НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” ________2023 р.

Дипломний проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”
спеціальності 121 “Інженерія програмного забезпечення”

на тему: Бот для автоматизації торгівлі віртуальними активами

Виконав: студент 4 курсу, групи ІП-94
(шифр групи)

Литвишко Нікіта Андрійович ________
(прізвище, ім’я, по батькові) (підпис)

Керівник ас. Валько В.В. ________
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) к. т. н. Волокита А. М. ________
(назва (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент ____________________ ________
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.

Студент ________
(підпис)

Київ – 2023 р

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
Завідувач кафедри
________ Сергій СТІРЕНКО
(підпис)
“__” ______________2023 р.

ЗАВДАННЯ

на бакалаврський дипломний проєкт студента

Литвишка Нікіти Андрійовича

1. Тема проєкту *Бот для автоматицації торгівлі віртуальними активами*

керівник проєкту ______ ас. Валько В. В. ______________________
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від 31 травня 2023 року №2102-с

2. Термін здачі студентом закінченого проєкту *15 червня 2023 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

Розділ 1. Аналіз предметної області

Розділ 2. Огляд та аналіз інстурментівв для створення бота автоматичної торгівлі на Dmarket

Розділ 3. Розробка системи

Розділ 4. Тестування розробленого програмного забезпечення

5. Перелік графічного матеріалу (з точним позначенням обов’язкових креслень) архітектура застосунку (структурна схема), діаграма використання (функціональна схема) , алгоритм дій програмного забезпечення (принципова схема).

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Волокита А. М.		

7. Дата видачі завдання «12» грудня 2022 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Термін виконання етапів проєкту	Примітки
1.	Затвердження теми проєкту	10.12.2022-15.12.2022	
2.	Вивчення та аналіз завдання	15.12.2022-15.03.2022	
3.	Розробка архітектури та загальної структури системи	15.03.2023-25.03.2023	
4.	Розробка структур окремих підсистем	25.03.2023-05.04.2023	
5.	Програмна реалізація системи	05.04.2023-15.04.2023	
6.	Оформлення пояснювальної записки	15.04.2023-11.06.2023	
7.	Захист програмного продукту	25.04.2023	
8.	Передзахист	11.06.2023	
9.	Захист	21.06.2023	

Студент-дипломник ____________ Нікіта ЛИТВИШКО

(підпис)

Керівник проєкту ____________ Володимир ВАЛЬКО

(підпис)

АНОТАЦІЯ

Пояснювальна записка дипломного проєкту складається з чотирьох розділів, містить 3 таблиці, 41 рисунків та 14 джерел – загалом 61 сторінок.

Дипломний проєкт присвячений розробці бота для автоматизованої торгівлі віртуальними активами.

Мета розробити бота для для автоматизованої торгівлі віртуальними активами.

Об'єкт дослідження: розробка бота для автоматизованої торгівлі віртуальними активами.

Предмет дослідження: використання бота для автоматизації торгівлі віртуальними активами.

Розділ 1 присвячений дослідженню предметної області.

Розділ 2 присвячений дослідженню інструментів для створення бота.

Розділ 3 присвячений розробці системи.

Розділ 4 присвячений тестуванню розробленого бота.

Програмне забезпечення впроваджено за допомогою API Dmarket.

ANNOTATION

The explanatory note of the diploma project consists of four sections, contains 3 tables, 41 figures and 14 sources - a total of 61 pages.

The diploma project is devoted to the development of a bot for automated trading of virtual assets.

The goal is to develop a bot for automated trading of virtual assets.

Research object: development of a bot for automated trading of virtual assets.

Research subject: using a bot to automate trading of virtual assets.

Chapter 1 is devoted to the study of the subject area.

Chapter 2 explores the tools for creating a bot.

Chapter 3 is devoted to system development.

Chapter 4 is devoted to testing the developed bot.

The software is implemented using the Dmarket API.

Довідник	Формат	Значення	Найменування	Кіл.листів	№ екземпляра	
			Документація загальна			
			Знову розроблена			
	A4	ІАЛЦ.467200.002 ТЗ	Бот для автоматичної торгівлі	3		
			віртуальними активами			
			Технічне завдання			
	A4	ІАЛЦ.467200.003 ПЗ	Бот для автоматичної торгівлі	61		
			віртуальними активами			
			Пояснювальна записка			
	A4	ІАЛЦ.467200.004 Д1	Бот для автоматичної торгівлі	1		
			віртуальними активами			
			Архітектура застосунку			
			(структурна схема)			
	A4	ІАЛЦ.467200.005 Д2	Бот для автоматичної торгівлі	1		
			віртуальними активами			
			Діаграма використання схема			
			(функціональна схема)			
	A4	ІАЛЦ.467200.006 Д3	Бот для автоматичної торгівлі	1		
			віртуальними активами			
			Алгоритм дій програмного			
			забезпечення			
			(принципова схема)			
	A4	ІАЛЦ.467200.007 Д4	Бот для автоматичної торгівлі	35		
			віртуальними активами			
			Текст програмного коду			

Змн.	Арк.	№ докум.	Підпис	Дата	ІАЛЦ.467200.001 ОА			
Розроб.		Литвишко Н.А			Бот для автоматизації торгівлі віртуальними активами Опис альбому	Літ.	Арк.	Акрушів
Перевір.							1	1
						КПІ ім. Ігоря Сікорського ФІОТ ІП-94		
Н. Контр.		Волокита А.М..						
Затверд.								

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Бот для автоматизації торгівлі віртуальними активами»

Київ – 2023

Зміст

1. НАЙМЕНУВАННЯ ТА ГАЛУЗЬ ЗАСТОСУВАННЯ

Дане технічне поширюється на дипломний проєкт бакалавра за темою: «Бот для автоматизації торгівлі віртуальними активами». Даний застосунок може бути практично використаний користувачами для торгівлі віртуаальними активами на Dmarket.

2. ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на дипломне проєктування, затверджене кафедрою інформатики та програмної інженерії Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського».

3. МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Розробка призначена для автоматизації та полегшенню торгівлі віртуальними активами на Dmarket.

Метою розробки є створення бота для автоматизації торгівлі віртуальними активами.

4. ДЖЕРЕЛА РОЗРОБКИ

Основними джерелами розробки є загальнодоступна документація існуючих програм, наукові статті та науково-технічна література з теорії і практики програмування

5. ТЕХНІЧНІ ВИМОГИ

Вимоги до розробленого продукту

- Можливість автоматичної роботи бота
- Гнучке та інтуітивно зрозуміле налаштування параметрів торгівлі
- Торгівля віртуальними активами

Вимоги до програмного забезпечення

- ОС Windows, Mac чи Linux.

- Python (версія 3.9)

6. СТАДІЇ І ЕТАПИ РОЗРОБКИ

№	Назва етапу	Термін виконання
1.	Затвердження теми роботи	10.12.2022-15.12.2022
2.	Вивчення та аналіз завдання	15.12.2022-15.03.2023
3.	Розробка архітектури та загальної структури системи	15.03.2023-25.03.2023
4.	Розробка структур окремих частин системи	25.03.2023-05.04.2023
5.	Програмна реалізація системи	05.04.2023-15.04.2023
6.	Виправлення помилок	15.04.2023-20.05.2023
7.	Оформлення пояснювальної записки	25.04.2023

ПОЯСНЮВАЛЬНА ЗАПИСКА

ДО ДИПЛОМНОГО ПРОЄКТУ

на тему “Бот для автоматизації торгівлі віртуальними активами”

Київ – 2023

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API — Application Programming Interface.

SQL — Structured Query Language.

FR - Functional Requirements.

NFR - Non Functional Requirements

БД - База даних

ВСТУП

Поява цифрових платформ і швидке зростання електронної комерції проклали шлях до революційних досягнень у різних секторах, у тому числі в онлайн-торгівлі. Серед цих платформ Dmarket став відомим ринком віртуальних предметів і активів в ігровій індустрії. Завдяки великій базі користувачів і різноманітному асортименту продуктів Dmarket забезпечує ідеальне середовище для трейдерів, які можуть брати участь у купівлі та продажу віртуальних товарів. Щоб оптимізувати ефективність торгівлі та використати ринкові можливості, розробці автоматизованих торгових ботів приділено значну увагу.

Актуальність цього дослідження полягає у зростаючій значущості алгоритмічної торгівлі та зростанні популярності Dmarket як видатної платформи для торгівлі віртуальними активами. Автоматизовані торгові боти можуть змінити досвід торгівлі, надаючи трейдерам передові інструменти для аналізу ринку, швидкості виконання та управління ризиками. Використовуючи потужність штучного інтелекту та алгоритмів машинного навчання, ці боти можуть обробляти величезні обсяги даних, ідентифікувати шаблони та здійснювати операції з мінімальним втручанням людини. Отже, ретельне дослідження розробки ботів для Dmarket має велике значення для трейдерів, розробників, дослідників і операторів платформ, які прагнуть оптимізувати свої торгові стратегії.

Предметом дослідження є - розробка бота для організації автоматичної торгівлі на Dmarket. Основна мета цього дослідження — розробка бота для організації автоматичної торгівлі на Dmarket, а також забезпечити повне розуміння процесу розробки та потенційних переваг автоматизованих торгових ботів на Dmarket. Використання потужності штучного інтелекту та алгоритмів машинного навчання дозволяє автоматизованим торговим ботам обробляти величезні обсяги даних, аналізувати ринкові шаблони та здійснювати операції з мінімальним втручанням людини. Глибоко вивчаючи предмет, це дослідження має на меті зробити внесок у наявні знання про алгоритмічну торгівлю та надати уявлення про ефективність і обмеження використання торгових ботів для автоматичної торгівлі на Dmarket. Крім того, це дослідження має на меті запропонувати бот трейдерам, розробникам і операторам платформ щодо підвищення їх торгових можливостей і прибутковості.

Основна мета дослідження полягає у розробці бота для автоматичної торгівлі на Dmarket, а також у вивченні процесу розробки та потенційних переваг таких автоматизованих торгових ботів. Шляхом глибокого вивчення предмета дослідження, можна зробити внесок у наші знання про алгоритмічну торгівлю та розкрити ефективність і обмеження використання ботів для автоматичної торгівлі на Dmarket.

У підсумку, розробка автоматизованих торгових ботів для платформи Dmarket є актуальним напрямком дослідження, оскільки вони можуть революціонізувати торговельний процес, забезпечуючи трейдерам передові інструменти та оптимізуючи їхні торгові стратегії. Результати цього дослідження можуть бути корисними для трейдерів, розробників і операторів платформи, які бажають використовувати автоматизовані торгові боти для підвищення ефективності та прибутковості своїх торгових операцій на Dmarket.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Віртуальні активи та їх значення

Віртуальні активи — це цифрові товари або предмети, які існують лише у віртуальному або онлайн-середовищі.[1] Ці активи можуть приймати різні форми, включаючи внутрішньо - ігрові предмети, скіни, віртуальну валюту, цифрові предмети колекціонування тощо. Вони мають цінність, тому що їх шукають геймери та інші користувачі, які хочуть покращити свій досвід онлайн або збирати унікальні предмети.

Віртуальні активи можна отримати різними способами, зокрема через гру, купівлю чи торгівлю. Деякі віртуальні активи створюють і контролюють розробники ігор, тоді як інші створюють користувачі або сторонні розробники.

Віртуальні активи набувають все більшого значення в ігровій індустрії, оскільки все більше геймерів прагнуть персоналізувати свій онлайн-досвід і виділитися з натовпу. Скіни та інші віртуальні предмети також можуть мати практичні переваги, такі як покращення ігрового процесу або розблокування нових функцій.

Окрім своєї цінності в іграх, віртуальні активи також набули значення в інших сферах, таких як цифрове мистецтво та колекціонування. Невзаємозамінні токени (NFT) — це тип віртуальних активів, який нещодавно набув популярності як спосіб підтвердження права власності на цифрове мистецтво та інші унікальні предмети.[1]

Загалом віртуальні активи стали важливою частиною онлайн-культури, пропонуючи користувачам спосіб покращити свій онлайн-досвід і виразити себе новими та унікальними способами. Їхнє значення та цінність, ймовірно, продовжуватимуть розвиватися разом із розвитком технологій та онлайн-культури.

1.1.1 Види віртуальних активів

Віртуальні активи — це цифрові товари або предмети, які існують лише у віртуальному або онлайн-середовищі. Ці активи можуть приймати різні форми, включаючи внутрішньо - ігрові предмети, скіни, віртуальну валюту, цифрові предмети колекціонування тощо. Ось деякі з найпоширеніших типів віртуальних активів[1]:

1. **Внутрішньо-ігрові предмети:** це віртуальні предмети, які можна отримати або заробити під час гри, наприклад зброя, броня та інше обладнання.
2. **Скіни:** скіни — це віртуальні предмети, які змінюють зовнішній вигляд предметів або персонажів у грі. Вони часто є суто косметичними, але також можуть мати практичні переваги, такі як покращення ігрового процесу.
3. **Віртуальна валюта:** це тип віртуального активу, який можна використовувати для покупки інших віртуальних предметів або послуг у грі чи віртуальному світі. Приклади включають золото World of Warcraft, долари Linden від Second Life і V-Bucks від Fortnite.
4. **Цифрові предмети колекціонування**: це унікальні віртуальні предмети, які часто створюються художниками чи дизайнерами та продаються обмеженим тиражем. Невзаємозамінні токени (NFT) — це тип цифрових предметів колекціонування, які останніми роками стають все більш популярними.
5. **Віртуальна нерухомість:** це віртуальна земля або власність у грі чи віртуальному світі. Гравці можуть володіти цією власністю та розвивати її, а в деяких випадках навіть можуть заробляти реальні гроші, продаючи її іншим гравцям.
6. **Віртуальні домашні тварини:** це віртуальні тварини або істоти, якими можна володіти та про яких можна піклуватися в грі чи віртуальному світі. Вони можуть бути суто косметичними або можуть мати практичні переваги, наприклад надання бонусів до ігрового процесу.

Загалом віртуальні активи можуть набувати різних форм і стали важливою частиною онлайн-культури, пропонуючи користувачам спосіб покращити свій досвід роботи в Інтернеті та виразити себе новими й унікальними способами.

1.2 Концепція торгівлі віртуальними активами

Торгівля віртуальними активами означає купівлю та продаж цифрових товарів або предметів, які існують лише у віртуальному або онлайн-середовищі.[2] Ці активи можуть приймати різні форми, включаючи внутрішньоігрові предмети, скіни, віртуальну валюту, цифрові предмети колекціонування тощо.

Концепція торгівлі віртуальними активами з'явилася в результаті зростання популярності онлайн-ігор та інших віртуальних світів. Гравці часто прагнуть персоналізувати свій онлайн-досвід, отримуючи рідкісні або унікальні предмети, а торгівля віртуальними активами надає їм можливість це зробити.

Вартість віртуальних активів визначається попитом і пропозицією. Рідкісні чи унікальні предмети, як правило, цінніші за звичайні, і їх вартість може коливатися залежно від низки факторів, зокрема оновлень гри, змін у базі гравців і змін у ширшій ігровій індустрії.

Торгувати віртуальними активами можна за допомогою різних каналів, включаючи онлайн-ринки та форуми, соціальні мережі та однорангові торгові платформи. Багато ігор і віртуальних світів також мають вбудовані системи торгівлі, які дозволяють гравцям торгувати віртуальними активами в самій грі.[2]

Однак торгівля віртуальними активами може бути ризикованою, оскільки часто існує мало регулювання чи контролю. Шахрайство та шахрайство є поширеним явищем у світі торгівлі віртуальними активами, і може бути важко перевірити справжність і вартість віртуальних активів. Крім того, вартість віртуальних активів може бути дуже мінливою, і трейдери повинні знати про пов’язані з цим ризики, перш ніж здійснювати будь-які операції.

Загалом торгівля віртуальними активами стала важливим аспектом онлайн-ігор і віртуальних світів, пропонуючи гравцям спосіб персоналізувати свій онлайн-досвід і потенційно заробляти гроші на торгівлі. Однак трейдери повинні усвідомлювати пов’язані з цим ризики та вживати заходів, щоб захистити себе від шахрайства та шахрайства.

1.3 Огляд мульти ігрової торгівлі

Торгівля кількома іграми означає купівлю та продаж віртуальних активів у кількох іграх або віртуальних світах.[3] Ця практика стає все більш популярною, оскільки геймери прагнуть розширити свою колекцію віртуальних предметів і скористатися можливостями торгівлі на різних платформах.

Однією з головних переваг торгівлі кількома іграми є можливість диверсифікувати свій портфель віртуальних активів. Торгуючи в кількох іграх, гравці можуть зменшити свій ризик, розподіляючи свої інвестиції на більш широкий діапазон активів. Це також може надати можливості для арбітражу, коли гравці можуть купити актив в одній грі за нижчою ціною та продати його за вищою ціною в іншій грі.

Торгівля в кількох іграх може здійснюватися за допомогою різних каналів, включаючи онлайн-ринки та форуми, соціальні мережі та однорангові торгові платформи. Багато ігор і віртуальних світів також мають вбудовані системи торгівлі, які дозволяють гравцям торгувати віртуальними активами в самій грі.

Однак торгівля кількома іграми також може бути складнішою, ніж торгівля в одній грі чи віртуальному світі. Кожна гра чи віртуальний світ може мати власні правила й положення щодо торгівлі, а також свою унікальну віртуальну економіку.[3] Трейдери повинні знати про ці відмінності та враховувати їх під час угод на декількох платформах.

Крім того, торгівля декількома іграми може бути ризикованою, ніж торгівля в одній грі, оскільки часто доступної інформації про вартість і автентичність віртуальних активів у різних іграх менше. Трейдери повинні бути старанними у своїх дослідженнях і належній обачності, щоб переконатися, що вони приймають обґрунтовані торгові рішення.

Загалом торгівля кількома іграми пропонує можливості для диверсифікації та потенційного арбітражу, але трейдери повинні знати про пов'язані з цим ризики та складності. Із зростанням онлайн-ігор і віртуальних світів торгівля кількома іграми, ймовірно, стане все більш важливим аспектом торгівлі віртуальними активами в майбутньому.

1.4 Огляд Dmarket

DMarket — це онлайн-ринок, який дозволяє гравцям торгувати віртуальними активами, такими як внутрішньоігрові предмети, скіни та інші цифрові товари.[4] Платформа була запущена в 2017 році і завоювала популярність серед ігрової спільноти завдяки зручному інтерфейсу та широкому вибору віртуальних активів.

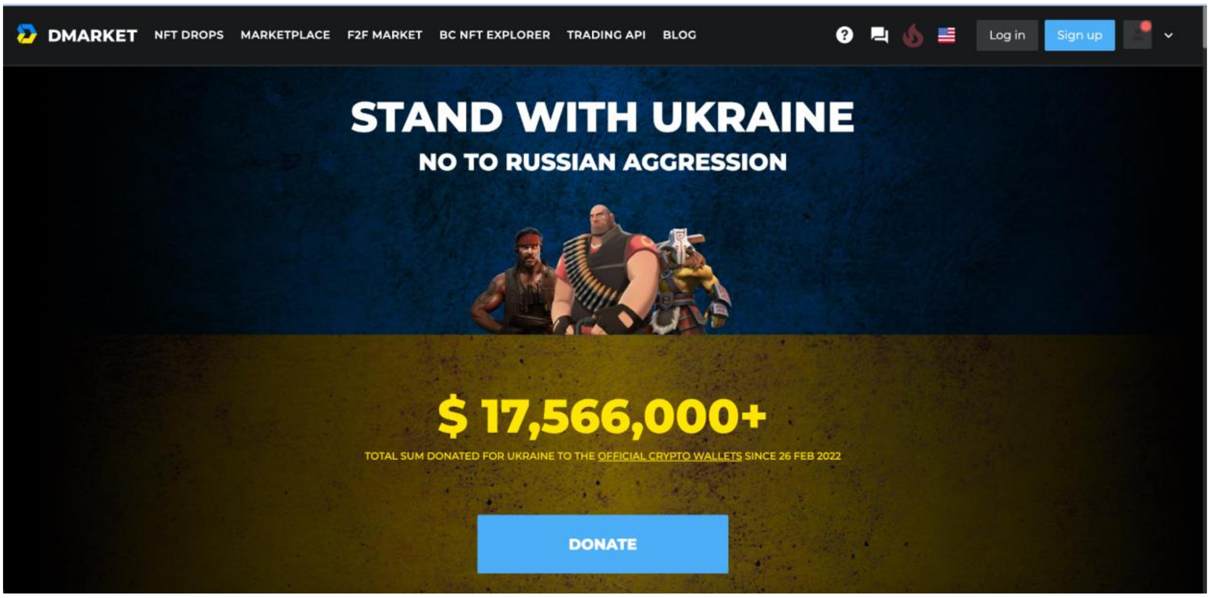


Рисунок 1.1 Головна сторінка Dmarket

Однією з унікальних особливостей DMarket є його технологія на основі блокчейну, яка забезпечує безпечні та прозорі транзакції між покупцями та продавцями. Ця технологія дозволяє здійснювати миттєві операції та усуває потребу в посередниках, роблячи платформу більш ефективною та економічно вигідною.

DMarket також пропонує різноманітні варіанти оплати, включаючи кредитну картку, криптовалюту та токени DMarket, які можна використовувати для придбання віртуальних активів за зниженою ставкою. Платформа також надає комплексну систему управління запасами, що дозволяє користувачам легко відстежувати свої віртуальні активи та відстежувати їхню вартість з часом.

