**Министерство науки и высшего образования Российской Федерации**

**Федеральное государственное автономное образовательное учреждение**

**высшего образования**

**«КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**

Институт вычислительной математики и информационных технологий

Кафедра системного анализа и информационных технологий

Направление подготовки: 02.03.02 — Фундаментальная информатика и

информационные технологии

Профиль: системный анализ и информационные технологии

КУРСОВАЯ РАБОТА

**Бот для биржевой торговли**

Студент 3 курса

Группа 09-131

«\_\_\_»\_\_\_\_\_\_\_\_\_\_2024 г. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Шарифуллин Р.Р.

Научный руководитель

канд.физ.-мат.наук., доцент,

доцент кафедры САИТ

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_2024 г. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Шаймухаметов Р.Р.

Казань – 2024

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_Toc167279965)

[1 Теоретический обзор и анализ предметной области 5](#_Toc167279966)

[1.1 Определение алгоритмической торговли и основные принципы 5](#_Toc167279967)

[1.2 Виды торговых ботов 6](#_Toc167279968)

[1.3 Преимущества и недостатки алгоритмической торговли 9](#_Toc167279969)

[1.4 Алгоритмические торговые стратегии 10](#_Toc167279970)

[2 Архитектура и проектирование торгового бота 14](#_Toc167279971)

[2.1 Выбор торговых платформ и API 14](#_Toc167279972)

[2.2 Выбор языка программирования и инструментов разработки 16](#_Toc167279973)

[2.3 Проектирование архитектуры программного обеспечения 18](#_Toc167279974)

[3 Реализация и тестирование 21](#_Toc167279975)

[3.1 Описание реализации основных компонентов 21](#_Toc167279978)

[3.2 Результаты и тестирование функциональности 27](#_Toc167279982)

[ЗАКЛЮЧЕНИЕ 31](#_Toc167279983)

[СПИСОК ЛИТЕРАТУРЫ 33](#_Toc167279984)

[ПРИЛОЖЕНИЕ 34](#_Toc167279985)

# ВВЕДЕНИЕ

В современном мире финансовых рынков появление автоматизированной торговли произвело революцию в этой системе. Появление сложных алгоритмов и торговых ботов позволило данной деятельности выйти за рамки классических методов, проложив путь к веку, в котором ключевые роли занимают скорость, эффективность и точность.

Мощное влияние автоматизированной торговли подчеркивается ее значительным вкладом в общий биржевой оборот, когда крупные инвестиционные компании совершают тысячи сделок в секунду, чтобы извлечь выгоду даже из малейшего движения рынка. Эта погоня привела к осознанию того, что даже миллисекунды могут стать решающим фактором в огромных прибылях и убытках. В результате чего традиционная роль трейдеров-людей была дополнена, если не вытеснена, алгоритмическими системами, которые совершают сделки с небывалой скоростью и объемом. Из всего вышеописанного, несомненно, можно сделать вывод об актуальности данной темы.

Несмотря на достижения алгоритмической торговли, многие трейдеры все еще сталкиваются с проблемой отсутствия доступных и простых в использовании торговых ботов. Большинство существующих вариантов оказываются слишком сложными и не всегда эффективными.

Требования к торговым ботам постоянно растут, и трейдеры ищут решения, которые бы сочетали в себе простоту использования, эффективность и надежность. Именно поэтому темой данной работы стала создание бота биржевой торговли.

Объектом данного исследования является процесс биржевой торговли.

Предметом исследования является автоматизация процесса анализа и биржевой торговли.

Цель данной курсовой работы заключается в приобретении необходимых навыков и знаний для создания программного продукта, направленного на автоматизированную торговлю с использованием эффективных методов кодирования. Для разработки и тестирования бота были выбраны криптовалютные биржи. Этот выбор обусловлен несколькими причинами. Во-первых, криптовалюты представляют собой динамичный и востребованный рынок. Во-вторых, использование криптовалюты обеспечивает широкий спектр торговых возможностей и вариативность стратегий, которые можно проверить и оптимизировать. Наконец, криптовалютные рынки работают круглосуточно, что позволяет боту работать без перерывов, что важно для тестирования его стабильности и эффективности в различных условиях рынка.

Для достижения этой цели были сформулированы следующие задачи**:**

* разработать и автоматизировать торговые стратегии, в том числе создание алгоритмов, осуществляющих сделки в соответствии с заданными параметрами и рыночными условиями;
* реализовать пользовательский интерфейс. Интерфейс будет служить торговым помощником, позволяя следить за сделками и рыночными данными в режиме реального времени;
* обеспечить прямое взаимодействие с API бирж для эффективного управления торговыми стратегиями;
* провести тестирование и оптимизацию разработанного программного продукта.

Обеспечить расширяемость и масштабируемость программного решения для работы с различными типами торговых счетов и криптовалютных бирж.

# ****1 Теоретический обзор и анализ предметной области****

# 1.1 Определение алгоритмической торговли и основные принципы

Алгоритмический трейдинг (алготрейдинг) — это торговля на бирже с использованием специального программного обеспечения (торгового бота), которая следует заранее определенному набору инструкций для облегчения выполнения сделки [1]. Бот — это компьютерная программа, выполняющая заданные действия без участия человека.

Определенный набор стратегий основывается на ряде факторов, включая выбранный инструмент, количество, таймфрейм, показатели индикаторов и т.д. Помимо потенциальной прибыли для трейдера, алгоритмическая торговля делает торговлю - более систематичной. Это достигается за счет исключения влияния человеческих эмоций на торговую деятельность.

Основной принцип работы торгового бота заключается в использовании заранее описанных стратегий, исходя из которых будет автоматически следить за ценой актива и выставлять ордера на покупку и продажу при выполнении заданных условий. Трейдеру больше нет нужды вручную следить за ценами и графиками или вводить ордера. Алгоритмическая торговая система выполняет эту функцию автоматически, правильно определяя торговые возможности.

# Виды торговых ботов

Первоначальным критерием классификации ботов является уровень их автоматизации. Он определяется количеством функций и задач, которые способен решать бот. Можно выделить следующие типы ботов (рисунок 1):

* бот, управляемый вручную. Ручное управление торговым ботом позволяет трейдеру самостоятельно оценить рынок и запустить соответствующую стратегию. Затем активируется робот, который торгует в соответствии с выбранной инструкцией;
* автоматические боты состоят из нескольких алгоритмов, разработанных с учетом специфики рынка. Алгоритмические торговые стратегии закладываются в код на этапе программирования. Впоследствии робот сам выбирает набор правил, основанных на рыночных индикаторах или паттернах, и активирует необходимую торговую модель;
* роботы с искусственным интеллектом, также известные как "генетические боты", способны не только подбирать подходящие алгоритмы, но и создавать стратегии на основе объемных исторических данных. Этот класс ботов представляет собой вершину технологического прогресса. Однако он также является самым сложным в создании и реализации. Большинство таких роботов работают в крупных инвестиционных фондах, а программный код считается коммерческой тайной.

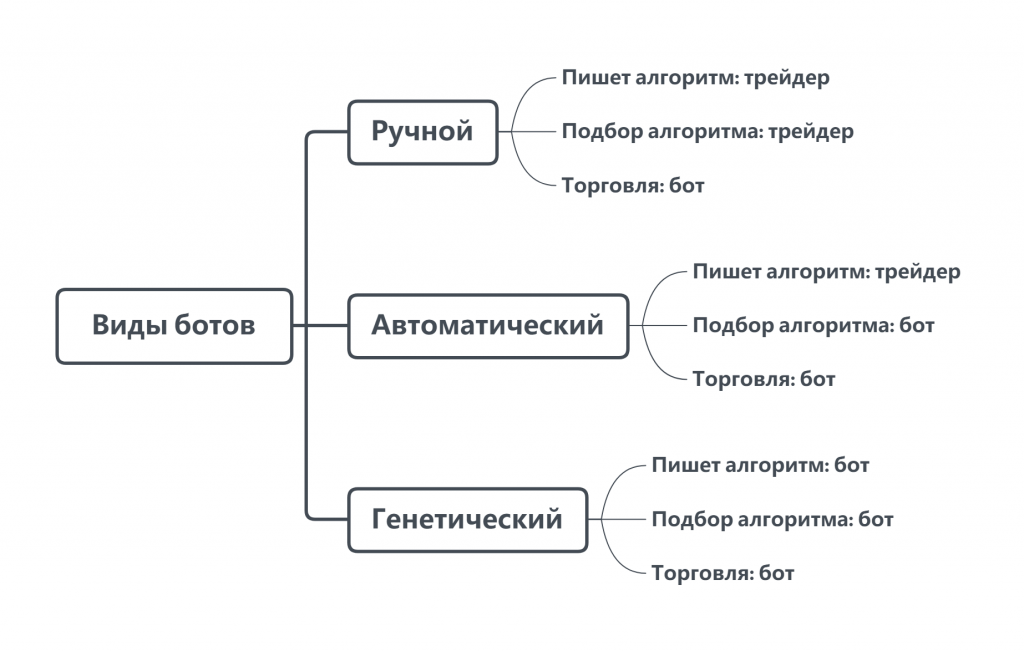


Рисунок 1 – Виды торговых ботов для алготрейдинга

Боты также можно классифицировать в зависимости от задач, для которых они были созданы:

* боты для алгоритмической торговли на финансовых рынках в рамках заданной стратегии. Это, пожалуй, самая важная группа роботов, которые ни используют технический анализ, индикаторы, паттерны и другие "триггеры" для открытия коротких или длинных позиций в соответствии заданной стратегией;
* боты для арбитража. Эти программы сравнивают цены активов на разных биржах или стоимость связанных инструментов на одной платформе. Они извлекают потенциальную прибыль из неэффективности рынка и расхождения цен;
* боты для страхования (хеджирования). Эти роботы открывают вторую позицию в соответствии со стратегией хеджирования. Они могут работать независимо или параллельно с трейдером, открывая как основную, так и страховую позицию;
* маркет-мейкеры. Используется фирмами, которые выступают в качестве маркет-мейкеров на бирже. Они поддерживают ликвидность и регулируют торговлю, размещая ордера на определенных ценовых уровнях;
* узконаправленные роботы, которые решают торговые задачи алгоритмически. Таким образом, на выходе программы получается список results, содержащий по словарю на каждый правильно декодированный штрихкод, из которого можно узнать текст штрихкода и координаты ограничивающего прямоугольника.

На рисунке 2 представлены целевые задачи алготрейдинга, исходя из которых подбирается и пишется конкретный бот.



Рисунок 2 – Целевые задачи алготрейдинга

Наконец, все алгоритмические торговые боты на бирже можно классифицировать по сложности программного кода (рисунок 3):

* Простые. Эти боты используют упрощенные стратегии, учитывают 1-3 индикатора технического анализа и имеют минимальный исходный код. Они предназначены для новичков в алгоритмической торговле;
* Продвинутые. У таких ботов нет готовой стратегии или она прописана неполностью. Трейдеру приходится самостоятельно настраивать параметры работы торгового бота;
* Скриптовые. Роботы, написанные на языке программирования. Функции написаны в виде скриптов - фрагментов кода, запускающих то или иное действие. Соответственно, трейдер должен хорошо разбираться в программировании.



Рисунок 3 – Виды роботов по настройке алготрейдинга

# Преимущества и недостатки алгоритмической торговли

Основным преимуществом алготрейдинга является автоматизация процесса торговли. Боты действуют в соответствии с заранее разработанной стратегией, исключая возможность эмоциональной предвзятости и человеческих ошибок. Они способны рассчитывать объем позиции и соблюдать риск-менеджмент, если это заложено в коде (рисунок 4) [2].

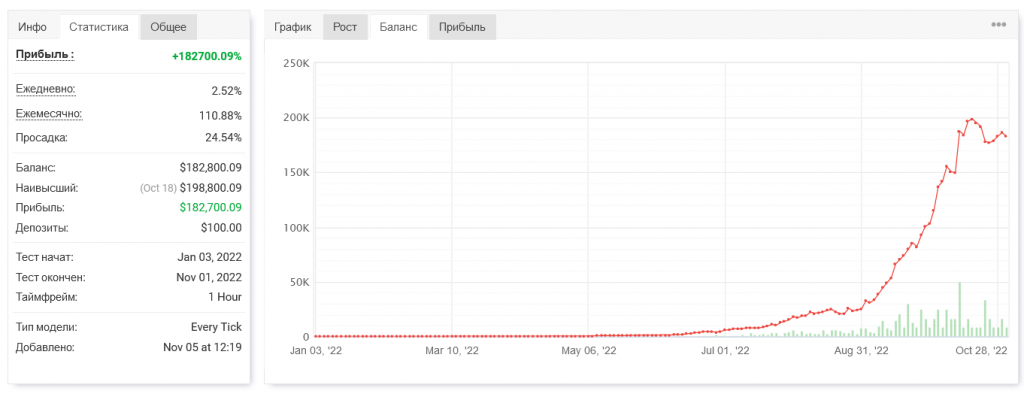


Рисунок 4 – Результаты работы торгового бота

Алгоритмы способны исполнять ордера быстрее, чем человек, что может стать значительным конкурентным преимуществом в определенных сценариях. На криптовалютном рынке можно торговать круглосуточно, что дает трейдеру вести торговлю, даже если его компьютер выключен.

