromDays(30)),
            To = DateTime.Now,
            // Get only finished candles
            IsFinishedOnly = true
        }
    };

    // Create a chart window
    _chartWindows.SafeAdd(subscription, key =>
    {
        var wnd = new ChartWindow
        {
            Title = $"{security.Code} {TimeSpan.FromMinutes(5)}"
        };
        wnd.MakeHideable();

        // Initialize indicators
        _sma = new SimpleMovingAverage() { Length = 11 };
        _macd = new MovingAverageConvergenceDivergence();

        // Initialize chart elements
        _smaChartElement = new ChartIndicatorElement();
        _macdChartElement = new ChartIndicatorElement();
        _candlesElem = new ChartCandleElement();

        // Set MACD display style as histogram
        _macdChartElement.DrawStyle = DrawStyles.Histogram;

        // Initialize chart areas
        _candlesArea = new ChartArea();
        _indicatorsArea = new ChartArea();

        // Add areas to the chart
        wnd.Chart.Areas.Add(_candlesArea);
        wnd.Chart.Areas.Add(_indicatorsArea);

        // Add elements to areas
        _candlesArea.Elements.Add(_candlesElem);
        _candlesArea.Elements.Add(_smaChartElement);
        _indicatorsArea.Elements.Add(_macdChartElement);

        // Bind chart elements to subscription for automatic drawing
        wnd.Chart.AddElement(_candlesArea, _candlesElem, subscription);
        wnd.Chart.AddElement(_candlesArea, _smaChartElement, subscription);
        wnd.Chart.AddElement(_indicatorsArea, _macdChartElement, subscription);

        return wnd;
    }).Show();

    // Start subscription to candles
    _connector.Subscribe(subscription);
}

```

5. In the [Connector.CandleReceived](#) event handler, we draw the candle and indicator values for each finished candle.


```

private void OnCandleReceived(Subscription subscription, ICandleMessage candle)
{
    var wnd = _chartWindows.TryGetValue(subscription);
    if (wnd == null)
        return;

    // Process only finished candles
    if (candle.State != CandleStates.Finished)
        return;

    // Calculate indicator values
    var smaValue = _sma.Process(candle);
    var macdValue = _macd.Process(candle);

    // Create data for drawing
    var data = new ChartDrawData();
    data
        .Group(candle.OpenTime)
        .Add(_candlesElem, candle)
        .Add(_smaChartElement, smaValue)
        .Add(_macdChartElement, macdValue);

    // Draw data on the chart in the user interface thread
    this.GuiAsync(() => wnd.Chart.Draw(data));
}

```

Example with automatic chart drawing

Starting with the latest versions of StockSharp, it is possible to set up automatic chart drawing without the need to explicitly call the Draw method. For this, when configuring Chart, the AddElement method is used, which links the chart element with a subscription:

```

private void SetupAutoDrawingChart()
{
    var security = SelectedSecurity;

    // Create chart elements
    var candleElement = new ChartCandleElement();
    var smaElement = new ChartIndicatorElement { Title = "SMA" };

    // Create chart areas
    var area = new ChartArea();

    // Add area to the chart
    Chart.Areas.Add(area);

    // Create a subscription to candles
    var subscription = new Subscription(
        DataType.TimeFrame(TimeSpan.FromMinutes(5)),
        security)
    {
        MarketData =
        {
            From = DateTime.Today.Subtract(TimeSpan.FromDays(30)),
            To = DateTime.Now
        }
    };

    // Bind elements to the chart area and subscription
    Chart.AddElement(area, candleElement, subscription);
    Chart.AddElement(area, smaElement, subscription);

    // Create an indicator
    var sma = new SimpleMovingAverage { Length = 14 };

    // Subscribe to the candle receiving event for indicator processing
}

```

```

_connector.CandleReceived += (sub, candle) =>
{
    if (sub == subscription && candle.State == CandleStates.Finished)
    {
        // Process the candle with the indicator and get the value
        var smaValue = sma.Process(candle);

        // Draw the indicator value
        var data = new ChartDrawData();
        data
            .Group(candle.OpenTime)
            .Add(smaElement, smaValue);

        this.GuiAsync(() => Chart.Draw(data));
    }
};

// Start subscription
_connector.Subscribe(subscription);
}

```

Displaying orders and trades on the chart

You can display order and trade markers directly on the chart:

```

// Create elements for displaying orders and trades
var orderElement = new ChartOrderElement();
var tradeElement = new ChartTradeElement();

// Add elements to the chart area
_candlesArea.Elements.Add(orderElement);
_candlesArea.Elements.Add(tradeElement);

// Subscribe to order and trade receiving events
_connector.OrderReceived += (sub, order) =>
{
    if (order.Security != _security)
        return;

    // Draw the order on the chart
    var data = new ChartDrawData();
    data.Group(order.Time).Add(orderElement, order);

    this.GuiAsync(() => Chart.Draw(data));
};

_connector.OwnTradeReceived += (sub, trade) =>
{
    if (trade.Order.Security != _security)
        return;

    // Draw the trade on the chart
    var data = new ChartDrawData();
    data.Group(trade.Time).Add(tradeElement, trade);

    this.GuiAsync(() => Chart.Draw(data));
};

```

Clearing the chart

To clear the chart, you can use the Reset method:

```

// Clear the entire chart
Chart.Reset();

```

```
// Clear a specific area  
_candlesArea.Reset();  
  
// Clear a specific element  
_candlesElem.Reset();
```

Candle chart panel

Candle chart panel

[ChartPanel](#) is an advanced component for creating stock charts, which has a toolbar and additional functionality. The charting technique with [ChartPanel](#) does not differ significantly from that of [Chart](#).

The following figures show the appearance of the component, as well as the functions of the toolbar buttons.

[image: Gui ChartPanel]

- 1 - Horizontal line;
- 2 - Vertical line;
- 3 - Prompt;
- 4 - Trend line;
- 5 - Area;
- 6 - This is a text.

Toolbar functionality

[image: Gui ChartPanelTools]

- 1 - Add panel;
- 2 - Enable autoscrolling;
- 3 - Enable autoscaling;
- 4 - Enable legend;
- 5 - Enable preview area;
- 6 - Enable crosshair;
- 7 - Enable prompt for bar;
- 8 - Show values on axis;
- 9 - Draw trend line;
- 10 - Draw pointer;
- 11 - Draw vertical line;
- 12 - Draw horizontal line;
- 13 - Select area;
- 14 - Text label;
- 15 - Show FPS stats;
- 16 - Orders registering;
- 17 - Save data;
- 18 - Publishing;

Equity curve chart

Equity curve chart

[EquityCurveChart](#) - the graphical component for displaying the equity curve.

Below is an example of using this component. The full example code is in the Samples/Testing/SampleHistoryTesting.

[image: Gui EquityCurveChart]

Example of the EquityCurveChart plotting

1. We add the [EquityCurveChart](#) graphical component to XAML. Assigning the **Curve** name to the component.

```
<Window x:Class="SampleRandomEmulation.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:loc="clr-namespace:StockSharp.Localization;assembly=StockSharp.Localization"
        Title="{x:Static loc:LocalizedStrings.XamlStr564}" Height="460" Width="604"
        xmlns:sx="clr-namespace:StockSharp.Xaml;assembly=StockSharp.Xaml"
        xmlns:charting="http://schemas.stocksharp.com/xaml">

    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="85*" />
            <ColumnDefinition Width="497*" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <Grid Grid.ColumnSpan="2">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="100" />
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="Auto" />
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto" />
                <RowDefinition Height="10" />
                <RowDefinition Height="Auto" />
                <RowDefinition Height="10" />
            </Grid.RowDefinitions>
            <Button x:Name="StartBtn" Content="{x:Static loc:LocalizedStrings.Str2421}" Grid.Row="0"
                Click="StartBtnClick" />
            <ProgressBar x:Name="TestingProcess" Grid.Column="1" Grid.Row="0" />
            <Button x:Name="Report" Content="{x:Static loc:LocalizedStrings.XamlStr432}" Grid.Row="0"
                Width="75" IsEnabled="False" Click="ReportClick" Grid.Column="2" Margin="0,0,0,-1" />
        </Grid>

        <Grid Grid.Row="1" Grid.ColumnSpan="2" Grid.Column="0">
            <Grid>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="180"/>
                    <ColumnDefinition Width="*" />
                </Grid.ColumnDefinitions>
                <sx:StatisticParameterGrid Grid.Column="0" x:Name="ParameterGrid" />
                <charting:EquityCurveChart Grid.Column="1" x:Name="EquityCurveChart" />
            </Grid>
        </Grid>
    </Grid>
</Window>
```

2. In the main window code we create a data source to draw the chart using the `EquityCurveChart.CreateCurve(System.String title, System.Windows.Media.Color color, System.Windows.Media.Color secondColor, Ecng.Drawing.DrawStyles style, System.Guid id )` method.

```
private ChartBandElement _pnl;  
private ChartBandElement _unrealizedPnL;  
private ChartBandElement _commissionCurve;  
  
.....  
  
public MainWindow()  
{  
    InitializeComponent();  
  
    _logManager.Listeners.Add(new FileLogListener("log.txt"));  
  
    _pnl = (ChartBandElement)EquityCurveChart.CreateCurve("PNL", Colors.Green, DrawStyles.Area);  
    _unrealizedPnL = (ChartBandElement)EquityCurveChart.CreateCurve("unrealizedPnL", Colors.Black,  
DrawStyles.Line);  
    _commissionCurve = (ChartBandElement)EquityCurveChart.CreateCurve("commissionCurve", Colors.Red,  
DrawStyles.Line);  
}
```

3. When the PnL value of the strategy is changed we add data to the data source. In this case we use the special `ChartDrawData` class.

```
_strategy.PnLChanged += () =>  
{  
    var data = new ChartDrawData();  
    data.Group(_strategy.CurrentTime)  
        .Add(_pnl, _strategy.PnL)  
        .Add(_unrealizedPnL, _strategy.PnLManager.UnrealizedPnL ?? 0)  
        .Add(_commissionCurve, _strategy.Commission ?? 0);  
  
    EquityCurveChart.Draw(data);  
};
```

Box chart

Box chart

BoxChart - is the special type of chart for displaying the volumes in the form of a numbers grid. To use this chart type, it is necessary to set the special style `ChartCandleElement.DrawStyle = ChartCandleDrawStyles.BoxVolume` This chart uses the information from the `PriceLevels` property as source data.

[image: Gui BoxChart]

Main properties

- `ChartCandleElement.Timeframe2Multiplier` - the multiplier factor that is applied to the main timeframe specified in the constructor to get a second timeframe. The displayed candles are joined together in groups of size corresponding to the second timeframe. The groups are drawn on the chart by the grid and frame of the respective colors.
- `ChartCandleElement.Timeframe3Multiplier` - is similarly to `ChartCandleElement.Timeframe2Multiplier`, but for the third timeframe. The 3rd timeframe is drawn on the chart using the grid of the corresponding color.
- `ChartCandleElement.FontColor` - the volumes values color on the chart.
- `ChartCandleElement.MaxVolumeColor` - the volumes values color on the chart for the maximum volume in the specified candle.
- `ChartCandleElement.Timeframe2Color` - the 2nd timeframe grid color.
- `ChartCandleElement.Timeframe2FrameColor` - the 2nd timeframe frame color.
- `ChartCandleElement.Timeframe3Color` - the 3rd timeframe grid color.

An example of using this type of chart is in *Samples/Common/SampleChart*.

To switch between the chart types, use the settings button (gear), located in the upper left corner of the chart.

Cluster chart

Cluster chart

ClusterChart - is the special type of chart to display the volumes in the form of clusters with the bar charts. To use this type of the chart it is necessary to set the special style [IChartCandleElement.DrawStyle = ChartCandleDrawStyles.ClusterProfile](#). This chart uses the information from the [PriceLevels](#) property as source data.

[image: Gui ClusterChart]

Main properties

- [ChartCandleElement.ClusterLineColor](#) - the basic cluster line color.
- [ChartCandleElement.ClusterTextColor](#) - the volumes values color on the chart.
- [ChartCandleElement.ClusterColor](#) - the main bars color in the clusters histograms.
- [ChartCandleElement.ClusterMaxColor](#) - the maximum volume bar color in the clusters histograms.

An example of using this type of chart is in *Samples/Common/SampleChart*.

To switch between the chart types, use the settings button (gear), located in the upper left corner of the chart.

Registering of orders from the chart

Registering of orders from the chart

S# allows to register orders from the chart, to activate this feature, you need to set the [Chart.OrderCreationMode](#) property to "True", it is disabled by default.

[image: API GUI Trading from chart]

Orders for purchase will be registered using the **Ctrl + Left Mouse Button** combination.

Orders for sale will be registered using the **Ctrl + Right Mouse Button** combination.

The resulting order can be intercepted through the event of a new order creation.