1.4.1 Функціонал Dmarket

DMarket пропонує низку функціональних можливостей як для покупців, так і для продавців, зокрема[4]:

- **Торгівля віртуальними активами:** DMarket дозволяє користувачам купувати та продавати віртуальні активи, такі як внутрішньоігрові предмети, скіни та інші цифрові товари. Платформа пропонує широкий вибір віртуальних активів із популярних ігор, включаючи CS:GO, Dota 2 і Team Fortress 2.
- **Безпечні транзакції:** DMarket використовує технологію на основі блокчейну для забезпечення безпечних і прозорих транзакцій між покупцями та продавцями. Ця технологія усуває потребу в посередниках, роблячи платформу більш ефективною та економічно вигідною.
- **Кілька варіантів оплати:** DMarket пропонує різноманітні варіанти оплати, включаючи кредитну картку, криптовалюту та токени DMarket, які можна використовувати для придбання віртуальних активів за зниженою ставкою.
- **Управління запасами:** DMarket надає комплексну систему управління запасами, що дозволяє користувачам легко відстежувати свої віртуальні активи та відстежувати їх вартість з часом.

- **Ігрова аналітика:** DMarket пропонує інструменти ігрової аналітики, які можуть допомогти розробникам оптимізувати свої ігри та покращити залучення гравців. Платформа надає детальні дані про поведінку гравців, включаючи купівельні звички та ігрову активність.
- **Управління даними:** DMarket також надає послуги керування даними, дозволяючи розробникам безпечно зберігати та керувати даними гравців.
- **Залучення гравців:** DMarket пропонує низку інструментів залучення гравців, включаючи обмін повідомленнями між гравцями та соціальні функції, щоб допомогти розробникам створити та підтримувати базу лояльних гравців.

Загалом, DMarket пропонує ряд функціональних функцій, які роблять його надійною та зручною платформою для геймерів, щоб купувати, продавати та торгувати віртуальними активами. Його технологія на основі блокчейну, широкий вибір варіантів оплати та комплексна система управління запасами роблять його популярним вибором серед ігрової спільноти. Крім того, інструменти платформи для аналізу ігор, управління даними та залучення гравців можуть допомогти розробникам оптимізувати свої ігри та покращити утримання гравців.

На додаток до свого ринку, DMarket пропонує ряд послуг, таких як аналітика ігор, керування даними та інструменти залучення гравців, які можуть допомогти розробникам оптимізувати свої ігри та покращити утримання гравців.

1.4.2 Торгівля скінами на Dmarket

Торгівля скінами на DMarket є простим процесом. Користувачі можуть переглядати велику колекцію скінів платформи з популярних ігор, таких як CS:GO, Dota 2 і Team Fortress 2.[4] Потім вони можуть вибрати скіни, які бажають придбати, і додати їх у свій кошик.

Продавати скіни на DMarket також легко. Користувачі можуть виставляти свої скіни на продаж на платформі та встановлювати бажану ціну. Торговий майданчик DMarket використовує технологію на основі блокчейну для забезпечення безпечних і прозорих транзакцій між покупцями та продавцями, що робить процес продажу безпечним і ефективним.

Однією з переваг торгівлі скінами на DMarket є комплексна система управління запасами платформи. Користувачі можуть легко відстежувати свої скіни та відстежувати їх вартість з часом. Це дозволяє їм приймати зважені рішення під час купівлі та продажу скінів, а також оптимізувати свої торгові стратегії.

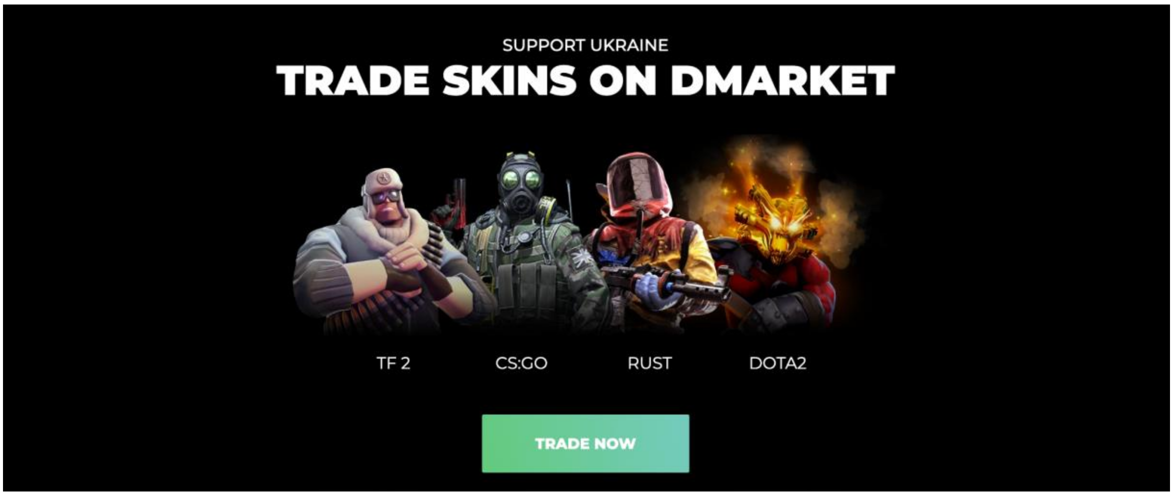


Рисунок 1.2 Торгівля скінами

1.4.3 Переваги та недоліки DMarket

DMarket — популярний онлайн-ринок для купівлі та продажу віртуальних активів, таких як внутрішньо-ігрові предмети, скіни та інші цифрові товари. Як і будь-яка інша платформа, DMarket має свої переваги та недоліки, які обговорюються нижче:

Переваги:

1. *Безпечна та прозора торгівля:* DMarket використовує технологію на основі блокчейну, щоб забезпечити безпечні та прозорі транзакції між покупцями та продавцями. Ця технологія усуває потребу в

посередниках, роблячи платформу більш ефективною та економічно вигідною.

2. *Широкий вибір віртуальних активів:* DMarket пропонує широкий вибір віртуальних активів із популярних ігор, зокрема CS:GO, Dota 2 і Team Fortress 2. Завдяки цьому гравцям буде легко знайти скіни та предмети, які вони шукають.
3. *Кілька варіантів оплати:* DMarket пропонує різноманітні варіанти оплати, включаючи кредитну картку, криптовалюту та токени DMarket, які можна використовувати для придбання віртуальних активів за зниженою ставкою.
4. *Комплексне управління запасами:* DMarket надає комплексну систему управління запасами, що дозволяє користувачам легко відстежувати свої віртуальні активи та відстежувати їх вартість з часом.
5. *Ігрова аналітика та керування даними:* DMarket пропонує інструменти для аналітики ігор та керування даними, які можуть допомогти розробникам оптимізувати свої ігри та покращити залучення гравців. Це може бути корисним для геймерів, оскільки це може призвести до кращого ігрового досвіду в цілому.
6. *Залучення гравців:* DMarket пропонує низку інструментів залучення гравців, включаючи обмін повідомленнями між гравцями та соціальні функції, щоб допомогти розробникам створити та підтримувати базу лояльних гравців.

Недоліки:

1. *Обмежений вибір ігор:* хоча DMarket пропонує широкий вибір віртуальних активів, вибір ігор на платформі дещо обмежений порівняно з іншими ринками.
2. *Високі комісії:* DMarket стягує комісію за транзакцію за кожен продаж, яка може бути вищою, ніж на інших ринках.
3. *Обмежена регіональна доступність:* DMarket доступний не в усіх регіонах, що може ускладнити доступ деяких геймерів до платформи.

4. *Волатильність цін:* вартість віртуальних активів на DMarket може сильно коливатися, що може ускладнити для гравців прогнозування вартості своїх активів з часом.

Загалом DMarket пропонує геймерам безпечну та надійну платформу для купівлі та продажу віртуальних активів. Його широкий вибір віртуальних активів, кілька варіантів оплати та комплексна система управління запасами роблять його популярним вибором серед ігрової спільноти. Однак обмежений вибір ігор платформи, високі комісії та обмежена регіональна доступність можуть бути недоліками для деяких користувачів.

1.5 Огляд конкурентної платформи Bitskins

BitSkins — це цифровий ринок, який дозволяє користувачам купувати та продавати внутрішньоігрові предмети для різноманітних популярних відеоігор, таких як CS:GO, Dota 2, Rust і Team Fortress 2. [5] Платформу було засновано в 2015 році і з тих пір стала популярне місце для геймерів, які хочуть торгувати цифровими предметами.

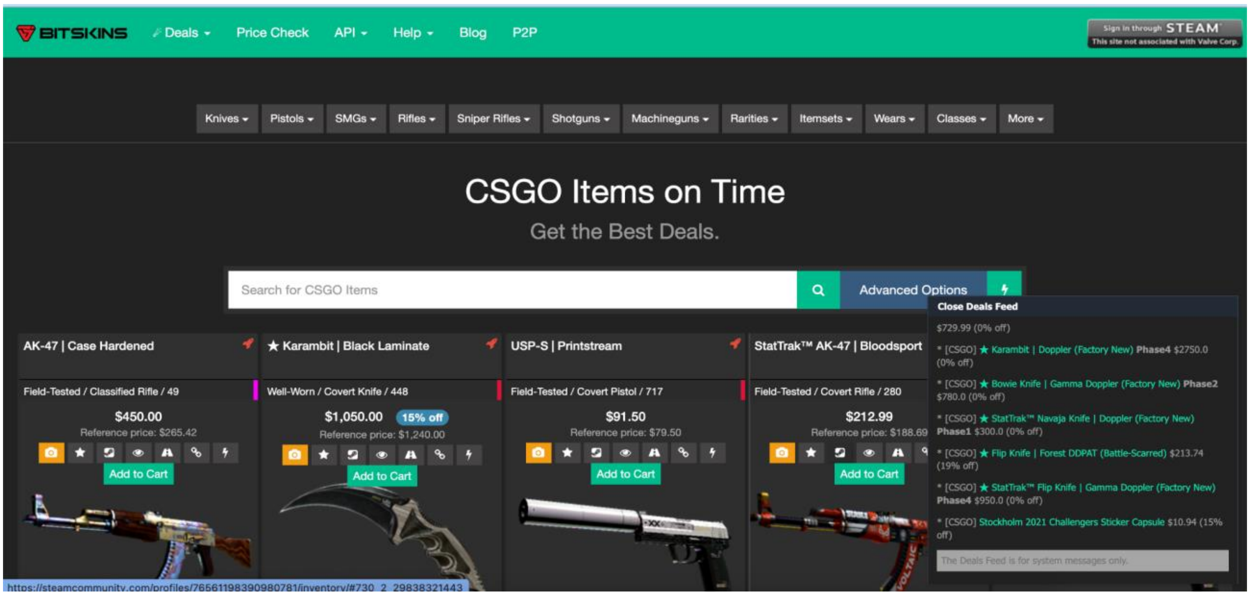


Рисунок 1.3 Огляд платформи Bitskins

На додаток до свого ринку BitSkins також пропонує конкурентну платформу для гравців CS:GO. [5] Ця платформа дозволяє гравцям змагатися один з одним у матчах на основі навичок за реальні гроші. Матчі проводяться за системою FACEIT, яка забезпечує чесну гру та точний підбір матчів.

Щоб взяти участь у конкурсній платформі BitSkins, користувачі повинні спочатку створити обліковий запис і внести кошти на свій гаманець BitSkins. Потім вони можуть вибрати гру, у яку хочуть грати, і вибрати матч на основі свого рівня навичок і бажаних ставок. BitSkins стягує невелику плату за кожен зіграний матч, яка вираховується з виграшу гравця.

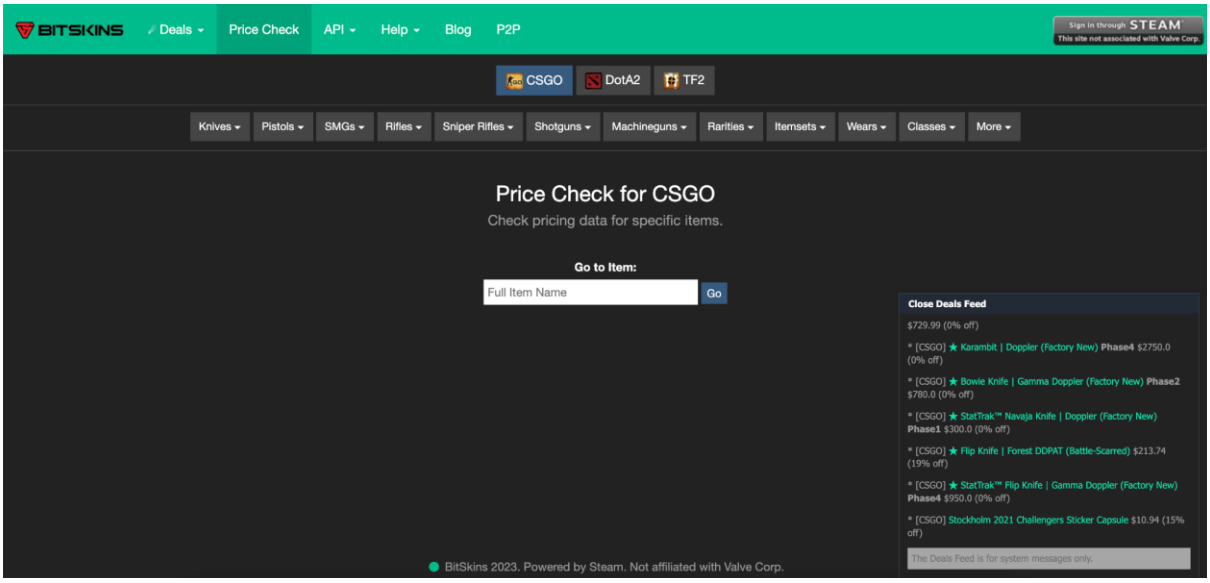


Рисунок 1.4 Перевірка ціни

Можна перевірити поточні ціни на певні предмети. Ціни на ігрові предмети можуть швидко коливатися залежно від попиту та пропозиції, і на них впливають різні фактори, як-от рідкість, стан і популярність. Крім того, ціни можуть відрізнятися в різних ринках і регіонах. Перш ніж купувати або продавати, важливо провести дослідження та порівняти ціни, а функція перевірки актуальної ціни на платформі BitSkins допомагає з цим.

1.5.1 Переваги і недоліки

Переваги BitSkins:

1. Широкий вибір ігор: BitSkins підтримує низку популярних відеоігор, включаючи CS:GO, Dota 2, Rust і Team Fortress 2, надаючи користувачам безліч варіантів вибору.

2. Конкурентна платформа: BitSkins пропонує конкурентоспроможну платформу для гравців CS:GO, що дозволяє їм грати в матчі на основі навичок за реальні гроші.
3. Проста у використанні: платформа BitSkins зручна та інтуїтивно зрозуміла, що дозволяє користувачам легко купувати та продавати цифрові товари.
4. Безпека: BitSkins серйозно ставиться до безпеки та використовує такі заходи, як двофакторна автентифікація, щоб захистити облікові записи користувачів і транзакції.
5. Низькі комісії: BitSkins стягує нижчі комісії порівняно з іншими торговими майданчиками, що означає, що користувачі можуть зберегти більшу частину своїх прибутків.

Недоліки BitSkins:

1. Обмежені варіанти оплати: BitSkins приймає платежі лише за допомогою певних методів оплати, що може бути незручним для деяких користувачів.
2. Коливання цін. Ціни на ігрові предмети можуть швидко коливатися залежно від попиту та пропозиції, що ускладнює користувачам передбачити вартість своїх предметів.
3. Немає відшкодування: BitSkins не пропонує відшкодування за транзакції, а це означає, що користувачі повинні бути впевнені у своїх покупках, перш ніж їх робити.
4. Ризик шахрайства: як і на будь-якому онлайн-ринку, існує ризик шахрайства та шахрайства, тому користувачі повинні бути обережними, купуючи чи продаючи цифрові товари.
5. Регіональні обмеження: BitSkins може бути недоступним у певних регіонах або країнах, що обмежує доступ для деяких користувачів.

Також в порівнянні з Dmarket, на BitSkins більше комісія та він менш популярний. Варто зазначити, що боти конкурентів не розглядались, тому що вони не в вільному доступі, недоцільно показувати свою робочу стратегію роботи бота.

Загалом платформа BitSkins надає гравцям зручний і безпечний спосіб купувати та продавати цифрові предмети, а також змагатися один з одним у матчах на основі навичок.

ВИСНОВОК ДО РОЗДІЛУ 1

Підсумовуючи, віртуальні активи – це цифрові елементи, які існують лише в онлайн-середовищах, таких як відеоігри, соціальні мережі та віртуальні світи. Вони бувають у багатьох формах, включаючи ігрові предмети, скіни, віртуальну валюту, цифрові предмети колекціонування, віртуальну нерухомість і віртуальних домашніх тварин.

Торгівля віртуальними активами стала популярною практикою, і багато гравців прагнуть розширити свій портфель віртуальних активів і скористатися можливостями торгівлі в різних іграх і на платформах. Торгівля з кількома іграми пропонує можливості для диверсифікації та потенційного арбітражу, але трейдери повинні знати про пов'язані з цим ризики та складності.

Оскільки онлайн-ігри та віртуальні світи продовжують розвиватися, віртуальні активи, ймовірно, стануть все більш важливим аспектом онлайн-культури та комерції.

Порівнюючи Dmarket та BitSkins, ми надали перевагу першому, адже він більш популярний та має невисоку комісію. Боти конкурентів не розглядались, бо їх немає у вільному доступі.

РОЗДІЛ 2. ОГЛЯД ТА АНАЛІЗ ІНСТРУМЕНТІВ ДЛЯ СТВОРЕННЯ БОТА ДЛЯ АВТОМАТИЧНОЇ ТОРГІВЛІ НА DMARKET

2.1 Характеристика та аналіз обраного проекту

Характеристики вибраного проекту, який передбачає розробку бота для автоматичної торгівлі на DMarket, розширено за рахунок кількох нових можливостей. До них належать:

1. **Торгівля кількома іграми:** бот буде розроблено для підтримки всіх ігор, доступних на DMarket, розширюючи його потенційне охоплення та збільшуючи різноманітність скінів і предметів, якими можна торгувати.
2. **Автоматичний аналіз бази даних скінів:** бот зможе автоматично аналізувати базу даних скінів для кожної гри, що дозволить йому бути в курсі останніх тенденцій і цін на кожен предмет.
3. **Розміщення замовлень на основі декількох параметрів:** бот зможе розміщувати замовлення на основі 15 різних параметрів, що дозволить йому приймати більш обґрунтовані та стратегічні торгові рішення. Він також може розміщувати замовлення, щоб боротися за перше місце, підвищуючи свої шанси на здійснення прибуткової торгівлі.
4. **Автоматичне відображення скінів для продажу:** Бот зможе автоматично відображати скіни для продажу після покупки, коригуючи ціни відповідно до своїх налаштувань і борючись за перше місце, щоб максимізувати прибуток.

Розширені можливості цього проекту надають кілька потенційних областей для аналізу, зокрема:

1. **Обсяг торгів:** Завдяки можливості торгувати кількома іграми та аналізувати базу даних скінів для кожної гри, бот має потенціал для створення великого обсягу торгів. Аналіз обсягу торгів за певний час може дати розуміння ефективності стратегій і алгоритмів бота.

2. **Прибутковість за іграми:** з підтримкою багато ігрової торгівлі може бути корисним аналізувати прибутковість бота для кожної гри окремо. Це може допомогти визначити, які ігри є найприбутковішими, а де стратегії бота, можливо, потрібно буде скоригувати.
3. **Ефективність параметрів:** маючи можливість розміщувати замовлення на основі кількох параметрів, буде важливо проаналізувати, які параметри є найбільш ефективними для отримання прибутку. Це може допомогти оптимізувати алгоритми бота та покращити його продуктивність з часом.
4. **Ефективність коригування ціни:** здатність бота автоматично коригувати ціни на основі своїх налаштувань і боротися за перше місце можна проаналізувати, щоб визначити його ефективність у максимізації прибутку. Це може допомогти визначити можливості для подальшої оптимізації та вдосконалення.

Загалом, розширені можливості проекту мають потенціал для підвищення його ефективності та прибутковості. Буде необхідний постійний моніторинг і аналіз, щоб переконатися, що бот працює оптимально та з часом здійснює прибуткові операції.

2.1.1 Функціональні вимоги

Функціональні вимоги — це функції та можливості, якими повинен володіти продукт або система, щоб задовольнити потреби користувачів. Вони описують, що повинен робити продукт або система і як він повинен виконувати певні завдання. [6]

Таблиця 2.1 Функціональні випоги

Вимога	Опис
FR1	Бот повинен мати можливість торгувати всіма іграми, доступними на DMarket.
FR2	Бот повинен мати можливість автоматично аналізувати базу даних скінів для кожної гри.
FR3	Бот повинен мати можливість розміщувати замовлення на основі 15 різних параметрів.
FR4	Бот повинен мати можливість розміщувати замовлення, щоб боротися за перше місце.
FR5	Бот повинен мати можливість автоматично відображати скіни для продажу після покупки.
FR6	Бот повинен мати можливість регулювати ціни відповідно до налаштувань.
FR7	Бот повинен мати можливість боротися за перше місце, щоб максимізувати прибуток.
FR8	Бот повинен мати можливість створювати звіти про торгову діяльність.

2.1.2 Нефункціональні вимоги

Нефункціональні вимоги описують, наскільки добре продукт або система повинні виконувати ці завдання. Вони визначають атрибути якості, якими має володіти продукт або система, такі як продуктивність, надійність, зручність використання та безпека.[6]

Таблиця 2.2 Нефукціональні вимоги

Вимога	Опис
NFR1	Продуктивність: бот повинен мати можливість обробляти великий обсяг торгової діяльності без значних затримок або простоїв.
NFR2	Надійність: бот повинен працювати надійно, тобто він повинен вміти обробляти помилки адекватно - ті ж самі помилки мережі,тощо.
NFR3	Безпека: бот повинен бути розроблений із відповідними заходами безпеки, щоб запобігти будь-якому несанкціонованому доступу або витоку даних.
NFR4	Зручність використання: бот має бути зручним та інтуїтивно зрозумілим у використанні.
NFR5	Масштабованість: бот повинен мати можливість масштабування вгору або вниз у міру збільшення або зменшення обсягу торгової діяльності.
NFR6	Ремонтопридатність: бот має бути розроблений таким чином, щоб його було легко підтримувати, з чітким і лаконічним кодом і документацією.
NFR7	Сумісність: бот повинен бути сумісний з усіма останніми версіями необхідного програмного забезпечення та операційних систем.
NFR8	Доступність: бот має бути доступний 24/7, щоб гарантувати, що торгову діяльність можна відстежувати та коригувати в будь-який час.