Основной недостаток - всевозможные сбои, по типу технических ошибок или некорректно запрограммированной стратегии.

Также в рамках алготрейдинга невозможно заниматься микротрейдингом.

# Алгоритмические торговые стратегии

Алгоритмические торговые стратегии включают в себя целый ряд методов, направленных на оптимизацию торговых результатов. К ним относятся стратегии следования за трендом, которые позволяют инвесторам извлекать выгоду из динамики рынка без необходимости прогнозирования; арбитражные возможности, которые позволяют инвесторам получать прибыль, используя неэффективность рынка; и ребалансировка индексных фондов, которая позволяет использовать предсказуемые периоды корректировки для получения прибыли.

Динамичный и конкурентный характер алгоритмической торговли требует постоянного совершенствования и тестирования стратегий для адаптации к меняющимся рыночным условиям и снижения рисков, связанных с системными сбоями, ошибками подключения и несовершенством алгоритмов. Преобразование сути стратегий алгоритмической торговли в формат структурированной таблицы позволяет комплексно сравнивать различные стратегии, проливая свет на их цели, механизмы и типичные применения, не прибегая к маркированным спискам. Такой подход способствует лаконичному и в то же время всестороннему обзору (таблица 1) [2].

Таблица 1 – Стратегии алготрейдинга

|  |  |  |  |
| --- | --- | --- | --- |
| **Тип стратегии** | **Цель** | **Механизм** | **Типичное применение** |
| Следование тренду | Извлекает выгоду из динамики изменения рынка | Использует скользящие средние и прорывы каналов для выявления рыночных тенденций и отслеживания их | Трейдеры, желающие получить не поддающиеся прогнозу стратегии |
| Арбитражная | Использует ценовые расхождения для получения безрисковой прибыли | Определяет разницу в ценах между акциями, зарегистрированными на двух биржах, или на разных рынках и воздействует на нее | Для трейдеров, которые хотят совершать быстрые сделки |
| Ребалансировка индексных фондов | Целью данной стратегии является получение прибыли от предсказуемых сделок | Она предвидит действия индексных фондов по ребалансировке и торгует ими, чтобы выровнять их с эталонными индексами | Для тех, кто стремится извлечь выгоду из периодов ребалансировки индексных фондов |

Математические модели, подтвержденные эмпирически, такие как дельта-нейтральная торговая стратегия, позволяют торговать комбинацией опционов и базовой ценной бумаги (таблица 2) [2]. (Дельта-нейтральная стратегия — это портфельная стратегия, состоящая из нескольких позиций с компенсирующими положительными и отрицательными дельтами. Дельта — это коэффициент, который сравнивает изменение цены актива, обычно рыночной ценной бумаги, с соответствующим изменением цены его производного инструмента. Таким образом, общая дельта рассматриваемых активов должна быть равна нулю.)

Таблица 2 – Стратегии, основанные на математических моделях

|  |  |  |
| --- | --- | --- |
| **Стратегия** | **Описание** | **Преимущества** |
| Стратегии на математических моделях | В этих стратегиях используются проверенные временем модели, такие как дельта-нейтральная торговля опционами и ценными бумагами. | Позволяет торговать на основе математической анализа и обеспечивает защиту от резких изменений цены актива на рынке. |
| Алгоритм доставки межбанковских цен (IPDA) | Использует исторические данные для анализа ликвидности и рыночного равновесия. | Он предлагает понимание оптимальных точек входа и динамики рынка. |
| Дельта-нейтральная торговая стратегия | Эта стратегия направлена на достижение нулевой общей дельты путем балансирования опционов и ценных бумаг. | Стратегия снижает риск, поддерживая нейтральность направления рынка. |

Продолжение таблицы 2

|  |  |  |
| --- | --- | --- |
| **Стратегия** | **Описание** | **Преимущества** |
| Стратегия средней реверсии | Использует возвращение цен на активы к их среднему значению или средней величине. | Она использует временные ценовые экстремумы для получения прибыли. |
| Средняя цена, взвешенная по объему (VWAP) | Этот метод исполнения ордеров основан на профилях исторического объема с целью оптимизации результата. | Эта стратегия минимизирует влияние рыночных сил и приводит исполнение ордеров в соответствие с преобладающей средней ценой. |
| Средневзвешенная по времени цена (TWAP) | Этот метод распределения позволяет снизить влияние рынка за счет распределения крупных ордеров во времени. | Он способствует плавному выходу на рынок сделок с большим объемом. |
| Процент от объема (POV) и недостаток реализации | Данная методология корректирует ордера в зависимости от объема в реальном времени, тем самым минимизируя затраты на исполнение. | Это оптимизирует участие в рынке и снижает стоимость исполнения. |

Продолжение таблицы 2

|  |  |  |
| --- | --- | --- |
| **Стратегия** | **Описание** | **Преимущества** |
| Алгоритмы поиска | Выявляет возможности для выставления крупных ордеров маркет-мейкерами | Система позволяет выгодно заполнять ордера, обнаружив заметные движения рынка. |

# 2 Архитектура и проектирование торгового бота

# Выбор торговых платформ и API

Торговая платформа — это программная система, инструмент, позволяющий инвесторам и трейдерам заключать сделки и контролировать свои счета в режиме реального времени.

Торговые платформы часто предлагают трейдеру свои дополнительные возможности для введения торговли. Это могут быть котировки в реальном времени и удобные для работы графики, новостные ленты, мгновенный доступ к огромному количеству потоковых и исторических финансовых данных, инструменты технического анализа и другие.

Существует два типа торговых платформ: коммерческие и проприетарные.

Коммерческие платформы предназначены для различных инвесторов. Они предоставляют удобные функции, такие как котировки в режиме реального времени, международные новости, графики и инструменты для исследований. Большинство популярных бирж являются коммерческими платформами.

Проприетарные платформы, напротив, настраиваются и разрабатываются крупными брокерскими компаниями и финансовыми учреждениями для собственного использования. Они предоставляют прямой доступ к рынкам, что дает учреждениям конкурентное преимущество. Однако, такие платформы недоступны для широкой публики [3].

При выборе торговой платформы для торговли важно учитывать как базовые функции, так и условия торговли, включая комиссии. Каждый трейдер имеет свои потребности, когда дело доходит до принятия торговых решений. В данной же работе для работы с криптовалютой было решено использовать такие торговые платформы, как Binance и Bitmex.

Преимущества биржи Binance:

* низкая комиссия за торговлю;
* использование технология исполнения ордеров, которая способна обработать 1,4 миллиона ордеров в секунду;
* реферальная система;
* комфортный интерфейс, переведенный на множество языков;
* наличие десктопного приложения для Windows и Mac;
* наличие мобильного приложения для Android и IOS;
* огромный выбор монет, которые торгуются в паре с BTC, ETH, BNB и USDT.

Биржа Binance не только предоставляет платформу для торговли криптовалютами, но и имеет собственный токен, известный как Binance Coin (BNB), изображенный на рисунке 5.



Рисунок 5 – Токен BNB

Этот токен был выпущен в 2017 году на базе блокчейна Binance Chain и начал использоваться в качестве средства оплаты комиссий на платформе Binance.

Основные преимущества биржи BitMEX:

* круглосуточная техническая поддержка;
* маржинальная торговля с высоким плечом;
* игра на понижении курса;
* низкие комиссии;
* высокий уровень безопасности.

Криптовалютная биржа BitMEX тоже имеет свой токен — BMEX [4].

API (Application Programming Interface) — это интерфейс, который в контексте торговых платформ и ботов играет важную роль, предоставляя возможность их связывать для получения актуальной информации о рынке и управления счетом. Торговые платформы API обеспечивают лучшие соединения, получая немедленный доступ к текущим рыночным данным, а также историческим, и генерировать правильные отчеты о ордерах и сделках [5].

В контексте Binance и BitMEX, обе платформы предоставляют API для своих пользователей. Трейдеры могут использовать API для создания собственных торговых роботов или для автоматизации определенных аспектов своей стратегии торговли.

# Выбор языка программирования и инструментов разработки

Одним из ключевых решений в процессе разработки программного обеспечения является выбор наиболее подходящего языка программирования для достижения желаемого результата. Это играет решающую роль в успешной разработке программного обеспечения. Различные языки программирования имеют свои уникальные особенности, сильные и слабые стороны, и области применения. При выборе языка программирования нужно учитывать множество факторов.

В данной работе в качестве языка программирования для разработки собственного бота для биржевой торговли был выбран Python. Python - отличный выбор для многих проектов, в том числе требующих надежной программной архитектуры, благодаря своей простоте, читабельности и универсальности.

Популярность Python объясняется его обширной стандартной библиотекой, которая предоставляет готовые модули для различных функций, что снижает потребность во внешних зависимостях. Кроме того, однозначный синтаксис и динамическая типизация способствуют быстрой разработке и сопровождению программного обеспечения. В контексте нашего торгового приложения обширная экосистема Python предлагает множество библиотек, включая Pandas для работы с данными, NumPy для численных вычислений, Requests и WebSocket для запросов и множество других модулей [6]. Вкратце рассмотрим основные модули:

Pandas — это библиотека Python для обработки и анализа данных. Она способствуют удобной и эффективной обработке и анализу табличных данных. Pandas позволяет загружать данные из различных источников, манипулировать с данными (такие как фильтрация, сортировка и группировка), а также проводить анализ и визуализацию данных.

NumPy представляет собой фундаментальную библиотеку для вычислений на Python. Она предоставляет многомерные массивы данных, при работе с которыми позволяет вычислять много высокоуровневых математических функций.

Библиотеки Requests и WebSocket для Python — это эффективные инструменты для работы с HTTP-запросами. Библиотеки предоставляют удобный интерфейс для взаимодействия с веб-серверами и получения данных из Интернета.

Сочетание этих и других не менее важных библиотек, которые будут использованы в нашей разработке, позволяет создавать сложные и эффективные программы для анализа данных, численных вычислений и веб-запросов. Это делает Python привлекательным выбором для широкого спектра задач, включая биржевую торговлю.

# Проектирование архитектуры программного обеспечения

Прежде чем приступать к разработке сложного приложения, необходимо спроектировать архитектуру программного обеспечения. Хорошо спроектированная архитектура обеспечивает возможность масштабирования системы, ее адаптацию к изменяющимся требованиям, поддержку и повторное использование.

Некоторые фундаментальные принципы, которые обычно используются при проектировании архитектуры программного обеспечения, следующие:

* разделение компонентов является основополагающим принципом архитектуры программного обеспечения. Программное обеспечение делится на отдельные компоненты или модули, каждый из которых выполняет определенную задачу;
* ответственность каждого компонента. Заключается в том, что каждый модуль должен отвечать за определенную функцию или задачу. Это способствует четкому разграничению ответственности и минимизации взаимозависимости между компонентами;
* модульность и возможность повторного использования;
* гибкость и расширяемость являются ключевыми факторами при проектировании любой системы;
* принцип разделения интерфейса и реализации.

В случае с ботом для биржевой торговли структура компонентов следующая.

Папка connectors: содержит модули, отвечающие за подключение к различным биржам. Здесь у нас имеются такие файлы, как bitmex.py и binance.py, каждый из которых отвечает за взаимодействие с API соответствующих бирж.

В папке interface содержится:

* autocomplete\_widget.py: обеспечивает функциональность автозаполнения пользовательских данных;
* logging\_component.py: ведение журнала для отладки и мониторинга;
* root\_component.py: определяет корневой компонент интерфейса;
* scrollable\_frame.py: реализует прокручиваемый фрейм для отображения контента;
* strategy\_component.py: управление стратегиями торговли;
* styling.py: конфигурации стилей интерфейса;
* trades\_component.py: отображение информации о сделках;
* watchlist\_component.py: управление списком активов пользователя.

Оставшиеся модули:

* database.py: взаимодействие с базой данных, облегчая хранение и поиск данных;
* info.log: хранит информацию о работе приложения для целей отладки и мониторинга;
* main.py: основная точка входа, инициализация и выполнение компонентов;
* models.py: определение моделей для обеспечения согласованности в работе с данными;
* strategies.py: реализация торговых стратегий приложения;
* utils.py: содержит вспомогательные функции.

Переход к диаграмме компонентов поможет нам визуализировать архитектуру системы и лучше понять взаимодействие между ее составными частями.

На рисунке 6 изображена данная диаграмма, которая описывает следующее:

* бот: этот компонент представляет основной функционал торгового бота. Он отвечает за обработку запросов на торговлю и принятие решений на основе полученных данных о рынке. Взаимодействует с другими компонентами для получения информации и сохранения результатов сделок;
* биржа: торговая платформа, на которой работает торговый бот. Он предоставляет информацию о текущем состоянии рынка и обрабатывает запросы на выполнение сделок от бота. Также взаимодействует с базой данных для хранения информации о сделках и с внешним API для получения актуальных данных о рынке;
* база данных: отвечает за хранение данных о сделках и состоянии биржи. Используется для сохранения результатов торговли, а также для хранения информации о пользователях и их предпочтениях. Взаимодействует как с ботом, так и с биржей для обмена данными;
* внешний API: Этот компонент предоставляет доступ к внешнему API для получения информации о рыночных условиях и других внешних данных, которые могут использоваться торговым ботом.