```
ChartPanel.CreateOrder += (chartArea,order) =>
{
    order.Portfolio = _portfolio;
    order.Security = _security;
    order.Volume = 1;

    _connector.RegisterOrder(order);
};
```

Registered orders will be displayed as a special element for displaying orders [ChartActiveOrdersElement](#).

Annotations

Annotations

[S#](#) provides the ability to add annotations to the chart in the form of text, lines, etc.

[image: ChartAnnotations]

Adding annotations is the same as adding any other information to the chart. First you need to create [ChartAnnotation](#) and add it to the chart area:

```
var _annotation = new ChartAnnotation { Type = ChartAnnotationTypes.BoxAnnotation };
Chart.AddElement(chartArea, _annotation);
```

After that, you need to initialize a new instance of the [AnnotationData](#) class, in which to describe the annotation and pass it to the [IChart.Draw\(StockSharp.Charting.IChartDrawData data\)](#) method for drawing on the chart:

```
var data = new ChartDrawData.AnnotationData
{
    X1 = new DateTimeOffset(2017, 10, 02, 8, 30, 0, TimeSpan.FromHours(1)),
    X2 = new DateTimeOffset(2017, 10, 02, 10, 30, 0, TimeSpan.FromHours(1)),
    Y1 = 193.5m,
    Y2 = 194m,
    IsVisible = true,
    Stroke = new SolidColorBrush(Color.FromRgb(0, 0, 255)),
    Thickness = new Thickness(3),
    Text = "New annotation",
    HorizontalAlignment = HorizontalAlignment.Stretch,
    VerticalAlignment = VerticalAlignment.Stretch,
    LabelPlacement = LabelPlacement.Axis,
    ShowLabel = true,
    CoordinateMode = AnnotationCoordinateMode.Absolute,
};
var drawData = new ChartDrawData();
drawData.Add(_annotation, data);
Chart.Draw(drawData);
```

Logging

Logging

[S#](#) offers several graphical components for displaying logs: [LogControl](#) and [Monitor](#).

When using the logging visual components, you need to use the [GuiLogListener](#) logger as a "listener". This logger provides streaming synchronization with the GUI when recording new [LogMessage](#) messages.

In order to implement the logging possibility in your own class, you need to implement the [ILogReceiver](#) interface. An easier way is to inherit from the [BaseLogReceiver](#) class, as shown in the *Samples/08_Misc/01_Logging* example:

```
private class TestSource : BaseLogReceiver
{
}
private readonly LogManager _logManager = new LogManager();
private readonly TestSource _testSource = new TestSource();
public MainWindow()
{
    InitializeComponent();
    // immediate flush
    _logManager.FlushInterval = TimeSpan.FromMilliseconds(1);
    // set test log source
    _logManager.Sources.Add(_testSource);
    // set .NET Trace system based source
    _logManager.Sources.Add(new Ecng.Logging.TraceSource());
    // write logs into MainWindow
    _logManager.Listeners.Add(new GuiLogListener(Monitor));
    // and file logs.txt
    _logManager.Listeners.Add(new FileLogListener
    {
        FileName = "logs",
    });
}
```

Recommended content

[Logging](#)

[Log panel](#)

[Extended log panel](#)

Log panel

Log panel

[LogControl](#) - the table to display the log messages. The toolbar buttons let you filter messages with different logging levels.

LogControl

[image: GUI LogControl]

Sample code

```
<Window x:Class="LoggingControls.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:sx="clr-namespace:StockSharp.Xaml;assembly=StockSharp.Xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <sx:LogControl x:Name="LogControl"/>
    </Grid>
</Window>
```

```
// creating a new instance of LogManager
_logManager = new LogManager();
// adding .NET tracing as a log source.
_logManager.Sources.Add(new Ecng.Logging.TraceSource());
// adding LogControl as a log listener.
_logManager.Listeners.Add(new GuiLogListener(LogControl));
.....
// sending test messages from the TraceSource:
Trace.TraceInformation("Info test message");
Trace.TraceWarning("Warning test message");
Trace.TraceError("Error test message");
```

Extended log panel

Extended log panel

Monitor - is the visual element where **LogControl** is used in conjunction with the **TreeView** hierarchical tree, in which log sources are shown. Initially, the component was designed to monitor the trading strategies. Therefore, by default, the "tree" includes the **Strategy** node. At the same time other sources can be used with this component.

[image: GUI Monitor]

Sample code

```
<Window x:Class="LoggingControls.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sx="clr-namespace:StockSharp.Xaml;assembly=StockSharp.Xaml"
    Title="MainWindow" Height="350" Width="525">
    <Grid>
        <sx:Monitor x:Name="Monitor" />
    </Grid>
</Window>
```

```
// creating a new instance of LogManager
_logManager = new LogManager();
// adding .NET tracing as a log source.
_logManager.Sources.Add(new Ecng.Logging.TraceSource());
// adding Monitor as a log listener.
_logManager.Listeners.Add(new GuiLogListener(Monitor));
```

Portfolios

Portfolios

To work with portfolios, [S#](#) offers the following graphical components:

- The combo box for portfolio selection - [PortfolioComboBox](#).
- The combo box with a button that opens a window with a list of portfolios - [PortfolioEditor](#).
- Portfolio list window - [PortfolioPickerWindow](#).
- A table that displays information about portfolios and positions - [PortfolioGrid](#).

Note that in all components for the portfolios list (the Portfolios property), the **ThreadSafeObservableCollection<TItem>** class (located in Ecng.Xaml) is used, which provides thread safety.

Recommended content

[Drop down list](#)

[Picker](#)

[Portfolio picker window](#)

[Table](#)

Drop down list

Drop down list

[PortfolioComboBox](#) is a combo box for portfolio selection.

[image: Gui PortfolioComboBox]

Main properties

- [PortfolioComboBox.Portfolios](#) - the list of portfolios.
- [PortfolioComboBox.SelectedPortfolio](#) - the selected portfolio.

Below is the code snippet with its use. The code example is taken from *Samples/InteractiveBrokers/SampleIB*.

```
<Window x:Class="SampleAsyncTransactions.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:loc="clr-namespace:StockSharp.Localization;assembly=StockSharp.Localization"
  xmlns:sx="clr-namespace:StockSharp.Xaml;assembly=StockSharp.Xaml"
  Title="{x:Static loc:LocalizedStrings.XamlStr567}" Height="116" Width="347">
  <Grid>
    <Grid.ColumnDefinitions>
      .....
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      .....
    </Grid.RowDefinitions>
    <Label Grid.Column="0" Grid.Row="0" Content="{x:Static loc:LocalizedStrings.XamlStr592}" />
    <TextBox x:Name="Path" Text="" Grid.Column="1" Grid.ColumnSpan="2" Grid.Row="0" />
    <Button x:Name="FindPath" Grid.Column="2" Grid.Row="0" Content="..." Width="25"
      HorizontalAlignment="Left" Click="FindPathClick" />
    <TextBlock Grid.Column="0" Grid.Row="1">
      <Run Text="{x:Static loc:LocalizedStrings.Account}" /><Run Text=":" />
    </TextBlock>
    <sx:PortfolioComboBox x:Name="Portfolios" Grid.Column="1" Grid.Row="1" />
    <Button x:Name="ConnectBtn" Grid.Column="0" Grid.Row="2" Content="{x:Static
loc:LocalizedStrings.Connect}" Click="ConnectClick" />
    <Button x:Name="ShowSecurities" Grid.Column="1" Grid.Row="2" IsEnabled="False" Content="{x:Static
loc:LocalizedStrings.Securities}" Click="ShowSecuritiesClick" />
  </Grid>
</Window>
```

```
.....
private readonly Connector _connector = new Connector();
.....
private void ConnectClick(object sender, RoutedEventArgs e)
{
  .....
  Portfolios.Portfolios = new PortfolioDataSource(_connector);
  .....
}
```

Picker

Picker

[PortfolioEditor](#) is a component consisting of the [PortfolioComboBox](#) combo box and a button that opens the [PortfolioPickerWindow](#).

Main properties and usage

See [Drop down list](#).

Portfolio picker window

Portfolio picker window

[PortfolioPickerWindow](#) is a window for selecting a portfolio. The window displays a list of portfolios and information about the cash positions of portfolios.

[image: GUI PortfolioPickerWindow]

Main properties

- [PortfolioPickerWindow.Portfolios](#) - the list of portfolios.
- [PortfolioPickerWindow.SelectedPortfolio](#) - the selected portfolio.

Below is the code snippet with its use.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    var wnd = new PortfolioPickerWindow();
    if (Portfolios != null)
        wnd.Portfolios = Portfolios;
    if (wnd.ShowDialog(this))
    {
        SelectedPortfolio = wnd.SelectedPortfolio;
    }
}
```

Table

Table

[PortfolioGrid](#) is a component that displays the status of portfolios and positions.

[image: GUI PortfolioGrid]

Main properties

- [PortfolioGrid.Positions](#) – the list of positions and portfolios.
- [PortfolioGrid.SelectedPosition](#) – the selected position.
- [PortfolioGrid.SelectedPositions](#) – selected positions.

The following code snippet demonstrates its usage. The code example is taken from *Samples/InteractiveBrokers/SampleIB*.

```
<Window x:Class="Sample.PortfoliosWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:loc="clr-namespace:StockSharp.Localization;assembly=StockSharp.Localization"
  xmlns:xaml="http://schemas.stocksharp.com/xaml"
  Title="{x:Static loc:LocalizedStrings.Portfolios}" Height="200" Width="470">
  <xaml:PortfolioGrid x:Name="PortfolioGrid" x:FieldModifier="public" />
</Window>
```

```
private readonly Connector _connector = new Connector();
private void ConnectClick(object sender, RoutedEventArgs e)
{
  .....
  _connector.PositionReceived += (sub, p) =>
  _portfoliosWindow.PortfolioGrid.Positions.TryAdd(position);
  .....
}
```

Instruments

Instruments

[S#](#) offers a large set of graphical components for working with instruments. This set includes:

- The component for creating the instrument – [SecurityCreateWindow](#).
- The component for displaying financial information on the instrument - [SecurityGrid](#).
- Components for searching and selecting instruments - [SecurityPicker](#), [SecurityPickerWindow](#) and [SecurityEditor](#).
- Components for creating continuous and index instruments - [ContinuousSecurityWindow](#) and [IndexSecurityWindow](#).

Recommended content

[Window](#)

[Table](#)

[Picker](#)

[Continuous futures](#)

[Index](#)

Window

Window

The [SecurityCreateWindow](#) component is a window for creating and editing an instrument. The component consists of two main elements: the special text field [SecurityIdTextBox](#) and the property editing grid [PropertyGridEx](#). You can access the created (edited) instrument with the [SecurityCreateWindow.Security](#) property.

Below is the appearance of the component and the code snippet with its use.

[image: Gui SecurityCreateWindow]

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    var dlg = new SecurityCreateWindow();
    var result = dlg.ShowDialog();
    if (result != null && (bool)result)
    {
        var security = dlg.Security;
    }
}
```

Table

Table

The [SecurityGrid](#) component is designed to display financial information (level1 fields) and its changes relating to instruments in a tabular form. The component allows you to select one or more instruments.

[image: GUI SecurityPicker2]

Main properties

- [SecurityGrid.Securities](#) - the list of instruments.
- [SecurityGrid.SelectedSecurity](#) - the selected instrument.
- [SecurityGrid.SelectedSecurities](#) - the list of selected instruments.
- [SecurityGrid.MarketDataProvider](#) - the provider of market data.

Please note that for the display of changes in market information, you must specify a provider of market data.

Below is the code snippet with its use.

In the figure, the [SecurityGrid](#) component is shown in the [SecurityPicker](#) graphical component.

```
<Window x:Class="SecurityGridSample.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:sx="clr-namespace:StockSharp.Xaml;assembly=StockSharp.Xaml"
  Title="MainWindow" Height="350" Width="525">
  <Grid>
    <sx:SecurityGrid x:Name="SecurityGrid"/>
  </Grid>
</Window>
```

```
private readonly Connector _connector = new Connector();
SecurityGrid.MarketDataProvider = _connector;
.....
_connector.SecurityReceived += (sub, security) =>
{
    SecurityGrid.Securities.Add(security);
};
.....
private void ColumnsFilter()
{
    string[] columns = { "Board", "BestAsk.Price", "BestAsk.Volume" };

    foreach (var column in SecurityGrid.Columns)
    {
        column.Visibility = columns.Contains(column.SortMemberPath) ? Visibility.Visible :
        Visibility.Collapsed;
    }
}
```

Picker

Picker

The [SecurityPicker](#) component is designed to find and select instruments. It supports both single and multiple choice. The component allows you to filter the list of instruments by their type. This component can also be used to display financial information (level1 fields), as shown in the [SecurityGrid](#) section.

[image: GUI SecurityPicker2]

[SecurityPicker](#) consists of:

1. A text field, to enter the code (or Id) of the instrument. When you enter, the list is filtered by the entered substring.
2. The special [SecurityTypeComboBox](#) combo box for filtering instruments by their type.
3. The [SecurityGrid](#) table to display the list of instruments.

Main properties

- [SecurityPicker.SelectionMode](#) - instrument selection mode: single, multiple.
- [SecurityPicker.ShowCommonStatColumns](#) - to display the main columns.
- [SecurityPicker.ShowCommonOptionColumns](#) - to display the main columns for options.
- [SecurityPicker.Title](#) - the title that is displayed at the top of the component.
- [SecurityPicker.Securities](#) - the list of instruments.
- [SecurityPicker.SelectedSecurity](#) - the selected instrument.
- [SecurityPicker.SelectedSecurities](#) - the list of selected instruments.
- [SecurityPicker.FilteredSecurities](#) - the list of filtered instruments.
- [SecurityPicker.ExcludeSecurities](#) - the list of hidden instruments.
- [SecurityPicker.SelectedType](#) - the selected instrument type.
- [SecurityPicker.SecurityProvider](#) - the provider of information about instruments.
- [SecurityPicker.MarketDataProvider](#) - the provider of market data.

Below is the code snippet with its use, taken from example *Samples/InteractiveBrokers/SampleIB*.