2.2 Вибір та аналіз доступних технологій та програмного забезпечення

2.2.1 Обрана мова програмування

Python — це мова програмування високого рівня, яка була вперше випущена в 1991 році. Її було розроблено так, щоб її було легко читати та писати, і вона здобула популярність завдяки своїй простоті, гнучкості та широкому спектру доступних бібліотек і фреймворків.[7]

Плюси:

- Легко освоїти: Python має простий і зрозумілий синтаксис, який полегшує навчання початківцям.
- Велика та активна спільнота: у Python є велика та активна спільнота розробників, які роблять внесок у її бібліотеки та фреймворки, що полегшує пошук ресурсів і отримання допомоги.
- Сумісність між платформами: код Python можна запускати на кількох платформах, включаючи Windows, macOS і Linux.
- Широка підтримка бібліотек: Python має широкий спектр бібліотек і фреймворків, які пропонують рішення майже для будь-яких завдань програмування, включаючи веб-розробку, аналіз даних, машинне навчання тощо.

Мінуси:

- Низька продуктивність: Python є інтерпретованою мовою, що означає, що він може бути повільнішим, ніж скомпільовані мови, такі як C++ або Java.
- Глобальне блокування інтерпретатора (GIL): GIL — це механізм у Python, який запобігає одночасному виконанню байт-кодів Python декількома потоками, що може вплинути на продуктивність у багатопоточних програмах.
- Слабкий у розробці мобільних пристроїв та ігор: Python не є найкращою мовою для розробки мобільних додатків або розробки ігор через його

нижчу продуктивність і відсутність вбудованої підтримки для мобільних платформ.

Переваги Python для створення ботів:

- Легкий у вивченні та використанні: Python має простий і легкий у вивченні синтаксис, що робить його популярним вибором для новачків.
- Підтримка великої бібліотеки: Python має широкий спектр бібліотек і фреймворків, які спрощують створення ботів для різноманітних завдань, таких як веб-скрапінг, чат-боти та автоматизація.
- Сумісність між платформами: код Python можна запускати на кількох платформах, що полегшує створення ботів, які працюють на різних операційних системах.
- Активна спільнота: Python має велике та активне співтовариство розробників, які вносять свій внесок у його бібліотеки та фреймворки, що полегшує пошук ресурсів і допомогу під час створення ботів.

2.2.2 Вибір інструменту для розробки бази даних

Інструменти SQL (мова структурованих запитів) — це програмні додатки, які дозволяють користувачам взаємодіяти з реляційними базами даних за допомогою команд SQL. Існує багато інструментів SQL, кожен з яких має свої функції та можливості. У цій відповіді я опишу та порівняю чотири популярні інструменти SQL: MySQL Workbench, Microsoft SQL Server Management Studio, PostgreSQL і SQLite.

1. **MySQL Workbench:** MySQL Workbench — це потужний інструмент SQL, який надає візуальний інтерфейс для проектування, розробки та керування базами даних MySQL. Він містить такі функції, як конструктор схем бази даних, редактор коду SQL, конструктор запитів та інструменти адміністрування сервера. MySQL Workbench — це безкоштовне програмне забезпечення з відкритим кодом, яке сумісне з Windows, macOS і Linux.[8]

2. **Microsoft SQL Server Management Studio (SSMS):** SSMS — це інструмент SQL, розроблений Microsoft для керування базами даних SQL Server. Він забезпечує графічний інтерфейс для керування та розробки баз даних SQL Server і включає такі функції, як редактор запитів, дизайнер баз даних, інструменти адміністрування сервера та інструменти моніторингу продуктивності. SSMS доступний лише для Windows, його можна безкоштовно завантажити та використовувати.[8]
3. **PostgreSQL:** PostgreSQL — це потужний інструмент SQL із відкритим кодом, який надає розширені функції та можливості для керування реляційними базами даних. Він включає такі функції, як повнотекстовий пошук, підтримку ГІС і підтримку типів даних JSON і XML. PostgreSQL сумісний із Windows, macOS і Linux і доступний безкоштовно за ліцензією PostgreSQL.[9]
4. **SQLite:** SQLite — це легкий інструмент SQL, який забезпечує простий і швидкий спосіб керування базами даних малого та середнього розміру. Він вбудований у програми та не потребує окремого серверного процесу, що полегшує його розгортання та керування. SQLite сумісний із Windows, macOS і Linux і доступний безкоштовно за ліцензією Public Domain License.[10]

З точки зору порівняння, кожен інструмент SQL має свої сильні та слабкі сторони, залежно від потреб і вимог користувача. MySQL Workbench і Microsoft SSMS — це потужні та багатофункціональні інструменти SQL, які добре підходять для керування великими та складними базами даних. PostgreSQL пропонує розширені функції та можливості для керування даними, тоді як SQLite є легким інструментом, який ідеально підходить для керування невеликими базами даних.

Загалом, обираючи інструмент SQL, важливо враховувати такі фактори, як розмір і складність бази даних, необхідні функції та можливості, а також операційну систему та платформу, що використовуються.

Таблиця 2.3 Порівняльна таблиця інструментів SQL

Інструмент SQL	**Особливості та можливості**	**Сумісність платформи**	**Ліцензія**
MySQL Workbench	Візуальний конструктор схем бази даних, редактор коду SQL, конструктор запитів, засоби адміністрування сервера	Windows, macOS, Linux	Безкоштовний і з відкритим кодом
Microsoft SQL Server Management Studio	Редактор запитів, дизайнер баз даних, інструменти адміністрування сервера, інструменти моніторингу продуктивності	вікна	безкоштовно
PostgreSQL	Розширені функції для керування реляційними базами даних, повнотекстовий пошук, підтримка ГІС, підтримка типів даних JSON і XML	Windows, macOS, Linux	Безкоштовний і з відкритим кодом
SQLite	Легкий, простий і швидкий спосіб керування базами даних малого та середнього розміру	Windows, macOS, Linux	Ліцензія суспільного надбання

З точки зору сумісності з платформами, MySQL Workbench, PostgreSQL і SQLite сумісні з Windows, mac OS і Linux, тоді як Microsoft SQL Server Management Studio доступний лише для Windows.

З точки зору функцій і можливостей, MySQL Workbench і Microsoft SQL Server Management Studio — це потужні та багатофункціональні інструменти SQL, які надають ряд інструментів для керування та розробки баз даних. PostgreSQL пропонує розширені функції та можливості для керування даними, такі як повнотекстовий пошук, підтримка ГІС і підтримка типів даних JSON і

XML. SQLite, з іншого боку, є легким інструментом, який ідеально підходить для керування невеликими базами даних.

З точки зору ліцензування, MySQL Workbench, PostgreSQL і SQLite доступні безкоштовно та з відкритим кодом, тоді як Microsoft SQL Server Management Studio є безкоштовним для використання, але не з відкритим кодом.

2.3 Специфікація API

API означає інтерфейс прикладного програмування. Це набір протоколів, процедур і інструментів для створення програмних додатків, які дозволяють різним компонентам програмного забезпечення взаємодіяти один з одним.[11] Простіше кажучи, API — це спосіб для різних програмних програм спілкуватися та обмінюватися даними один з одним.

API дозволяють розробникам створювати потужні програми, які можна інтегрувати з іншими програмними компонентами чи службами. Наприклад, платформи соціальних медіа, такі як Facebook і Twitter, надають API, які дозволяють розробникам створювати програми, які можуть отримувати доступ до їхніх платформ і взаємодіяти з ними. Це дозволяє розробникам створювати програми, які можуть публікувати оновлення, отримувати дані користувачів і виконувати інші дії від імені користувачів.

Існують різні типи API, зокрема:

1. REST API: Representational State Transfer (REST) API – це популярний тип API, який використовує запити HTTP для взаємодії з веб-службами.[12] REST API розроблені як прості, масштабовані та легкі, і вони використовуються багатьма веб-додатками та службами.
2. SOAP API: API простого протоколу доступу до об’єктів (SOAP) — ще один тип API, який використовує XML для взаємодії з веб-службами. SOAP API складніші за REST API і часто використовуються в корпоративних програмах.[11]

3. API GraphQL: API GraphQL — це новий тип API, який використовує мову запитів для запиту й отримання певних даних із сервера. [11] GraphQL API розроблено, щоб бути більш гнучкими та ефективними, ніж REST API, і вони стають все більш популярними для створення веб-додатків.

API є важливими компонентами для створення сучасних програмних додатків. Вони дозволяють розробникам створювати складні додатки, які можна інтегрувати з різноманітними службами та платформами, а також дозволяють компаніям пропонувати свої послуги та дані іншим розробникам і додаткам, стимулюючи інновації та зростання індустрії програмного забезпечення.

2.3.1 Торговий API DMarket

DMarket API — це RESTful API, який надає розробникам доступ до ринку DMarket на основі блокчейну. Ринок дозволяє користувачам купувати, продавати та торгувати внутрішньоігровими предметами в різних іграх і на платформах.[13]

Документацію для API DMarket організовано в кілька розділів, кожен з яких відповідає окремому набору кінцевих точок API. Ці кінцеві точки дозволяють розробникам взаємодіяти з ринком DMarket різними способами, наприклад отримувати інформацію про користувачів, елементи та транзакції, а також виконувати такі дії, як розміщення замовлень і створення списків.

Документація починається з огляду основних функцій і можливостей API. Він надає інформацію про те, як автентифікувати запити до API за допомогою ключів API та маркерів OAuth 2.0, а також описує різні типи запитів, які можна надсилати до API.

Перший розділ документації присвячений автентифікації. Тут надається детальна інформація про те, як отримати ключ API або маркер OAuth 2.0 і як використовувати ці облікові дані для автентифікації запитів до API.

Наступний розділ документації стосується користувачів. Він надає інформацію про те, як отримати інформацію про користувачів, включаючи інформацію про їхні профілі та баланси на рахунку. Він також містить відомості про те, як оновлювати інформацію про користувача та переказувати кошти між обліковими записами користувачів.

Розділ документації з елементами містить інформацію про те, як отримати інформацію про ігрові предмети, доступні на ринку DMarket, наприклад їх назву, опис і ціну. Він також містить відомості про те, як шукати певні предмети та як отримати інформацію про рідкість і популярність конкретних предметів.

Розділ документації зі списками містить інформацію про те, як створювати, оновлювати та видаляти списки для ігрових предметів на ринку DMarket. Він містить відомості про те, як вказати ціну та кількість елементів у списку, а також як керувати статусом списків.

Розділ замовлень документації містить інформацію про те, як розміщувати, оновлювати та скасовувати замовлення на купівлю та продаж ігрових предметів на ринку DMarket. Він містить відомості про те, як вказати тип замовлення (купівля чи продаж), товар, яким торгують, і ціну товару.

Розділ документації про транзакції містить інформацію про те, як отримати інформацію про транзакції на ринку DMarket, включаючи їхній статус, ціну та учасників. Він також містить відомості про те, як отримати інформацію про історію транзакцій і як фільтрувати транзакції на основі конкретних критеріїв.

Загалом, документація DMarket API містить вичерпний посібник із використання DMarket API. Він містить детальну інформацію про те, як взаємодіяти з ринком DMarket, і надає численні приклади запитів і відповідей API, щоб допомогти розробникам почати використовувати API у власних програмах і службах.

2.3.2 Опис використання API DMarket

Клацніть свій аватар і виберіть опцію Параметри облікового запису (див. рис. 2.1)

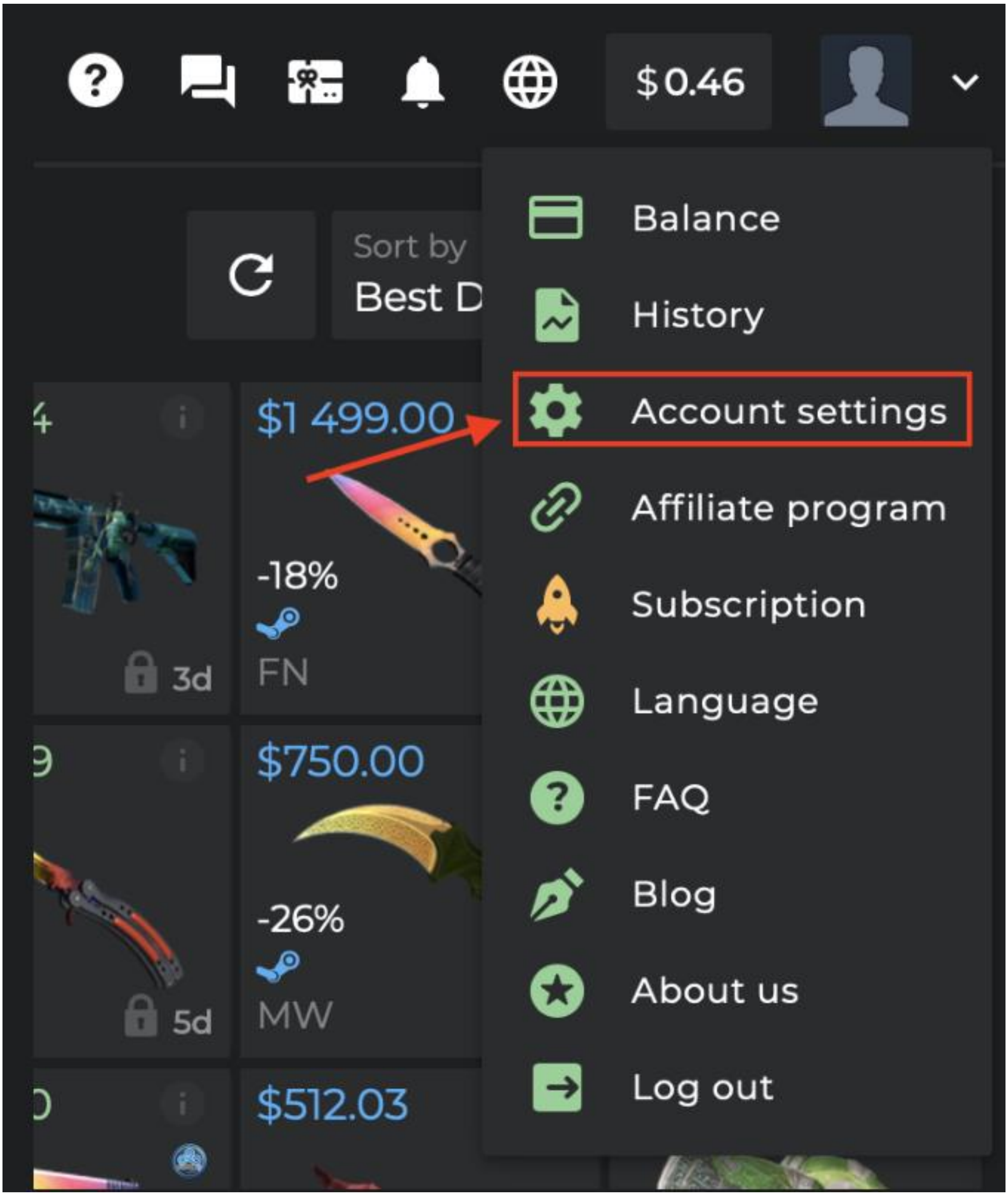


Рисунок 2.1 Налаштування аккаунта

Виберіть розділ «Trading API» і натисніть кнопку «Generate Trading API keys». (див. рис. 2.2)

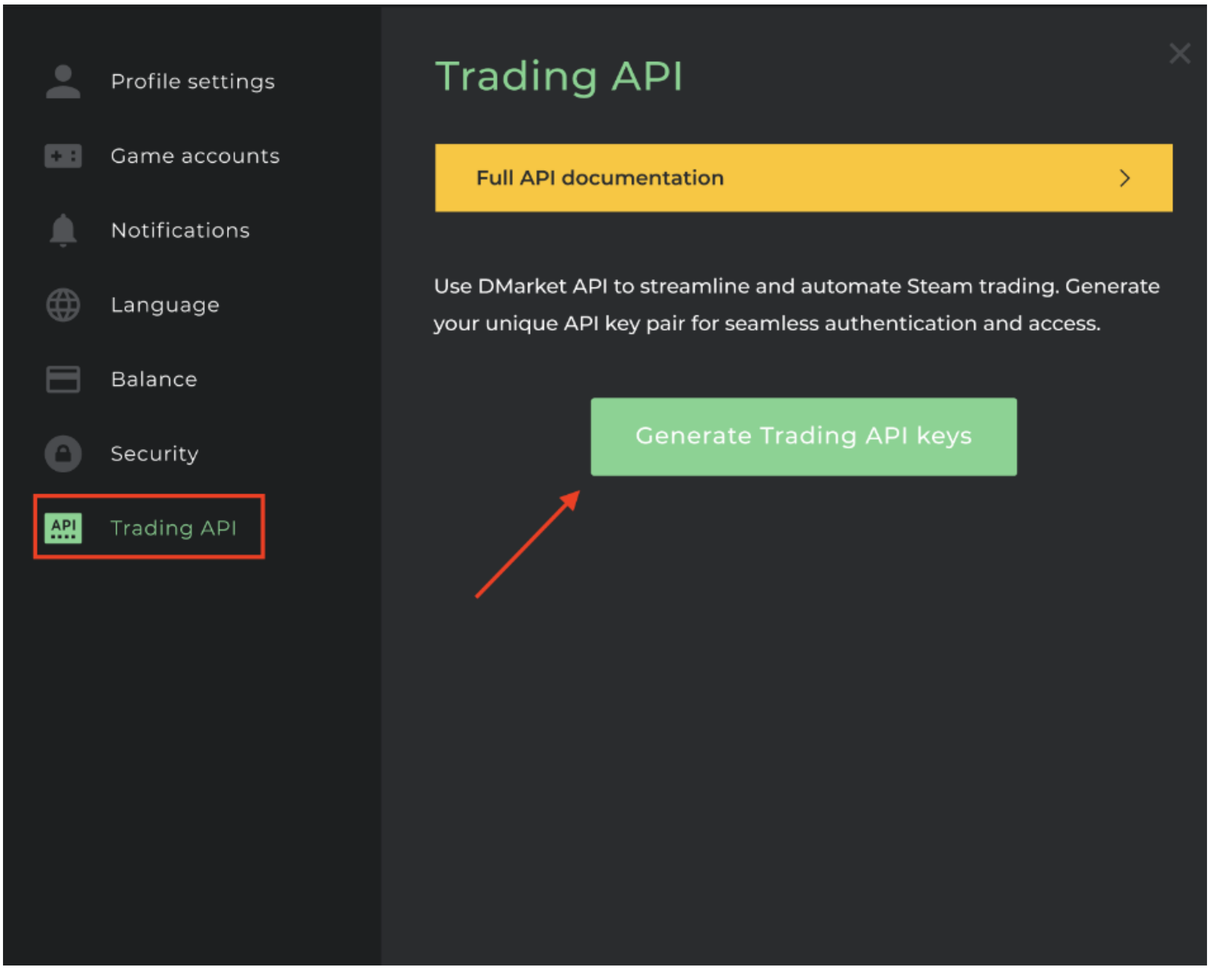


Рисунок 2.2 Генерація API ключів

Після створення ключів API вони з’являються на сторінці.

Зверніть увагу: приватний ключ API відображається лише один раз після його створення. Переконайтеся, що ви зберегли його в безпечному місці, до якого ніхто, крім вас, не має доступу. (див. рис.2.3)

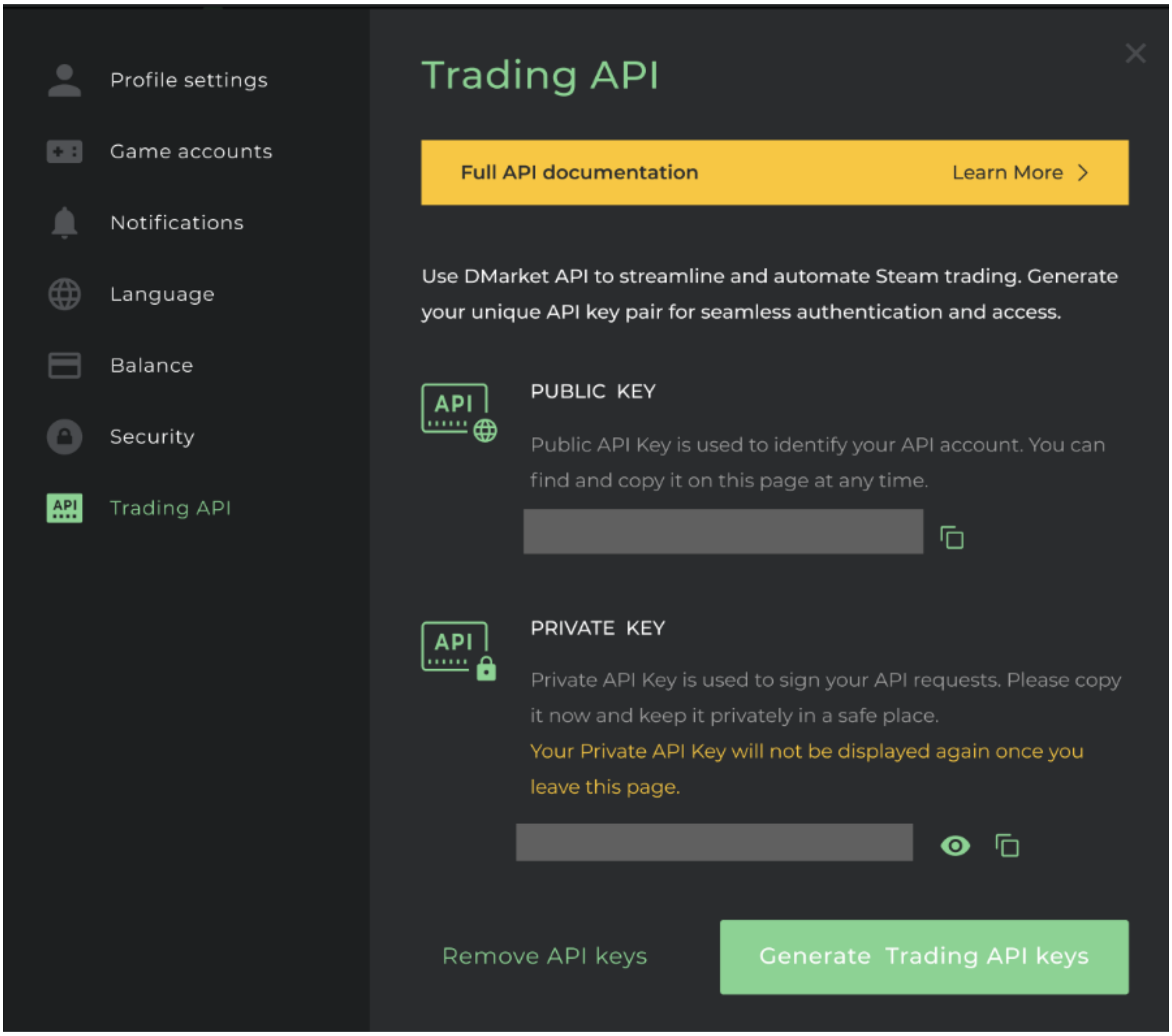


Рисунок 2.3 Отримані API ключі

Після отримання API ключів їх можна використовувати для роботи бота

ВИСНОВОК ДО РОЗДІЛУ 2

Виходячи з характеристик і вимог, наведених для бота для автоматичної торгівлі на DMarket, стає зрозуміло, що це складний проект, який вимагає високого рівня технічної експертизи як у програмуванні, так і в розробці бази даних. Бот повинен мати можливість торгувати в кількох іграх, аналізувати бази даних скінів, розміщувати замовлення на основі багатьох параметрів, динамічно коригувати ціни та створювати звіти про торгову діяльність.