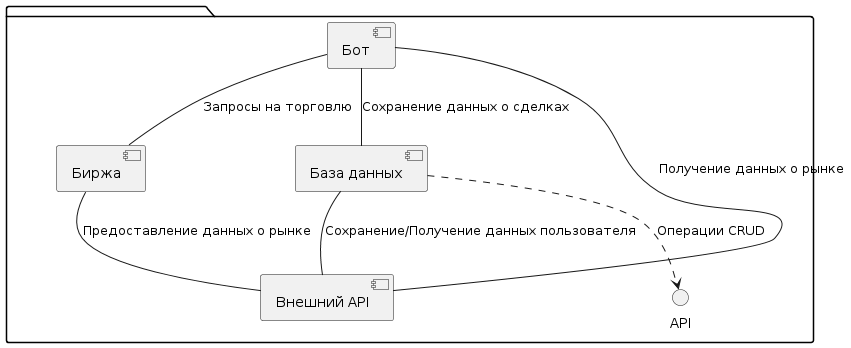


Рисунок 6 – Диаграмма компонентов

Таким образом, представленная диаграмма показывает, как каждый компонент системы взаимодействует друг с другом для обеспечения функциональности торгового бота.

# Реализация и тестирование



# Описание реализации основных компонентов

Первым делом описание реализации компонентов системы нужно начать с коннекторов, которые играют ключевую роль в установлении связи с внешними сервисами.

В данной работе присутствуют два основных коннектора: binance и bitmex, в каждом из котором реализован класс для работы с соответствующей торговой платформой (рисунки 7, 8).

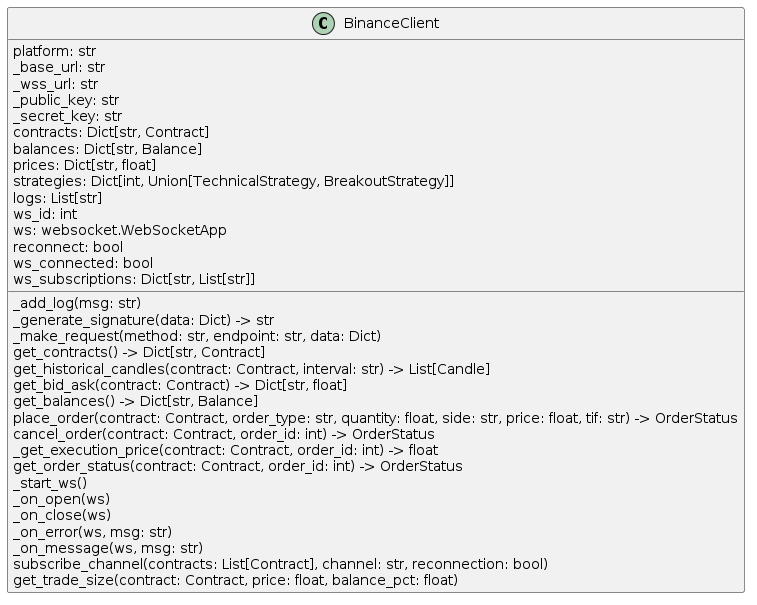


Рисунок 7 – Класс BinanceClient

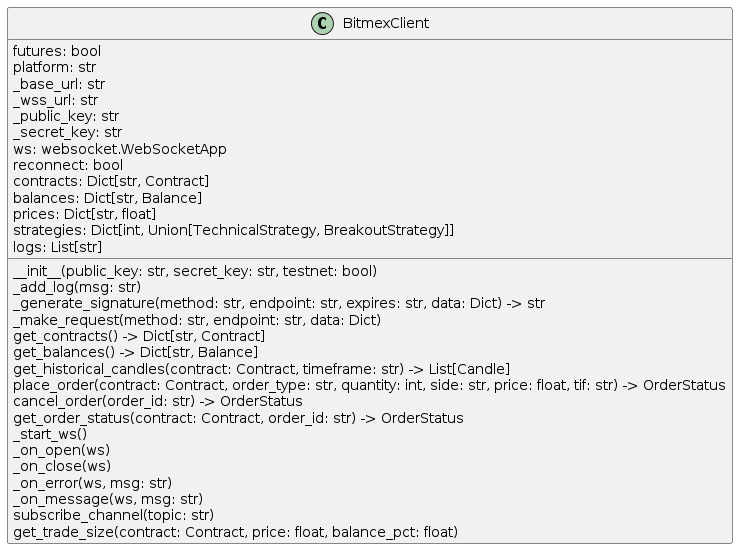


Рисунок 8 – Класс BitmexClient

Оба класса предоставляют доступ к различным функциям бирж, таким как получение информации о контрактах, балансах, ценах, исторических свечах, размещение и отмена ордеров, а также работу с веб-сокетами для получения данных.

Каждый из коннекторов использует основной принцип связывания через API биржи. Бот аутентифицируется на бирже с помощью публичного и секретного API ключей, передаваемых в заголовке (X-MBX-APIKEY) при каждом запросе. Для выполнения различных операций он отправляет HTTP-запросы к API биржи с использованием методов GET, POST и DELETE. Ответы на запросы анализируются для получения необходимых данных.

Также программа поддерживает постоянное WebSocket соединение с сервером биржи, подписываясь на определенные каналы данных (например, bookTicker и aggTrade) для конкретных торговых пар (symbols). При возникновении событий, связанных с этими каналами (например, изменение цены или новая сделка), сервер биржи отправляет данные через WebSocket соединение, и бот их анализирует.

Следующим важны компонентом является модуль strategies, в котором реализован класс Strategy, от которого наследуются классы BreakoutStrategy и TechnicalStrategy (рисунок 9).

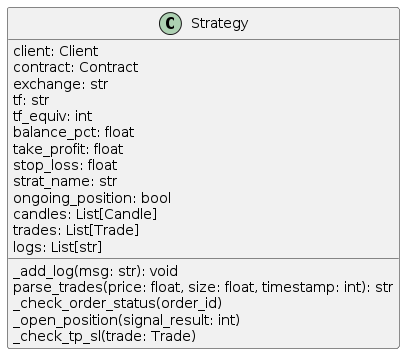


Рисунок 9 – Класс Strategy

Strategy — это базовый класс стратегии, который содержит общие методы и атрибуты для всех стратегий. Он определяет логику открытия и закрытия позиций, проверку статуса ордеров и другие общие операции.

На основе базового класса реализован класс — TechnicalStrategy. Этот класс представляет техническую стратегию, основанную на индикаторах, таких как RSI (Индекс относительной силы) и MACD (Схождение/Расхождение скользящих средних). Он использует данные о свечах (Candle) для расчета индикаторов и определения торговых сигналов. Метод \_check\_signal() вычисляет сигнал на основе текущих значений индикаторов, а метод check\_trade() открывает позицию, если сигнал соответствует условиям стратегии. Ниже представлена реализация методов вычисления индикаторов:

def \_rsi(self) -> float:  
 close\_list = []  
 for candle in self.candles:  
 close\_list.append(candle.close)  
 closes = pd.Series(close\_list)  
 delta = closes.diff().dropna()  
 up, down = delta.copy(), delta.copy()  
 up[up < 0] = 0  
 down[down > 0] = 0  
 avg\_gain = up.ewm(com=(self.\_rsi\_length - 1), min\_periods=self.\_rsi\_length).mean()  
 avg\_loss = down.abs().ewm(com=(self.\_rsi\_length - 1), min\_periods=self.\_rsi\_length).mean()  
 rs = avg\_gain / avg\_loss  
 rsi = 100 - 100 / (1 + rs)  
 rsi = rsi.round(2)  
 return rsi.iloc[-2]  
  
def \_macd(self) -> Tuple[float, float]:  
 close\_list = []  
 for candle in self.candles:  
 close\_list.append(candle.close)   
 closes = pd.Series(close\_list)   
 ema\_fast = closes.ewm(span=self.\_ema\_fast).mean()  
 ema\_slow = closes.ewm(span=self.\_ema\_slow).mean()  
 macd\_line = ema\_fast - ema\_slow  
 macd\_signal = macd\_line.ewm(span=self.\_ema\_signal).mean()  
 return macd\_line.iloc[-2], macd\_signal.iloc[-2]

BreakoutStrategy — класс, представляющий собой стратегию прорыва, основанную на выходе цены за пределы предыдущих критических уровней, таких как максимумы и минимумы. Он также использует данные о свечах для определения торговых сигналов. Метод \_check\_signal() проверяет, произошел ли прорыв, а метод check\_trade() открывает позицию в случае обнаружения сигнала.

def \_check\_signal(self) -> int:  
 if self.candles[-1].close > self.candles[-2].high and self.candles[-1].volume > self.\_min\_volume:  
 return 1  
 elif self.candles[-1].close < self.candles[-2].low and self.candles[-1].volume > self.\_min\_volume:  
 return -1  
 else:  
 return 0  
  
def check\_trade(self, tick\_type: str):  
  
 if not self.ongoing\_position:  
 signal\_result = self.\_check\_signal()  
  
 if signal\_result in [1, -1]:  
 self.\_open\_position(signal\_result)

Оба класса наследуют общую функциональность от класса Strategy, что обеспечивает единый интерфейс и структуру для различных стратегий. Это позволяет легко добавлять новые стратегии или расширять существующие.

Компонент models представляет собой определение классов для обеспечения согласованности в работе с данными:

* класс Balance содержит данные о балансе трейдера на бирже. Он используется для контроля и управления балансом пользователя, включая информацию о начальной марже, поддерживаемой марже, балансе маржи, балансе кошелька и нереализованной прибыли;
* класс Candle содержит информацию о свечном графике, включая данные об открытии, закрытии, минимуме, максимуме и объеме за заданный временной интервал. Он используется для хранения и обработки свечных данных, включая временную метку, цену открытия, максимальную и минимальную цены, закрывающую цену и объем;
* класс Contract представляет контракт на бирже, который может быть фьючерсным контрактом, контрактом на разницу цен или другим типом контракта. Он используется для хранения информации о контракте, такой как символ, базовый актив, котируемый актив, точность цены, количество, размер шага и т.д.;
* класс OrderStatus отображает состояние заказа на бирже. Он используется для отслеживания статуса выполнения заказа, средней цены выполнения и количества выполненных заказов;
* класс Trade содержит информацию о сделке, совершенной на бирже. Он используется для отслеживания информации о времени сделки, контракте, стратегии, направлении (покупка/продажа), цене входа, статусе, прибыли и т.д.

Компонент database содержит в себе класс WorkspaceData. Данный класс предоставляет удобный способ работы с базой данных SQLite. Он позволяет сохранять и извлекать данные из базы данных, что особенно полезно для сохранения и загрузки настроек, стратегий торговли и другой информации.

И наконец, заключительное, это компоненты, связанные с интерфейсом. Среди этих компонентов — autocomplete\_widget, который облегчает пользователю ввод данных; logging\_component, который позволяет отслеживать события и ошибки приложения в режиме реального времени; root\_component, который является базовым элементом пользовательского интерфейса, объединяющим другие компоненты и управляющим их размещением; scrollable\_frame, обеспечивающий возможность прокрутки содержимого для удобного просмотра больших объемов данных; strategy\_component, предназначенный для отображения и управления информацией о торговых стратегиях; styling, определяющий внешний вид и стиль интерфейса; trades\_component, отображающий информацию о сделках пользователя; и watchlist\_component, предназначенный для отслеживания списка инструментов или активов на рынке. Вместе эти компоненты создают полнофункциональное приложение для анализа и управления финансовыми данными и операциями на бирже.



# Результаты и тестирование функциональности

На рисунке 10 представлен конечный вид реализованного бота для биржевой торговли, являющийся результатом данной работы.