```
<Window x:Class="Sample.SecuritiesWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:loc="clr-namespace:StockSharp.Localization;assembly=StockSharp.Localization"
  xmlns:xaml="http://schemas.stocksharp.com/xaml"
  Title="{x:Static loc:LocalizedStrings.Securities}" Height="415" Width="1081">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="*" />
      <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <xaml:SecurityPicker x:Name="SecurityPicker" x:FieldModifier="public"
      SecuritySelected="SecurityPicker_OnSecuritySelected" ShowCommonStatColumns="True" />
  </Grid>
</Window>
```

```

private void ConnectClick(object sender, RoutedEventArgs e)
{
    .....
    _connector.SecurityReceived += (sub, security) =>
    _securitiesWindow.SecurityPicker.Securities.Add(security);
    _securitiesWindow.SecurityPicker.MarketDataProvider = _connector;
    .....
}
private void SecurityPicker_OnSecuritySelected(Security security)
{
    NewStopOrder.IsEnabled = NewOrder.IsEnabled =
    Level1.IsEnabled = Depth.IsEnabled = security != null;
}

```

Continuous futures

Continuous futures

[ContinuousSecurityWindow](#) - is a visual editor for creating *continuous* ([ContinuousSecurity](#)) instruments. See [Continuous futures](#).

[image: HydraGluingCSCustom]

This component includes:

- Special [SecurityIdTextBox](#) text field, which generates a *continuous* instrument with the input of Id - [Code]@[Board].
- The [SecurityJumpsEditor](#) component is a special DataGrid for working with instruments that are part of a *continuous* instrument. The instruments are wrapped in the [SecurityJump](#), class, which has two properties: [SecurityJump.Security](#) and [SecurityJump.Date](#) (roll forward). The added instruments are stored in the [SecurityJumpsEditor.Jumps](#) list. The component has the [SecurityJumpsEditor.Validate](#) function to check the correctness of the component instruments.
- Buttons for adding/removing instruments.
- **Auto** button allows you to automatically create a *continuous* instrument.
- **Ok** button completes the creation of a *continuous* instrument.

Main properties

- [ContinuousSecurityWindow.Security](#) – continuous instrument
- [ContinuousSecurityWindow.SecurityStorage](#) – provider of information about instruments.

Below is the code snippet with its use.

```
private void CreateContinuousSecurity_OnClick(object sender, RoutedEventArgs e)
{
    _continuousSecurityWindow = new ContinuousSecurityWindow
    {
        SecurityStorage = _entityRegistry.Securities,
        Security = new ContinuousSecurity { Board = ExchangeBoard.Associated }
    };
    if (!_continuousSecurityWindow.ShowModal(this))
        return;
    _continuousSecurity = _continuousSecurityWindow.Security;
    ContinuousSecurity.Content = _continuousSecurity.Id;
    var first = _continuousSecurity.InnerSecurities.First();
    var gluingSecurity = new Security
    {
        Id = _continuousSecurity.Id,
        Code = _continuousSecurity.Code,
        Board = ExchangeBoard.Associated,
        Type = _continuousSecurity.Type,
        VolumeStep = first.VolumeStep,
        PriceStep = first.PriceStep,
        ExtensionInfo = new Dictionary<object, object> { { "GluingSecurity", true } }
    };
    if (_entityRegistry.Securities.ReadById(gluingSecurity.Id) == null)
    {
        _entityRegistry.Securities.Save(gluingSecurity);
    }
}
```

Recommended content

Index

Index

[IndexEditor](#) - Graphic control for editing [ExpressionIndexSecurity](#).

[ExpressionIndexSecurity](#) - is a special type of index security based on a combination of several securities using mathematical formulas. This type has the [ExpressionIndexSecurity.Expression](#) property, which stores the formula in text form and the list of underlying [ExpressionIndexSecurity.InnerSecurityIds](#) securities.

[image: GUI IndexSecurityWindow]

Basic properties

- [IndexEditor.Securities](#) - all available securities.
- [IndexEditor.Text](#) - mathematical formula of the index.

To use [IndexEditor](#), first you need to register a special service:

```
...
ConfigManager.RegisterService<ICompilerService>(new RoslynCompilerService());
...
```

Next, the securities necessary for index calculation should be passed to [IndexEditor](#):

```
...
IndexEditor.Securities.AddRange(SecurityProvider.LookupAll());
SecurityProvider.Added += OnAdded;
...
private void OnAdded(IEnumerable<Security> securities)
{
    IndexEditor.Securities.AddRange(securities);
}
```

Market-data

Market-data

There are graphical components to display all kinds of market data in [S#](#):

- Table to display trades - [TradeGrid](#).
- Table to display Level1 - [Level1Grid](#).
- Component to display an order book - [MarketDepthControl](#).
- Table to display an order log - [OrderLogGrid](#).
- Tables to display news - [NewsGrid](#), [NewsPanel](#).

Recommended content

[Ticks](#)

[Level1](#)

[Order book](#)

[Order log](#)

[News](#)

Ticks

Tick Trades

[image: GUI TradeGrid]

[TradeGrid](#) - a trade table.

Main properties

- [TradeGrid.Trades](#) - list of trades.
- [TradeGrid.SelectedTrade](#) - selected trade.
- [TradeGrid.SelectedTrades](#) - selected trades.

Below are code fragments demonstrating its usage:

```
<Window x:Class="Sample.TradesWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:loc="clr-namespace:StockSharp.Localization;assembly=StockSharp.Localization"
  xmlns:xaml="http://schemas.stocksharp.com/xaml"
  Title="{x:Static loc:LocalizedStrings.Str985}" Height="284" Width="544">
  <xaml:TradeGrid x:Name="TradeGrid" x:FieldModifier="public" />
</Window>
```

```
public class TradesWindow
{
    private readonly Connector _connector;
    private readonly Security _security;
    private Subscription _tickSubscription;

    public TradesWindow(Connector connector, Security security)
    {
        InitializeComponent();

        _connector = connector;
        _security = security;

        // Subscribe to tick trade reception event
        _connector.TickTradeReceived += OnTickReceived;

        // Create a subscription to tick trades
        _tickSubscription = new Subscription(DataType.Ticks, security);

        // Start subscription
        _connector.Subscribe(_tickSubscription);
    }

    // Handler for tick trade reception event
    private void OnTickReceived(Subscription subscription, ITickTradeMessage tick)
    {
        // Check if the trade belongs to our subscription
        if (subscription != _tickSubscription)
            return;

        // Add the trade to TradeGrid in the user interface thread
        thisGuiAsync(() => TradeGrid.Trades.Add(tick));
    }

    // Method for unsubscribing when the window is closed
    public void Unsubscribe()
    {
    }
```

```

    if (_tickSubscription != null)
    {
        _connector.TickTradeReceived -= OnTickReceived;
        _connector.UnSubscribe(_tickSubscription);
        _tickSubscription = null;
    }
}
}

```

Displaying own trades

```

public class MyTradesWindow
{
    private readonly Connector _connector;

    public MyTradesWindow(Connector connector)
    {
        InitializeComponent();

        _connector = connector;

        // Subscribe to own trade reception event
        _connector.OwnTradeReceived += OnOwnTradeReceived;

        // Create a subscription to transaction data
        var myTradesSubscription = new Subscription(DataType.Transactions, null);

        // Start subscription
        _connector.Subscribe(myTradesSubscription);
    }

    // Handler for own trade reception event
    private void OnOwnTradeReceived(Subscription subscription, MyTrade myTrade)
    {
        // Add own trade to TradeGrid in the user interface thread
        this.GuiAsync(() => TradeGrid.Trades.Add(myTrade));
    }
}

```

Getting historical tick trades

```

// Method for getting historical tick trades
public void LoadHistoricalTicks(Security security, DateTime from, DateTime to)
{
    // Clear current trades
    TradeGrid.Trades.Clear();

    // Create a subscription to historical tick trades
    var historySubscription = new Subscription(DataType.Ticks, security)
    {
        MarketData =
        {
            // Specify time period for historical data
            From = from,
            To = to
        }
    };

    // Subscribe to tick trade reception event
    _connector.TickTradeReceived += OnHistoricalTickReceived;

    // Start subscription
    _connector.Subscribe(historySubscription);
}

```

```

// Handler for historical tick trade reception event
private void OnHistoricalTickReceived(Subscription subscription, ITickTradeMessage tick)
{
    // Add tick to TradeGrid in the user interface thread
    this.GuiAsync(() =>
    {
        TradeGrid.Trades.Add(tick);

        // Update statistics
        UpdateTradeStatistics();
    });
}

// Method for updating trade statistics
private void UpdateTradeStatistics()
{
    int totalTrades = TradeGrid.Trades.Count;
    decimal totalVolume = TradeGrid.Trades.Sum(t => t.Volume);
    decimal averagePrice = TradeGrid.Trades.Any()
        ? TradeGrid.Trades.Average(t => t.Price)
        : 0;

    // Update interface statistics elements
    TotalTradesLabel.Content = $"Total trades: {totalTrades}";
    TotalVolumeLabel.Content = $"Total volume: {totalVolume}";
    AveragePriceLabel.Content = $"Average price: {averagePrice:F2}";
}

```

Filtering trades by volume

```

// Method for filtering trades by minimum volume
public void FilterTicksByVolume(decimal minVolume)
{
    // Save filter value
    _minVolumeFilter = minVolume;

    // Update tick trade reception event handler
    _connector.TickTradeReceived -= OnTickReceived;
    _connector.TickTradeReceived += OnFilteredTickReceived;
}

// Handler for tick trade reception event with volume filtering
private void OnFilteredTickReceived(Subscription subscription, ITickTradeMessage tick)
{
    // Check if the trade belongs to the selected instrument
    if (tick.SecurityId != _security.ToSecurityId())
        return;

    // Apply volume filter
    if (tick.Volume < _minVolumeFilter)
        return;

    // Add the trade to TradeGrid in the user interface thread
    this.GuiAsync(() => TradeGrid.Trades.Add(tick));

    // If it's a large trade, you can highlight it or send a notification
    if (tick.Volume >= _largeVolumeThreshold)
    {
        NotifyLargeVolumeTrade(tick);
    }
}

// Method for large trade notification
private void NotifyLargeVolumeTrade(ITickTradeMessage tick)
{
    // Output information about the large trade
    Console.WriteLine($"Large trade: {tick.SecurityId}, {tick.ServerTime}, Price: {tick.Price}, Volume:

```

```
{tick.Volume}");  
  
// You can add sound or visual notification  
this.GuiAsync(() =>  
{  
    // Example of visual highlighting in the list  
    var tradeItem = TradeGrid.Trades.LastOrDefault();  
    if (tradeItem != null)  
    {  
        TradeGrid.SelectedTrade = tradeItem;  
        HighlightTrade(tradeItem);  
    }  
});  
}
```

Level1

Level1

[image: GUI Level1Grid]

[Level1Grid](#) - a table for displaying Level1 fields. This table uses data in the form of [Level1ChangeMessage](#) messages.

Main properties

- [Level1Grid.MaxCount](#) - maximum number of messages to display.
- [Level1Grid.Messages](#) - list of messages added to the table.
- [Level1Grid.SelectedMessage](#) - selected message.
- [Level1Grid.SelectedMessages](#) - selected messages.

Below are code fragments demonstrating its usage:

```
<Window x:Class="Membrane02.Level1Window"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:sx="http://schemas.stocksharp.com/xaml"
  xmlns:local="clr-namespace:Membrane02"
  mc:Ignorable="d"
  Title="Level1Window" Height="300" Width="300" Closing="Window_Closing">
  <Grid>
    <sx:Level1Grid x:Name="Level1Grid" />
  </Grid>
</Window>
```

```
public Level1Window()
{
    InitializeComponent();
    _connector = MainWindow.This.Connector;

    // Subscribe to Level1 data reception event
    _connector.Level1Received += OnLevel1Received;

    // Create a subscription to Level1 data if not already subscribed
    var security = MainWindow.This.SelectedSecurity;
    if (!_connector.Subscriptions.Any(s =>
        s.DataType == DataType.Level1 &&
        s.SecurityId == security.ToSecurityId()))
    {
        var subscription = new Subscription(DataType.Level1, security);
        _connector.Subscribe(subscription);
    }
}

private void OnLevel1Received(Subscription subscription, Level1ChangeMessage level1Message)
{
    // Check if the message belongs to the selected instrument
    if (level1Message.SecurityId != MainWindow.This.SelectedSecurity.ToSecurityId())
        return;