Окрім функціональних вимог, бот також має відповідати декільком нефункціональним вимогам, таким як висока продуктивність, надійність, безпека, зручність використання, масштабованість, ремонтопридатність, сумісність і доступність. Ці вимоги гарантують, що бот не тільки ефективний у торгівлі, але й безпечний, простий у використанні та може підтримуватися та масштабуватися з часом.

Загалом, бот для автоматичної торгівлі на DMarket — це важливе завдання, яке потребує ретельного планування, виконання та постійного обслуговування. Однак, якщо це зробити належним чином, він може стати дуже прибутковим інструментом для торгівлі скінами на DMarket.

РОЗДІЛ 3. РОЗРОБКА СИСТЕМИ

3.1 Опис взаємодії програмних компонентів

Існує декілька умов роботи бота, а саме:

1) Бот працює автоматично;
2) Для запуску бота потрібно налаштувати файл з конфігурацією, вказати приватний та публічний ключі;
3) Уся інформація робота бота має записуватися у логи;
4) Використовується API Dmarket.

3.2 Файл налаштувань «config.py»

У файлі налаштовується конфігурація реєстратора з двома обробниками:

- один, який виводить повідомлення журналу до стандартної помилки з розфарбуванням і рівнем INFO,
- і інший, який записує повідомлення журналу у файл "log/info.log" без серіалізації та рівня INFO

1. Визначаються константи для URL-адреси API та торгівлі, списку ігор і назви бази даних.
2. Визначається список bad_items, які потрібно виключити, і два класи для таймерів і параметрів. BAD_ITEMS - список слів, які зазвичай асоціюються з предметами, які не бажано чи вигідно купувати чи продавати на ігровому ринку. Ці товари, ймовірно, матимуть низький попит, низьку вартість або високу пропозицію, що може призвести до труднощів із їх продажем або низької норми прибутку.
3. Визначаються параметри купівлі та продажу предметів на ігровому ринку, включаючи баланс стоп-замовлень, ціновий діапазон, відсоток прибутку, кількість продажів тощо.

Опис змінних

- "sink": sys.stderr - виведення логів у консоль
- "sink": "log/info.log" - виведення логів у файл
- 'level': INFO – це рівень логів. Можливі рівні: TRACE, DEBUG, INFO, SUCCESS, WARNING, ERROR, CRITICAL. Кожен наступний зліва направо рівень забороняє виведення логів нижчого рівня. Тобто, якщо вказаний рівень INFO, повідомлення з рівнем TRACE, DEBUG виводитися не будуть.
- GAMES = [Games.CS, Games.DOTA, Games.RUST] - список ігор, за якими проводитиметься торгівля.
- Доступні значення: Games.CS, Games.DOTA, Games.RUST, Games.TF2
- PREV_BASE = 60*60*4 - оновлення бази скінів кожні PREV_BASE секунд
- ORDERS_BASE = 60*10 - оновлення бази ордерів кожні ORDERS_BASE секунд
- BAD_ITEMS – список слів. Якщо слово входить у назву предмета, він не буде куплений.

Опис параметрів виставлення ордерів

- BuyParams - параметри виставлення ордерів
- STOP_ORDERS_BALANCE = 1000 - Зупиняти виставлення ордерів при балансі на 10 доларів менше мінімальної ціни ордера
- MIN_AVG_PRICE = 400 – Мінімальна середня ціна за останні 20 покупок предмета в центах. Предмети з нижчою ціною не додаватимуться до бази скінів.
- MAX_AVG_PRICE = 3500 – Максимальна середня ціна за останні 20 покупок предмета в центах. Предмети з вищою ціною не додаватимуться до бази скінів.
- FREQUENCY = True - Включити алгоритм високочастотної торгівлі. За значення True слід вказувати параметр PROFIT_PERCENT = 6 або менше, а параметр GOOD_POINTS_PERCENT = 50 або вище

- MIN_PRICE = 300 – мінімальна середня ціна. Нижче за цю ціну ордер виставлятися не буде.
- MAX_PRICE = 3000 – максимальна середня ціна. Вище за цю ціну ордер на предмет не поставиться.
- PROFIT_PERCENT = 7 – мінімальний профіт, який ми хочемо отримати, якщо купимо предмет за ціною поточного першого ордера.
- GOOD_POINTS_PERCENT = 50 - відсоток точок в історії 20 останніх продажів, що відповідають параметру PROFIT_PERCENT = 7. У даному випадку, якщо мені 50 відсотків точок продавалися з профітом менше 7 відсотків, то ордер на такий предмет ставитися не буде.
- AVG_PRICE_COUNT = 7 - вирахування середньої ціни за останніми 7 продажами для формування передбачуваного профіту.
- ALL_SALES = 100 - мінімальна кількість продажів за весь період, нижче за який ордер виставлятися не буде
- DAYS_COUNT = 20 - щонайменше SALE_COUNT = 15 продажів за DAYS_COUNT = 20 днів. Відбір за популярністю
- SALE_COUNT = 15 - не менше SALE_COUNT = 15 продажів за DAYS_COUNT = 20 днів. Відбір за популярністю
- LAST_SALE = 2 - останній продаж не пізніше LAST_SALE днів тому
- FIRST_SALE = 15 - перша покупка не пізніше FIRST_SALE днів тому
- MAX_COUNT_SELL_OFFERS = 30 – максимальна кількість виставлених предметів на продаж. Понад 30 ордер не поставиться.
- BOOST_PERCENT = 24 - видаляємо до 3 точок, які вищі за середню ціну на 24 відсотки
- BOOST_POINTS = 3 - видаляємо до 3 точок, які вищі за середню ціну на 24 відсотки
- MAX_THRESHOLD = 1 – Максимальне підвищення ціни на MAX_THRESHOLD відсотків від поточного ордера. Максимальне підвищення ціни нашого ордера від ціни поточного першого ордера

- MIN_THRESHOLD = 3 - Максимальне зниження ціни нашого ордера від ціни поточного. Задають межі зміни ціни для ордера
- SellParams - параметри виставлення на продаж
- MIN_PERCENT = 4 - мінімальний відсоток профіту
- MAX_PERCENT = 12 - максимальний відсоток профіту

3.3 Опис функцій main.py

Основний модуль, містить кілька функцій, які використовуються в боті, який взаємодіє з API DMarket для виконання дій, пов'язаних із купівлею та продажем віртуальних предметів (скінів).

Ось короткий опис кожної функції:

- process_main_item_database(): асинхронна функція, яка регулярно оновлює базу даних скінів бота.

```
async def process_main_item_database():
    while True:
        logger.info('Skin database processing')
        try:
            await skin_analyzer.update_skin_base()
            await asyncio.sleep(skin_analyzer.base_update_interval)
        except Exception as e:
            logger.exception(f' Failed to update primary database: {e}. Sleep 5 seconds.')
            await asyncio.sleep(5)
```

- Рисунок 3.3.1 Оновлення бази скінів

- process_orders(): асинхронна функція, яка оновлює список замовлень бота та перевіряє, чи достатньо балансу для розміщення нових замовлень.

```
async def process_orders():
    await asyncio.sleep(5)
    while True:
        try:
            logger.debug(f'{market_api.balance}')
            if market_api.balance > orders_manager.order_analyzer.min_buy_price + BuyConfig.LIMIT_ORDERS_BALANCE:
                await orders_manager.manage_orders()
                await asyncio.sleep(5)
            else:
                user_targets = await orders_manager.market_api.get_user_targets(limit='1000')
                inactive_targets = await orders_manager.market_api.get_user_targets(limit='1000', status='TargetStatusInactive')
                await orders_manager.market_api.delete_user_targets(user_targets.Items + inactive_targets.Items)
                logger.debug('There is not enough money to place orders, we postpone analytics')
                await asyncio.sleep(60 * 5)
        except Exception as e:
            logger.error(f' Failed to get order database: {e}. Sleep 30 seconds.')
            await asyncio.sleep(5)
```

Рисунок 3.3.2 Оновлення списка замовлень

- process_history(): асинхронна функція, яка регулярно зберігає історію інвентаризації бота.

```
async def process_history():
    while True:
        try:
            await market_history.update_skin_history()
            await asyncio.sleep(60*15)
        except Exception as e:
            logger.error(f' Failed to get history: {e}. Sleep 10 seconds.')
            await asyncio.sleep(30)
```

Рисунок 3.3.3 Збереження історії бота

- process_sell_offers(): асинхронна функція, яка додає нові товари до списку товарів для продажу бота.

```
async def process_sell_offers():
    while True:
        try:
            await sell_offers.create_sell_offers()
            await asyncio.sleep(60*10)
        except Exception as e:
            logger.error(f' Failed to put up for sale: {e}. Sleep 10 seconds.')
            await asyncio.sleep(30)
```

Рисунок 3.3.4 Додавання нових товарів

- process_update_offers(): асинхронна функція, яка оновлює список товарів для продажу бота.

```
async def process_update_offers():
    while True:
        try:
            await sell_offers.update_active_offers()
            await asyncio.sleep(5)
        except Exception as e:
            logger.error(f' Failed to update sellable items: {e}. Sleep 10 seconds.')
            await asyncio.sleep(30)
```

Рисунок 3.3.5 Оновлення списку товарів

- all_processes(): асинхронна функція, яка виконує всі функції бота одночасно.

```python
async def all_processes():
    processes = await asyncio.gather(
            market_api.process_check_money(),
            process_history(),
            process_orders(),
            process_sell_offers(),
            process_update_offers(),
            process_main_item_database(), return_exceptions=True
            )
    return processes
```

Рисунок 3.3.6 Запуск процесів бота

- main(): функція, яка ініціалізує бота та запускає основний цикл, доки його не перерве користувач (наприклад, натиснувши Ctrl+C).

```python
def main():
    try:
        logger.info('Bot start')
        event_loop = asyncio.get_event_loop()
        event_loop.run_until_complete(all_processes())
    except KeyboardInterrupt:
        asyncio.run(market_api.close())
        logger.info('Bot stop')
```

Рисунок 3.3.7 Оновлення бази скінів

3.4 Опис функцій calculate_profit.py

```python
from datetime import datetime
from db.orm import SkinOfferManager
from config import logger

all_skins = [i for i in SkinOfferManager.get_all_skins() if i.sellTime]
logger.info(f'Number of skins sold: {len(all_skins)}')
full_profit = 0
from_date = datetime(2023, 4, 1)
for i in all_skins:
    if i.sellTime >= from_date:
        item_profit = round(i.sellPrice - i.buyPrice, 2)
        full_profit += item_profit
        profit_percent = round(item_profit / i.buyPrice * 100, 2)
        logger.info(f'{i.buyTime} - {i.sellTime} {i.title} {i.buyPrice} '
                    f'{round(i.sellPrice, 2)} {item_profit} {profit_percent}')

logger.info(f'Full profit: {round(full_profit, 2)} $.')
```

Рисунок 3.4.1 Підрахунок доходу

У цьому модулі зчитуються дані з таблиці бази даних, що містить інформацію про продані товари, і обчислює прибуток, отриманий від продажу цих товарів.

Зокрема, код імпортує datetime модуль і SkinOfferManager функцію з db.orm модуля, а також модуль logger з config. Потім він вибирає всі елементи з таблиці бази даних, які мають sellTime атрибут, який вказує, що вони були продані. Кількість вибраних елементів реєструється на консолі.

Потім код встановлює from_date змінну на 1 квітня 2023 року та повторює вибрані елементи. Для кожного елемента перевіряється, чи sellTime атрибут більше або дорівнює змінній from. Якщо задовольняє умова, він обчислює отриманий прибуток шляхом віднімання атрибута buyPrice від sellPrice атрибута та записує інформацію про товар на консоль, включаючи час купівлі та продажу, назву предмета, ціни купівлі та продажу, отриманий прибуток, і прибуток у відсотках від ціни покупки.

В кінці обчислюється загальний отриманий прибуток шляхом підсумовування всіх окремих прибутків і реєструє це на консолі.

3.5 Опис функцій init_db.py

```
from pathlib import Path
from playhouse.sqlite_ext import SqliteExtDatabase
from config import DB_NAME

DB_PATH = str(Path(__file__).resolve().parent) + DB_NAME
db = SqliteExtDatabase(DB_PATH, check_same_thread=False)
```

Рисунок 3.5.1 Ініціалізація бази даних

Шлях до бази даних створюється з використанням DB_NAME константи з config модуля та шляху до поточного файлу сценарію. Екземпляр бази даних зберігається в db змінній.

Короткий опис функцій і модулів, які використовуються в цьому модулі:

- Path: Клас з pathlib модуля, який представляє шлях до файлу або каталогу. Метод resolve()повертає абсолютний шлях до файлу або каталогу.
- SqliteExtDatabase: клас із playhouse.sqlite_ext модуля, який представляє з'єднання з базою даних SQLite. Він розширює SqliteDatabase клас із того самого модуля додатковими функціями, такими як підтримка розширень JSON і FTS5.
- DB_NAME: константа з config модуля, що вказує назву файлу бази даних SQLite.
- check_same_thread: Параметр конструктора SqliteExtDatabase, який визначає, чи має бути дозволено використовувати підключення до бази даних кількома потоками чи ні. Встановлення цього значення False дозволяє використовувати з'єднання різними потоками.

3.6 Опис функцій schemas.py

Цей модуль визначає схему бази даних за допомогою Peewee ORM. Він визначає дві моделі Skin і SkinOffer, які успадковують базову модель BaseModel. Моделі визначають поля для різноманітної інформації, пов'язаної зі скінами та пропозиціями скінів у грі.

```python
class Skin(BaseModel):
    title = CharField()
    game = CharField()
    LastSales = JSONField()
    avg_price = FloatField()
    update_time = DateTimeField()


class SkinOffer(BaseModel):
    title = CharField(null=True)
    AssetID = CharField()
    game = CharField(null=True)
    buyPrice = FloatField(null=True)
    buyTime = DateTimeField(null=True)
    OfferID = CharField(null=True)
    sellTime = DateTimeField(null=True)
    sellPrice = FloatField(null=True)
    fee = IntegerField(default=3)
```

Рисунок 3.6.1 Моделі Skin та SkinOffer

Модель Skin Має поля для назви скіна, гри, до якої він належить, поле JSON для історії продажів скіна, поле середньої ціни та поле часу оновлення.

У SkinOffer моделі є поля для назви пропозиції, ідентифікатора активу, гри, ціни покупки, часу покупки, ідентифікатора пропозиції, часу продажу, ціни продажу та комісії.

Крім того, код визначає настроювання JSONField, яке серіалізує та десеріалізує дані JSON для Last Sales поля в Skin моделі.

3.7 Опис функцій orm.py

Тут описані функції Python, які взаємодіють із базою даних за допомогою Peewee ORM для виконання операцій CRUD (Create, Read, Update, Delete) у двох моделях: Skin і SkinOffer.

SkinManager клас має такі методи:

- add_all_skins(items: List[SkinHistory]): створює список об’єктів Skin у базі даних.

```python
def add_all_skins(items: List[SkinHistory]):
    skins = [Skin(**i.dict()) for i in items]
    with db.atomic():
        Skin.bulk_create(skins, batch_size=500)
```

Рисунок 3.7.1 Сторуння списку об’єктів Skin

- is_skin_exists(item: MarketOffer): перевіряє наявність скіна в базі даних.

```python
def is_skin_exists(item: MarketOffer):
    skin = Skin.select().where(Skin.title == item.title)
    if skin:
        return True
    return False
```

Рисунок 3.7.2 Перевірка наявності в базі даних

- update_or_create_skins_by_title(items: List[SkinHistory]): оновлює існуючі скіни або створює нові на основі списку об’єктів SkinHistory.

```python
def update_or_create_skins_by_title(items: List[SkinHistory]):
    updated_skins = list()
    created_skins = list()
    for item in items:
        try:
            skin = Skin.get(Skin.title == item.title)
            it = item.dict()
            skin.avg_price = it['avg_price']
            skin.LastSales = it['LastSales']
            skin.update_time = it['update_time']
            updated_skins.append(skin)
        except DoesNotExist:
            created_skins.append(Skin(**item.dict()))
    with db.atomic():
        Skin.bulk_update(updated_skins,
                         fields=[Skin.avg_price, Skin.LastSales, Skin.update_time],
                         batch_size=500)
    with db.atomic():
        Skin.bulk_create(created_skins, batch_size=500)
```

Рисунок 3.7.3 Оновлення скінів

- get_all_skins() -> List[SkinHistory]: отримує всі об’єкти Skin з бази даних і повертає список об’єктів SkinHistory.

```python
def get_all_skins() -> List[SkinHistory]:
    skins = Skin.select()
    return [SkinHistory.from_orm(skin) for skin in skins]
```

Рисунок 3.7.4 Отримання всіх скінів з бази даних

- get_skins_updated_before (now, delta) -> List[SkinHistory]: отримує об’єкти Skin, які востаннє оновлювалися до визначеного періоду часу, і повертає список об’єктів SkinHistory.

```
def get_skins_updated_before(now, delta) -> List[SkinHistory]:
    skins = Skin.select().where(Skin.update_time < datetime.fromtimestamp(now - delta))
    if skins:
        return [SkinHistory.from_orm(skin) for skin in skins]
    return []
```

Рисунок 3.7.5 Отримання скінів до часу

SkinOfferManager клас має такі методи:

- add_skin(item: SellOffer) -> None: створює новий об’єкт SkinOffer у базі даних.

```
def add_skin(item: SellOffer) -> None:
    new_skin = SkinOffer.create(
        title=item.title,
        game=item.game,
        AssetID=item.AssetID,
        buyPrice=item.buyPrice,
        buyTime=item.buyTime,
        OfferID=item.OfferID,
        sellTime=item.sellTime,
        sellPrice=item.sellPrice
    )
    new_skin.save()
```

Рисунок 3.7.6 Додавання скіна в бд

- update_sold_skins(skins: List[SkinOffer]): оновлює список об’єктів SkinOffer із назвою, ціною продажу, часом продажу та ідентифікатором пропозиції.

```
def update_sold_skins(skins: List[SkinOffer]):
    if not skins:
        return
    with db.atomic():
        SkinOffer.bulk_update(skins, fields=[SkinOffer.title, SkinOffer.sellPrice,
                                             SkinOffer.sellTime, SkinOffer.OfferID])
```

Рисунок 3.7.7 Оновлення списку проданих скінів

- get_unsold_skins() -> List[SellOffer]: отримує всі об'єкти SkinOffer із бази даних, де час продажу не встановлено, і повертає список об'єктів SellOffer.

```
def get_unsold_skins() -> List[SellOffer]:
    skins = SkinOffer.select().where(SkinOffer.sellTime == None)
    return [SellOffer.from_orm(s) for s in skins]
```

Рисунок 3.7.8 Отримання непроданих скінів

- get_all_skins () -> List[SkinOffer]: отримує всі об'єкти SkinOffer із бази даних і повертає список об'єктів SkinOffer.

```
def get_all_skins() -> List[SkinOffer]:
    skins = SkinOffer.select()
    return skins
```

Рисунок 3.7.9 Отримання всіх скінів

- clear_all_skins (): видаляє всі об'єкти SkinOffer з бази даних.

```
def clear_all_skins():
    skins = SkinOffer.select()
    for s in skins:
        s.delete_instance()
```

Рисунок 3.7.10 Видалення скінів з бд

- modify_skin_by_asset(skin: SellOffer): оновлює об'єкт SkinOffer із зазначеним AssetID із часом продажу, ціною продажу та ідентифікатором пропозиції.

```
def modify_skin_by_asset(skin: SellOffer):
    try:
        item = SkinOffer.get(SkinOffer.AssetID == skin.AssetID)
        item.OfferID = skin.OfferID
        item.sellTime = skin.sellTime
        item.sellPrice = skin.sellPrice
        item.save()
    except DoesNotExist:
        pass
```

Рисунок 3.7.11 Оновлення пропозиції на продаж

3.8 Опис функцій модуля helpers.py

```python
from pyti.simple_moving_average import simple_moving_average as sma
from typing import List
from api.interfaces import LastSale


def calculate_moving_average(history: List[LastSale]) -> list:
    cost = [i.Price.Amount for i in history]
    cost.reverse()
    averaged_values = [i for i in list(sma(cost, 5))]
    averaged_values.reverse()
    return averaged_values
```

Рисунок 3.8.1 Розрахунок SMA

Ця функція обчислює 5-періодне просте ковзне середнє (SMA) на основі списку об’єктів LastSale. Він використовує simple_moving_average функцію з pity бібліотеки для обчислення SMA. Функція спочатку витягує дані про ціну зі history списку, змінює порядок цін на зворотний (таким чином, щоб остання ціна була в кінці списку), обчислює SMA за допомогою sma, а потім повертає список SMA до початкового порядку. Функція повертає SMA у вигляді списку.