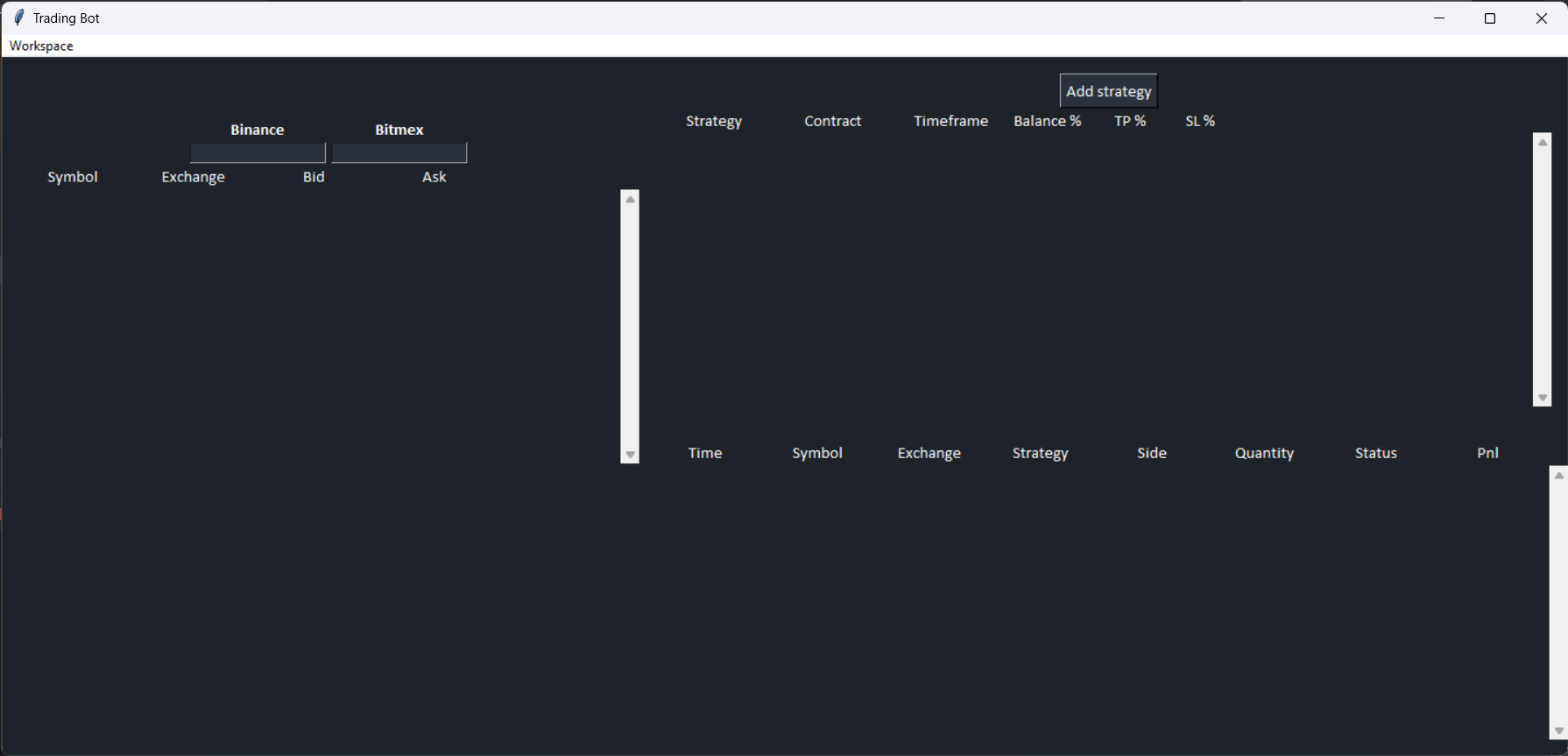


Рисунок 10 – Вид реализованного бота

В левой части окна созданной программы, на рисунке 11, представлен блок, в котором можно выводить любую доступную из выпадающего списка криптовалюту одной из указанных бирж с текущей ценой покупки и продажи. Цены представлены в режиме реального времени и постоянно обновляются.

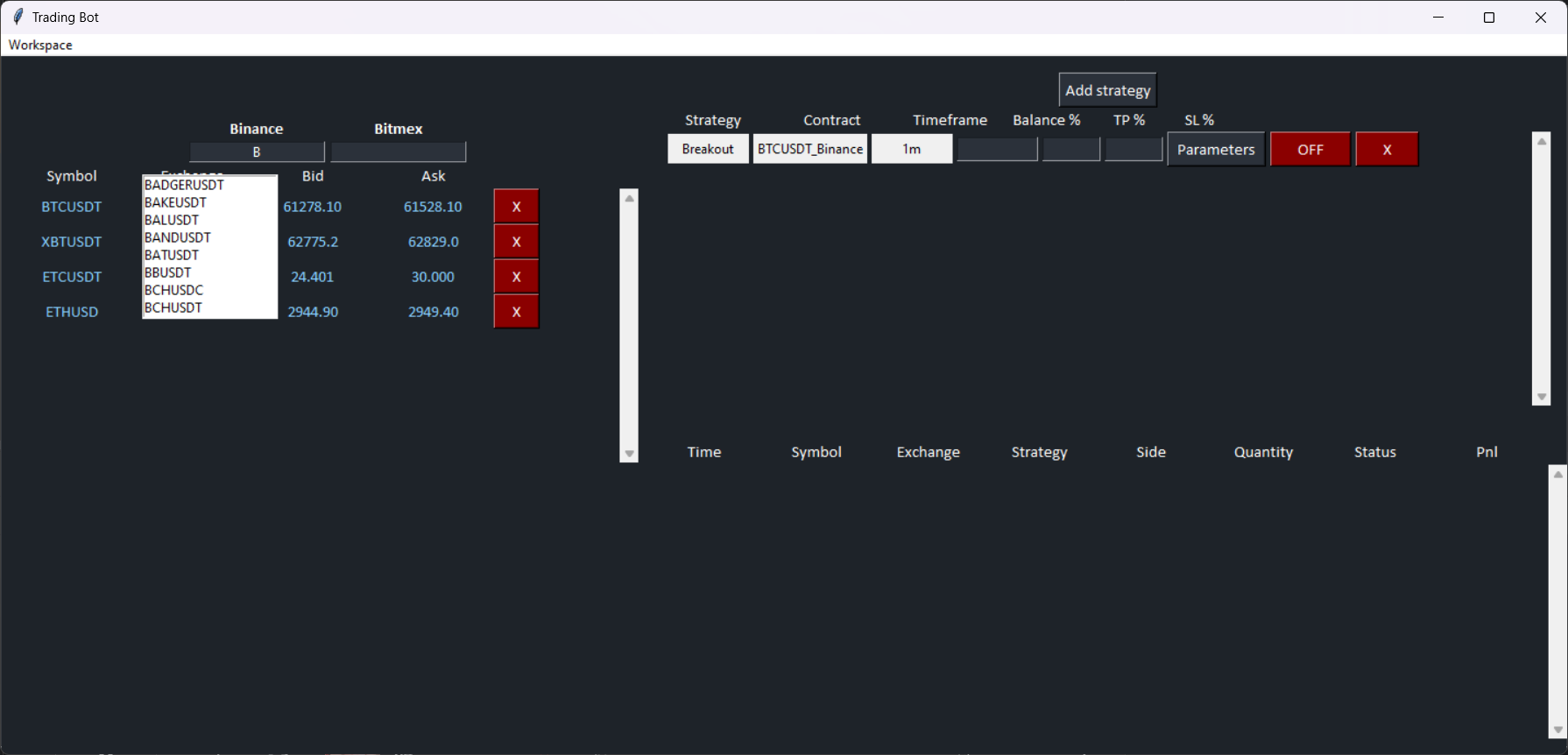


Рисунок 11 – Получение криптовалюты с ценой покупки и продажи

Чуть правее расположен блок добавления стратегий (рисунок 12).

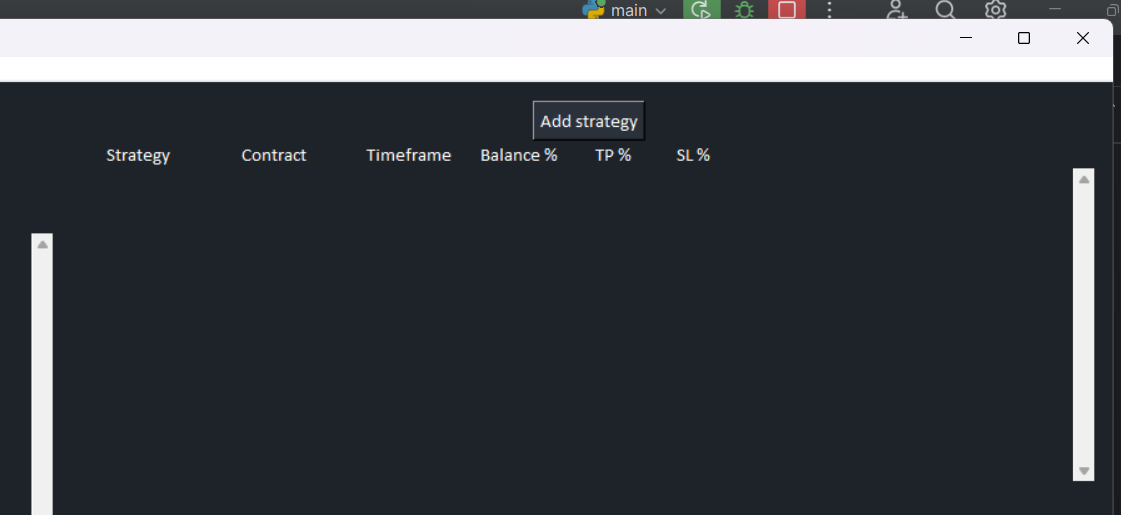


Рисунок 12 – Блок возможности добавления стратегий

Здесь трейдер имеет возможность добавить новую стратегию, настроить ее под себя, указав такие параметры, как тип стратегии, криптовалюту, таймфрейм, баланс в %, тейк-профит в % (ордер, позволяющий забрать прибыль), стоп-лосс в % (ордер, позволяющий указать уровень убытков, в случаях движения рынка не в пользу трейдера) и различные параметры в зависимости от выбранной стратегии. После можно запустить стратегию для начала автоматизированного процесса торговли (рисунки 13, 14).

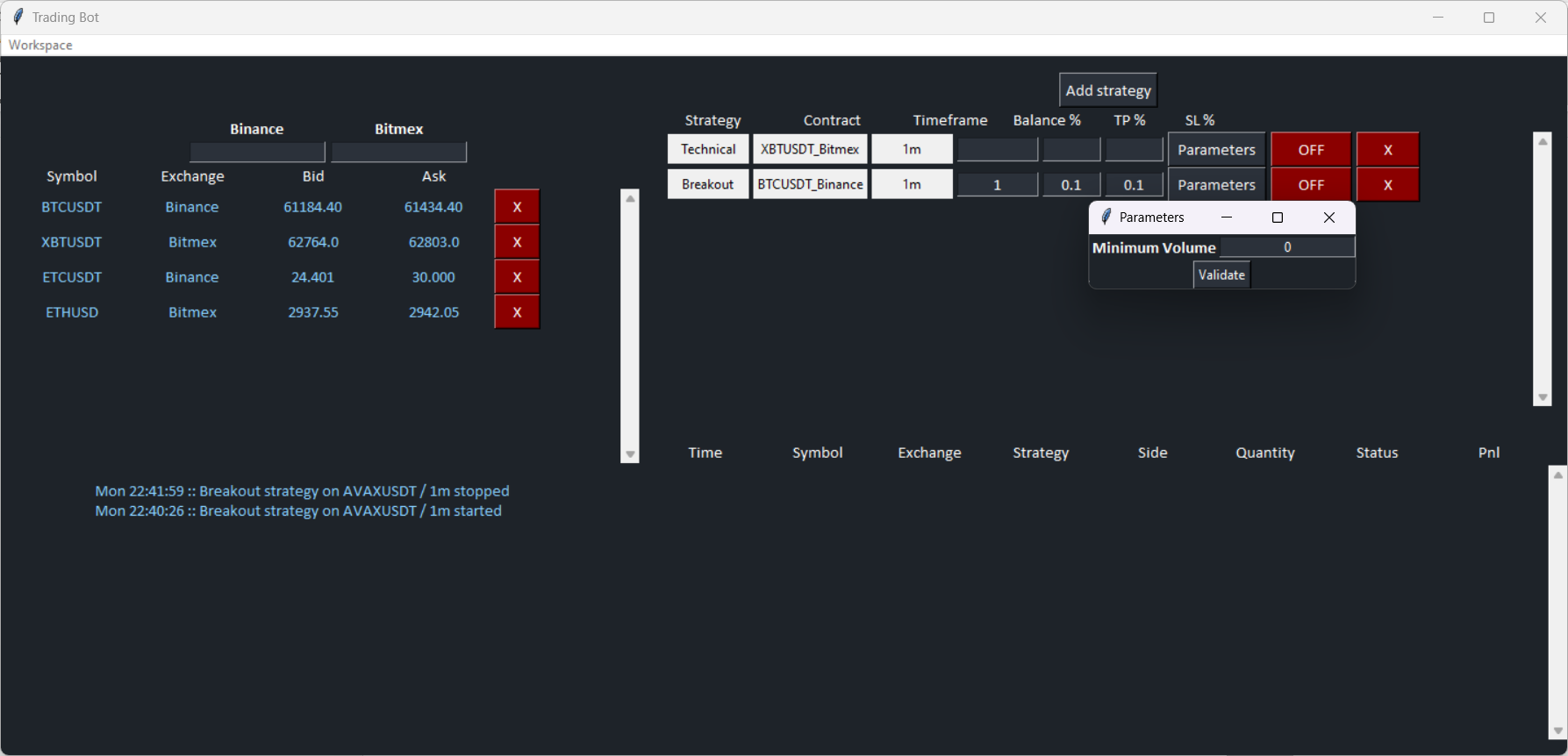


Рисунок 13 – Добавление стратегии Breakout

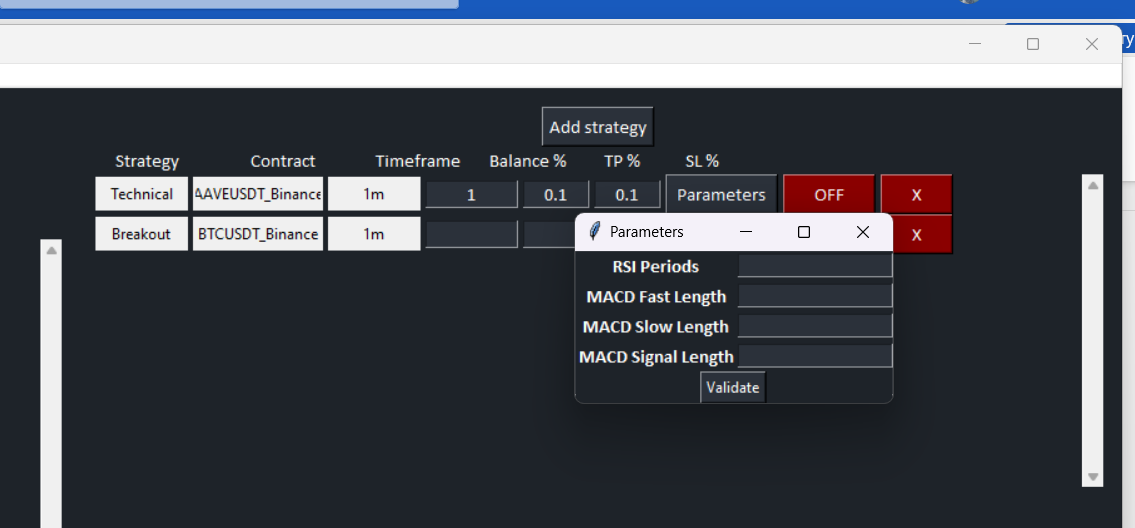


Рисунок 14 – Добавление стратегии Technical

На рисунке 15 представлен процесс работы на примере стратегии Breakout. Ниже блока добавления стратегий расположена часть, где введется запись об открытии или закрытии сделки с указанием времени, названии криптовалюты, биржи, стратегии, длины позиции, количества, статуса и Pnl (величина, указывающая разницу между прибылью и убытками).