    // Add the message to Level1Grid
    this.GuiAsync(() => Level1Grid.Messages.Add(level1Message));
}
```



```

private void Window_Closing(object sender, System.ComponentModel.CancelEventArgs e)
{
    // Unsubscribe from events when the window is closing
    if (_connector != null)
        _connector.Level1Received -= OnLevel1Received;
}

```

Recommended way to process Level1 data

```

// Creating a subscription to Level1 using multiple instruments
public void SubscribeToLevel1(IEnumerable<Security> securities)
{
    foreach (var security in securities)
    {
        var subscription = new Subscription(DataType.Level1, security);
        _connector.Subscribe(subscription);
    }

    // Subscribe to Level1 data reception event
    _connector.Level1Received += OnLevel1Received;
}

// Handler for Level1 data reception event
private void OnLevel1Received(Subscription subscription, Level1ChangeMessage level1Message)
{
    // Check if we need to process this particular message
    if (IsLevel1Needed(subscription))
    {
        // Update GUI in the user interface thread
        this.GuiAsync(() =>
        {
            // Add the message to Level1Grid
            Level1Grid.Messages.Add(level1Message);

            // Process changes in Level1 fields
            foreach (var change in level1Message.Changes)
            {
                switch (change.Key)
                {
                    case Level1Fields.LastTradePrice:
                        // Process last trade price change
                        var lastPrice = (decimal)change.Value;
                        Console.WriteLine($"Last price {security.Code}: {lastPrice}");
                        break;

                    case Level1Fields.BestBidPrice:
                        // Process best bid price change
                        var bestBid = (decimal)change.Value;
                        Console.WriteLine($"Best bid {security.Code}: {bestBid}");
                        break;

                    case Level1Fields.BestAskPrice:
                        // Process best ask price change
                        var bestAsk = (decimal)change.Value;
                        Console.WriteLine($"Best ask {security.Code}: {bestAsk}");
                        break;
                }
            }
        });
    }
}

```

Order book

Order Book

[image: GUI MarketDepthControl]

[MarketDepthControl](#) - a graphical component for displaying the order book. The component allows displaying quotes and own orders.

Main properties and methods

- [MarketDepthControl.MaxDepth](#) - order book depth.
- [MarketDepthControl.IsBidsOnTop](#) - display bids on top.
- [MarketDepthControl.UpdateFormat](#)([StockSharp.BusinessEntities.Security](#) security) - update the format of price and volume display using the instrument.
- [MarketDepthControl.ProcessOrder](#)([StockSharp.BusinessEntities.Order](#) order, [System.Decimal](#) price, [System.Decimal](#) balance, [StockSharp.Messages.OrderStates](#) state) - process an order.
- [MarketDepthControl.UpdateDepth](#)([StockSharp.Messages.IOrderBookMessage](#) message, [StockSharp.BusinessEntities.Security](#) security) - update the order book using a message.

Below are code fragments demonstrating its usage:

```
<Window x:Class="SampleBarChart.QuotesWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:aml="http://schemas.stocksharp.com/xaml"
  Title="QuotesWindow" Height="600" Width="280">
  <aml:MarketDepthControl x:Name="DepthCtrl" x:FieldModifier="public" />
</Window>
```

```
public class MarketDepthWindow
{
    private readonly Connector _connector;
    private readonly Security _security;
    private Subscription _depthSubscription;

    public MarketDepthWindow(Connector connector, Security security)
    {
        InitializeComponent();

        _connector = connector;
        _security = security;

        // Configure order book formatting
        DepthCtrl.UpdateFormat(security);

        // Subscribe to order book reception event
        _connector.OrderBookReceived += OnMarketDepthReceived;

        // Create a subscription to order book for the selected instrument
        _depthSubscription = new Subscription(DataType.MarketDepth, security);

        // Start subscription
        _connector.Subscribe(_depthSubscription);
    }

    // Handler for order book reception event
    private void OnMarketDepthReceived(Subscription subscription, IOrderBookMessage depth)
    {
    }
```

```

// Check if the order book belongs to our subscription
if (subscription != _depthSubscription)
    return;

// Update the order book in the user interface thread
this.GuiAsync(() => DepthCtrl.UpdateDepth(depth, _security));
}

// Method for unsubscribing when the window is closed
public void Unsubscribe()
{
    if (_depthSubscription != null)
    {
        _connector.OrderBookReceived -= OnMarketDepthReceived;
        _connector.UnSubscribe(_depthSubscription);
        _depthSubscription = null;
    }
}
}

```

Displaying own orders in the order book

```

public class MarketDepthWithOrdersWindow
{
    private readonly Connector _connector;
    private readonly Security _security;

    public MarketDepthWithOrdersWindow(Connector connector, Security security)
    {
        InitializeComponent();

        _connector = connector;
        _security = security;

        // Configure order book formatting
        DepthCtrl.UpdateFormat(security);

        // Subscribe to order book and order reception events
        _connector.OrderBookReceived += OnMarketDepthReceived;
        _connector.OrderReceived += OnOrderReceived;

        // Create a subscription to order book
        var depthSubscription = new Subscription(DataType.MarketDepth, security);
        _connector.Subscribe(depthSubscription);

        // If necessary, create a subscription to orders
        var ordersSubscription = new Subscription(DataType.Transactions, null);
        _connector.Subscribe(ordersSubscription);
    }

    // Handler for order book reception event
    private void OnMarketDepthReceived(Subscription subscription, IOrderBookMessage depth)
    {
        if (depth.SecurityId != _security.ToSecurityId())
            return;

        // Update the order book in the user interface thread
        this.GuiAsync(() => DepthCtrl.UpdateDepth(depth, _security));
    }

    // Handler for order reception event
    private void OnOrderReceived(Subscription subscription, Order order)
    {
        if (order.Security != _security)
            return;

        // Display the order in the order book
    }
}

```

```

this.GuiAsync(() => DepthCtrl.ProcessOrder(
    order,
    order.Price,
    order.Balance,
    order.State));
}
}

```

Getting best prices from the order book

```

// Method to get best prices from the order book
public (decimal? BestBid, decimal? BestAsk) GetBestPrices(IOrderBookMessage depth)
{
    if (depth == null)
        return (null, null);

    var bestBid = depth.GetBestBid()?.Price;
    var bestAsk = depth.GetBestAsk()?.Price;

    return (bestBid, bestAsk);
}

// Using the method to display spread
private void OnMarketDepthReceived(Subscription subscription, IOrderBookMessage depth)
{
    if (depth.SecurityId != _security.ToSecurityId())
        return;

    // Get best prices
    var (bestBid, bestAsk) = GetBestPrices(depth);

    // Calculate and display spread
    if (bestBid.HasValue && bestAsk.HasValue)
    {
        var spread = bestAsk.Value - bestBid.Value;
        var spreadPercent = bestBid.Value > 0 ? spread / bestBid.Value * 100 : 0;

        this.GuiAsync(() =>
        {
            SpreadLabel.Content = $"Spread: {spread:F2} ({spreadPercent:F2}%)";
        });
    }

    // Update order book
    this.GuiAsync(() => DepthCtrl.UpdateDepth(depth, _security));
}

```

Order log

Order Log

[image: GUI orderlog]

[OrderLogGrid](#) - a graphical component for displaying the order log ([OrderLogItem](#)).

Main properties and methods

- [OrderLogGrid.LogItems](#) - list of order log items.
- [OrderLogGrid.SelectedLogItem](#) - selected order log item.
- [OrderLogGrid.SelectedLogItems](#) - selected order log items.

Below are code fragments demonstrating its usage:

```
<Window x:Class="SampleITCH.OrdersLogWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:loc="clr-namespace:StockSharp.Localization;assembly=StockSharp.Localization"
  xmlns:xaml="http://schemas.stocksharp.com/xaml"
  Title="{x:Static loc:LocalizedStrings.OrderLog}" Height="750" Width="900">
  <xaml:OrderLogGrid x:Name="OrderLogGrid" x:FieldModifier="public" />
</Window>
```

```
public class OrderLogWindow
{
    private readonly Connector _connector;
    private readonly Security _security;
    private Subscription _orderLogSubscription;

    public OrderLogWindow(Connector connector, Security security)
    {
        InitializeComponent();

        _connector = connector;
        _security = security;

        // Subscribe to order log item reception event
        _connector.OrderLogItemReceived += OnOrderLogItemReceived;

        // Create a subscription to order log
        _orderLogSubscription = new Subscription(DataType.OrderLog, security);

        // Start subscription
        _connector.Subscribe(_orderLogSubscription);
    }

    // Handler for order log item reception event
    private void OnOrderLogItemReceived(Subscription subscription, OrderLogItem item)
    {
        // Check if the log item belongs to our subscription
        if (subscription != _orderLogSubscription)
            return;

        // Add the item to OrderLogGrid in the user interface thread
        this.GuiAsync(() => OrderLogGrid.LogItems.Add(item));
    }

    // Method for unsubscribing when the window is closed
    public void Unsubscribe()
    {
    }
```

```

if (_orderLogSubscription != null)
{
    _connector.OrderLogItemReceived -= OnOrderLogItemReceived;
    _connector.UnSubscribe(_orderLogSubscription);
    _orderLogSubscription = null;
}
}
}

```

Order log filtering

```

// Creating a subscription to order log with filtering
public void SubscribeOrderLog(Security security, DateTime from, DateTime to)
{
    // Create a subscription to order log
    var orderLogSubscription = new Subscription(DataType.OrderLog, security)
    {
        MarketData =
        {
            // Specify time period for historical data
            From = from,
            To = to
        }
    };

    // Subscribe to order log item reception event
    _connector.OrderLogItemReceived += OnFilteredOrderLogItemReceived;

    // Start subscription
    _connector.Subscribe(orderLogSubscription);
}

// Handler for order log item reception event with filtering
private void OnFilteredOrderLogItemReceived(Subscription subscription, OrderLogItem item)
{
    // Check subscription type
    if (subscription.DataType != DataType.OrderLog)
        return;

    // Filter by price (example)
    if (item.Price < _minPrice || item.Price > _maxPrice)
        return;

    // Add the item to OrderLogGrid in the user interface thread
    this.GuiAsync(() =>
    {
        OrderLogGrid.LogItems.Add(item);

        // Limit the number of displayed items
        while (OrderLogGrid.LogItems.Count > _maxItems)
            OrderLogGrid.LogItems.RemoveAt(0);
    });
}

```

Order log dynamics analysis

```

// Class for analyzing order log dynamics
public class OrderLogAnalyzer
{
    private readonly Connector _connector;
    private readonly Security _security;
    private readonly OrderLogGrid _orderLogGrid;

    // Counters for analysis
    private int _buyCount = 0;

```

```

private int _sellCount = 0;
private decimal _buyVolume = 0;
private decimal _sellVolume = 0;

public OrderLogAnalyzer(Connector connector, Security security, OrderLogGrid orderLogGrid)
{
    _connector = connector;
    _security = security;
    _orderLogGrid = orderLogGrid;

    // Subscribe to order log item reception event
    _connector.OrderLogItemReceived += OnOrderLogItemReceived;

    // Create a subscription to order log
    var subscription = new Subscription(DataType.OrderLog, security);

    // Start subscription
    _connector.Subscribe(subscription);
}

// Handler for order log item reception event
private void OnOrderLogItemReceived(Subscription subscription, OrderLogItem item)
{
    if (item.SecurityId != _security.ToSecurityId())
        return;

    // Analyze order log item
    if (item.Side == Sides.Buy)
    {
        _buyCount++;
        _buyVolume += item.Volume;
    }
    else if (item.Side == Sides.Sell)
    {
        _sellCount++;
        _sellVolume += item.Volume;
    }

    // Update interface with analysis results
    this.GuiAsync(() =>
    {
        // Add item to OrderLogGrid
        _orderLogGrid.LogItems.Add(item);

        // Update statistics
        UpdateStatistics();
    });
}

// Update statistics
private void UpdateStatistics()
{
    BuyCountLabel.Content = $"Buys: {_buyCount}";
    SellCountLabel.Content = $"Sells: {_sellCount}";
    BuyVolumeLabel.Content = $"Buy volume: {_buyVolume}";
    SellVolumeLabel.Content = $"Sell volume: {_sellVolume}";

    // Calculate imbalance
    var volumeImbalance = _buyVolume - _sellVolume;
    var imbalancePercent = (_buyVolume + _sellVolume) > 0
        ? volumeImbalance / (_buyVolume + _sellVolume) * 100
        : 0;

    ImbalanceLabel.Content = $"Imbalance: {volumeImbalance:F2} ({imbalancePercent:F2}%)";
}
}

```

News

[NewsGrid](#) - a table for displaying news.

Main properties

- [NewsGrid.News](#) - list of news.
- [NewsGrid.FirstSelectedNews](#) - selected news item.
- [NewsGrid.SelectedNews](#) - selected news items.
- [NewsGrid.SubscriptionProvider](#) - news provider.

Below are code fragments demonstrating its usage:

```
<Window x:Class="SampleAlfa.NewsWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:loc="clr-namespace:StockSharp.Localization;assembly=StockSharp.Localization"
        xmlns:xaml="http://schemas.stocksharp.com/xaml"
        Title="{x:Static loc:LocalizedStrings.News}" Height="300" Width="1050">
    <xaml:NewsPanel x:Name="NewsPanel"/>
</Window>
```

```
private readonly Connector _connector = new Connector();
private void ConnectClick(object sender, RoutedEventArgs e)
{
    // Other connection actions

    // Set news provider
    _newsWindow.NewsPanel.SubscriptionProvider = _connector;

    // Subscribe to news reception event
    _connector.NewsReceived += OnNewsReceived;

    // Create a subscription to news
    var newsSubscription = new Subscription(DataType.News);
    _connector.Subscribe(newsSubscription);

    // Perform connection
    _connector.Connect();
}

// Handler for news reception event
private void OnNewsReceived(Subscription subscription, News news)
{
    // Add news to NewsGrid in user interface thread
    this.GuiAsync(() => _newsWindow.NewsPanel.NewsGrid.News.Add(news));
}
```

News Filtering

```
// Creating a subscription to news with filtering
public void SubscribeToFilteredNews(string source = null, DateTime? from = null)
{
    // Create a subscription to news
    var newsSubscription = new Subscription(DataType.News)
    {
        MarketData =
```



```

{
    // Set starting date for historical news
    From = from ?? DateTime.Today.AddDays(-7),

    // Optionally set news source
    NewsSource = source
}
};

// Subscribe to news reception event
_connector.NewsReceived += OnFilteredNewsReceived;

// Start the subscription
_connector.Subscribe(newsSubscription);
}

// Handler for filtered news reception events
private void OnFilteredNewsReceived(Subscription subscription, News news)
{
    // Check source filter
    if (subscription.MarketData.NewsSource != null &&
        !string.Equals(news.Source, subscription.MarketData.NewsSource,
            StringComparison.OrdinalIgnoreCase))
        return;

    // Add news to NewsGrid
    this.GuiAsync(() => _newsWindow.NewsPanel.NewsGrid.News.Add(news));

    // Output news information
    Console.WriteLine($"News: {news.Headline}");
    Console.WriteLine($"Source: {news.Source}");
    Console.WriteLine($"Time: {news.ServerTime}");
    if (!string.IsNullOrEmpty(news.Story))
        Console.WriteLine($"Text: {news.Story}");
}

```

Searching News by Keywords

```

// Method for filtering news by keywords
public void FilterNewsByKeywords(IEnumerable<string> keywords)
{
    var keywordsList = keywords.ToList();

    // If already subscribed to news,
    // just set the handler
    _connector.NewsReceived += (subscription, news) =>
    {
        // Check if the news headline contains any of the keywords
        bool containsKeyword = keywordsList.Any(keyword =>
            news.Headline.IndexOf(keyword, StringComparison.OrdinalIgnoreCase) >= 0);

        if (containsKeyword)
        {
            // Add news to NewsGrid
            this.GuiAsync(() => _newsWindow.NewsPanel.NewsGrid.News.Add(news));

            // Display notification
            ShowNotification($"New news on topic: {news.Headline}");
        }
    };
}

```

Trading

Trading

This section presents the components associated with own trading transactions: orders ([OrderGrid](#), [OrderWindow](#)), conditional orders ([OrderConditionalWindow](#)) and own trades ([MyTradeGrid](#)).

Recommended content

[Orders](#)

[New order creation](#)

[New stop order creation](#)

[Own trades](#)

Orders

Orders

[OrderGrid](#) is a table for displaying orders and conditional orders. In addition, the context menu of this table contains commands for operations with orders: registration, replacement, and cancellation of orders. Selecting a menu item generates events: [OrderGrid.OrderRegistering](#), [OrderGrid.OrderReRegistering](#), or [OrderGrid.OrderCanceling](#) respectively.

[image: GUI OrderGrid]

The operation itself (registration, replacement, cancellation) is not performed. The corresponding code needs to be written in the event handlers manually.

Main Members

- [OrderGrid.Orders](#) - list of orders.
- [OrderGrid.SelectedOrder](#) - selected order.
- [OrderGrid.SelectedOrders](#) - selected orders.
- [OrderGrid.AddRegistrationFail](#)([StockSharp.BusinessEntities.OrderFail](#) fail) - method that adds an order registration error message to the comment field.
- [OrderGrid.OrderRegistering](#) - order registration event (occurs after selecting the corresponding context menu item).
- [OrderGrid.OrderReRegistering](#) - order replacement event (occurs after selecting the corresponding context menu item).
- [OrderGrid.OrderCanceling](#) - order cancellation event (occurs after selecting the corresponding context menu item).

Below are code fragments showing its use. The code example is taken from *Samples/01_Basic/03_Orders*.

```
<Window x:Class="Sample.OrdersWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:loc="clr-namespace:StockSharp.Localization;assembly=StockSharp.Localization"
xmlns:xaml="http://schemas.stocksharp.com/xaml"
Title="{x:Static loc:LocalizedStrings.Orders}" Height="410" Width="930">
<xaml:OrderGrid x:Name="OrderGrid" x:FieldModifier="public"
    OrderCanceling="OrderGrid_OnOrderCanceling"
    OrderReRegistering="OrderGrid_OnOrderReRegistering" />
</Window>
```

```
private readonly Connector _connector = new Connector();

private void ConnectClick(object sender, RoutedEventArgs e)
{
    // Other code during connection...

    // Subscribe to the order received event
    _connector.OrderReceived += (subscription, order) =>
    {
        // Add orders to the OrderGrid table
        _ordersWindow.OrderGrid.Orders.TryAdd(order);
    };

    // To connect the connector
    _connector.Connect();
}
```

```

}

// Cancels all selected orders
private void OrderGrid_OnOrderCanceling(IEnumerable<Order> orders)
{
    // Iterate through selected orders and cancel each one
    foreach (var order in orders)
    {
        _connector.CancelOrder(order);
    }
}

// Opens an order editing window and performs replacement of the selected order
private void OrderGrid_OnOrderReRegistering(Order order)
{
    var window = new OrderWindow
    {
        Title = LocalizedStrings.Str2976Params.Put(order.TransactionId),
        SecurityProvider = _connector,
        MarketDataProvider = _connector,
        Portfolios = new PortfolioDataSource(_connector),
        Order = order.ReRegisterClone(newVolume: order.Balance)
    };

    if (window.ShowDialog(this))
        _connector.ReRegisterOrder(order, window.Order);
}

```

Working with Orders through Subscriptions

The modern approach to working with orders involves using subscriptions:

```

// Subscribe to the order received event
_connector.OrderReceived += OnOrderReceived;

// Order received handler
private void OnOrderReceived(Subscription subscription, Order order)
{
    // Check if the order belongs to the subscription we're interested in
    if (subscription == _ordersSubscription)
    {
        // Add the order to the table
        _ordersWindow.OrderGrid.Orders.TryAdd(order);

        // Additional order processing
        Console.WriteLine($"Order received: {order.TransactionId}, Status: {order.State}");

        // If the order is in a final state, update the UI
        if (order.State == OrderStates.Done || order.State == OrderStates.Failed)
        {
            this.GuiAsync(() => {
                // Update interface for completed orders
            });
        }
    }
}

```

Canceling Orders

```

// Modern approach to order cancellation
private void CancelOrder(Order order)
{
    try
    {
        _connector.CancelOrder(order);
    }
}

```

```

    // Log the action
    _logManager.AddInfoLog($"Order cancellation command sent {order.TransactionId}");
}
catch (Exception ex)
{
    _logManager.AddErrorLog($"Error when canceling order: {ex.Message}");
}
}

// Mass cancellation of orders
private void CancelAllOrders()
{
    var activeOrders = _ordersWindow.OrderGrid.Orders
        .Where(o => o.State == OrderStates.Active)
        .ToArray();

    foreach (var order in activeOrders)
    {
        CancelOrder(order);
    }
}

```

Handling Order Registration and Cancellation Errors

```

// Subscribe to order registration failures
_connector.OrderRegisterFailReceived += OnOrderRegisterFailed;

// Order registration failure handler
private void OnOrderRegisterFailed(Subscription subscription, OrderFail fail)
{
    // Add error information to OrderGrid
    _ordersWindow.OrderGrid.AddRegistrationFail(fail);

    // Log the error
    _logManager.AddErrorLog($"Order registration error: {fail.Error}");

    // Notify the user
    this.GuiAsync(() =>
    {
        MessageBox.Show(this,
            $"Failed to register order: {fail.Error}",
            "Registration Error",
            MessageBoxButton.OK,
            MessageBoxImage.Error);
    });
}

```

Own trades

Own trades

[MyTradeGrid](#) - a table to display own trades.

[image: GUI MytradeGrid]

Main members

- [MyTradeGrid.Trades](#) - list of trades.
- [MyTradeGrid.SelectedTrade](#) - the selected trade.
- [MyTradeGrid.SelectedTrades](#) - selected trades.

Below is the code snippet with its use. The code example is taken from *Samples/InteractiveBrokers/SampleIB*.

```
<Window x:Class="Sample.MyTradesWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:loc="clr-namespace:StockSharp.Localization;assembly=StockSharp.Localization"
  xmlns:xaml="http://schemas.stocksharp.com/xaml"
  Title="{x:Static loc:LocalizedStrings.MyTrades}" Height="284" Width="644">
  <xaml:MyTradeGrid x:Name="TradeGrid" x:FieldModifier="public" />
</Window>
```

```
private readonly Connector _connector = new Connector();
private void ConnectClick(object sender, RoutedEventArgs e)
{
    .....
    _connector.OwnTradeReceived += trade => _myTradesWindow.TradeGrid.Trades.Add(trade);
    .....
}
```

New order creation

New order creation

[OrderWindow](#) - window for creating an order.

[image: GUI OrderWindow]

If the connection supports registering a conditional order (stop-loss, take-profit), then in this window you can register a conditional order with advanced conditions by setting the **Advanced conditions** flag.

Basic properties

- [OrderWindow.Portfolios](#) - list of portfolios.
- [OrderWindow.MarketDataProvider](#) - market data provider.
- [OrderWindow.SecurityProvider](#) - security information provider.
- [OrderWindow.Order](#) - created order.

Code snippets using it are shown below. Sample code taken from *Samples/01_Basic/03_Orders*.

```
...
private readonly Connector _connector = new Connector();
...
private void NewOrderClick(object sender, RoutedEventArgs e)
{
    var wnd = new OrderWindow
    {
        Order = new Order { Security = SecurityPicker.SelectedSecurity },
        SecurityProvider = _connector,
        MarketDataProvider = _connector,
        Portfolios = new PortfolioDataSource(_connector),
    };
    if (wnd.ShowDialog(this))
        _connector.RegisterOrder(wnd.Order);
}
```

New stop order creation

New stop order creation

[OrderConditionalWindow](#) - the window for creating a conditional order.

[image: GUI OrderConditionalWindow]

Main properties

- [OrderConditionalWindow.Portfolios](#) - the list of portfolios.
- [OrderConditionalWindow.SecurityProvider](#) - provider of information about instruments.
- [OrderConditionalWindow.MarketDataProvider](#) - provider of market data.
- [OrderConditionalWindow.Adapter](#) - message adapter.
- [OrderConditionalWindow.Order](#) - the created order.

Below is a code snippet showing its use. The code example is taken from *Samples/InteractiveBrokers/SampleIB*.

```
...
private readonly Connector _connector = new Connector();
...
private void NewStopOrderClick(object sender, RoutedEventArgs e)
{
    var wnd = new OrderConditionalWindow
    {
        Order = new Order
        {
            Security = SecurityPicker.SelectedSecurity,
            Type = OrderTypes.Conditional,
            ExpiryDate = DateTime.Today
        },
        SecurityProvider = _connector,
        MarketDataProvider = _connector,
        Portfolios = new PortfolioDataSource(_connector),
        Adapter = _connector.Adapter
    };
    if (wnd.ShowDialog(this))
        _connector.RegisterOrder(wnd.Order);
}
```


Table of available market data

Table of available market data

[MarketDataGrid](#) - a table displaying available market data.

[image: API GUI MarketDataGrid]

The following is an example of the code for adding a [MarketDataGrid](#) table to the screen form.

```
<Window x:Class="MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:xaml="http://schemas.stocksharp.com/xaml"
  mc:Ignorable="d"
  Title="MainWindow" Height="662" Width="787" Left="10" Top="10">
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="180"/>
      <ColumnDefinition Width="180"/>
      <ColumnDefinition Width="923*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition Height="30"/>
      <RowDefinition/>
    </Grid.RowDefinitions>
    <xaml:MarketDataGrid x:Name="MarketDataGrid" Grid.Row="1" Grid.ColumnSpan="3" />
    <Button Grid.Row="0" Grid.Column="0" x:Name="Setting" Content="Setting" Click="Setting_Click" />
    <Button Grid.Row="0" Grid.Column="1" x:Name="Connect" Content="Connect" Click="Connect_Click" />
  </Grid>
</Window>
```

Buy/Sell panel

Buy/Sell panel

[BuySellPanel](#) - A special panel enabling you to quickly register an order at the best prices.

[image: API GUI BuySell]

Below is an example of the code for adding the [BuySellPanel](#).