3.9 Опис функцій модуля sell_skins.py

Даний модуль є частиною торгового бота, який взаємодіє з API DMarket для керування скінами, пропозиціями та транзакціями.

- Клас MarketHistory визначає методи для отримання скінів із бази даних, збереження проданих скінів у базі даних і оновлення бази даних проданими скінами.
- Клас SellOffers визначає методи додавання скінів для продажу шляхом створення пропозицій на ринку DMarket і оновлення цін існуючих пропозицій на основі агрегованих ринкових цін.

В даному модулі реалізовано функціонал:

- Отримання скінів з бази даних, які зараз не продаються.

- Отримання інвентарю користувача та визначення ціну продажу для кожного товару.
- Створення нових пропозицій на ринку для кожного товару, який ще не продається.
- Отримання існуючих пропозицій для скінів, які ще не продані.
- Оновлення ціни існуючих пропозицій відповідно до агрегованих ринкових цін.

Існують деякі константи та параметри, визначені в config модулі, такі як PUBLIC_KEY, PRIVATE_KEY і GAMES_FOR_TRADE, які є списком ідентифікаторів гри. Об'єкт logger використовується для реєстрації повідомлень про налагодження.

3.10 Опис функцій модуля orders.py

Цей код визначає клас OrderAnalyzer із методами для аналізу замовлень скінів на DMarket. Клас має різні параметри, які використовуються для фільтрації та вибору скінів для покупки.

Метод filter_skins_by_sales_date_range отримує список SkinHistory об'єктів і повертає відфільтрований список оболонок, які відповідають певним умовам, пов'язаним з історією їх продажу.

```
def filter_skins_by_sales_date_range(self, skins: List[SkinHistory]) -> List[SkinHistory]:
    filtered_skins = list()
    for i in skins:
        sales_list = list()
        first_sale_days_ago = int(i.LastSales[-1].Date.timestamp())
        last_sale_days_ago = int(i.LastSales[0].Date.timestamp())
        if first_sale_days_ago < (time() - self.first_sale_days_ago * 60 * 60 * 24):
            if last_sale_days_ago > (time() - self.last_sale_days_ago * 60 * 60 * 24):
                for j in i.LastSales:
                    if int(j.Date.timestamp()) > (time() - self.number_of_days * 60 * 60 * 24):
                        sales_list.append(j)
                if len(sales_list) >= self.number_of_sales:
                    filtered_skins.append(i)
    return filtered_skins
```

Рисунок 3.10.1 Фільтрація скінів по продажам

Метод control_boosted_sales бере список SkinHistory об'єктів і повертає відфільтрований список оболонок, які відповідають певним умовам, пов'язаним із останніми цінами продажу.

```
def control_boosted_sales(self, skins: List[SkinHistory]) -> List[SkinHistory]:
    filtered_skins = list()
    for item in skins:
        moving_average = calculate_moving_average(item.LastSales)
        number_of_abnormal_points = 0
        try:
            for i in range(len(moving_average[:-4])):
                if item.LastSales[i].Price.Amount > \
                        moving_average[i] * (1 + self.abnormal_price / 100):
                    item.LastSales.pop(i)
                    number_of_abnormal_points += 1
            if number_of_abnormal_points <= self.abnormal_points:
                filtered_skins.append(item)
        except IndexError:
            pass
    return filtered_skins
```

Рисунок 3.10.2 Фільтрація скінів за критеріями

Метод filter_profitable_skins бере список SkinHistory об’єктів, обчислює загальну ціну кожного скіна та фільтрує скіни, які відповідають певним умовам, пов’язаним з історією продажу та ціною.

```
async def filter_profitable_skins(self, skins: List[SkinHistory]) -> List[SkinOrder]:
    filtered_skins = list()
    sorted_skins = sorted(skins, key=lambda x: x.title)
    titles_list = [i.title for i in sorted_skins]
    agr_prices = await self.bot.get_agregated_prices(titles_list)
    sorted_agr_prices = sorted(agr_prices, key=lambda x: x.MarketHashName)

    for item, prices in zip(sorted_skins, sorted_agr_prices):
        best_order_price = prices.Orders.BestPrice*100
        number_of_points = math.ceil(len(item.LastSales) / 100 * self.pos_points)
        counter = 0
        for i in item.LastSales:
            price_with_fee = i.Price.Amount * (1 - self.bot.SELL_FEE/100)
            if price_with_fee > best_order_price * (1 + self.acceptable_profit / 100):
                counter += 1
        if counter >= number_of_points:
            if prices.Offers.Count <= self.max_on_sale_offers:
                filtered_skins.append(SkinOrder(title=item.title, bestOrder=int(best_order_price), game=item.game))
    return filtered_skins
```

Рисунок 3.10.3 Фільтрація скінів за ціною

Метод filter_frequency_skins подібний до методу, filter_profitable_skins але фільтрує скіни на основі їх частоти продажу та ціни.

```python
async def filter_frequency_skins(self, skins: List[SkinHistory]) -> List[SkinOrder]:
    filtered_skins = list()
    skins = sorted(skins, key=lambda x: x.title)
    for item in skins:
        top_offer_price, top_target_price, total_offers, total_targets, profit_margin, avg_profit_margin = \
            await self.calculate_market_offers_profit(item)
        if avg_profit_margin > self.acceptable_profit and profit_margin > self.acceptable_profit:
            desired_sell_price = top_target_price * (1 + self.acceptable_profit / 100)
            count = 0
            points_count = math.ceil(len(item.LastSales) / 100 * self.pos_points)
            for i in item.LastSales:
                price_with_fee = i.Price.Amount * (1 - self.bot.SELL_FEE / 100)
                if price_with_fee > desired_sell_price:
                    count += 1
            if count >= points_count:
                if total_offers <= self.max_on_sale_offers:
                    filtered_skins.append(SkinOrder(title=item.title, bestOrder=int(top_target_price*100), game=item.game))
    return filtered_skins
```

Рисунок 3.10.4 Фільтрація скінів за частотою продажів

Нарешті, get_first_and_second_price метод отримує список CumulativePrice об’єктів і повертає першу та другу найкращі пропозиції для скінів.

```python
def get_first_and_second_price(info: List[CumulativePrice]) -> tuple:
    number_of_offers = len(info)
    if number_of_offers == 0:
        price_of_the_best_offer = 0
        price_of_second_best_offer = 0
    else:
        the_best_offer = info[0]
        if number_of_offers == 1:
            second_best_offer = the_best_offer
        else:
            if the_best_offer.Amount == 1:
                second_best_offer = info[1]
            else:
                second_best_offer = the_best_offer
        price_of_the_best_offer = the_best_offer.Price
        price_of_second_best_offer = second_best_offer.Price
    return price_of_the_best_offer, price_of_second_best_offer, number_of_offers
```

Рисунок 3.10.5 Отримання двох найкращих пропозицій

3.11 Опис функцій модуля skin_analyzer.py

Цей модуль визначає SkinAnalyzer клас, який використовується для аналізу та оновлення бази даних історій скінів для різних ігор на DMarket, цифровому ринку для геймерів.

Клас SkinAnalyzer має кілька методів:

- __init__: Ініціалізує клас різними параметрами, такими як об’єкт API DMarket, мінімальна та максимальна ціни для аналізу скінів і різні параметри для купівлі скінів.
- is_valid_skin_name: статичний метод, який перевіряє, чи немає назви оформлення у списку поганих елементів.
- fetch_market_items: отримує список ринкових пропозицій для певної гри та цінового діапазону та повертає лише унікальні скіни зі списку. Цей метод використовує API DMarket для надсилання HTTP-запитів і отримання даних.
- select_filtered_skins: фільтрує список скінів, порівнюючи їхні середні ціни з певним ціновим діапазоном, і оновлює історію скінів у базі даних останніми даними про продажі. Цей метод також використовує DMarket API для отримання даних про продажі для кожного скіна.
- select_filtered_skins: оновлює базу даних новими скінами, які відповідають мінімальним критеріям для аналізу, викликаючи fetch_market_items та select_filtered_skins методи для кожної гри зі GAMES_FOR_TRADE списку.
- update_skin_base: оновлює базу даних новими даними про продаж скінів, які нещодавно не оновлювалися, викликаючи метод select_filtered_skins для списку скінів, отриманих із бази даних. Цей метод викликається періодично на основі base_update_interval параметра.

У коді використовуються зовнішні бібліотеки, такі як pydantic, itertools і datetime для надання додаткових функцій і структур даних. Модуль logger використовується для реєстрації інформації та помилок під час виконання коду.

3.12 Діаграма діяльності

Діаграма послідовності (sequence diagram) - це графічний інструмент моделювання взаємодії між об'єктами в рамках системи. Вона використовується для візуалізації послідовності повідомлень, які передаються між об'єктами протягом певного сценарію взаємодії.

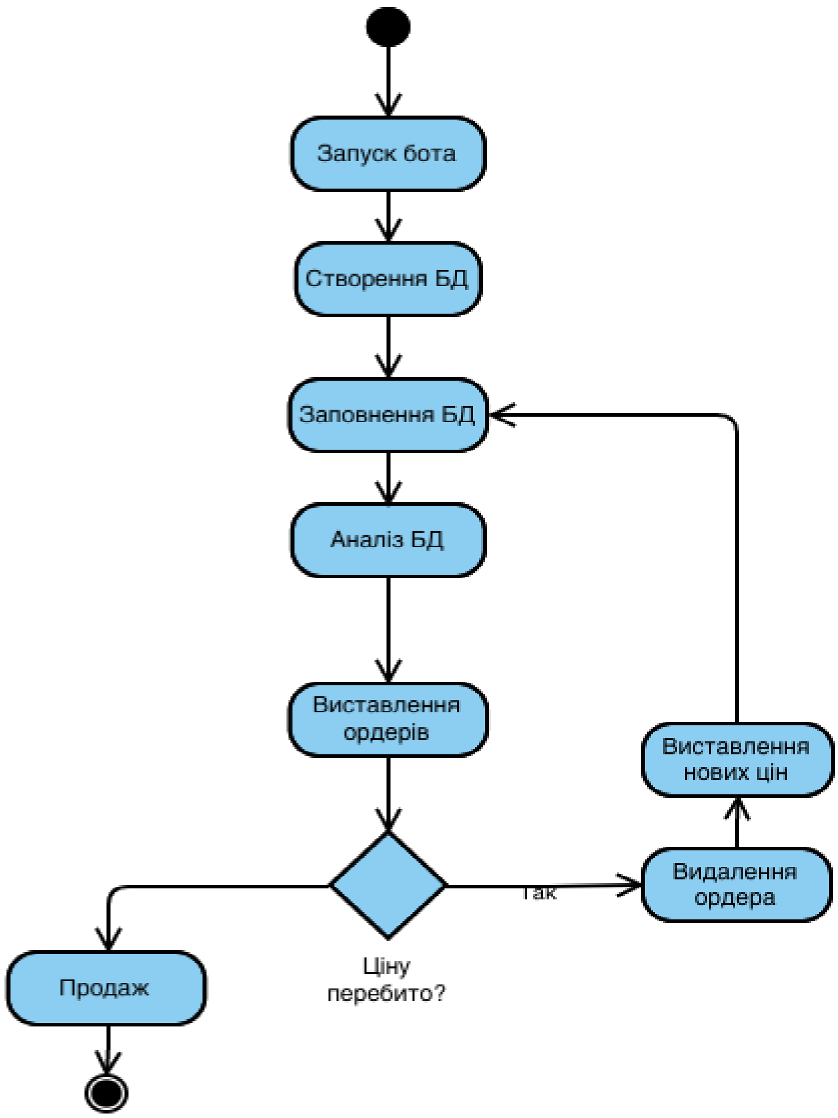


Рисунок 3.12.1 Діаграма діяльності

На діаграмі вище. зображено дії що виконуються ботом в процесі торгівлі віртуальними скінами. Для початку запускаться бот, далі створюється база даних для зберігання скінів і подальшого їх аналізу, після аналізу відбувається виставлення ордерів, якщо ціну перебито іншим користувачем, то ордер видаляється і створюється новий з іншими цінами для всіх скінів, якщо ж ні, відбувається продаж скіна.

ВИСНОВКИ ДО РОЗДІЛУ 3

Таким чином, у цьому розділі було продемонстровано результати розробки бота. Бот для автоматизаціх торгівлі на Dmarket реалізує функціонал, що дозволяє автоматизувати процес купівлі та продажу предметів на Dmarket. Для цього він використовує API Dmarket, яке надає доступ для виконання необхідних операцій на платформі Була представлена та пояснена конфігурація роботи бота для його коректної роботи. Детально описаний принцип роботи функцій, класів та їх методів. В розділі продемостровані фрагменти коду у вигляді скріншотів, що відповідають виконанню описаних процесів.

РОЗДІЛ 4. ТЕСТУВАННЯ РОЗРОБЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Контрольний приклад

На початку роботи з ботом, потрібно його запустити.

```
2023-05-31 14:37:33.650 | INFO     | __main__:main:86 - Bot start
C:\Users\steam\Downloads\bot\main.py:87: DeprecationWarning: There is no current event loop
  event_loop = asyncio.get_event_loop()
2023-05-31 14:37:33.669 | INFO     | __main__:process_main_item_database:64 - Skin database processing
2023-05-31 14:40:18.392 | INFO     | components.skin_analyzer:refresh_data:83 - Total items analyzed: 53
2023-05-31 14:40:18.400 | INFO     | components.skin_analyzer:update_skin_base:91 - No skins to update
2023-05-31 14:40:18.402 | INFO     | components.skin_analyzer:update_skin_base:94 - Skins database updated 2.75 minutes.
2023-05-31 14:40:20.280 | INFO     | components.buy_skins:process_skins_analysis:152 - NUMBER OF SKINS 53
2023-05-31 14:40:20.424 | INFO     | components.buy_skins:process_skins_analysis:160 - REMAINING AFTER ANALYSIS BY PARAMETERS 3
```

Рисунок 4.1 Запуск бота

Після запуску бота таблиця БД заповнюється скінами, які в подальшому будуть виставлені на продажу.

Структура БД | Данные | Прагмы | SQL

Таблица: skin   Filter in any column

	id	title	game	LastSales	avg_price	update_time
	Фи...	Фильтр	Фил...	Фильтр	Фильтр	Фильтр
1	1	Desert Eagle | Light Rail (Battle-...	a8db	[{"Date": ...	54.55	2023-05-31 14:38:11.121044
2	2	Desert Eagle | Light Rail (Field-Tested)	a8db	[{"Date": ...	51.55	2023-05-31 14:38:11.173376
3	3	Desert Eagle | Oxide Blaze (Minimal ...	a8db	[{"Date": ...	57.35	2023-05-31 14:38:12.361587
4	4	Desert Eagle | Sputnik (Field-Tested)	a8db	[{"Date": ...	57.1	2023-05-31 14:38:12.553640
5	5	Dual Berettas | Flora Carnivora (Battl...	a8db	[{"Date": ...	51.3	2023-05-31 14:38:14.533282
6	6	Dual Berettas | Flora Carnivora (Well...	a8db	[{"Date": ...	50.2	2023-05-31 14:38:15.588067
7	7	FAMAS | Faulty Wiring (Factory New)	a8db	[{"Date": ...	70.45	2023-05-31 14:38:17.141658
8	8	Five-SeveN | Buddy (Field-Tested)	a8db	[{"Date": ...	53.45	2023-05-31 14:38:18.698578
9	9	G3SG1 | Dream Glade (Minimal Wear)	a8db	[{"Date": ...	50.25	2023-05-31 14:38:23.027928
10	10	Galil AR | Rocket Pop (Field-Tested)	a8db	[{"Date": ...	50.25	2023-05-31 14:38:26.346779
11	11	Galil AR | Sage Spray (Field-Tested)	a8db	[{"Date": ...	107.35	2023-05-31 14:38:27.409492
12	12	Glock-18 | Clear Polymer (Factory ...	a8db	[{"Date": ...	68.9	2023-05-31 14:38:28.995850
13	13	Glock-18 | High Beam (Minimal Wear)	a8db	[{"Date": ...	72.65	2023-05-31 14:38:29.282619
14	14	Glock-18 | Winterized (Factory New)	a8db	[{"Date": ...	58.1	2023-05-31 14:38:32.009347
15	15	M4A1-S | Fizzy POP (Field-Tested)	a8db	[{"Date": ...	54.25	2023-05-31 14:38:33.735385
16	16	MAC-10 | Button Masher (Minimal ...	a8db	[{"Date": ...	51.05	2023-05-31 14:38:36.475706
17	17	MP5-SD | Agent (Field-Tested)	a8db	[{"Date": ...	53.55	2023-05-31 14:38:43.216921
18	18	MP9 | Army Sheen (Factory New)	a8db	[{"Date": ...	157.6	2023-05-31 14:38:47.538974
19	19	Negev | Lionfish (Minimal Wear)	a8db	[{"Date": ...	53.45	2023-05-31 14:38:49.353208
20	20	Negev | Loudmouth (Well-Worn)	a8db	[{"Date": ...	61.2	2023-05-31 14:38:50.515886
21	21	P250 | Cyber Shell (Minimal Wear)	a8db	[{"Date": ...	51.6	2023-05-31 14:38:59.206276
22	22	Sawed-Off | Full Stop (Well-Worn)	a8db	[{"Date": ...	53.25	2023-05-31 14:39:23.513463
23	23	Souvenir Five-SeveN | Coolant (Facto...	a8db	[{"Date": ...	61.7	2023-05-31 14:39:27.817759
24	24	Souvenir MP7 | Motherboard (Factory	a8db	[{"Date":	58.4	2023-05-31 14:39:29.592944

1 - 24 из 53   Перейти к: 1

Рисунок 4.2 Скіни, що аналізуються ботом

Наступним кроком є виставлення ордерів за допомогою бота.

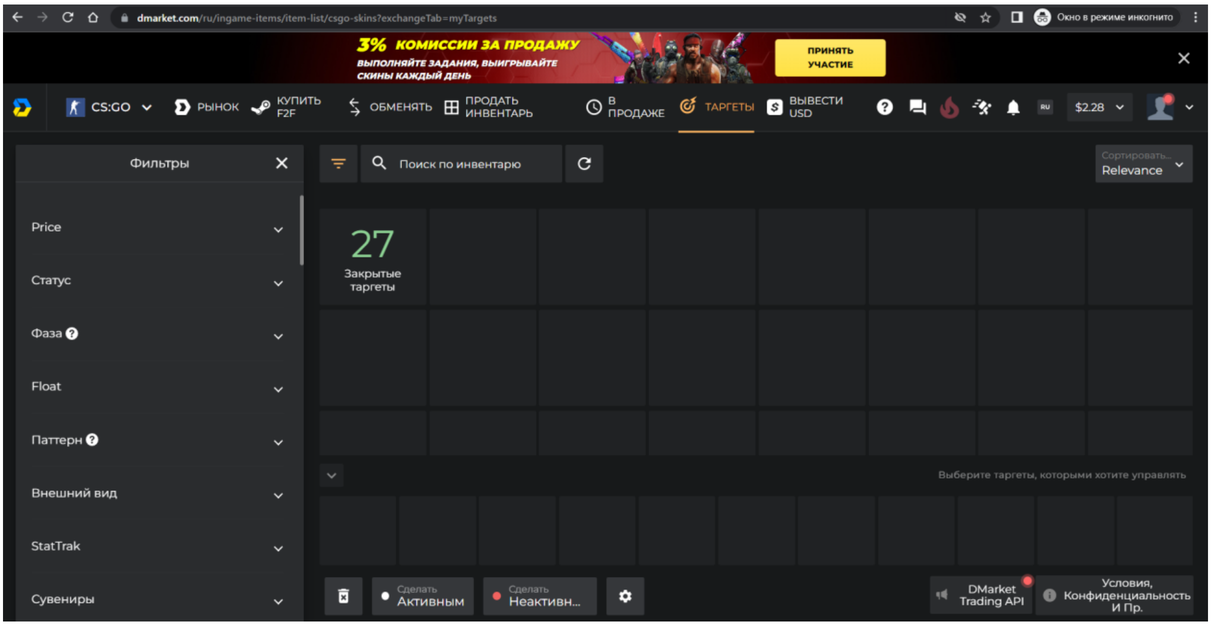


Рисунок 4.3 До запуску бота

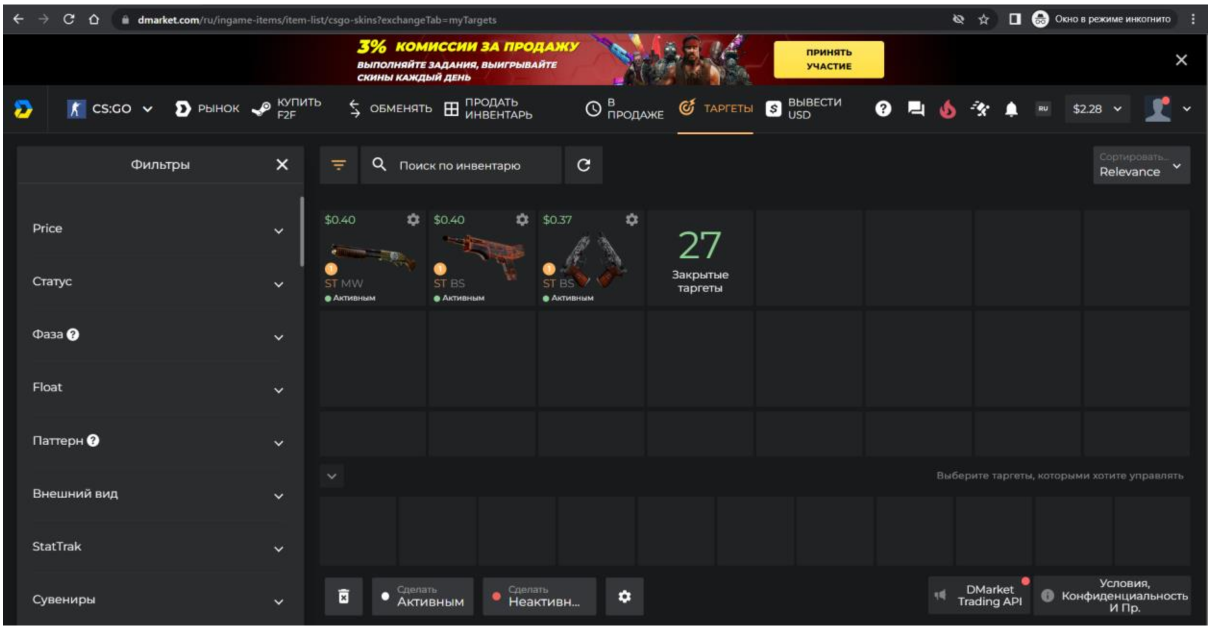


Рисунок 4.4 Виставлені ордери

На прикладі одного з предмета, бот виставив ордер на 0,4, за цією ціною в нього є всі шанси щоб купити даний предмет, та отримати бажаний прибуток після його продажу.