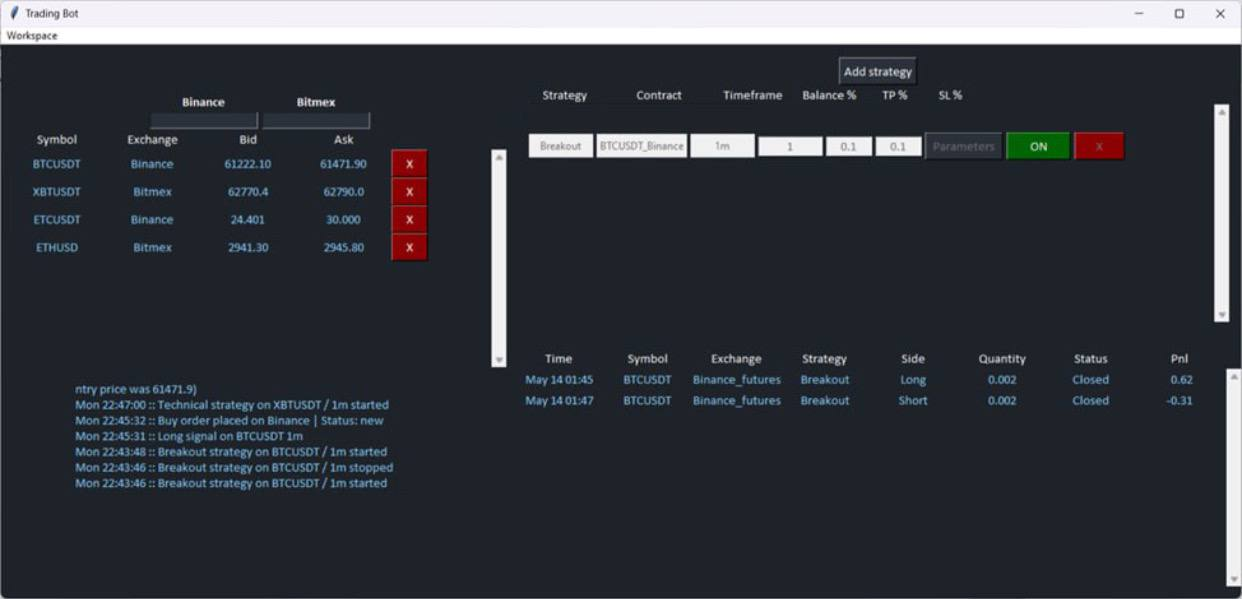


Рисунок 15 – Процесс работы бота

В нижнем левом углу находится блок оповещений с возможность прокрутки при работе стратегий (рисунок 16).

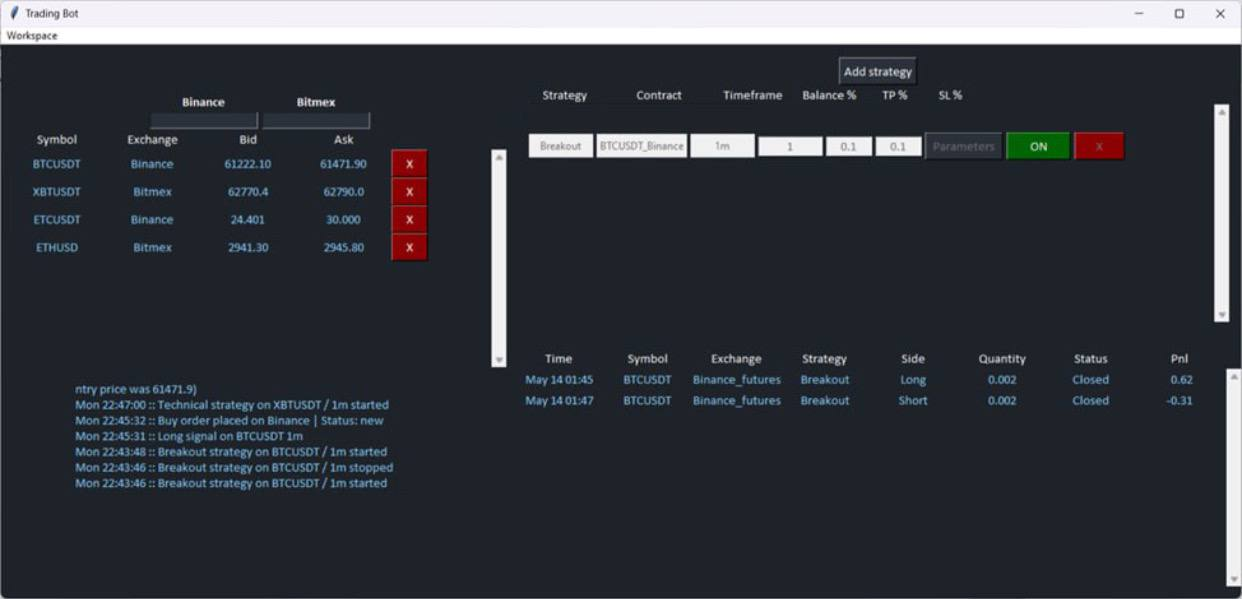


Рисунок 16 – Блок всевозможных оповещений

В левом верхнем углу окна программного расположена кнопка Workspace, представленная на рисунке 17, с помощью которой можно сохранить процесс торговли со всеми настройками и стратегиями торговли в базу данных.

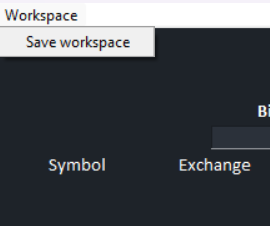


Рисунок 17 – Кнопка сохранения в базу данных

Эффективность бота для биржевой торговли напрямую зависит от тщательной настройки со стороны трейдера. Правильная настройка бота включает в себя определение четких правил входа, выхода и управления рисками, учитывая особенности рынка и торговые цели. Только при грамотной настройке бот может демонстрировать высокую эффективность, помогая трейдеру в достижении его финансовых целей.

# ЗАКЛЮЧЕНИЕ

Исходя из проделанной работы, полученных результатов и всего вышеизложенного, можно сделать следующие выводы:

1. разработаны и автоматизированы торговые стратегии, такие, как Breakout и Technical, способные совершать сделки в соответствии с заданными параметрами и условиями рынка;
2. реализован информативный и удобный пользовательский интерфейс, облегчающий пользователю взаимодействие с ботом;
3. торговый бот обеспечивает прямое взаимодействие с API бирж. В данной работе использовалось биржи: Binance и BiTMEX. Такой принцип позволяет пользователю эффективно управлять своими торговыми стратегиями. Это повышает скорость и надежность операций;
4. разработанный программный продукт прошел тщательное тестирование и оптимизацию для обеспечения его стабильности и эффективности в различных рыночных условиях;
5. программное решение было спроектировано с возможностью расширения и масштабируемости, что позволяет ему работать с различными торговыми платформами.

Таким образом, созданный программный продукт является высокоэффективным инструментом для автоматизированной торговли на криптовалютных рынках.

За время выполнения курсовой работы были реализованы следующие компетенции:

| Шифр компетенции | Расшифровка приобретаемой компетенции | Расшифровка освоения компетенции |
| --- | --- | --- |
| УК-6 | Способен управлять своим временем, выстраивать и реализовывать траекторию саморазвития на основе принципов образования в течение всей жизни | Студент по своей инициативе и желанию определился с темой курсовой работы, выполнил, и представил работу в срок |
| ПК-4 | Разработка требований и проектирование программного обеспечения | Программное обеспечение спроектировано в соответствии с требованиями, которые были определены совместно студентом и научным руководителем |
| ПК-5 | Оценка и выбор варианта архитектуры программного средства | Для выбора используемых алгоритмов для торговли были изучены и проанализированы его различные виды |
| ПК-6 | Разработка тестовых случаев, проведение тестирования и исследование результатов | Для проведения тестирования торгового бота были использованы исторические данные. Вместо основного счета, использовался демосчет. В курсовой работе сделаны выводы исходя из результатов |

# СПИСОК ЛИТЕРАТУРЫ

1. Basics of Algorithmic Trading: Concepts and Examples [Электронный ресурс]. ⎯ 2017. ⎯ URL: https://www.investopedia.com/articles/active-trading/101014/basics-algorithmic-trading-concepts-and-examples.asp (дата обращения 08.05.2024).
2. Basics Algorithmic Trading Strategies: Concepts and Examples [Электронный ресурс]. ⎯ 2015. ⎯ URL: https://tradingstrategycourse.com/basics-algorithmic-trading/ (дата обращения 08.05.2024).
3. What Is a Trading Platform? Definition, Examples, and Features [Электронный ресурс]. ⎯ 2019. ⎯ URL: https://www.investopedia.com/terms/t/trading-platform.asp (дата обращения 09.05.2024).
4. Binance vs Bitmex: что лучше выбрать [Электронный ресурс]. ⎯ 2020. ⎯ URL: https://profinvestment.com/binance-vs-bitmex/ (дата обращения 09.05.2024).
5. Best Trading API Brokers & Platforms in 2024 [Электронный ресурс]. ⎯ 2024. ⎯ URL: https://www.wikijob.co.uk/trading/forex/brokers-with-api-access (дата обращения 09.05.2024).
6. Преимущества использования языка Python для алгоритмической торговли [Электронный ресурс]. ⎯ 2021. ⎯ URL: https://dzen.ru/a/ZXNQdrUrMW9Cl4Nn (дата обращения 10.05.2024).

# ПРИЛОЖЕНИЕ

import logging  
import requests  
import time  
import typing  
import collections  
from urllib.parse import urlencode  
import hmac  
import hashlib  
import websocket  
import json  
import threading  
from models import \*  
from strategies import TechnicalStrategy, BreakoutStrategy

logger = logging.getLogger()

class BinanceClient:  
 def \_\_init\_\_(self, public\_key: str, secret\_key: str, testnet: bool, futures: bool):  
 self.futures = futures  
 if self.futures:  
 self.platform = "binance\_futures"  
 if testnet:  
 self.\_base\_url = "https://testnet.binancefuture.com"  
 self.\_wss\_url = "wss://stream.binancefuture.com/ws"  
 else:  
 self.\_base\_url = "https://fapi.binance.com"  
 self.\_wss\_url = "wss://fstream.binance.com/ws"  
 else:  
 self.platform = "binance\_spot"  
 if testnet:  
 self.\_base\_url = "https://testnet.binance.vision"  
 self.\_wss\_url = "wss://testnet.binance.vision/ws"  
 else:  
 self.\_base\_url = "https://api.binance.com"  
 self.\_wss\_url = "wss://stream.binance.com:9443/ws"  
 self.\_public\_key = public\_key  
 self.\_secret\_key = secret\_key  
 self.\_headers = {'X-MBX-APIKEY': self.\_public\_key}  
 self.contracts = self.get\_contracts()  
 self.balances = self.get\_balances()  
  
 self.prices = dict()  
 self.strategies: typing.Dict[int, typing.Union[TechnicalStrategy, BreakoutStrategy]] = dict()  
 self.logs = []  
 self.\_ws\_id = 1  
 self.ws: websocket.WebSocketApp  
 self.reconnect = True  
 self.ws\_connected = False  
 self.ws\_subscriptions = {"bookTicker": [], "aggTrade": []}  
 t = threading.Thread(target=self.\_start\_ws)  
 t.start()  
 logger.info("Binance Futures Client successfully initialized")  
  
 def \_add\_log(self, msg: str):  
  
 logger.info("%s", msg)  
 self.logs.append({"log": msg, "displayed": False})  
  
 def \_generate\_signature(self, data: typing.Dict) -> str:  
 return hmac.new(self.\_secret\_key.encode(), urlencode(data).encode(), hashlib.sha256).hexdigest()  
  