```
<Window x:Class="MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:xaml="http://schemas.stocksharp.com/xaml"
  mc:Ignorable="d"
  Title="MainWindow" Height="662" Width="787" Left="10" Top="10">
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="180"/>
    <ColumnDefinition Width="180"/>
    <ColumnDefinition Width="923*"/>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="30"/>
    <RowDefinition/>
  </Grid.RowDefinitions>
  <xaml:BuySellPanel x:Name="BuySellPanel" Grid.Row="1" Grid.ColumnSpan="3"/>
  <Button Grid.Row="0" Grid.Column="0" x:Name="Setting" Content="Setting" Click="Setting_Click" />
  <Button Grid.Row="0" Grid.Column="1" x:Name="Connect" Content="Connect" Click="Connect_Click" />
</Grid>
</Window>
```

To fill the panel with data, you have to specify the market data source and the instrument source.

```
private void Connect_Click(object sender, RoutedEventArgs e)
{
    ...
    BuySellPanel.SecurityProvider = _connector;
    BuySellPanel.MarketDataProvider = _connector;
    ...
}
```

Strategies

Strategies

This section contains the components that are used when working with strategies.

- The table for displaying the statistics of the strategy - [StrategiesStatisticsPanel](#).

Recommended content

[Statistics](#)

Statistics

[StrategiesStatisticsPanel](#) - is a table to display parameters and statistics of the strategies.

[image: GUI StrategiesStatisticsPanel]

Main properties and methods

- [StrategiesStatisticsPanel.SelectedStrategy](#) - the selected strategy.
- [StrategiesStatisticsPanel.SelectedStrategies](#) - selected strategies.
- [StrategiesStatisticsPanel.AddStrategy](#)([StockSharp.Algo.Strategies.Strategy](#) strategy) - adds strategies to the table.
- [StrategiesStatisticsPanel.SetColumnVisibility](#)([System.String](#) name, [System.Windows.Visibility](#) visibility) - sets the visibility of the columns in the table.

Below is the code snippet with its use. The code example is taken from *Samples/Testing/SampleHistoryTestingParallel*.

```
<Window x:Class="SampleHistoryTestingParallel.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:loc="clr-namespace:StockSharp.Localization;assembly=StockSharp.Localization"
    xmlns:charting="http://schemas.stocksharp.com/xaml"
    Title="{x:Static loc:LocalizedStrings.XamlStr563}" Height="430" Width="525">

    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="100" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="10" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="10" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <Label Grid.Column="0" Grid.Row="0" Content="{x:Static loc:LocalizedStrings.XamlStr593}" />
        <TextBox x:Name="HistoryPath" Text="" Grid.Column="1" Grid.Row="0" />
        <Button x:Name="FindPath" Grid.Column="2" Grid.Row="0" Content="..." Width="25"
            HorizontalAlignment="Left" Click="FindPathClick" />
        <Button x:Name="StartBtn" Content="{x:Static loc:LocalizedStrings.Str2421}" Grid.Row="2"
            Grid.Column="0" Click="StartBtnClick" />
        <ProgressBar x:Name="TestingProcess" Grid.Column="1" Grid.Row="2" />
        <TabControl Grid.Row="4" Grid.ColumnSpan="3" Grid.Column="0" >
            <TabItem Header="{x:Static loc:LocalizedStrings.Equity}">
                <charting:EquityCurveChart x:Name="Curve" />
            </TabItem>
            <TabItem Header="{x:Static loc:LocalizedStrings.Str436}">
                <charting:StrategiesStatisticsPanel x:Name="Stat" ShowProgress="False"/>
            </TabItem>
        </TabControl>
    </Grid>
</Window>
```

```
var strategies = periods
    .Select(period =>
```

```

{
    ...
    // create strategy based SMA
    var strategy = new SmaStrategy(series, new SimpleMovingAverage { Length = period.Item1 }, new
SimpleMovingAverage { Length = period.Item2 })
    {
        Volume = 1,
        Security = security,
        Portfolio = portfolio,
        Connector = connector,
        // by default interval is 1 min,
        // it is excessively for time range with several months
        UnrealizedPnLInterval = ((stopTime - startTime).Ticks / 1000).To<TimeSpan>()
    };
    ...
    var curveItems = Curve.CreateCurve(LocalizedStrings.Str3026Params.Put(period.Item1,
period.Item2), period.Item3, DrawStyles.Line);
    strategy.PnLChanged += () =>
    {
        var data = new EquityData
        {
            Time = strategy.CurrentTime,
            Value = strategy.PnL,
        };
        this.GuiAsync(() => curveItems.Add(data));
    };
    Stat.AddStrategies(new[] { strategy });
    return strategy;
})
.ToEx(periods.Length);

```

Options

Options

The [S#](#) includes several graphic components to work with options.

- [OptionDesk](#) - the option desk.
- [OptionPositionChart](#) - the chart showing the position and the options Greeks regarding the underlying asset.

Recommended content

[Greeks](#)

[Volatility trading](#)

Option desk

Option desk

The [OptionDesk](#) graphical component is a table for displaying the option desk. It shows “Greeks”, implied volatility, the theoretical price, the best offer and bid for Put and Call options.

Below is the **OptionCalculator** example, which uses this component. The source code of the example can be found in the *Samples/06_Strategies/09_LiveOptionsQuoting* folder.

[image: option desk]

OptionCalculator example

1. In the XAML code, add the [OptionDesk](#) element and assign it the **Desk** name.

```
<Window x:Class="OptionCalculator.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:loc="clr-namespace:StockSharp.Localization;assembly=StockSharp.Localization"
        xmlns:xaml="http://schemas.stocksharp.com/xaml"
        Title="{x:Static loc:LocalizedStrings.XamlStr396}" Height="400" Width="1030">
    <Grid Margin="5,5,5,5">

        .....

        <xaml:OptionDesk x:Name="Desk" Grid.Row="6" Grid.ColumnSpan="3" Grid.Column="0" />

    </Grid>
</Window>
```

2. In the C# code, create a connection and subscribe to the necessary events.

```
...
public readonly Connector Connector = new Connector();
...
// subscribe on connection successfully event
Connector.Connected += () =>
{
    // update gui labels
    this.GetAsync(() => ChangeConnectStatus(true));
};
// subscribe on disconnection event
Connector.Disconnected += () =>
{
    // update gui labels
    this.GetAsync(() => ChangeConnectStatus(false));
};
// subscribe on connection error event
Connector.ConnectionError += error => this.GetAsync(() =>
{
    // update gui labels
    ChangeConnectStatus(false);
    MessageBox.Show(this, error.ToString(), LocalizedStrings.ErrorConnection);
});
// fill underlying asset's list
Connector.SecurityReceived += (sub, security) =>
{
    if (security.Type == SecurityTypes.Future)
        _assets.Add(security);
};
Connector.Level1Received += (sub, security) =>
{
```

```

    if (_model.UnderlyingAsset == security || _model.UnderlyingAsset.Id ==
security.UnderlyingSecurityId)
        _isDirty = true;
};
// subscribing on tick prices and updating asset price
Connector.TickTradeReceived += (sub, trade) =>
{
    if (_model.UnderlyingAsset == trade.Security || _model.UnderlyingAsset.Id ==
trade.Security.UnderlyingSecurityId)
        _isDirty = true;
};
Connector.NewPosition += position => this.GuiAsync(() =>
{
    var asset = SelectedAsset;
    if (asset == null)
        return;
    var assetPos = position.Security == asset;
    var newPos = position.Security.UnderlyingSecurityId == asset.Id;
    if (!assetPos && !newPos)
        return;
    if (assetPos)
        PosChart.AssetPosition = position;
    if (newPos)
        PosChart.Positions.Add(position);
    RefreshChart();
});
Connector.PositionChanged += position => this.GuiAsync(() =>
{
    if ((PosChart.AssetPosition != null && PosChart.AssetPosition == position) ||
PosChart.Positions.Cache.Contains(position))
        RefreshChart();
});
try
{
    if (File.Exists(_settingsFile))
        Connector.Load(new JsonSerializer<SettingsStorage>().Deserialize(_settingsFile));
}
...

```

3. When connecting, set the message provider for market data.

```

private void ConnectClick(object sender, RoutedEventArgs e)
{
    if (!_isConnected)
    {
        ConnectBtn.IsEnabled = false;
        _model.Clear();
        _model.MarketDataProvider = Connector;
        ...
        Connector.Connect();
    }
    else
        Connector.Disconnect();
}

```

4. When receiving instruments, we add the underlying assets to the list.

```

// fill underlying asset's list
Connector.SecurityReceived += (sub, security) =>
{
    if (security.Type == SecurityTypes.Future)
        _assets.Add(security);
};

```

5. When selecting an instrument:

- Fill the array with a chain of options, where the selected instrument acts as the underlying asset;

- Assign this array to the [OptionDeskModel.Options](#) property;
- Clear the options board values using the [OptionDeskModel.Clear](#) method.

```
private void Assets_OnSelectionChanged(object sender, SelectionChangedEventArgs e)
{
    var asset = SelectedAsset;
    _model.UnderlyingAsset = asset;
    _model.Clear();
    _options.Clear();
    var options = asset.GetDerivatives(Connector);
    foreach (var security in options)
    {
        _model.Add(security);
        _options.Add(security);
    }
    ProcessPositions();
}
```

Recommended content

[Greeks](#)

Position chart

Position chart

The [OptionPositionChart](#) graphical component is a chart showing the position and the options “Greeks” related to the underlying asset.

The following is the `SampleOptionQuoting` example, in which this chart is used. The source code of the example can be found in the `Samples/06_Strategies/09_LiveOptionsQuoting` folder.

[image: option volsmile]

SampleOptionQuoting example

1. In the XAML code, add the [OptionPositionChart](#) element and assign it the **PosChart** name.

```
<Window x:Class="OptionCalculator.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:loc="clr-namespace:StockSharp.Localization;assembly=StockSharp.Localization"
        xmlns:xaml="http://schemas.stocksharp.com/xaml"
        Title="{x:Static loc:LocalizedStrings.XamlStr396}" Height="400" Width="1030">
    <Grid Margin="5,5,5,5">

        .....

        <xaml:OptionPositionChart x:Name="PosChart" Grid.Row="7" Grid.Column="0" Grid.ColumnSpan="6"
    />
    </Grid>
</Window>
```

2. In the C# code, create a connection and subscribe to the necessary events.

```
...
public readonly Connector Connector = new Connector();
...
// subscribe on connection successfully event
Connector.Connected += () =>
{
    // update gui labels
    this.GuiAsync(() => ChangeConnectStatus(true));
};
// subscribe on disconnection event
Connector.Disconnected += () =>
{
    // update gui labels
    this.GuiAsync(() => ChangeConnectStatus(false));
};
// subscribe on connection error event
Connector.ConnectionError += error => this.GuiAsync(() =>
{
    // update gui labels
    ChangeConnectStatus(false);
    MessageBox.Show(this, error.ToString(), LocalizedStrings.ErrorConnection);
});
// fill underlying asset's list
Connector.SecurityReceived += (sub, security) =>
{
    if (security.Type == SecurityTypes.Future)
        _assets.Add(security);
};
Connector.Level1Received += (sub, security) =>
{
```

```

    if (_model.UnderlyingAsset == security || _model.UnderlyingAsset.Id ==
security.UnderlyingSecurityId)
        _isDirty = true;
};
// subscribing on tick prices and updating asset price
Connector.TickTradeReceived += (sub, trade) =>
{
    if (_model.UnderlyingAsset == trade.Security || _model.UnderlyingAsset.Id ==
trade.Security.UnderlyingSecurityId)
        _isDirty = true;
};
Connector.NewPosition += position => this.GuiAsync(() =>
{
    var asset = SelectedAsset;
    if (asset == null)
        return;
    var assetPos = position.Security == asset;
    var newPos = position.Security.UnderlyingSecurityId == asset.Id;
    if (!assetPos && !newPos)
        return;
    RefreshChart();
});
Connector.PositionChanged += position => this.GuiAsync(() =>
{
    if ((PosChart.AssetPosition != null && PosChart.AssetPosition == position) ||
PosChart.Positions.Cache.Contains(position))
        RefreshChart();
});
try
{
    {
        if (File.Exists(_settingsFile))
            Connector.Load(new JsonSerializer<SettingsStorage>().Deserialize(_settingsFile));
    }
}
...

```

3. When connecting, set the initial control settings:

1. Resetting the model of the [OptionPositionChart.Model](#) control;
2. Redrawing the chart with the initial values
[OptionPositionChart.Refresh\(System.Nullable<System.Decimal> assetPrice,](#)
[System.Nullable<System.DateTimeOffset> currentTime, System.Nullable<System.DateTimeOffset>](#)
[expiryDate \);](#)
3. Specifying message provider for market data and instruments.

```

private void ConnectClick(object sender, RoutedEventArgs e)
{
    if (!_isConnected)
    {
        ConnectBtn.IsEnabled = false;
        ...
        PosChart.Model = null;
        ...
        PosChart.MarketDataProvider = Connector;
        PosChart.SecurityProvider = Connector;
        PosChart.PositionProvider = Connector;
        Connector.Connect();
    }
    else
        Connector.Disconnect();
}

```

4. When receiving instruments, we add the underlying assets to the list.