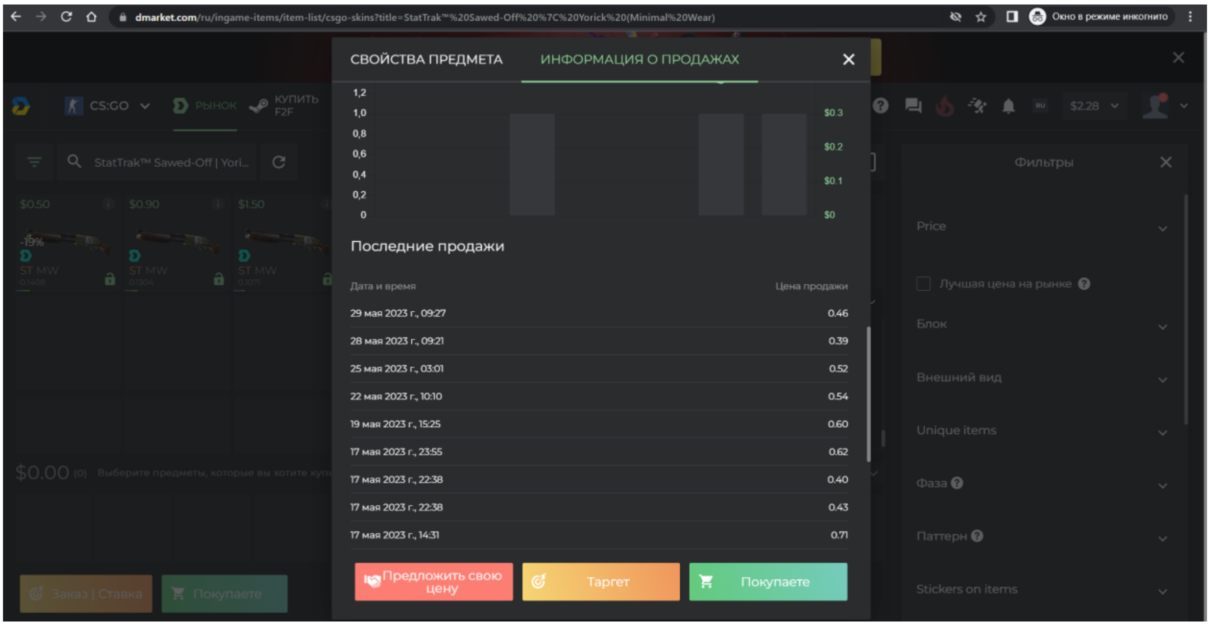


Рисунок 4.5 Купівля предмету

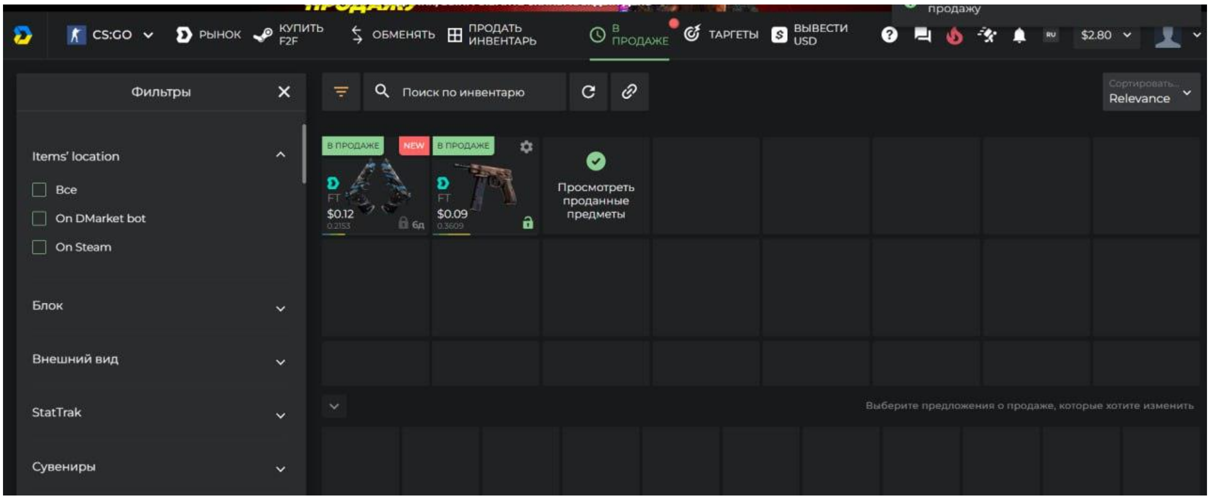


Рисунок 4.6 Виставлення на продаж купленого скіна

В таблиці skinoffers відображаються скіни, які були виставлені на продаж, ціну їх покупки та ціну продажу

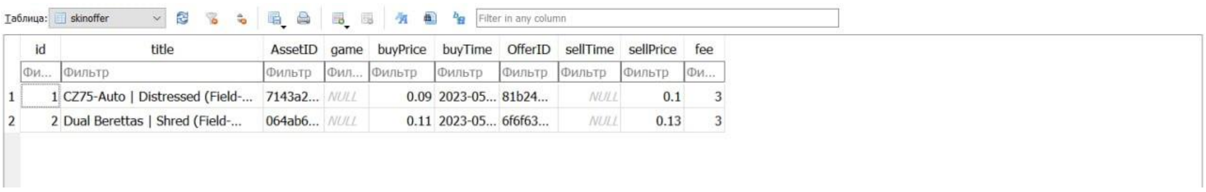


Таблица: skinoffer

	id	title	AssetID	game	buyPrice	buyTime	OfferID	sellTime	sellPrice	fee
	Фи...	Фильтр	Фильтр	Фил...	Фильтр	Фильтр	Фильтр	Фильтр	Фильтр	Фи...
1	1	CZ75-Auto | Distressed (Field-...	7143a2...	NULL	0.09	2023-05...	81b24...	NULL	0.1	3
2	2	Dual Berettas | Shred (Field-...	064ab6...	NULL	0.11	2023-05...	6f6f63...	NULL	0.13	3

Рисунок 4.7 Таблиця skinoffers

Слід зазначити, що з 53 скінів він виставив ордери тільки на 3, тому що використовуються строгі налаштування до відбору скінів. Цей бот - інстремент для торгівлі, який можна гнучко налаштовувати під стратегію торгівлі.

ВИСНОВКИ ДО РОЗДІЛУ 4

В даному розділі було описано контрольний приклад роботи бота автоматичної торгівлі віртуальними активами. Згідно з результатів, можна побачити, що бот знайшов потенційно привабливі предмети на сайті Dmarket, виконав аналіз за заданою конфігурацією та з 53 предметів виставив ордери на покупку тільки на 3 з них, оскільки використовувались строгі налаштування до відбору скінів, для найбільш безпечної торгівлі. Це не дозволить ризикувати у виборі предметів та допоможе торгувати більш стабільно наявними активами.

Було показано, що бот правильно проаналізував скіни, і така стратегія працює в реальних умовах. Окрім цього було продемонстровано, як бот виставляє предмети на продаж, в першу чергу, шляхом аналізу ціни купівлі предмета та пропозицій конкрурентів.

ВИСНОВОК

В ході написання дипломної роботи, було визначено, що віртуальні активи – це цифрові елементи, які існують лише в онлайн-середовищах, таких як відеоігри, соціальні мережі та віртуальні світи. Вони бувають у багатьох формах, включаючи ігрові предмети, скіни, віртуальну валюту, цифрові предмети колекціонування, віртуальну нерухомість і віртуальних домашніх тварин.

Торгівля віртуальними активами стала популярною практикою, і багато гравців прагнуть розширити свій портфель віртуальних активів і скористатися можливостями торгівлі в різних іграх і на платформах. Торгівля з кількома іграми пропонує можливості для диверсифікації та потенційного арбітражу, але трейдери повинні знати про пов’язані з цим ризики та складності.

Порівнюючи Dmarket та BitSkins, ми надали перевагу першому, адже він більш популярний та має невисоку комісію. Боти конкурентів не розглядались, бо їх немає у вільному доступі.

Було розроблено функціональні вимоги, бот повинен мати можливість торгувати в кількох іграх, аналізувати бази даних скінів, розміщувати замовлення на основі багатьох параметрів, динамічно коригувати ціни та створювати звіти про торгову діяльність.

Крім функціональних вимог, бот також має відповідати декільком нефункціональним вимогам, таким як висока продуктивність, надійність, безпека, зручність використання, масштабованість, ремонтопридатність, сумісність і доступність.

У результаті розробки бота було реалізовано функціонал для автоматичної торгівлі на Dmarket. Представлена та пояснена конфігурація роботи бота. Детально описаний принцип роботи функцій та класів. Продемонстровано фрагменти коду для описаних процесів у вигляді скріншотів.

Згідно з результатів, можемо зробити висновок, що бот працює правильно, адже в контрольному прикладі було виявлено, що що з 65 скінів він виставив ордери тільки на 4, тому що використовуються строгі налаштування до відбору скінів.

Підсумовуючи, це дослідження є комплексним дослідженням розробки автоматизованих торгових ботів для Dmarket. Досліджуючи тему під різними кутами, нам вдалось висвітлити алгоритмічну торгівлю, водночас було розроблено бот для віртуальної торгівлі активами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. What is virtual asset? [Електронний ресурс] – Режим доступу до ресурсу: https://www.techtarget.com/whatis/definition/virtual-asset.
2. "Virtual currency and the gaming industry: Trading real world dollars for virtual world fun" by Kathleen M. Garvin, published in the Journal of Business & Technology Law.
3. "Virtual currency and the gaming industry: Trading real world dollars for virtual world fun" by Kathleen M. Garvin, published in the Journal of Business & Technology Law.
4. Офіційний сайт - Dmarket [Електронний ресурс] – Режим доступу до ресурсу: dmarket.com.
5. Офіційний сайт - Bitskins [Електронний ресурс] – Режим доступу до ресурсу: https://bitskins.com/price
6. Pavel G. What are Functional and Non-Functional Requirements and How to Document These [Електронний ресурс] / Gorbachenko Pavel // Enkonix – Режим доступу до ресурсу: https://enkonix.com/blog/functional-requirements-vs-non-functional/.
7. What is Python? [Електронний ресурс] – Режим доступу до ресурсу: https://www.python.org/doc/essays/blurb/.
8. MySQL [Електронний ресурс] – Режим доступу до ресурсу: https://www.mysql.com/products/workbench/.
9. PostgreSQL: The World's Most Advanced Open Source Relational Database [Електронний ресурс] – Режим доступу до ресурсу: https://www.postgresql.org/.
10. What Is SQLite? [Електронний ресурс] – Режим доступу до ресурсу: https://sqlite.org/index.html.
11. Коротко про API та його тестування [Електронний ресурс] – Режим доступу до ресурсу: https://qagroup.com.ua/publications/korotko-pro-ari-ta-jogo-testuvannia/.

12."Virtual currency and the gaming industry: Trading real world dollars for virtual world fun" by Kathleen M. Garvin, published in the Journal of Business & Technology Law.
13.DMarket trading API [Електронний ресурс] – Режим доступу до ресурсу: https://docs.dmarket.com/v1/swagger.html?_gl=1*zgjp6w*_ga*NDA5MjM0MjY2LjE2ODAyOTg5MzU.*_ga_J73817TZWR*MTY4MDI5ODkzNS4xLjEuMTY4MDI5OTIzMi42MC4wLjA.
14.UML для бізнес-моделювання: для чого потрібні діаграми процесів [Електронний ресурс] – Режим доступу до ресурсу: https://evergreens.com.ua/ua/articles/uml-diagrams.html.

ДОДАТОК 1

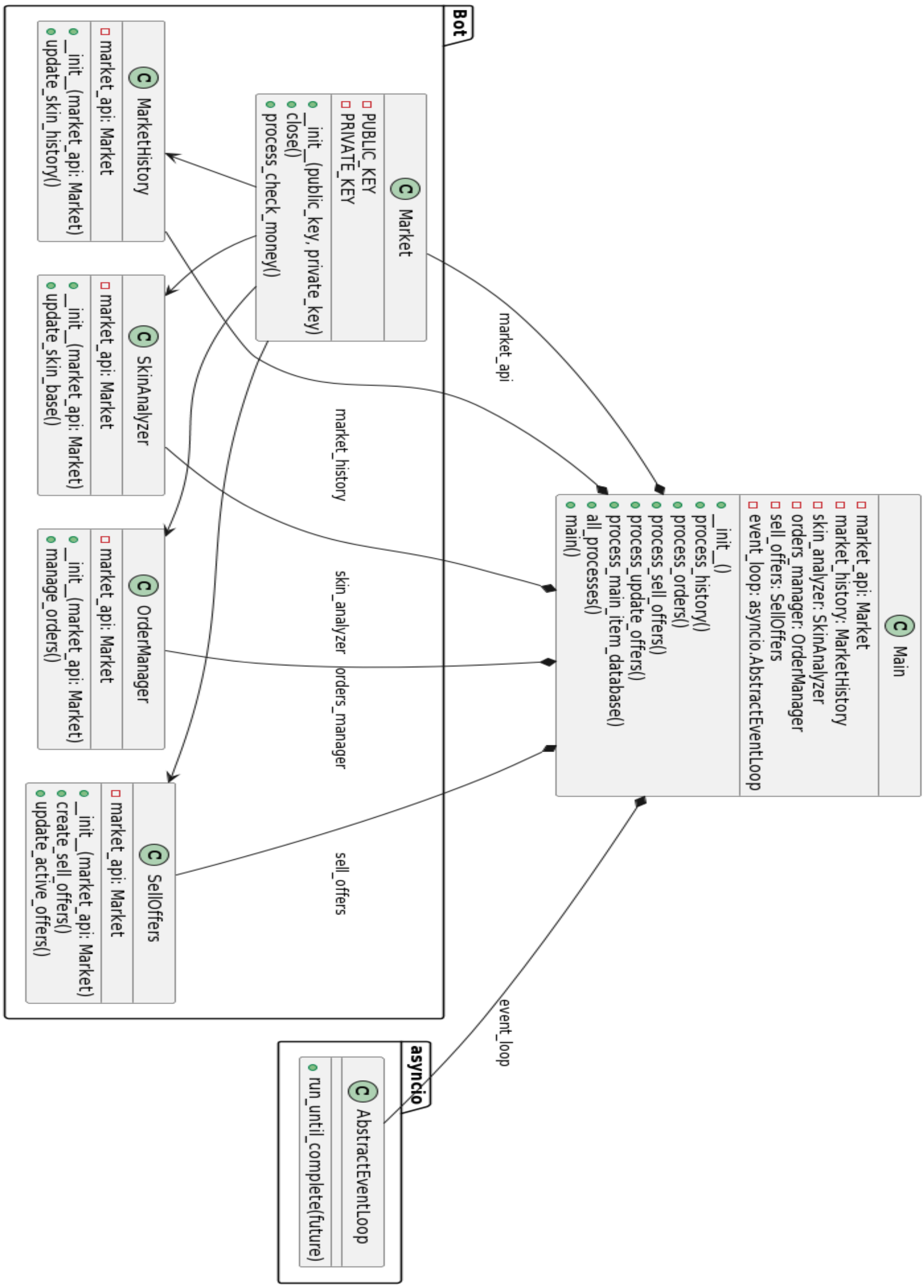
Бот для автоматизації торгівлі віртуальними активами

Архітектура застосунку

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ - 2023



					ІАЛЦ.467200.004 Д1			
Змн.	Арк.	№ докум.	Підпис	Дата				
Розроб.		Литвишко Н.А			*Бот для автоматизації торгівлі віртуальними активами* Архітектура застосунку (структурна схема)	Літ.	Арк.	Акрушів
Перевір.							1	1
						КПІ ім. Ігоря Сікорського ФІОТ ІП-94		
Н. Контр.		Волокита А.М.						
Затверд.								

ДОДАТОК 2

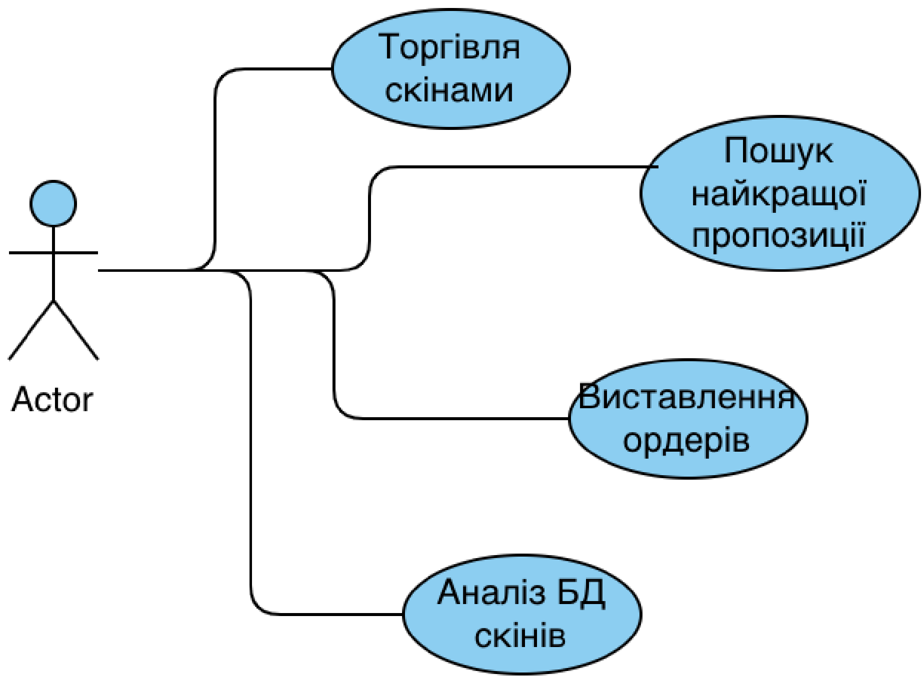
Бот для автоматизації торгівлі віртуальними активами

Діаграма використання

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ - 2023



					ІАЛЦ.467200.005 Д2			
Змн.	Арк.	№ докум.	Підпис	Дата				
Розроб.		*Литвишко Н.А*			*Бот для автоматизації торгівлі віртуальними активами* Діаграма використання (функціональна схема)	Літ.	Арк.	Акрушів
Перевір.							1	1
						КПІ ім. Ігоря Сікорського ФІОТ ІП-94		
Н. Контр.		*Волокита А.М.*						
Затверд.								