 def \_make\_request(self, method: str, endpoint: str, data: typing.Dict):  
 if method == "GET":  
 try:  
 response = requests.get(self.\_base\_url + endpoint, params=data, headers=self.\_headers)  
 except Exception as e: # Takes into account any possible error, most likely network errors  
 logger.error("Connection error while making %s request to %s: %s", method, endpoint, e)  
 return None  
 elif method == "POST":  
 try:  
 response = requests.post(self.\_base\_url + endpoint, params=data, headers=self.\_headers)  
 except Exception as e:  
 logger.error("Connection error while making %s request to %s: %s", method, endpoint, e)  
 return None  
 elif method == "DELETE":  
 try:  
 response = requests.delete(self.\_base\_url + endpoint, params=data, headers=self.\_headers)  
 except Exception as e:  
 logger.error("Connection error while making %s request to %s: %s", method, endpoint, e)  
 return None  
 else:  
 raise ValueError()  
  
 if response.status\_code == 200: # 200 is the response code of successful requests  
 return response.json()  
 else:  
 logger.error("Error while making %s request to %s: %s (error code %s)",  
 method, endpoint, response.json(), response.status\_code)  
 return None  
  
 def get\_contracts(self) -> typing.Dict[str, Contract]:  
 if self.futures:  
 exchange\_info = self.\_make\_request("GET", "/fapi/v1/exchangeInfo", dict())  
 else:  
 exchange\_info = self.\_make\_request("GET", "/api/v3/exchangeInfo", dict())  
 contracts = dict()  
 if exchange\_info is not None:  
 for contract\_data in exchange\_info['symbols']:  
 contracts[contract\_data['symbol']] = Contract(contract\_data, self.platform)  
 return collections.OrderedDict(sorted(contracts.items())) # Sort keys of the dictionary alphabetically  
  
 def get\_historical\_candles(self, contract: Contract, interval: str) -> typing.List[Candle]:  
 data = dict()  
 data['symbol'] = contract.symbol  
 data['interval'] = interval  
 data['limit'] = 1000 # The maximum number of candles is 1000 on Binance Spot  
 if self.futures:  
 raw\_candles = self.\_make\_request("GET", "/fapi/v1/klines", data)  
 else:  
 raw\_candles = self.\_make\_request("GET", "/api/v3/klines", data)  
 candles = []  
 if raw\_candles is not None:  
 for c in raw\_candles:  
 candles.append(Candle(c, interval, self.platform))  
 return candles  
  
 def get\_bid\_ask(self, contract: Contract) -> typing.Dict[str, float]:  
 data = dict()  
 data['symbol'] = contract.symbol  
  
 if self.futures:  
 ob\_data = self.\_make\_request("GET", "/fapi/v1/ticker/bookTicker", data)  
 else:  
 ob\_data = self.\_make\_request("GET", "/api/v3/ticker/bookTicker", data)  
 if ob\_data is not None:  
 if contract.symbol not in self.prices: # Add the symbol to the dictionary if needed  
 self.prices[contract.symbol] = {'bid': float(ob\_data['bidPrice']), 'ask': float(ob\_data['askPrice'])}  
 else:  
 self.prices[contract.symbol]['bid'] = float(ob\_data['bidPrice'])  
 self.prices[contract.symbol]['ask'] = float(ob\_data['askPrice'])  
 return self.prices[contract.symbol]  
  
 def get\_balances(self) -> typing.Dict[str, Balance]:  
 data = dict()  
 data['timestamp'] = int(time.time() \* 1000)  
 data['signature'] = self.\_generate\_signature(data)  
 balances = dict()  
 if self.futures:  
 account\_data = self.\_make\_request("GET", "/fapi/v2/account", data)  
 else:  
 account\_data = self.\_make\_request("GET", "/api/v3/account", data)  
 if account\_data is not None:  
 if self.futures:  
 for a in account\_data['assets']:  
 balances[a['asset']] = Balance(a, self.platform)  
 else:  
 for a in account\_data['balances']:  
 balances[a['asset']] = Balance(a, self.platform)  
 return balances  
  
 def place\_order(self, contract: Contract, order\_type: str, quantity: float, side: str, price=None, tif=None) -> OrderStatus:  
 data = dict()  
 data['symbol'] = contract.symbol  
 data['side'] = side.upper()  
 data['quantity'] = round(int(quantity / contract.lot\_size) \* contract.lot\_size, 8) # int() to round down  
 data['type'] = order\_type.upper() # Makes sure the order type is in uppercase  
 if price is not None:  
 data['price'] = round(round(price / contract.tick\_size) \* contract.tick\_size, 8)  
 data['price'] = '%.\*f' % (contract.price\_decimals, data['price']) # Avoids scientific notation  
 if tif is not None:  
 data['timeInForce'] = tif  
 data['timestamp'] = int(time.time() \* 1000)  
 data['signature'] = self.\_generate\_signature(data)  
 if self.futures:  
 order\_status = self.\_make\_request("POST", "/fapi/v1/order", data)  
 else:  
 order\_status = self.\_make\_request("POST", "/api/v3/order", data)  
 if order\_status is not None:  
 if not self.futures:  
 if order\_status['status'] == "FILLED":  
 order\_status['avgPrice'] = self.\_get\_execution\_price(contract, order\_status['orderId'])  
 else:  
 order\_status['avgPrice'] = 0  
 order\_status = OrderStatus(order\_status, self.platform)  
 return order\_status  
  
 def cancel\_order(self, contract: Contract, order\_id: int) -> OrderStatus:  
 data = dict()  
 data['orderId'] = order\_id  
 data['symbol'] = contract.symbol  
  
 data['timestamp'] = int(time.time() \* 1000)  
 data['signature'] = self.\_generate\_signature(data)  
 if self.futures:  
 order\_status = self.\_make\_request("DELETE", "/fapi/v1/order", data)  
 else:  
 order\_status = self.\_make\_request("DELETE", "/api/v3/order", data)  
 if order\_status is not None:  
 if not self.futures:  
 # Get the average execution price based on the recent trades  
 order\_status['avgPrice'] = self.\_get\_execution\_price(contract, order\_id)  
 order\_status = OrderStatus(order\_status, self.platform)  
 return order\_status  
  
 def \_get\_execution\_price(self, contract: Contract, order\_id: int) -> float:  
 data = dict()  
 data['timestamp'] = int(time.time() \* 1000)  
 data['symbol'] = contract.symbol  
 data['signature'] = self.\_generate\_signature(data)  
 trades = self.\_make\_request("GET", "/api/v3/myTrades", data)  
 avg\_price = 0  
 if trades is not None:  
 executed\_qty = 0  
 for t in trades:  
 if t['orderId'] == order\_id:  
 executed\_qty += float(t['qty'])  
 for t in trades:  
 if t['orderId'] == order\_id:  
 fill\_pct = float(t['qty']) / executed\_qty  
 avg\_price += (float(t['price']) \* fill\_pct) # Weighted sum  
 return round(round(avg\_price / contract.tick\_size) \* contract.tick\_size, 8)  
  
 def get\_order\_status(self, contract: Contract, order\_id: int) -> OrderStatus:  
 data = dict()  
 data['timestamp'] = int(time.time() \* 1000)  
 data['symbol'] = contract.symbol  
 data['orderId'] = order\_id  
 data['signature'] = self.\_generate\_signature(data)  
 if self.futures:  
 order\_status = self.\_make\_request("GET", "/fapi/v1/order", data)  
 else:  
 order\_status = self.\_make\_request("GET", "/api/v3/order", data)  
 if order\_status is not None:  
 if not self.futures:  
 if order\_status['status'] == "FILLED":  
 # Get the average execution price based on the recent trades  
 order\_status['avgPrice'] = self.\_get\_execution\_price(contract, order\_id)  
 else:  
 order\_status['avgPrice'] = 0  
 order\_status = OrderStatus(order\_status, self.platform)  
 return order\_status  
  
 def \_on\_message(self, ws, msg: str):  
 data = json.loads(msg)  
 if "u" in data and "A" in data:  
 data['e'] = "bookTicker" # For Binance Spot, to make the data structure uniform with Binance Futures  
 if "e" in data:  
 if data['e'] == "bookTicker":  
 symbol = data['s']  
 if symbol not in self.prices:  
 self.prices[symbol] = {'bid': float(data['b']), 'ask': float(data['a'])}  
 else:  
 self.prices[symbol]['bid'] = float(data['b'])  
 self.prices[symbol]['ask'] = float(data['a'])  
  
 # PNL Calculation  
 try:  
 for b\_index, strat in self.strategies.items():  
 if strat.contract.symbol == symbol:  
 for trade in strat.trades:  
 if trade.status == "open" and trade.entry\_price is not None:  
 if trade.side == "long":  
 trade.pnl = (self.prices[symbol]['bid'] - trade.entry\_price) \* trade.quantity  
 elif trade.side == "short":  
 trade.pnl = (trade.entry\_price - self.prices[symbol]['ask']) \* trade.quantity  
 except RuntimeError as e: # Handles the case the dictionary is modified while loop through it  
 logger.error("Error while looping through the Binance strategies: %s", e)  
 if data['e'] == "aggTrade":  
 symbol = data['s']  
 for key, strat in self.strategies.items():  
 if strat.contract.symbol == symbol:  
 res = strat.parse\_trades(float(data['p']), float(data['q']), data['T']) # Updates candlesticks  
 strat.check\_trade(res)  
  
 def subscribe\_channel(self, contracts: typing.List[Contract], channel: str, reconnection=False):  
 if len(contracts) > 200:  
 logger.warning("Subscribing to more than 200 symbols will most likely fail. "  
 "Consider subscribing only when adding a symbol to your Watchlist or when starting a "  
 "strategy for a symbol.")  
 data = dict()  
 data['method'] = "SUBSCRIBE"  
 data['params'] = []  
 if len(contracts) == 0:  
 data['params'].append(channel)  
 else:  
 for contract in contracts:  
 if contract.symbol not in self.ws\_subscriptions[channel] or reconnection: data['params'].append(contract.symbol.lower() + "@" + channel)  
 if contract.symbol not in self.ws\_subscriptions[channel]:  
 self.ws\_subscriptions[channel].append(contract.symbol)  
 if len(data['params']) == 0:  
 return  
 data['id'] = self.\_ws\_id  
 try:  
 self.ws.send(json.dumps(data)) # Converts the JSON object (dictionary) to a JSON string  
 logger.info("Binance: subscribing to: %s", ','.join(data['params']))  
 except Exception as e:  
 logger.error("Websocket error while subscribing to @bookTicker and @aggTrade: %s", e)  
 self.\_ws\_id += 1

import logging  
import requests  
import time  
import typing  
import collections  
from urllib.parse import urlencode  
import hmac  
import hashlib  
import websocket  
import json  
import dateutil.parser  
import threading  
from models import \*  
from strategies import TechnicalStrategy, BreakoutStrategy

logger = logging.getLogger()  
  
class BitmexClient:  
 def \_\_init\_\_(self, public\_key: str, secret\_key: str, testnet: bool):  
 self.futures = True  
 self.platform = "bitmex" # Just to have more homogeneous connectors, even if self.platform is not used  
 if testnet:  
 self.\_base\_url = "https://testnet.bitmex.com"  
 self.\_wss\_url = "wss://testnet.bitmex.com/realtime"  
 else:  
 self.\_base\_url = "https://www.bitmex.com"  
 self.\_wss\_url = "wss://www.bitmex.com/realtime"  
 self.\_public\_key = public\_key  
 self.\_secret\_key = secret\_key  
 self.ws: websocket.WebSocketApp  
 self.reconnect = True  
 self.contracts = self.get\_contracts()  
 self.balances = self.get\_balances()  
 self.prices = dict()  
 self.strategies: typing.Dict[int, typing.Union[TechnicalStrategy, BreakoutStrategy]] = dict()  
 self.logs = []  
 t = threading.Thread(target=self.\_start\_ws)  
 t.start()  
 logger.info("Bitmex Client successfully initialized")

def \_add\_log(self, msg: str):  
 logger.info("%s", msg)  
 self.logs.append({"log": msg, "displayed": False})  
  
 def \_generate\_signature(self, method: str, endpoint: str, expires: str, data: typing.Dict) -> str:  
 message = method + endpoint + "?" + urlencode(data) + expires if len(data) > 0 else method + endpoint + expires  
 return hmac.new(self.\_secret\_key.encode(), message.encode(), hashlib.sha256).hexdigest()  
  
 def \_make\_request(self, method: str, endpoint: str, data: typing.Dict):  
 headers = dict()  
 expires = str(int(time.time()) + 5)  
 headers['api-expires'] = expires  
 headers['api-key'] = self.\_public\_key  
 headers['api-signature'] = self.\_generate\_signature(method, endpoint, expires, data)  
 if method == "GET":  
 try:  
 response = requests.get(self.\_base\_url + endpoint, params=data, headers=headers)  
 except Exception as e:  
 logger.error("Connection error while making %s request to %s: %s", method, endpoint, e)  
 return None  
 elif method == "POST":  
 try:  
 response = requests.post(self.\_base\_url + endpoint, params=data, headers=headers)  
 except Exception as e:  
 logger.error("Connection error while making %s request to %s: %s", method, endpoint, e)  
 return None  
 elif method == "DELETE":  
 try:  
 response = requests.delete(self.\_base\_url + endpoint, params=data, headers=headers)  
 except Exception as e:  
 logger.error("Connection error while making %s request to %s: %s", method, endpoint, e)  
 return None  
 else:  
 raise ValueError()  
 if response.status\_code == 200:  
 return response.json()  
 else:  
 logger.error("Error while making %s request to %s: %s (error code %s)",  
 method, endpoint, response.json(), response.status\_code)  
 return None  
  