```

Connector.SecurityReceived += (sub, security) =>
{
    if (security.Type == SecurityTypes.Future)

```

```

        _assets.Add(security);
    };

```

5. Upon changing the Level1 of the underlying instrument or options, as well as getting a new trade we set the `_isDirty` flag. This allows to call the `RefreshChart` method (see below) in the timer event (the code is omitted) to redraw the chart. Thus we control the frequency of redrawing.

```

Connector.Level1Received += (sub, security) =>
{
    if (_model.UnderlyingAsset == security || _model.UnderlyingAsset.Id ==
        security.UnderlyingSecurityId)
        _isDirty = true;
};
Connector.TickTradeReceived += (sub, trade) =>
{
    if (_model.UnderlyingAsset == trade.Security || _model.UnderlyingAsset.Id ==
        trade.Security.UnderlyingSecurityId)
        _isDirty = true;
};

```

6. In the new position occurrence event handler, we call `RefreshChart` to redraw the chart.

```

Connector.NewPosition += position => this.GuiAsync(() =>
{
    var asset = SelectedAsset;
    if (asset == null)
        return;
    var assetPos = position.Security == asset;
    var newPos = position.Security.UnderlyingSecurityId == asset.Id;
    if (!assetPos && !newPos)
        return;
    RefreshChart();
});
Connector.PositionChanged += position => this.GuiAsync(() =>
{
    if ((PosChart.AssetPosition != null && PosChart.AssetPosition == position) ||
        PosChart.Positions.Cache.Contains(position))
        RefreshChart();
});

```

7. This method redraws the chart:

```

private void RefreshChart()
{
    var asset = SelectedAsset;
    var trade = asset.LastTrade;
    if (trade != null)
        PosChart.Refresh(trade.Price);
}

```

Recommended content

[Volatility trading](#)

Object properties editing table

Object properties editing table

[PropertyGridEx](#) - the table for editing object properties. The component includes a set of additional editors for system types and [S#](#) types.

[image: GUI PropertyDataGridEx]

[PropertyGridEx](#) has its own editors for the following types:

- [Unit](#).
- [Security](#).
- [Portfolio](#).
- [ExchangeBoard](#).
- [Exchange](#).
- **ExtensionInfo** directory.
- [System.TimeSpan](#), [System.DateTime](#) and [System.DateTimeOffset](#).
- [System.Net.EndPoint](#) and [System.Net.IPAddress](#).
- [System.Security.SecureString](#).
- [System.Text.Encoding](#).
- [System.Enum](#).

Main properties

- [PropertyGridEx.SecurityProvider](#) - the provider of information about instruments.
- [PropertyGridEx.ExchangeInfoProvider](#) - the provider of site information.
- [PropertyGridEx.Portfolios](#) - the list of available portfolios.
- **SelectedObject** - the object whose properties will be displayed in the table.

Below is the code snippet with its use. The code example is taken from *Samples/Fix/SampleFix*.

```
<Window x:Class="SampleFix.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:loc="clr-namespace:StockSharp.Localization;assembly=StockSharp.Localization"
    xmlns:propertyGrid="http://schemas.stocksharp.com/xaml"
    Title="{x:Static loc:LocalizedStrings.XamlStr540}" Height="110" Width="512">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="24" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <StackPanel Grid.Column="0" Grid.Row="0" Grid.ColumnSpan="5" Orientation="Horizontal">
```

```

<xctk:DropDownButton Content="{x:Static loc:LocalizedStrings.TransactionSession}">
  <xctk:DropDownButton.DropDownContent>
    <propertyGrid:PropertyGridEx x:Name="TransactionSessionSettings" />
  </xctk:DropDownButton.DropDownContent>
</xctk:DropDownButton>
<xctk:DropDownButton Content="{x:Static loc:LocalizedStrings.MarketDataSession}">
  <xctk:DropDownButton.DropDownContent>
    <propertyGrid:PropertyGridEx x:Name="MarketDataSessionSettings" />
  </xctk:DropDownButton.DropDownContent>
</xctk:DropDownButton>
</StackPanel>

  <Button x:Name="ConnectBtn" Background="LightPink" Grid.Column="0" Grid.Row="1" Grid.RowSpan="2"
Content="{x:Static loc:LocalizedStrings.Connect}" Click="ConnectClick" />
  <Button x:Name="ShowSecurities" Grid.Column="1" Grid.Row="1" IsEnabled="False" Content="{x:Static
loc:LocalizedStrings.Securities}" Click="ShowSecuritiesClick" />
  <Button x:Name="ShowPortfolios" Grid.Column="2" Grid.Row="1" IsEnabled="False" Content="{x:Static
loc:LocalizedStrings.Portfolios}" Click="ShowPortfoliosClick" />
  <Button x:Name="ShowStopOrders" Grid.Column="3" Grid.Row="1" IsEnabled="False" Content="{x:Static
loc:LocalizedStrings.StopOrders}" Click="ShowStopOrdersClick" />
  <Button x:Name="ShowNews" Grid.Column="4" Grid.Row="1" IsEnabled="False" Content="{x:Static
loc:LocalizedStrings.News}" Click="ShowNewsClick" />

  <Button x:Name="ShowTrades" Grid.Column="1" Grid.Row="2" IsEnabled="False" Content="{x:Static
loc:LocalizedStrings.Ticks}" Click="ShowTradesClick" />
  <Button x:Name="ShowMyTrades" Grid.Column="2" Grid.Row="2" IsEnabled="False" Content="{x:Static
loc:LocalizedStrings.MyTrades}" Click="ShowMyTradesClick" />
  <Button x:Name="ShowOrders" Grid.Column="3" Grid.Row="2" IsEnabled="False" Content="{x:Static
loc:LocalizedStrings.Orders}" Click="ShowOrdersClick" />
</Grid>
</Window>

```

```

private readonly Connector _connector = new Connector();
public MainWindow()
{
  InitializeComponent();
  Title = Title.Put("FIX");
  _ordersWindow.MakeHideable();
  _myTradesWindow.MakeHideable();
  _tradesWindow.MakeHideable();
  _securitiesWindow.MakeHideable();
  _stopOrdersWindow.MakeHideable();
  _portfoliosWindow.MakeHideable();
  _newsWindow.MakeHideable();
  if (File.Exists(_settingsFile))
  {
    _connector._connector.Load(new JsonSerializer<SettingsStorage>().Deserialize(_settingsFile));
  }
  MarketDataSessionSettings.SelectedObject =
((ChannelMessageAdapter)_connector.MarketDataAdapter).InnerAdapter;
  TransactionSessionSettings.SelectedObject =
((ChannelMessageAdapter)_connector.TransactionAdapter).InnerAdapter;
  Instance = this;
  _connector.LogLevel = LogLevels.Debug;
  _logManager.Sources.Add(_connector);
  _logManager.Listeners.Add(new FileLogListener { LogDirectory = "StockSharp_Fix" });
}

```

Connection settings window

Connection settings window

[ConnectorWindow](#) - a special window for configuring adapters for connecting a connector.

[image: API GUI ConnectorWindow]

Here is the connection settings window. From the drop-down list (opens with the '+' button), you need to select the necessary adapters and configure their properties in the properties window located on the right.

This window should be called through the [Extensions.Configure](#)([StockSharp.Algo.Connector](#) connector, [System.Windows.Window](#) owner), extension method, into which the [Connector](#) and the parent window are passed. If the configuration is successful, the [Extensions.Configure](#)([StockSharp.Algo.Connector](#) connector, [System.Windows.Window](#) owner) extension method will return 'true'. Below is the code to call the connector connection settings window and save the settings to a file.

```
private void Setting_Click(object sender, RoutedEventArgs e)
{
    if (_connector.Configure(this))
    {
        new JsonSerializer<SettingsStorage>().Serialize(_connector.Save(), _connectorFile);
    }
}
```

| The connection correctness can be checked using the **Check** button.

The result of this window will be to create and add adapters to the list of *internal adapters* of the [Connector.Adapter](#) property.

For more information about saving and loading connector settings, see [Save and load settings](#).

Risk settings window

Risk settings window

[AlertSettingsWindow](#) - A special window for configuring risk control.

[image: API GUI RiskWindow]

The following is an example of the code to call the risk control settings window for the strategy.

```
private void RiskButton_OnClick(object sender, RoutedEventArgs e)
{
    var wnd = new RiskWindow();
    wnd.Rules.AddRange(Strategy.RiskManager.Rules.Select(r => r.Clone()));
    if (!wnd.ShowDialog(this))
        return;
    Strategy.RiskManager.Rules.Clear();
    Strategy.RiskManager.Rules.AddRange(wnd.Rules);
}
```


Commission settings window

Commission settings window

[CommissionWindow](#) - A special window for setting the rules for charging a commission.

[image: API CommissionWindow]

Below is an example of the code to call a window for setting the rules for charging a commission.

```
private void RiskButton_OnClick(object sender, RoutedEventArgs e)
{
    var wnd = new CommissionWindow();
    wnd.Rules.AddRange(Strategy.RiskManager.Rules.Select(r => r.Clone()));
    if (!wnd.ShowDialog(this))
        return;
    Strategy.RiskManager.Rules.Clear();
    Strategy.RiskManager.Rules.AddRange(wnd.Rules);
}
```

Notification settings window

Notification settings window

[AlertSettingsWindow](#) - window for configuring notifications for certain events

[image: API GUI AlertWindow]

You can configure notifications about changing the following data types: Portfolio, Client code, Broker, Depository, Server time, Transaction, Data type, Cancel, Order ID, Order ID (string), Order ID (platform), Derivative, Derivative (string), Price, Volume (order), Volume (trade), Visible volume, Direction, Balance, Order type, Status, Comment, Order message, System order, Order expiration time, Execution condition, Price, Trade initiator, Open interest, Error, Condition, Uptrend, Commission, Delay, Slippage, Identifier (user), Currency, P/L, Position, Market maker.

Notifications can be of the following form:

- **Window** - a small pop-up window with a message will appear in the corner of the screen.
- **Melody** - the melody will play.
- **SMS** - a message will be sent by SMS.
- **Email** - a message will be sent by email.
- **Speech** - a message will be spoken by the computer-generated voice.
- **Log** - a message will be sent to the Designer Logs window.
- **Disabled** - a notification will not be displayed.

Themes (light and dark)

Themes of S# graphic components

For all S# graphic elements, there are several different themes. Below are the two most popular themes.

[image: API GUI Thems 01]

[image: API GUI Thems 02]

To install the application theme, just write one line. For example, to set the VisualStudio 2017 dark theme, you have to specify the line:

```
...  
ThemeExtensions.ApplyDefaultTheme();  
...
```

Since all S# graphics elements are based on **DevExpress** graphic elements, you need to add the appropriate **DevExpress** libraries (**DevExpress.Xpf.Core**, **DevExpress.Xpf.Themes.VS2017Dark**, etc.)

Shell

Shell is a ready-made graphical framework with the ability to change quickly to your needs and with fully open source code in the C# language (enough basic knowledge).

You don't need to waste time creating a GUI (Graphical User Interface). A ready-made robot is created faster without sacrificing application usability. Basic functionality for testing, trading, connecting to data sources, displaying charts, portfolios, positions, orders and trades is already in the box.

[image: Shell Title 00]

The main features of the trading robot shell:

1. Full source code. Ideal for custom robots, or for your solutions.
2. **Support for more than 70 connections** to various exchanges of the world [Connectors](#).
3. Flexible user interface.
4. Testing strategies (statistics, equity, report).
5. Save and restore strategy settings.
6. Simultaneous launch of strategies.
7. Detailed information about the strategy operation (orders, trades, position, profit, logs, etc.). Everything is clear and simple.
8. Running strategies on schedule.

Recommended content

[Installing Shell](#)

Installing Shell

Installing Shell

Install and remove apps.

Quick start

Quick start

Shell - is focused on programming in **C#** language in Visual Studio environment or similar.

When you start the **Shell** project, the Solution Explorer displays the Shell project:

[image: Shell Quick start 00]

The **Strategies** folder contains three strategies included in **Shell**, a shell for default strategies, as well as some auxiliary interfaces.

The project can be launched without any preliminary preparation to see how it operates on the strategies already included in **Shell**.

After launch, we will see a window like this:

[image: Shell Quick start 01]

The connection settings and connection buttons, as well as the save button for the current Shell configuration, are located at the top of the screen. The main tabs are also located there.

Go to the connection settings and select the necessary connection. How to set up a connection is described in **Connections settings**.

The next step is to connect by clicking the **Connect** button *[image: Designer The quick access toolbar 00]*.

After connecting, on the **Common** tab, you can see the portfolios, securities, orders, and own trades received from the connection.

[image: Shell Quick start 02]

Go to the Real-time tab and click the **Add** *[image: Designer Creation tool 00]* button to add a strategy for trading.

[image: Shell Quick start 03]

Once the strategy is added, fill in its basic parameters such as **Security**, **Portfolio**, etc. Click the Start strategy button to launch it.

[image: Shell Quick start 04]

Similar to the **Real-time** tab, you can run a strategy test on historical data on the **Emulation** tab.

[image: Shell Quick start 05]

Common

Common

The [Common](#) tab allows you to view general information on portfolios, orders, trades, and securities. You can also set the theme of the [Shell](#) interface here.

[image: Shell Common 00]

On the **Securities** tab, you can upload securities from the local storage and modify security settings.

[image: Shell Common 01]

Recommended content

[Real-time](#)

Real-time

Real-time

The [Real-time](#) tab allows you to manage strategies launched in trading.

[image: Shell realtime 00]

By clicking the **Add** *[image: Designer Creation tool 00]* button, you can add a strategy to start trading. Each added strategy is opened on a separate tab and added to the list of strategies on the **Strategies** tab.

On the **Strategies** tab, you can delete, start, stop, and configure strategies. The same can be done on an individual strategy tab. The individual strategy tab has more extensive information on the strategy, statistics, a list of registered orders and trades of the strategy, etc.

Recommended content

[Common](#)

Emulation

Emulation

The [Emulation](#) tab allows you to test strategies on historical data.

[image: Shell emulation 00]

By clicking the **Add** *[image: Designer Creation tool 00]* button, you can add a strategy for testing. Each added strategy is opened on an individual tab.

An individual strategy tab has extensive information about the strategy, statistics, a list of registered orders and trades of the strategy, etc. On this tab, you can configure testing parameters, the path to historical data, the start and end dates of testing, and the format of the historical data storage.

Recommended content

[RemoteManager](#)

RemoteManager

The [RemoteManager](#) tab allows you to enable remote control mode. To enable this mode, you must go to the user configuration menu.

[image: Shell RemoteManager 00]

In the window that appears, set your **login** and **password**.

[image: Shell RemoteManager 01]

Then you need to enable the **server mode**.

[image: Shell RemoteManager 02]

You can now connect to Shell from another Shell.

To do this, you need to run **another Shell**. In it, go to the connection settings.

[image: Shell RemoteManager 03]

In the window that opens, set up the FIX connection.

[image: Shell RemoteManager 04]

Then press the Connect button.

[image: Shell RemoteManager 05]

When connected, all existing strategies in the Shell server will be available in the Shell client.

[image: Shell RemoteManager 06]

By clicking the Add button, you can add another strategy to trade.

[image: Shell RemoteManager 07]

Because the Shell client supports multiple servers, when adding a strategy you must select the server on the left. All strategies available on the server will appear on the right.

[image: Shell RemoteManager 08]

After adding a strategy, it will appear in the list of strategies.

[image: Shell RemoteManager 09]

When selecting a strategy, there will be tabs on the right with the strategy settings, as well as its statistics.

After changing the strategy settings, be sure to click the Apply changes button, otherwise the changes will not be applied to the strategy.

[image: Shell RemoteManager 10]

If the strategy has a command other than Start/Stop, then to apply it you must set it in the next field.

[image: Shell RemoteManager 11]

And click the send command button.

To set your command in the strategy, you need to override the [Strategy.ApplyCommand](#)([StockSharp.Messages.CommandMessage](#) cmdMsg) method.

```
public virtual void ApplyCommand(CommandMessage cmdMsg)
```

The [Strategy](#) base class only controls the strategy start and stop.

Recommended content

[Connections settings](#)

Connections settings

Connections settings

See [Connections settings](#).

Connector settings

Connector settings

See [Connector settings](#).

Create strategy

Create strategy

To create your own strategy, create a folder for your strategy in the Strategies folder.

[image: Shell custom strategy 00]

Using SmaStrategy as an example, create the strategy itself.

If you need to add your own testing or monitoring panel for a strategy, then the strategy must implement the IHaveTestControl and IHaveMonitoringControl interfaces, respectively.

```
public class SmaStrategy : Strategy, IHaveMonitoringControl, IHaveTestControl
{
    ...
    #region MonitoringControl
    public BaseStudioControl AddMonitoringPanel()
    {
        var usercontrol = new SmaMonitoringControl();
        usercontrol.Init(this);
        return usercontrol;
    }
    #endregion
    #region TestingControl
    public BaseStudioControl AddTestPanel()
    {
        var usercontrol = new SmaTestingControl();
        usercontrol.Init(this);
        return usercontrol;
    }
    #endregion
    ...
}
```

You also need to create the panels themselves. How to build your own testing or monitoring panel is described in [Create strategy panel](#).

If there are enough testing or monitoring panels for the strategy, which are used for default strategies, then you do not need to implement the IHaveTestControl and IHaveMonitoringControl interfaces. Shell will run the default testing or monitoring panels by itself.

[image: Shell custom strategy 01]

In order for the created strategy to be available in the strategy selection window, it must be added to the **DictionaryStrategies** dictionary of the main window.

```
...
//-----
DictionaryStrategies = new ObservableDictionary<Guid, Strategy>
{
    { new SmaStrategy().GetTypeId(), new SmaStrategy() },
    { new StairsTrendStrategy().GetTypeId(), new StairsTrendStrategy() },
    { new StairsCountertrendStrategy().GetTypeId(), new StairsCountertrendStrategy() }
};
//-----
...
```

In order for the strategy to be saved and then loaded, you need to set the strategy parameter in the strategy designer.

```

public class SmaStrategy : Strategy, IHaveMonitoringControl, IHaveTestControl
{
    ...
    public SmaStrategy()
    {
        ...
        this.Param("TypeId", GetType().GUID);
        ...
    }
    ...
}

```

To save additional fields, you must override the **Load** and **Save** methods.

```

public class SmaStrategy : Strategy, IHaveMonitoringControl, IHaveTestControl
{
    ...
    #region Load
    public override void Load(SettingsStorage storage)
    {
        base.Load(storage);
        try
        {
            _securityStr = storage.GetValue<string>(nameof(Security));
            _portfolioStr = storage.GetValue<string>(nameof(Portfolio));
            LongSmaLength = storage.GetValue<int>(nameof(LongSmaLength));
            ShortSmaLength = storage.GetValue<int>(nameof(ShortSmaLength));
            Series.CandleType = storage.GetValue(nameof(Series.CandleType), Series.CandleType);
            Series.Arg = storage.GetValue(nameof(Series.Arg), Series.Arg);
        }
        catch (Exception e)
        {
            e.LogError();
        }
    }
    #endregion
    #region Save
    public override void Save(SettingsStorage storage)
    {
        base.Save(storage);
        storage.SetValue(nameof(Security), Security?.Id);
        storage.SetValue(nameof(Portfolio), Portfolio?.Name);
        storage.SetValue(nameof(LongSmaLength), LongSmaLength);
        storage.SetValue(nameof(ShortSmaLength), ShortSmaLength);
        if (Series.CandleType != null)
            storage.SetValue(nameof(Series.CandleType), Series.CandleType.GetTypeName(false));
        if (Series.Arg != null)
            storage.SetValue(nameof(Series.Arg), Series.Arg);
    }
    #endregion
    ...
}

```

Recommended content

[Create strategy panel](#)

Create strategy panel

Create strategy panel

Custom panels are special controls created by S# to facilitate working with DevExpress elements.

First, you need to create a simple UserControl in the XAML folder of your strategy.

[image: Shell custom strategy panel 00]

[image: Shell custom strategy panel 01]

Replace UserControl with `controls:BaseStudioControl`.

```
<controls:BaseStudioControl>
...
</controls:BaseStudioControl>
```

Then implement your own panel logic similar to the existing strategy panels.

In order for the [Real-time](#) panel to see the strategy in your panel, your strategy must be set as a property:

```
public partial class SmaMonitoringControl
{
    ...
    public Strategy Strategy { get; set; }
    ...
}
```

To save the strategy settings, you must override the **Load** and **Save** methods in the panel.

```
public partial class SmaMonitoringControl
{
    ...
    public override void Load(SettingsStorage storage)
    {
        base.Load(storage);
        try
        {
            Strategy =
MainWindow.Instance.CreateStrategy(storage.GetValue<SettingsStorage>(nameof(Strategy)));
            Init(Strategy);
        }
        catch (Exception e)
        {
            e.LogError();
        }
    }
    public override void Save(SettingsStorage storage)
    {
        base.Save(storage);
        storage.SetValue(nameof(Strategy), Strategy.Save());
    }
    ...
}
```

Recommended content

[Create strategy](#)

Run strategies from Designer

Run strategies from Designer

[Shell](#) can run strategies created in [Designer](#).

To launch a strategy created in [Designer](#), select it on the [Real-time](#) tab and click the **Add Designer strategy** button. In the window that appears, choose the strategy file exported from [Designer](#).

[image: Shell run Designer strategy 00]

After that, it will appear in the list of available strategies.

[image: Shell run Designer strategy 01]

By selecting the added strategy, you can set the necessary parameters and launch it for trading.

[image: Shell run Designer strategy 02]

Recommended content

[Create strategy](#)

Store

Store

The **Store** is our marketplace and showcase for products created by the company and the community. It helps developers promote their work and makes it easy for users to discover, download, and update extensions for the platform.

Why publish

Sharing your work in the Store lets developers reach a wider audience, gather feedback, and monetize their ideas. It also grows the overall ecosystem and provides other traders with ready-to-use tools.

Uploading content

Anyone can publish their own robot, trading strategy, or indicator. Publication is performed through the [Installer](#):

1. Prepare the package with your project files.
2. Open the Installer and sign in.
3. Choose **Publish** and follow the step-by-step wizard to upload the package, description, and screenshots.
4. Decide whether the item will be free or specify a price.

After submission the material is reviewed by our team. Once approved it becomes available in the Store catalogue.

Monetization and commission

Authors can distribute their work for free or sell it. Payments are processed by the company, which retains a commission from each sale. To learn the current commission rate and payout terms, please contact us at info@stocksharp.com.

MATLAB

MATLAB

[image: matlab stocksharp]

The [MATLAB](#) is designed for traders who develop trading algorithms within the [MATLAB MathWorks](#) environment.

The integration connector [MATLAB](#) allows to connect to all brokers and exchanges that are supported by the [S#](#) (+30 connection types, all world markets). MATLAB scripts will be able to get real-time data from the markets and to send trading orders in them.

Detailed settings, scripts, and documentation is available to customers who have purchased the [MATLAB](#).

UDP Dumper

UDP Dumper

The app **UDPDumper** is designed to accumulate UDP packets. Using this program, you can check the valid of the network settings made by a broker or exchange. And also, accumulate data for further tests of a UDP-based connectors (e.g., [FAST](#) or SBE).

To install need use [Installer](#).

Setup and run

1. On first launch, the app shows the following:*[image: Dumper 1]*
2. To add network feeds, you can add both manually and automatically load all feeds from the config files of exchanges. To do this, click on the button:*[image: Dumper 2]*
3. In the window that appears, you need to find the desired config file from the exchange and open it:*[image: Dumper 3]*
4. All feeds with IP addresses and ports settings will be loaded from a file:*[image: Dumper 4]*
5. It is necessary to select the necessary feeds, and click on the download start button:*[image: Dumper 5]*
6. В случае успешных настроек программа начнет получать UDP датаграммы и записывать на диск. Программа будет писать кол-во полученных байтов для каждого потока:*[image: Dumper 6]*
[!CAUTION] If the settings are successful, the program will start receiving UDP datagrams and writing them to disk. The app will write the number of bytes received for each feed.
7. The app **UDPDumper** is written with a graphical interface. If the program is launched without a graphical interface (as well as running under Linux operating systems, etc.) you can use the **UDPDumper.Console**, program, which is a console and cross-platform version.

The app **UDPDumper.Console** takes as a parameter the path to the file created by the UI version (exactly the UI version, and **not an exchange config**):

```
StockSharp.UdpDumper.Console.exe settings.json
```

8. To test on the accumulated data of the connector, you can use the dump mode. More details [Dump mode](#).

Source codes

Source codes

Currently, the [S#](#) core is being developed as part of the Open Source community, is available as source code, and is hosted in the [GitHub/StockSharp](#) repository.

[Instructions for using GitHub](#)

List of components available with source code:

- Common classes for creating your own connections.
- Format of the market data storage.
- Trading simulator.
- History simulator (backtester).
- Indicators (more than 70) of technical analysis.
- Algorithms for calculating profit-loss, slippage, delay.
- Algorithms for building candles of any time frame, as well as non-time-based candles (tick, range, etc.).
- Logging.
- Import and export.

The source codes of all closed components, as well as ready-made programs, are available upon purchase. For more information about the cost of source codes, see [Source Code Cost](#).

Recommended content

[Installation instruction](#)

Reference materials

Reference materials

Materials from [S#](#):

- Advanced training - [StockSharp Education](#).
- Source code of S# core - [GitHub/StockSharp](#).
- Free and paid products - [Shell](#).
- YouTube channel - [StockSharp YouTube](#).
- Facebook - [StockSharp Facebook](#).
- Twitter - [StockSharp Twitter](#).

Contacts:

- Platform site - <https://stocksharp.com>.
- Blog - [StockSharp Blog](#).
- Telegram chat - [Chat](#).