ДОДАТОК З

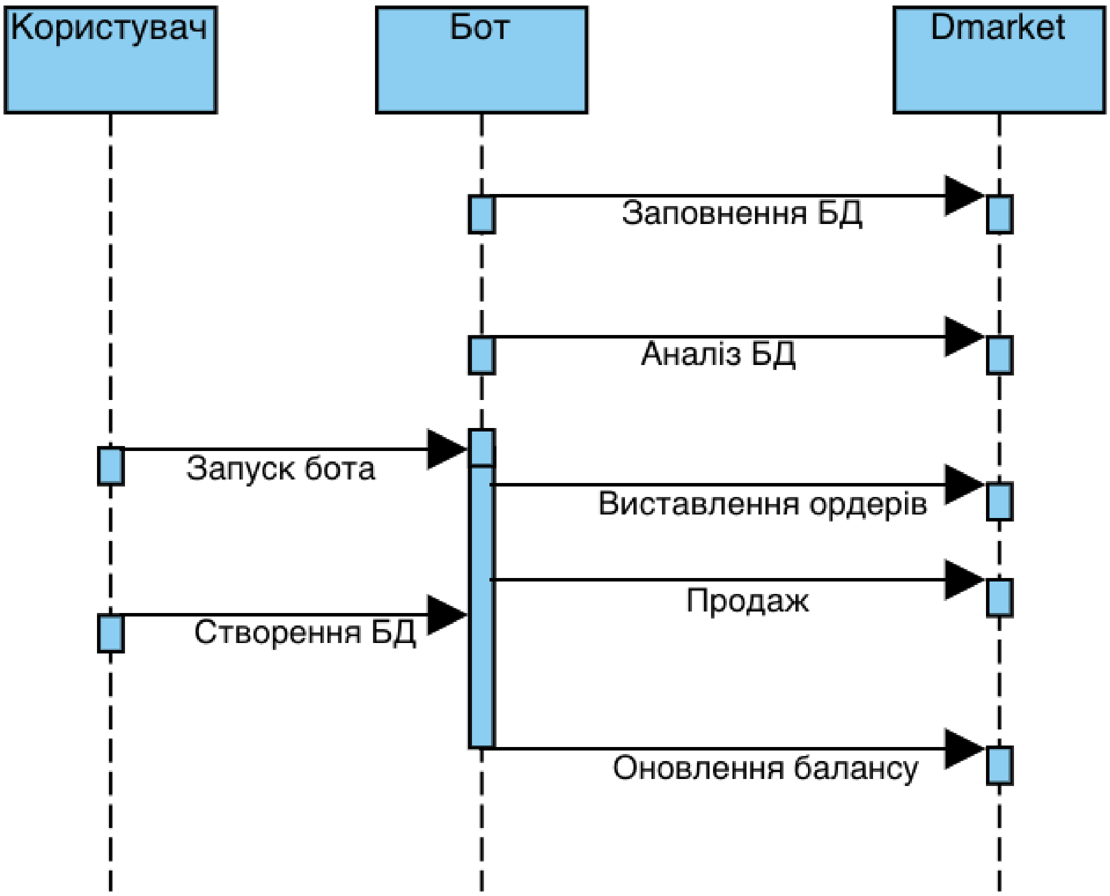
Бот для автоматизації торгівлі віртуальними активами

Алгоритм дій програмного забезпечення

ІАЛЦ.467200.006 ДЗ

Аркушів 1

Київ - 2023



					ІАЛЦ.467200.006 ДЗ			
Змн.	Арк.	№ докум.	Підпис	Дата				
Розроб.		*Литвишко Н.А*			*Бот для автоматизації торгівлі віртуальними активами*	Літ.	Арк.	Акрушів
Перевір.							1	1
					Алгоритм дій програмного забезпечення (принципова схема)	***КПІ ім. Ігоря Сікорського ФІОТ ІП-94***		
Н. Контр.		*Волокита А.М.*						
Затверд.								

ДОДАТОК 4

Бот для автоматизації торгівлі віртуальними активами

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 35

Київ - 2023

main.py

```
import asyncio
import logging as logger
from config import *
from config import PUBLIC_KEY, PRIVATE_KEY
from api.market import Market
from components.skin_analyzer import SkinAnalyzer
from components.buy_skins import OrderManager
from components.sell_skins import MarketHistory, SellOffers

market_api = Market(PUBLIC_KEY, PRIVATE_KEY)
market_history = MarketHistory(market_api)
skin_analyzer = SkinAnalyzer(market_api)
orders_manager = OrderManager(market_api)
sell_offers = SellOffers(market_api)

async def process_history():
    while True:
        try:
            await market_history.update_skin_history()
            await asyncio.sleep(60*15)
        except Exception as e:
            logger.error(f' Failed to get history: {e}. Sleep 10 seconds.')
            await asyncio.sleep(30)

async def process_orders():
    await asyncio.sleep(5)
    while True:
        try:
            logger.debug(f'{market_api.balance}')
            if market_api.balance > orders_manager.order_analyzer.min_buy_price +
BuyConfig.LIMIT_ORDERS_BALANCE:
                await orders_manager.manage_orders()
                await asyncio.sleep(5)
            else:
                user_targets = await
orders_manager.market_api.get_user_targets(limit='1000')
```

```
            inactive_targets = await
orders_manager.market_api.get_user_targets(limit='1000',
status='TargetStatusInactive')
            await orders_manager.market_api.delete_user_targets(user_targets.Items
+ inactive_targets.Items)
            logger.debug('There is not enough money to place orders, we postpone
analytics')
            await asyncio.sleep(60 * 5)
      except Exception as e:
         logger.error(f' Failed to get order database: {e}. Sleep 30 seconds.')
         await asyncio.sleep(5)

async def process_sell_offers():
   while True:
      try:
         await sell_offers.create_sell_offers()
         await asyncio.sleep(60*10)
      except Exception as e:
         logger.error(f' Failed to put up for sale: {e}. Sleep 10 seconds.')
         await asyncio.sleep(30)

async def process_update_offers():
   while True:
      try:
         await sell_offers.update_active_offers()
         await asyncio.sleep(5)
      except Exception as e:
         logger.error(f' Failed to update sellable items: {e}. Sleep 10 seconds.')
         await asyncio.sleep(30)

async def process_main_item_database():
   while True:
      logger.info('Skin database processing')
      try:
         await skin_analyzer.update_skin_base()
         await asyncio.sleep(skin_analyzer.base_update_interval)
      except Exception as e:
         logger.exception(f' Failed to update primary database: {e}. Sleep 5
seconds.')
```

```
        await asyncio.sleep(5)

async def all_processes():
    processes = await asyncio.gather(
        market_api.process_check_money(),
        process_history(),
        process_orders(),
        process_sell_offers(),
        process_update_offers(),
        process_main_item_database(), return_exceptions=True
        )
    return processes

def main():
    try:
        logger.info('Bot start')
        event_loop = asyncio.get_event_loop()
        event_loop.run_until_complete(all_processes())
    except KeyboardInterrupt:
        asyncio.run(market_api.close())
        logger.info('Bot stop')

if __name__ == '__main__':
    main()
```

config.py

```
import sys
from loguru import logger
from api.interfaces import Games

logger_config = {
    "handlers": [
        {"sink": sys.stderr, 'colorize': True, 'level': 'INFO'},
        {"sink": "log/info.log", "serialize": False, 'level': 'INFO'},
    ]
}
logger.configure(**logger_config)

PUBLIC_KEY =
'58c4f56da6236deafe82d42df534569e72ae278b54fe378310e571be9c505bc2'
```

```
PRIVATE_KEY =
'c78506e47f8babafd25beef985ec6fed333bbe0fc0e31787cfc592a97fed136458c4f5
6da6236deafe82d42df534569e72ae278b54fe378310e571be9c505bc2'

API_URL = "https://api.dmarket.com"

GAMES_FOR_TRADE = [Games.CS]
DB_NAME = '/skins_1.db'

INVALID_SKINS = ['key', 'pin', 'sticker', 'case', 'operation', 'pass', 'capsule',
'package', 'challengers',
            'patch', 'music', 'kit', 'graffiti']

class TimeControl:
    PREVIOUS_BASE = 60 * 60 * 5
    ORDER_BASE = 60 * 10

class PriorParams:
    POPULAR_LEVEL = 3
    MIN_AVERAGE_PRICE = 50
    MAX_AVERAGE_PRICE = 299

class BuyConfig:
    LIMIT_ORDERS_BALANCE = 10
    HIGH_FREQUENCY_TRADING = True
    MIN_PRICE = 10
    MAX_PRICE = 300

    DESIRED_PROFIT_PERCENTAGE = 0.1
    POSITIVE_POINTS_PERCENTAGE = 15
    AVERAGE_RECENT_SALES_PRICE = 7

    TOTAL_SALES = 80
    NUMBER_OF_DAYS = 23
    NUMBER_OF_SALES = 11
    LAST_SALE_DAYS_AGO = 3
    FIRST_SALE_DAYS_AGO = 20

    MAX_ON_SALE_OFFERS = 20
```

```
    ABNORMAL_PRICE_PERCANTAGE = 25
    NUMBER_OF_ABNORMAL_PRICE_POINTS = 3

    UPPER_THRESHOLD_PERCENTAGE = 0.1
    LOWER_THRESHOLD_PERCENTAGE = 3

class SellConfig:
    MIN_PROFIT_PERCENTAGE = 0.1
    MAX_PROFIT_PERCENTAGE = 5
```

helpers.py

```
from pyti.simple_moving_average import simple_moving_average as sma
from typing import List
from api.interfaces import LastSale

def calculate_moving_average(history: List[LastSale]) -> list:
    cost = [i.Price.Amount for i in history]
    cost.reverse()
    averaged_values = [i for i in list(sma(cost, 5))]
    averaged_values.reverse()
    return averaged_values
```

sell_skins.py

```
import datetime
from typing import List
from db.orm import SkinOfferManager, SkinOffer
from api.interfaces import SellOffer, CreateOffer, CreateOffers, LastPrice,
EditOffer, EditOffers, \
    DeleteOffers, DeleteOffer, OfferDetails
from api.market import Market
from config import PUBLIC_KEY, PRIVATE_KEY, SellConfig, logger,
GAMES_FOR_TRADE
from time import time

class MarketHistory:
    def __init__(self, bot: Market):
        self.bot = bot
```

```
    @staticmethod
    def get_unsold_skins() -> List[SkinOffer]:
        skins = SkinOfferManager.get_all_skins()
        if skins:
            return [i for i in skins if not i.sellTime]
        return list()

    async def update_skin_history(self):
        get_user_buy_history = await self.bot.get_user_targets_closed(limit='100')
        get_user_buy_history = get_user_buy_history.Trades
        get_user_buy_history = [SellOffer(AssetID=i.AssetID,
buyPrice=i.Price.Amount) for i in get_user_buy_history]
        sold_skins = []
        for game in GAMES_FOR_TRADE:
            get_user_sell_history = await
self.bot.get_user_offers(status='OfferStatusSold', game=game, limit='100')
            get_user_sell_history = get_user_sell_history.Items
            sold_skins += get_user_sell_history
        get_user_sell_history = [SellOffer(AssetID=i.AssetID,
OfferID=i.Offer.OfferID,
                                 sellPrice=i.Offer.Price.Amount,
sellTime=datetime.datetime.now(),
                                 title=i.Title, game=i.GameID) for i in sold_skins]
        get_owned_asset_ids = [i.AssetID for i in SkinOfferManager.get_all_skins()]
        for i in get_user_buy_history:
            if i.AssetID not in get_owned_asset_ids:
                SkinOfferManager.add_skin(i)
        unsold_skins = self.get_unsold_skins()
        for skin in unsold_skins:
            for i in get_user_sell_history:
                if skin.AssetID == i.AssetID:
                    skin.title = i.title
                    skin.sellPrice = i.sellPrice * (1 - skin.fee / 100)
                    skin.OfferID = i.OfferID
                    skin.sellTime = i.sellTime
                    skin.game = i.game
                    break
        SkinOfferManager.update_sold_skins(unsold_skins)
```

```
class SellOffers:
    def __init__(self, bot: Market):
        self.bot = bot
        self.min_profit = SellConfig.MIN_PROFIT_PERCENTAGE
        self.max_profit = SellConfig.MAX_PROFIT_PERCENTAGE

    async def create_sell_offers(self):
        unsold_skins = SkinOfferManager.get_unsold_skins()
        inventory_skins = []
        items_to_sell = []
        for game in GAMES_FOR_TRADE:
            inventory_items = await self.bot.get_user_items(game=game)
            inventory_skins += inventory_items.objects
        for i in inventory_skins:
            fee = 3
            if 'custom' in i.fees['dmarket']['sell']:
                fee = int(i.fees['dmarket']['sell']['custom']['percentage'])
            if i.inMarket:
                items_to_sell.append(SellOffer(AssetID=i.itemId, title=i.title,
game=i.gameId, fee=fee))
        offers_to_create = []
        for i in items_to_sell:
            for j in unsold_skins:
                if i.AssetID == j.AssetID:
                    price = j.buyPrice * (1 + self.max_profit / 100 + i.fee / 100)
                    i.sellPrice = price
            try:
                offers_to_create.append(CreateOffer(AssetID=i.AssetID,
                                                    Price=LastPrice(Currency='USD',
Amount=round(i.sellPrice, 2))))
            except TypeError:
                pass
        add_result = await
self.bot.create_user_offers(CreateOffers(Offers=offers_to_create))
        if add_result.Result:
            for i in add_result.Result:
                for j in items_to_sell:
                    if i.CreateOffer.AssetID == j.AssetID:
```

```
                j.sellPrice = i.CreateOffer.Price.Amount
                j.OfferID = i.OfferID
                SkinOfferManager.update_skin_offer_id(j)
        logger.debug(f'Add to sell: {add_result}')

    @staticmethod
    def calculate_sell_price(max_price, min_price, current_price) -> float:
        if current_price < min_price:
            offer_price = min_price
        elif min_price < current_price <= max_price:
            offer_price = current_price - 0.01
        else:
            offer_price = max_price
        return offer_price

    async def update_active_offers(self):
        active_offers = sorted([i for i in SkinOfferManager.get_unsold_skins() if
i.OfferID],
                               key=lambda x: x.title)
        offers_to_update = list()
        for i in active_offers:
            itemid = OfferDetails(items=[i.AssetID])
            details = await self.bot.details_user_offers(request_body=itemid)
            min_listed_price = details.objects[0].minListedPrice.amount / 100
            if i.sellPrice != min_listed_price:
                max_sell_price = i.buyPrice * (1 + self.max_profit / 100 + i.fee / 100)
                min_sell_price = i.buyPrice * (1 + self.min_profit / 100 + i.fee / 100)
                updated_sell_price = self.calculate_sell_price(max_sell_price,
min_sell_price, min_listed_price)
                if round(updated_sell_price, 2) != round(i.sellPrice, 2):
                    i.sellPrice = updated_sell_price
                    offers_to_update.append(EditOffer(OfferID=i.OfferID,
AssetID=i.AssetID,
                                                      Price=LastPrice(Currency='USD',
Amount=round(i.sellPrice, 2))))

        updated_offers = await
self.bot.edit_user_offers(EditOffers(Offers=offers_to_update))
        for i in updated_offers.Result:
```

```
         for j in active_offers:
             if i.EditOffer.AssetID == j.AssetID:
                 j.sellPrice = i.EditOffer.Price.Amount
                 j.OfferID = i.NewOfferID
                 logger.debug(f'{i.EditOffer.AssetID} {j.AssetID}')
                 SkinOfferManager.update_skin_offer_id(j)
     logger.debug(f'UPDATE OFFERS: {updated_offers}')

 async def remove_all_active_offers(self):
     active_offers = await self.bot.get_user_offers(status='OfferStatusActive')
     offers_to_remove = [DeleteOffer(itemId=i.AssetID, offerId=i.Offer.OfferID,
price=i.Offer.Price) for i in active_offers.Items]
     await self.bot.delete_user_offers(DeleteOffers(objects=offers_to_remove))
```

skin_analyzer.py

```
import datetime
from itertools import groupby
from time import time
from pydantic import ValidationError
from typing import List, Union

from api.market import Market
from db.orm import SkinManager
from api.interfaces import MarketOffer, Games, SkinHistory
from config import logger, PriorParams, BuyConfig, TimeControl,
INVALID_SKINS, GAMES_FOR_TRADE

class SkinAnalyzer:
    def __init__(self, bot: Market):
        self.bot = bot
        self.base_update_interval = TimeControl.PREVIOUS_BASE
        self.min_average_price = PriorParams.MIN_AVERAGE_PRICE
        self.max_average_price = PriorParams.MAX_AVERAGE_PRICE
        self.skin_manager = SkinManager()
        self.min_buy_price = BuyConfig.MIN_PRICE
        self.max_buy_price = BuyConfig.MAX_PRICE

    @staticmethod
    def is_valid_skin_name(item_name: str):
        for i in INVALID_SKINS:
```

```
            if i in item_name.lower():
                return False
        return True

    async def fetch_market_items(self, min_avg_price: int, max_avg_price: int,
game: Games) -> List[MarketOffer]:
        market_items = await self.bot.get_market_items(price_from=min_avg_price,
price_to=max_avg_price, game=game)
        cursor = market_items.cursor
        while cursor:
            another_market_offers = await
self.bot.get_market_items(price_from=min_avg_price, price_to=max_avg_price,
                                                    cursor=cursor, game=game)
            market_items.objects += another_market_offers.objects
            cursor = another_market_offers.cursor

            if len(market_items.objects) > 10000:
                break
        market_items.objects = sorted(market_items.objects, key=lambda x: x.title)
        skins = [list(group)[0] for _, group in groupby(market_items.objects, lambda
x: x.title)]
        return [i for i in skins if self.is_valid_skin_name(i.title)]

    async def select_filtered_skins(self, skins: List[Union[MarketOffer,
SkinHistory]], min_avg_price: int, max_avg_price: int) -> \
            List[SkinHistory]:
        selected_skins = list()
        counter = 0
        for i in skins:
            if isinstance(i, MarketOffer):
                game = Games(i.gameId)
            else:
                game = Games(i.game)
            try:
                last_sales = await self.bot.get_last_sales(i.title, game=game)
                if len(last_sales.LastSales) == 20:
                    cost = [i.Price.Amount for i in last_sales.LastSales]
                    avg_price = sum(cost)
                    quantity = len(cost)
```

```
                    avg_price = avg_price / quantity
                    if min_avg_price <= avg_price <= max_avg_price:
                        try:
                            skin_history = SkinHistory(title=i.title, game=game.value,
LastSales=last_sales.LastSales,
                                                   avg_price=avg_price,
update_time=datetime.datetime.now())
                            selected_skins.append(skin_history)
                        except ValidationError as e:
                            logger.error(e.json())
            except Exception as e:
                logger.error(f'Exception in skin base{e}')
            if counter % 500 == 0:
                logger.debug(f'Game {game}. {counter} skins analyzed.')
            counter += 1
        return selected_skins

    async def refresh_data(self):
        refreshed_skins = list()
        for game in GAMES_FOR_TRADE:
            market_items = await self.fetch_market_items(self.min_average_price,
self.max_average_price, game)
            logger.debug(game)
            market_items = [i for i in market_items if not
self.skin_manager.is_skin_exists(i)]
            refreshed_skins += await self.select_filtered_skins(market_items,
self.min_average_price, self.max_average_price)
        self.skin_manager.add_all_skins(refreshed_skins)
        logger.info(f'Total items analyzed: {len(refreshed_skins)}')

    async def update_skin_base(self):
        timenow = time()
        await self.refresh_data()
        updated_skins = [i for i in
self.skin_manager.get_skins_updated_before(timenow, self.base_update_interval)
                         if self.min_buy_price < i.avg_price < self.max_buy_price]
        if not updated_skins:
            logger.info('No skins to update')
```

```
        selected_skins = await self.select_filtered_skins(updated_skins,
self.min_average_price, self.max_average_price)
        self.skin_manager.update_or_create_skins_by_title(selected_skins)
        logger.info(f'Skins database updated {round((time() - timenow) / 60, 2)}
minutes.')
```