 def get\_contracts(self) -> typing.Dict[str, Contract]:  
 instruments = self.\_make\_request("GET", "/api/v1/instrument/active", dict())  
 contracts = dict()  
 if instruments is not None:  
 for s in instruments:  
 contracts[s['symbol']] = Contract(s, "bitmex")  
 return collections.OrderedDict(sorted(contracts.items())) # Sort keys of the dictionary alphabetically  
  
 def get\_balances(self) -> typing.Dict[str, Balance]:  
 data = dict()  
 data['currency'] = "all"  
 margin\_data = self.\_make\_request("GET", "/api/v1/user/margin", data)  
 balances = dict()  
 if margin\_data is not None:  
 for a in margin\_data:  
 balances[a['currency']] = Balance(a, "bitmex")  
 return balances  
  
 def get\_historical\_candles(self, contract: Contract, timeframe: str) -> typing.List[Candle]:  
 data = dict()  
 data['symbol'] = contract.symbol  
 data['partial'] = True  
 data['binSize'] = timeframe  
 data['count'] = 500  
 data['reverse'] = True  
 raw\_candles = self.\_make\_request("GET", "/api/v1/trade/bucketed", data)  
 candles = []  
 if raw\_candles is not None:  
 for c in reversed(raw\_candles):  
 if c['open'] is None or c['close'] is None: # Some candles returned by Bitmex miss data  
 continue  
 candles.append(Candle(c, timeframe, "bitmex"))  
 return candles  
  
 def place\_order(self, contract: Contract, order\_type: str, quantity: int, side: str, price=None, tif=None) -> OrderStatus:  
 data = dict()  
 data['symbol'] = contract.symbol  
 data['side'] = side.capitalize()  
 data['orderQty'] = round(quantity / contract.lot\_size) \* contract.lot\_size  
 data['ordType'] = order\_type.capitalize()  
 if price is not None:  
 data['price'] = round(round(price / contract.tick\_size) \* contract.tick\_size, 8)  
 if tif is not None:  
 data['timeInForce'] = tif  
 order\_status = self.\_make\_request("POST", "/api/v1/order", data)  
 if order\_status is not None:  
 order\_status = OrderStatus(order\_status, "bitmex")  
 return order\_status  
  
 def cancel\_order(self, order\_id: str) -> OrderStatus:  
 data = dict()  
 data['orderID'] = order\_id  
  
 order\_status = self.\_make\_request("DELETE", "/api/v1/order", data)  
 if order\_status is not None:  
 order\_status = OrderStatus(order\_status[0], "bitmex")  
 return order\_status  
  
 def get\_order\_status(self, contract: Contract, order\_id: str) -> OrderStatus:  
 data = dict()  
 data['symbol'] = contract.symbol  
 data['reverse'] = True  
 order\_status = self.\_make\_request("GET", "/api/v1/order", data)  
 if order\_status is not None:  
 for order in order\_status:  
 if order['orderID'] == order\_id:  
 return OrderStatus(order, "bitmex")

def \_on\_message(self, ws, msg: str):  
 data = json.loads(msg)  
 if "table" in data:  
 if data['table'] == "instrument":  
 for d in data['data']:  
 symbol = d['symbol']  
 if symbol not in self.prices:  
 self.prices[symbol] = {'bid': None, 'ask': None}  
 if 'bidPrice' in d:  
 self.prices[symbol]['bid'] = d['bidPrice']  
 if 'askPrice' in d:  
 self.prices[symbol]['ask'] = d['askPrice']  
 # PNL Calculation  
 try:  
 for b\_index, strat in self.strategies.items():  
 if strat.contract.symbol == symbol:  
 for trade in strat.trades:  
 if trade.status == "open" and trade.entry\_price is not None:  
  
 if trade.side == "long":  
 price = self.prices[symbol]['bid']  
 else:  
 price = self.prices[symbol]['ask']  
 multiplier = trade.contract.multiplier  
  
 if trade.contract.inverse:  
 if trade.side == "long":  
 trade.pnl = (1 / trade.entry\_price - 1 / price) \* multiplier \* trade.quantity  
 elif trade.side == "short":  
 trade.pnl = (1 / price - 1 / trade.entry\_price) \* multiplier \* trade.quantity  
 else:  
 if trade.side == "long":  
 trade.pnl = (price - trade.entry\_price) \* multiplier \* trade.quantity  
 elif trade.side == "short":  
 trade.pnl = (trade.entry\_price - price) \* multiplier \* trade.quantity  
 except RuntimeError as e:  
 logger.error("Error while looping through the Bitmex strategies: %s", e)  
 if data['table'] == "trade":  
 for d in data['data']:  
 symbol = d['symbol']  
 ts = int(dateutil.parser.isoparse(d['timestamp']).timestamp() \* 1000)  
 for key, strat in self.strategies.items():  
 if strat.contract.symbol == symbol:  
 res = strat.parse\_trades(float(d['price']), float(d['size']), ts)  
 strat.check\_trade(res)

import tkinter as tk  
from tkinter.messagebox import askquestion  
import logging  
import json  
from connectors.bitmex import BitmexClient  
from connectors.binance import BinanceClient  
from interface.styling import \*  
from interface.logging\_component import Logging  
from interface.watchlist\_component import Watchlist  
from interface.trades\_component import TradesWatch  
from interface.strategy\_component import StrategyEditor  
  
logger = logging.getLogger() # This will be the same logger object as the one configured in main.py  
  
class Root(tk.Tk):  
 def \_\_init\_\_(self, binance: BinanceClient, bitmex: BitmexClient):  
 super().\_\_init\_\_()  
 self.binance = binance  
 self.bitmex = bitmex  
 self.title("Trading Bot")  
 self.protocol("WM\_DELETE\_WINDOW", self.\_ask\_before\_close)  
 self.configure(bg=BG\_COLOR)  
 # Create the menu, sub menu and menu commands  
 # You can use the menu when you don't want to overload the interface with too many buttons  
 self.main\_menu = tk.Menu(self)  
 self.configure(menu=self.main\_menu)  
 self.workspace\_menu = tk.Menu(self.main\_menu, tearoff=False)  
 self.main\_menu.add\_cascade(label="Workspace", menu=self.workspace\_menu)  
 self.workspace\_menu.add\_command(label="Save workspace", command=self.\_save\_workspace)  
 # Separates the root component in two blocks  
 self.\_left\_frame = tk.Frame(self, bg=BG\_COLOR)  
 self.\_left\_frame.pack(side=tk.LEFT)  
 self.\_right\_frame = tk.Frame(self, bg=BG\_COLOR)  
 self.\_right\_frame.pack(side=tk.LEFT)  
 # Creates and places components at the top and bottom of the left and right frame  
 self.\_watchlist\_frame = Watchlist(self.binance.contracts, self.bitmex.contracts, self.\_left\_frame, bg=BG\_COLOR)  
 self.\_watchlist\_frame.pack(side=tk.TOP, padx=10)  
 self.logging\_frame = Logging(self.\_left\_frame, bg=BG\_COLOR)  
 self.logging\_frame.pack(side=tk.TOP, pady=15) # Space a bit the components with vertical padding  
 self.\_strategy\_frame = StrategyEditor(self, self.binance, self.bitmex, self.\_right\_frame, bg=BG\_COLOR)  
 self.\_strategy\_frame.pack(side=tk.TOP, pady=15)  
 self.\_trades\_frame = TradesWatch(self.\_right\_frame, bg=BG\_COLOR)  
 self.\_trades\_frame.pack(side=tk.TOP, pady=15)  
 self.\_update\_ui() # Starts the infinite interface update loop  
  
 def \_ask\_before\_close(self):  
 result = askquestion("Confirmation", "Do you really want to exit the application?")  
 if result == "yes":  
 self.binance.reconnect = False # Avoids the infinite reconnect loop in \_start\_ws()  
 self.bitmex.reconnect = False  
 self.binance.ws.close()  
 self.bitmex.ws.close()  
 self.destroy() # Destroys the UI and terminates the program as no other thread is running  
  
 def \_update\_ui(self):  
 # Logs  
 for log in self.bitmex.logs:  
 if not log['displayed']:  
 self.logging\_frame.add\_log(log['log'])  
 log['displayed'] = True  
 for log in self.binance.logs:  
 if not log['displayed']:  
 self.logging\_frame.add\_log(log['log'])  
 log['displayed'] = True  
 # Trades and Logs  
 for client in [self.binance, self.bitmex]:  
 try: # try...except statement to handle the case when a dictionary is updated during the following loops  
 for b\_index, strat in client.strategies.items():  
 for log in strat.logs:  
 if not log['displayed']: self.logging\_frame.add\_log(log['log'])  
 log['displayed'] = True  
 # Update the Trades component (add a new trade, change status/PNL)  
 for trade in strat.trades:  
 if trade.time not in self.\_trades\_frame.body\_widgets['symbol']: self.\_trades\_frame.add\_trade(trade)  
 if "binance" in trade.contract.exchange:  
 precision = trade.contract.price\_decimals  
 else:  
 precision = 8 # The Bitmex PNL is always is BTC, thus 8 decimals  
 pnl\_str = "{0:.{prec}f}".format(trade.pnl, prec=precision)  
 self.\_trades\_frame.body\_widgets['pnl\_var'][trade.time].set(pnl\_str) self.\_trades\_frame.body\_widgets['status\_var'][trade.time].set(trade.status.capitalize()) self.\_trades\_frame.body\_widgets['quantity\_var'][trade.time].set(trade.quantity)  
 except RuntimeError as e:  
 logger.error("Error while looping through strategies dictionary: %s", e)  
 # Watchlist prices  
 try:  
 for key, value in self.\_watchlist\_frame.body\_widgets['symbol'].items():  
 symbol = self.\_watchlist\_frame.body\_widgets['symbol'][key].cget("text")  
 exchange = self.\_watchlist\_frame.body\_widgets['exchange'][key].cget("text")  
 if exchange == "Binance":  
 if symbol not in self.binance.contracts:  
 continue  
 if symbol not in self.binance.ws\_subscriptions["bookTicker"] and self.binance.ws\_connected: self.binance.subscribe\_channel([self.binance.contracts[symbol]], "bookTicker")  
 if symbol not in self.binance.prices: self.binance.get\_bid\_ask(self.binance.contracts[symbol])  
 continue  
 precision = self.binance.contracts[symbol].price\_decimals  
 prices = self.binance.prices[symbol]  
 elif exchange == "Bitmex":  
 if symbol not in self.bitmex.contracts:  
 continue  
 if symbol not in self.bitmex.prices:  
 continue  
 precision = self.bitmex.contracts[symbol].price\_decimals  
 prices = self.bitmex.prices[symbol]  
 else:  
 continue  
 if prices['bid'] is not None:  
 price\_str = "{0:.{prec}f}".format(prices['bid'], prec=precision) self.\_watchlist\_frame.body\_widgets['bid\_var'][key].set(price\_str)  
 if prices['ask'] is not None:  
 price\_str = "{0:.{prec}f}".format(prices['ask'], prec=precision)  
 self.\_watchlist\_frame.body\_widgets['ask\_var'][key].set(price\_str)  
 except RuntimeError as e:  
 logger.error("Error while looping through watchlist dictionary: %s", e)  
 self.after(1500, self.\_update\_ui)  
  
 def \_save\_workspace(self):  
 # Watchlist  
 watchlist\_symbols = []  
 for key, value in self.\_watchlist\_frame.body\_widgets['symbol'].items():  
 symbol = value.cget("text")  
 exchange = self.\_watchlist\_frame.body\_widgets['exchange'][key].cget("text")  
 watchlist\_symbols.append((symbol, exchange,))  
 self.\_watchlist\_frame.db.save("watchlist", watchlist\_symbols)  
 # Strategies  
 strategies = []  
 strat\_widgets = self.\_strategy\_frame.body\_widgets  
 for b\_index in strat\_widgets['contract']: # Loops through the rows of a column (not necessarily the 'contract' one  
 strategy\_type = strat\_widgets['strategy\_type\_var'][b\_index].get()  
 contract = strat\_widgets['contract\_var'][b\_index].get()  
 timeframe = strat\_widgets['timeframe\_var'][b\_index].get()  
 balance\_pct = strat\_widgets['balance\_pct'][b\_index].get()  
 take\_profit = strat\_widgets['take\_profit'][b\_index].get()  
 stop\_loss = strat\_widgets['stop\_loss'][b\_index].get()  
 # Extra parameters are all saved in one column as a JSON string because they change based on the strategy  
 extra\_params = dict()  
 for param in self.\_strategy\_frame.extra\_params[strategy\_type]:  
 code\_name = param['code\_name']  
 extra\_params[code\_name] = self.\_strategy\_frame.additional\_parameters[b\_index][code\_name]  
 strategies.append((strategy\_type, contract, timeframe, balance\_pct, take\_profit, stop\_loss,  
 json.dumps(extra\_params),))  
 self.\_strategy\_frame.db.save("strategies", strategies)  
 self.logging\_frame.add\_log("Workspace saved")

import sqlite3  
import typing  
  
class WorkspaceData:  
 def \_\_init\_\_(self):  
 self.conn = sqlite3.connect("database.db")  
 self.conn.row\_factory = sqlite3.Row # Makes the data retrieved from the database accessible by their column name  
 self.cursor = self.conn.cursor()  
 self.cursor.execute("CREATE TABLE IF NOT EXISTS watchlist (symbol TEXT, exchange TEXT)")  
 self.cursor.execute("CREATE TABLE IF NOT EXISTS strategies (strategy\_type TEXT, contract TEXT,"  
 "timeframe TEXT, balance\_pct REAL, take\_profit REAL, stop\_loss REAL, extra\_params TEXT)")  
 self.conn.commit() # Saves the changes  
  
 def save(self, table: str, data: typing.List[typing.Tuple]):  
 self.cursor.execute(f"DELETE FROM {table}")  
 table\_data = self.cursor.execute(f"SELECT \* FROM {table}")  
 columns = [description[0] for description in table\_data.description] # Lists the columns of the table  
 # Creates the SQL insert statement dynamically  
 sql\_statement = f"INSERT INTO {table} ({', '.join(columns)}) VALUES ({', '.join(['?'] \* len(columns))})"  
 self.cursor.executemany(sql\_statement, data)  
 self.conn.commit()  
 def get(self, table: str) -> typing.List[sqlite3.Row]:  
 self.cursor.execute(f"SELECT \* FROM {table}")  
 data = self.cursor.fetchall()  
 return data

import logging  
from typing import \*  
import time  
from threading import Timer  
import pandas as pd  
from models import \*

if TYPE\_CHECKING: # Import the connector class names only for typing purpose (the classes aren't actually imported)  
 from connectors.bitmex import BitmexClient  
 from connectors.binance import BinanceClient

logger = logging.getLogger()  
  
# TF\_EQUIV is used in parse\_trades() to compare the last candle timestamp to the new trade timestamp  
TF\_EQUIV = {"1m": 60, "5m": 300, "15m": 900, "30m": 1800, "1h": 3600, "4h": 14400}  
  
class Strategy:  
 def \_\_init\_\_(self, client: Union["BitmexClient", "BinanceClient"], contract: Contract, exchange: str,  
 timeframe: str, balance\_pct: float, take\_profit: float, stop\_loss: float, strat\_name):  
 self.client = client  
 self.contract = contract  
 self.exchange = exchange  
 self.tf = timeframe  
 self.tf\_equiv = TF\_EQUIV[timeframe] \* 1000  
 self.balance\_pct = balance\_pct  
 self.take\_profit = take\_profit  
 self.stop\_loss = stop\_loss  
 self.strat\_name = strat\_name  
 self.ongoing\_position = False  
 self.candles: List[Candle] = []  
 self.trades: List[Trade] = []  
 self.logs = []  
  
 def \_add\_log(self, msg: str):  
 logger.info("%s", msg)  
 self.logs.append({"log": msg, "displayed": False})  
  
 def parse\_trades(self, price: float, size: float, timestamp: int) -> str:  
 timestamp\_diff = int(time.time() \* 1000) - timestamp  
 if timestamp\_diff >= 2000:  
 logger.warning("%s %s: %s milliseconds of difference between the current time and the trade time",  
 self.exchange, self.contract.symbol, timestamp\_diff)  
 last\_candle = self.candles[-1]  
 # Same Candle  
 if timestamp < last\_candle.timestamp + self.tf\_equiv:  
 last\_candle.close = price  
 last\_candle.volume += size  
 if price > last\_candle.high:  
 last\_candle.high = price  
 elif price < last\_candle.low:  
 last\_candle.low = price  
 # Check Take profit / Stop loss  
 for trade in self.trades:  
 if trade.status == "open" and trade.entry\_price is not None:  
 self.\_check\_tp\_sl(trade)  
 return "same\_candle"  
 # Missing Candle(s)  
 elif timestamp >= last\_candle.timestamp + 2 \* self.tf\_equiv:  
 missing\_candles = int((timestamp - last\_candle.timestamp) / self.tf\_equiv) - 1  
 logger.info("%s missing %s candles for %s %s (%s %s)", self.exchange, missing\_candles, self.contract.symbol,  
 self.tf, timestamp, last\_candle.timestamp)  
 for missing in range(missing\_candles):  
 new\_ts = last\_candle.timestamp + self.tf\_equiv  
 candle\_info = {'ts': new\_ts, 'open': last\_candle.close, 'high': last\_candle.close,  
 'low': last\_candle.close, 'close': last\_candle.close, 'volume': 0}  
 new\_candle = Candle(candle\_info, self.tf, "parse\_trade")  
 self.candles.append(new\_candle)  
 last\_candle = new\_candle  
 new\_ts = last\_candle.timestamp + self.tf\_equiv  
 candle\_info = {'ts': new\_ts, 'open': price, 'high': price, 'low': price, 'close': price, 'volume': size}  
 new\_candle = Candle(candle\_info, self.tf, "parse\_trade")  
 self.candles.append(new\_candle)  
 return "new\_candle"  
 # New Candle  
 elif timestamp >= last\_candle.timestamp + self.tf\_equiv:  
 new\_ts = last\_candle.timestamp + self.tf\_equiv  
 candle\_info = {'ts': new\_ts, 'open': price, 'high': price, 'low': price, 'close': price, 'volume': size}  
 new\_candle = Candle(candle\_info, self.tf, "parse\_trade")  
 self.candles.append(new\_candle)  
 logger.info("%s New candle for %s %s", self.exchange, self.contract.symbol, self.tf)  
 return "new\_candle"  
  
 def \_check\_order\_status(self, order\_id):  
 order\_status = self.client.get\_order\_status(self.contract, order\_id)  
 if order\_status is not None:  
 logger.info("%s order status: %s", self.exchange, order\_status.status)  
 if order\_status.status == "filled":  
 for trade in self.trades:  
 if trade.entry\_id == order\_id:  
 trade.entry\_price = order\_status.avg\_price  
 trade.quantity = order\_status.executed\_qty  
 break  
 return  
 t = Timer(2.0, lambda: self.\_check\_order\_status(order\_id))  
 t.start()  
  
 def \_open\_position(self, signal\_result: int):  
 # Short is not allowed on Spot platforms  
 if self.client.platform == "binance\_spot" and signal\_result == -1:  
 return  
 trade\_size = self.client.get\_trade\_size(self.contract, self.candles[-1].close, self.balance\_pct)  
 if trade\_size is None:  
 return  
 order\_side = "buy" if signal\_result == 1 else "sell"  
 position\_side = "long" if signal\_result == 1 else "short"  
 self.\_add\_log(f"{position\_side.capitalize()} signal on {self.contract.symbol} {self.tf}")  
 order\_status = self.client.place\_order(self.contract, "MARKET", trade\_size, order\_side)  
 if order\_status is not None:  
 self.\_add\_log(f"{order\_side.capitalize()} order placed on {self.exchange} | Status: {order\_status.status}")  
 self.ongoing\_position = True  
 avg\_fill\_price = None  
 if order\_status.status == "filled":  
 avg\_fill\_price = order\_status.avg\_price  
 else:  
 t = Timer(2.0, lambda: self.\_check\_order\_status(order\_status.order\_id))  
 t.start()  
 new\_trade = Trade({"time": int(time.time() \* 1000), "entry\_price": avg\_fill\_price,  
 "contract": self.contract, "strategy": self.strat\_name, "side": position\_side,  
 "status": "open", "pnl": 0, "quantity": order\_status.executed\_qty, "entry\_id": order\_status.order\_id})  
 self.trades.append(new\_trade)  
  
 def \_check\_tp\_sl(self, trade: Trade):  
 tp\_triggered = False  
 sl\_triggered = False  
 price = self.candles[-1].close  
 if trade.side == "long":  
 if self.stop\_loss is not None:  
 if price <= trade.entry\_price \* (1 - self.stop\_loss / 100):  
 sl\_triggered = True  
 if self.take\_profit is not None:  
 if price >= trade.entry\_price \* (1 + self.take\_profit / 100):  
 tp\_triggered = True  
 elif trade.side == "short":  
 if self.stop\_loss is not None:  
 if price >= trade.entry\_price \* (1 + self.stop\_loss / 100):  
 sl\_triggered = True  
 if self.take\_profit is not None:  
 if price <= trade.entry\_price \* (1 - self.take\_profit / 100):  
 tp\_triggered = True  
 if tp\_triggered or sl\_triggered:  
 self.\_add\_log(f"{'Stop loss' if sl\_triggered else 'Take profit'} for {self.contract.symbol} {self.tf} "  
 f"| Current Price = {price} (Entry price was {trade.entry\_price})")  
 order\_side = "SELL" if trade.side == "long" else "BUY"  
 if not self.client.futures:  
 # Make sure we don't sell more than what's in the available balance on Binance Spot  
 current\_balances = self.client.get\_balances()  
 if current\_balances is not None:  
 if order\_side == "SELL" and self.contract.base\_asset in current\_balances:  
 trade.quantity = min(current\_balances[self.contract.base\_asset].free, trade.quantity)  
 order\_status = self.client.place\_order(self.contract, "MARKET", trade.quantity, order\_side)  
 if order\_status is not None:  
 self.\_add\_log(f"Exit order on {self.contract.symbol} {self.tf} placed successfully")  
 trade.status = "closed"  
 self.ongoing\_position = False  
  
class TechnicalStrategy(Strategy):  
 def \_\_init\_\_(self, client, contract: Contract, exchange: str, timeframe: str, balance\_pct: float, take\_profit: float,  
 stop\_loss: float, other\_params: Dict):  
 super().\_\_init\_\_(client, contract, exchange, timeframe, balance\_pct, take\_profit, stop\_loss, "Technical")  
 self.\_ema\_fast = other\_params['ema\_fast']  
 self.\_ema\_slow = other\_params['ema\_slow']  
 self.\_ema\_signal = other\_params['ema\_signal']  
 self.\_rsi\_length = other\_params['rsi\_length']  
  
 def \_rsi(self) -> float:  
 close\_list = []  
 for candle in self.candles:  
 close\_list.append(candle.close)  
 closes = pd.Series(close\_list)  
 # Calculate the different between the value of one row and the value of the row before  
 delta = closes.diff().dropna()  
 up, down = delta.copy(), delta.copy()  
 up[up < 0] = 0  
 down[down > 0] = 0 # Keep only the negative change, others are set to 0  
 avg\_gain = up.ewm(com=(self.\_rsi\_length - 1), min\_periods=self.\_rsi\_length).mean()  
 avg\_loss = down.abs().ewm(com=(self.\_rsi\_length - 1), min\_periods=self.\_rsi\_length).mean()  
 rs = avg\_gain / avg\_loss # Relative Strength  
 rsi = 100 - 100 / (1 + rs)  
 rsi = rsi.round(2)  
 return rsi.iloc[-2]  
  
 def \_macd(self) -> Tuple[float, float]:  
 close\_list = []  
 for candle in self.candles:  
 close\_list.append(candle.close) # Use only the close price of each candlestick for the calculations  
 closes = pd.Series(close\_list) # Converts the close prices list to a pandas Series.  
 ema\_fast = closes.ewm(span=self.\_ema\_fast).mean() # Exponential Moving Average method  
 ema\_slow = closes.ewm(span=self.\_ema\_slow).mean()  
 macd\_line = ema\_fast - ema\_slow  
 macd\_signal = macd\_line.ewm(span=self.\_ema\_signal).mean()  
 return macd\_line.iloc[-2], macd\_signal.iloc[-2]  
  
 def \_check\_signal(self):  
 macd\_line, macd\_signal = self.\_macd()  
 rsi = self.\_rsi()  
 if rsi < 30 and macd\_line > macd\_signal:  
 return 1  
 elif rsi > 70 and macd\_line < macd\_signal:  
 return -1  
 else:  
 return 0

def check\_trade(self, tick\_type: str):  
 if tick\_type == "new\_candle" and not self.ongoing\_position:  
 signal\_result = self.\_check\_signal()  
 if signal\_result in [1, -1]:  
 self.\_open\_position(signal\_result)

class BreakoutStrategy(Strategy):  
 def \_\_init\_\_(self, client, contract: Contract, exchange: str, timeframe: str, balance\_pct: float, take\_profit: float,  
 stop\_loss: float, other\_params: Dict):  
 super().\_\_init\_\_(client, contract, exchange, timeframe, balance\_pct, take\_profit, stop\_loss, "Breakout")  
 self.\_min\_volume = other\_params['min\_volume']

def \_check\_signal(self) -> int:  
 if self.candles[-1].close > self.candles[-2].high and self.candles[-1].volume > self.\_min\_volume:  
 return 1  
 elif self.candles[-1].close < self.candles[-2].low and self.candles[-1].volume > self.\_min\_volume:  
 return -1  
 else:  
 return 0  
  
 def check\_trade(self, tick\_type: str):  
 if not self.ongoing\_position:  
 signal\_result = self.\_check\_signal()  
 if signal\_result in [1, -1]:  
 self.\_open\_position(signal\_result)