buy_skins.py

```
from itertools import groupby
from api.market import Market
from time import time
from db.orm import SkinManager
from config import logger, BuyConfig, TimeControl, GAMES_FOR_TRADE
from typing import List, Tuple
from api.interfaces import SkinHistory, SkinOrder, Target, CreateTarget, \
    CreateTargets, LastPrice, TargetAttributes, CumulativePrice
import math
from components.helpers import calculate_moving_average
from config import INVALID_SKINS
from api.errors import RequestLimitException

class OrderAnalyzer:
    def __init__(self, bot: Market):
        self.bot = bot

        self.frequency_trading_on = BuyConfig.HIGH_FREQUENCY_TRADING

        self.max_buy_price = BuyConfig.MAX_PRICE
        self.min_buy_price = BuyConfig.MIN_PRICE

        self.recent_sales_price = BuyConfig.AVERAGE_RECENT_SALES_PRICE
        self.acceptable_profit = BuyConfig.DESIRED_PROFIT_PERCENTAGE
        self.pos_points = BuyConfig.POSITIVE_POINTS_PERCENTAGE
        self.first_sale_days_ago = BuyConfig.FIRST_SALE_DAYS_AGO
        self.last_sale_days_ago = BuyConfig.LAST_SALE_DAYS_AGO
        self.number_of_days = BuyConfig.NUMBER_OF_DAYS
        self.number_of_sales = BuyConfig.NUMBER_OF_SALES
        self.max_on_sale_offers = BuyConfig.MAX_ON_SALE_OFFERS

        self.abnormal_price = BuyConfig.ABNORMAL_PRICE_PERCANTAGE
```

```
        self.abnormal_points =
BuyConfig.NUMBER_OF_ABNORMAL_PRICE_POINTS

        self.upper_threshold = BuyConfig.UPPER_THRESHOLD_PERCENTAGE
        self.lower_threshold = BuyConfig.LOWER_THRESHOLD_PERCENTAGE

    def filter_skins_by_sales_date_range(self, skins: List[SkinHistory]) ->
List[SkinHistory]:
        filtered_skins = list()
        for i in skins:
            sales_list = list()
            first_sale_days_ago = int(i.LastSales[-1].Date.timestamp())
            last_sale_days_ago = int(i.LastSales[0].Date.timestamp())
            if first_sale_days_ago < (time() - self.first_sale_days_ago * 60 * 60 * 24):
                if last_sale_days_ago > (time() - self.last_sale_days_ago * 60 * 60 * 24):
                    for j in i.LastSales:
                        if int(j.Date.timestamp()) > (time() - self.number_of_days * 60 * 60
* 24):
                            sales_list.append(j)
                    if len(sales_list) >= self.number_of_sales:
                        filtered_skins.append(i)
        return filtered_skins

    def control_boosted_sales(self, skins: List[SkinHistory]) -> List[SkinHistory]:
        filtered_skins = list()
        for item in skins:
            moving_average = calculate_moving_average(item.LastSales)
            number_of_abnormal_points = 0
            try:
                for i in range(len(moving_average[:-4])):
                    if item.LastSales[i].Price.Amount > \
                            moving_average[i] * (1 + self.abnormal_price / 100):
                        item.LastSales.pop(i)
                        number_of_abnormal_points += 1
                if number_of_abnormal_points <= self.abnormal_points:
                    filtered_skins.append(item)
            except IndexError:
                pass
        return filtered_skins
```

```
    async def filter_profitable_skins(self, skins: List[SkinHistory]) ->
List[SkinOrder]:
        filtered_skins = list()
        sorted_skins = sorted(skins, key=lambda x: x.title)
        titles_list = [i.title for i in sorted_skins]
        agr_prices = await self.bot.get_agregated_prices(titles_list)
        sorted_agr_prices = sorted(agr_prices, key=lambda x: x.MarketHashName)

        for item, prices in zip(sorted_skins, sorted_agr_prices):
            best_order_price = prices.Orders.BestPrice*100
            number_of_points = math.ceil(len(item.LastSales) / 100 * self.pos_points)
            counter = 0
            for i in item.LastSales:
                price_with_fee = i.Price.Amount * (1 - self.bot.SELL_FEE/100)
                if price_with_fee > best_order_price * (1 + self.acceptable_profit / 100):
                    counter += 1
            if counter >= number_of_points:
                if prices.Offers.Count <= self.max_on_sale_offers:
                    filtered_skins.append(SkinOrder(title=item.title,
bestOrder=int(best_order_price), game=item.game))
        return filtered_skins

    @staticmethod
    def get_first_and_second_price(info: List[CumulativePrice]) -> tuple:
        number_of_offers = len(info)
        if number_of_offers == 0:
            price_of_the_best_offer = 0
            price_of_second_best_offer = 0
        else:
            the_best_offer = info[0]
            if number_of_offers == 1:
                second_best_offer = the_best_offer
            else:
                if the_best_offer.Amount == 1:
                    second_best_offer = info[1]
                else:
                    second_best_offer = the_best_offer
            price_of_the_best_offer = the_best_offer.Price
```

```
            price_of_second_best_offer = second_best_offer.Price
        return price_of_the_best_offer, price_of_second_best_offer,
number_of_offers

    async def calculate_market_offers_profit(self, skin: SkinHistory):
        price_levels = await self.bot.get_cumulative_price_levels(skin.title,
skin.game)
        recent_sales = skin.LastSales[0:self.recent_sales_price]
        average_price = sum([s.Price.Amount/100 for s in
recent_sales])/len(recent_sales)
        top_offer_price, second_offer, total_offers =
self.get_first_and_second_price(price_levels.Offers)
        top_target_price, second_target, total_targets =
self.get_first_and_second_price(price_levels.Targets)
        if top_offer_price == 0 or (top_target_price -
second_target)/top_offer_price*100 > 3:
            top_target_price = second_target
        if second_offer == 0 or (second_offer - top_offer_price)/second_offer*100 >
3:
            top_offer_price = second_offer
        profit_margin = -(top_target_price - (1 - self.bot.SELL_FEE/100) *
top_offer_price) / top_target_price * 100
        avg_profit_margin = -(top_target_price - (1 - self.bot.SELL_FEE/100) *
average_price) / top_target_price * 100
        return top_offer_price, top_target_price, total_offers, total_targets,
profit_margin, round(avg_profit_margin, 2)

    async def filter_frequency_skins(self, skins: List[SkinHistory]) ->
List[SkinOrder]:
        filtered_skins = list()
        skins = sorted(skins, key=lambda x: x.title)
        for item in skins:
            top_offer_price, top_target_price, total_offers, total_targets, profit_margin,
avg_profit_margin = \
                await self.calculate_market_offers_profit(item)
            if avg_profit_margin > self.acceptable_profit and profit_margin >
self.acceptable_profit:
                desired_sell_price = top_target_price * (1 + self.acceptable_profit / 100)
                count = 0
```

```
                points_count = math.ceil(len(item.LastSales) / 100 * self.pos_points)
                for i in item.LastSales:
                    price_with_fee = i.Price.Amount * (1 - self.bot.SELL_FEE / 100)
                    if price_with_fee > desired_sell_price:
                        count += 1
                if count >= points_count:
                    if total_offers <= self.max_on_sale_offers:
                        filtered_skins.append(SkinOrder(title=item.title,
bestOrder=int(top_target_price*100), game=item.game))
        return filtered_skins

    async def process_skins_analysis(self) -> List[SkinOrder]:
        timestamp = time()
        analyzed_skins = list()
        available_skins = []
        for game in GAMES_FOR_TRADE:
            available_skins += [i for i in SkinManager.get_all_skins() if
self.min_buy_price < i.avg_price < self.max_buy_price
                      and i.game == game.value]
        if available_skins:
            logger.info(f'NUMBER OF SKINS {len(available_skins)}')
        if available_skins:
            available_skins = self.filter_skins_by_sales_date_range(available_skins)
            available_skins = self.control_boosted_sales(available_skins)
            if self.frequency_trading_on:
                available_skins = await self.filter_frequency_skins(available_skins)
            else:
                available_skins = await self.filter_profitable_skins(available_skins)
            logger.info(f'REMAINING AFTER ANALYSIS BY PARAMETERS
{len(available_skins)}')
            for skin in available_skins:
                skin.maxPrice = int(skin.bestOrder*(1 + self.upper_threshold/100))
                skin.minPrice = int(skin.bestOrder*(1 - self.lower_threshold/100))
                analyzed_skins.append(skin)
        logger.debug(f'The database of orders was updated {round(time() -
timestamp, 2)} sec.')
        return analyzed_skins

class OrderManager:
```

```
    def __init__(self, bot: Market):
        self.bot = bot
        self.order_analyzer = OrderAnalyzer(self.bot)

    @staticmethod
    def calculate_optimal_price(min_price, max_price, best_price):
        if best_price > max_price:
            order_price = max_price
        elif min_price < best_price <= max_price:
            order_price = best_price + 1
        else:
            order_price = min_price
        return order_price

    @staticmethod
    def separate_targets(skins: List[SkinOrder], targets: List[Target]) ->
Tuple[List[SkinOrder], List[Target], List[Target]]:
        valid_targets = [i for i in targets if i.Title in [s.title for s in skins]]
        invalid_targets = [i for i in targets if i.Title not in [s.title for s in skins]]
        remaining_skins = [i for i in skins if i.title not in [s.Title for s in targets]]
        return remaining_skins, valid_targets, invalid_targets

    async def place_market_orders(self, item: SkinOrder):
        market_offer = await self.bot.get_market_items(title=item.title, limit=1,
game=item.game)
        if market_offer.objects and market_offer.objects[0].title == item.title:
            market_offer = market_offer.objects[0]
            last_price = LastPrice(Currency='USD', Amount=item.bestOrder/100)
            target_attributes = [TargetAttributes(Name='name',
Value=market_offer.extra.name),
                                 TargetAttributes(Name='title', Value=market_offer.title),
                                 TargetAttributes(Name='category',
Value=market_offer.extra.category),
                                 TargetAttributes(Name='gameId', Value=market_offer.gameId),
                                 TargetAttributes(Name='categoryPath',
Value=market_offer.extra.categoryPath),
                                 TargetAttributes(Name='image', Value=market_offer.image)]
            if market_offer.extra.exterior:
```

```
                target_attributes.append(TargetAttributes(Name='exterior',
Value=market_offer.extra.exterior))
            create_target = CreateTarget(Amount='1', Price=last_price,
Attributes=target_attributes)
            create_targets = CreateTargets(Targets=[create_target])
            created_order = await self.bot.create_user_targets(create_targets)
            return created_order
        return []

    async def verify_market_offers(self, item: SkinOrder):
        market_offers = await self.bot.get_offers_by_title(title=item.title, limit=3)
        market_offers = sorted(market_offers.objects, key=lambda x:
int(x.price.USD))
        offer_price_list = [i.price.USD for i in market_offers]
        desired_sell_price = item.bestOrder * (1 +
self.order_analyzer.acceptable_profit / 100)
        if any([desired_sell_price <= p for p in offer_price_list]):
            return True
        return False

    async def manage_orders(self):
        timestamp = time()
        analyzed_skins = await self.order_analyzer.process_skins_analysis()
        user_targets = await self.bot.get_user_targets(limit='1000')
        grouped_titles = [list(j) for _, j in groupby(user_targets.Items, key=lambda x:
x.Title)]
        inactive_targets = await self.bot.get_user_targets(limit='1000',
status='TargetStatusInactive')
        remaining_skins, valid_targets, invalid_targets =
self.separate_targets(analyzed_skins, user_targets.Items)
        for title in grouped_titles:
            if len(title) > 1:
                invalid_targets += title[1:]
        await self.bot.delete_user_targets(invalid_targets + inactive_targets.Items)
        for skin in remaining_skins:
            if self.bot.balance > skin.bestOrder:
                for i in INVALID_SKINS:
                    if i in skin.title.lower():
                        continue
```

```
                if await self.verify_market_offers(skin):
                    await self.place_market_orders(skin)
        if valid_targets:
            for i in valid_targets:
                for j in analyzed_skins:
                    if i.Title == j.title:
                        if i.Price.Amount*100 != j.bestOrder:
                            order_price = self.calculate_optimal_price(j.minPrice,
j.maxPrice, j.bestOrder)
                            j.bestOrder = order_price
                            if await self.verify_market_offers(j):
                                await self.bot.delete_user_targets([i])
                                await self.place_market_orders(j)

        logger.debug(f'Orders were updated {round(time() - timestamp, 2)} sec.')
```

orm.py

```
from typing import List
from datetime import datetime
from peewee import DoesNotExist

from db.schemas import Skin, SkinOffer, db
from api.interfaces import SkinHistory, MarketOffer, SellOffer

db.connect()
Skin.create_table()
SkinOffer.create_table()
db.close()

class SkinManager:
    @staticmethod
    def add_all_skins(items: List[SkinHistory]):
        skins = [Skin(**i.dict()) for i in items]
        with db.atomic():
            Skin.bulk_create(skins, batch_size=500)

    @staticmethod
    def is_skin_exists(item: MarketOffer):
```

```
        skin = Skin.select().where(Skin.title == item.title)
        if skin:
            return True
        return False

    @staticmethod
    def update_or_create_skins_by_title(items: List[SkinHistory]):
        updated_skins = list()
        created_skins = list()
        for item in items:
            try:
                skin = Skin.get(Skin.title == item.title)
                it = item.dict()
                skin.avg_price = it['avg_price']
                skin.LastSales = it['LastSales']
                skin.update_time = it['update_time']
                updated_skins.append(skin)
            except DoesNotExist:
                created_skins.append(Skin(**item.dict()))
        with db.atomic():
            Skin.bulk_update(updated_skins,
                             fields=[Skin.avg_price, Skin.LastSales, Skin.update_time],
                             batch_size=500)
        with db.atomic():
            Skin.bulk_create(created_skins, batch_size=500)

    @staticmethod
    def get_all_skins() -> List[SkinHistory]:
        skins = Skin.select()
        return [SkinHistory.from_orm(skin) for skin in skins]

    @staticmethod
    def get_skins_updated_before(now, delta) -> List[SkinHistory]:
        skins = Skin.select().where(Skin.update_time < datetime.fromtimestamp(now
- delta))
        if skins:
            return [SkinHistory.from_orm(skin) for skin in skins]
        return []
```

```
class SkinOfferManager:

    @staticmethod
    def add_skin(item: SellOffer) -> None:
        new_skin = SkinOffer.create(
            title=item.title,
            game=item.game,
            AssetID=item.AssetID,
            buyPrice=item.buyPrice,
            buyTime=item.buyTime,
            OfferID=item.OfferID,
            sellTime=item.sellTime,
            sellPrice=item.sellPrice
        )
        new_skin.save()

    @staticmethod
    def update_sold_skins(skins: List[SkinOffer]):
        if not skins:
            return
        with db.atomic():
            SkinOffer.bulk_update(skins, fields=[SkinOffer.title, SkinOffer.sellPrice,
                                              SkinOffer.sellTime, SkinOffer.OfferID])

    @staticmethod
    def get_unsold_skins() -> List[SellOffer]:
        skins = SkinOffer.select().where(SkinOffer.sellTime == None)
        return [SellOffer.from_orm(s) for s in skins]

    @staticmethod
    def get_all_skins() -> List[SkinOffer]:
        skins = SkinOffer.select()
        return skins

    @staticmethod
    def clear_all_skins():
        skins = SkinOffer.select()
        for s in skins:
            s.delete_instance()
```

```
    @staticmethod
    def modify_skin_by_asset(skin: SellOffer):
        try:
            item = SkinOffer.get(SkinOffer.AssetID == skin.AssetID)
            item.OfferID = skin.OfferID
            item.sellTime = skin.sellTime
            item.sellPrice = skin.sellPrice
            item.save()
        except DoesNotExist:
            pass

    @staticmethod
    def update_skin_offer_id(skin: SellOffer):
        try:
            item = SkinOffer.get(SkinOffer.AssetID == skin.AssetID)
            item.OfferID = skin.OfferID
            item.title = skin.title
            item.fee = skin.fee
            item.sellPrice = skin.sellPrice
            # item.sell_time = skin.sell_time
            # item.sell_price = skin.sell_price
            # item.update_time = skin.update_time
            item.save()
        except DoesNotExist:
            pass
```

init_db.py

```
from pathlib import Path
from playhouse.sqlite_ext import SqliteExtDatabase
from config import DB_NAME

DB_PATH = str(Path(__file__).resolve().parent) + DB_NAME
db = SqliteExtDatabase(DB_PATH, check_same_thread=False)
```

market.py

```
import asyncio
```

```
import aiohttp
import requests
import json
from datetime import datetime
from asyncio import CancelledError
from typing import List, Union
from furl import furl
from nacl.bindings import crypto_sign
from config import API_URL, logger
from api.errors import *
from api.interfaces import Balance, Games, LastSales, SalesHistory, MarketOffers, 
AggregatedTitle, \
    UserTargets, ClosedTargets, Target, UserItems, CreateOffers, 
CreateOffersResponse, EditOffers, EditOffersResponse, \
    DeleteOffers, CreateTargets, CumulativePrices, OfferDetails, 
OfferDetailsResponse

class Market:
    def __init__(self, public_key: str, private_key: str):
        self.PUBLIC_KEY = public_key
        self.PRIVATE_KEY = private_key
        self.SELL_FEE = 3
        self.balance = 0
        self.session = aiohttp.ClientSession()

    async def close(self):
        return await self.session.close()

    def generate_request_headers(self, http_method: str, api_path: str, 
query_params: dict = None, request_body: dict = None) -> dict:
        timestamp = str(round(datetime.now().timestamp()))
        request_string = http_method + api_path
        request_string = str(furl(request_string).add(query_params))
        if request_body:
            request_string += json.dumps(request_body)
        request_string += timestamp
        signature_prefix = "dmar ed25519 "
        encoded_request = request_string.encode('utf-8')
        private_key_bytes = bytes.fromhex(self.PRIVATE_KEY)
```

```
        signature_bytes = crypto_sign(encoded_request, private_key_bytes)
        signature = signature_bytes[:64].hex()
        headers = {
            "X-Api-Key": self.PUBLIC_KEY,
            "X-Request-Sign": signature_prefix + signature,
            "X-Sign-Date": timestamp
        }
        return headers

    @staticmethod
    def handle_error(http_status: int, headers: dict, response_msg: str):
        if http_status == 400:
            raise InvalidRequestException()
        if http_status == 502 or http_status == 500:
            raise GatewayErrorException()
        if http_status == 429:
            raise RequestLimitException()
        if http_status == 401:
            raise InvalidAPIKeyException()
        if http_status != 200 and 'application/json' not in headers['content-type']:
            raise InvalidResponseException(response_msg)

    async def verify_response(self, response: aiohttp.ClientResponse or
requests.Response) -> dict:
        headers = dict(response.headers)
        if 'RateLimit-Remaining' not in headers:
            await asyncio.sleep(5)
        if 'RateLimit-Remaining' in headers and headers['RateLimit-Remaining'] in
['1', '0']:
            await asyncio.sleep(int(headers['RateLimit-Reset']))
        if isinstance(response, requests.Response):
            http_status = response.status_code
            self.handle_error(http_status, headers, response.text)
            request_body = response.json()
        else:
            http_status = response.status
            self.handle_error(http_status, headers, response.text)
            request_body = await response.json()
```

```
        return request_body

    async def perform_api_call(self, url: str, http_method: str, headers: dict, params: dict = None, request_body: dict = None,
                               is_async: bool = True) -> dict:
        if not is_async:
            response = requests.get(url, params=params, headers=headers)
            await asyncio.sleep(0.001)
            return await self.verify_response(response)
        if http_method == 'GET':
            async with self.session.get(url, params=params, headers=headers) as response:
                data = await self.verify_response(response)
                return data
        elif http_method == 'DELETE':
            async with self.session.delete(url, params=params, json=request_body, headers=headers) as response:
                data = await self.verify_response(response)
                return data
        else:
            async with self.session.post(url, params=params, json=request_body, headers=headers) as response:
                data = await self.verify_response(response)
                return data

    async def user(self):
        http_method = 'GET'
        path = '/account/v1/user'
        headers = self.generate_request_headers(http_method, path)
        base_url = API_URL
        request_url = base_url + path
        response = await self.perform_api_call(request_url, http_method, headers)
        return response

    async def get_balance(self):
        http_method = 'GET'
        path = '/account/v1/balance'
        headers = self.generate_request_headers(http_method, path)
        base_url = API_URL
```

```
        request_url = base_url + path
        response = await self.perform_api_call(request_url, http_method, headers)
        if 'usd' in response:
            self.balance = Balance(**response).usd
            logger.debug(f'Balance: {self.balance}')
            return self.balance
        else:
            logger.debug(f'{response}')

    async def process_check_money(self) -> None:
        while True:
            try:
                await self.get_balance()
                await asyncio.sleep(60*5)
            except KeyboardInterrupt:
                break
            except CancelledError:
                break
            except Exception:
                continue
        return

    async def get_last_sales(self, skin_title: str, game: Games = Games.CS,
currency: str = 'USD') -> LastSales:
        http_method = 'GET'
        query_params = {'GameID': game.value, 'Title': skin_title, 'Currency':
currency}
        path = '/marketplace-api/v1/last-sales'
        headers = self.generate_request_headers(http_method, path, query_params)
        base_url = API_URL
        request_url = base_url + path
        response = await self.perform_api_call(request_url, http_method, headers,
query_params)
        return LastSales(**response)

    async def get_sales_history(self, skin_title: str, game: Games = Games.CS,
currency: str = 'USD',
                            period: str = '1M') -> SalesHistory:
        http_method = 'GET'
```

```
        query_params = {'GameID': game.value, 'Title': skin_title, 'Currency':
currency, 'Period': period}
        path = '/marketplace-api/v1/sales-history'
        headers = self.generate_request_headers(http_method, path, query_params)
        base_url = API_URL
        request_url = base_url + path
        response = await self.perform_api_call(request_url, http_method, headers,
query_params)
        return SalesHistory(**response)

    async def get_market_items(self, game: Games = Games.CS, title: str = '', limit:
int = 100, offset: int = 0,
                               orderby: str = 'price', orderdir: str = 'asc', tree_filters: str = '',
                               currency: str = 'USD', price_from: int = 0, price_to: int = 0, types:
str = 'dmarket',
                               cursor: str = '') -> MarketOffers:
        http_method = 'GET'
        path = '/exchange/v1/market/items'
        query_params = {'gameId': game.value, 'title': title, 'limit': limit, 'orderBy':
orderby, 'currency': currency,
                        'offset': offset, 'orderDir': orderdir, 'treeFilters': tree_filters, 'priceFrom':
price_from,
                        'priceTo': price_to, 'types': types, 'cursor': cursor}
        headers = self.generate_request_headers(http_method, path, query_params)
        base_url = API_URL
        request_url = base_url + path
        response = await self.perform_api_call(request_url, http_method, headers,
query_params)
        return MarketOffers(**response)

    async def get_agregated_prices(self, titles: List[str], limit: int = 100, offset: str =
None) -> List[AggregatedTitle]:
        http_method = "GET"
        path = '/price-aggregator/v1/aggregated-prices'
        if len(titles) > 100:
            additional_skins = await self.get_agregated_prices(titles[100:])
        else:
            additional_skins = []
        query_params = {'Titles': titles[:100], 'Limit': limit}
```

```
        if offset:
            query_params['Offset'] = offset
        headers = self.generate_request_headers(http_method, path, query_params)
        base_url = API_URL
        request_url = base_url + path
        response = await self.perform_api_call(request_url, http_method, headers,
query_params, is_async=False)
        return [AggregatedTitle(**i) for i in response['AggregatedTitles']] +
additional_skins

    async def get_offers_by_title(self, title: str, limit: int = 100, cursor: str = '') ->
MarketOffers:
        http_method = 'GET'
        path = '/exchange/v1/offers-by-title'
        query_params = {'Title': title, 'Limit': limit, 'Cursor': cursor}
        headers = self.generate_request_headers(http_method, path, query_params)
        base_url = API_URL
        request_url = base_url + path
        response = await self.perform_api_call(request_url, http_method, headers,
query_params)
        return MarketOffers(**response)

    async def get_user_targets(self, game: Games = Games.CS, price_from: float =
None, price_to: float = None,
                               title: str = None, target_id: str = None, status: str =
'TargetStatusActive',
                               limit: str = '100', cursor: str = '', currency: str = 'USD'):
        http_method = 'GET'
        path = '/marketplace-api/v1/user-targets'
        query_params = {'BasicFilters.Status': status, 'GameId': game.value,
'BasicFilters.Currency': currency,
                        'Limit': limit}
        if price_from:
            query_params['BasicFilters.PriceFrom'] = price_from
        if price_to:
            query_params['BasicFilters.PriceTo'] = price_to
        if title:
            query_params['BasicFilters.Title'] = title
        if target_id:
```

Змн.	Арк.	№ докум.	Підпис	Дата	ІАЛЦ.467200.007 Д4	Арк. 28

```
            query_params['BasicFilters.TargetID'] = target_id
        if cursor:
            query_params['Cursor'] = cursor
        headers = self.generate_request_headers(http_method, path, query_params)
        base_url = API_URL
        request_url = base_url + path
        response = await self.perform_api_call(request_url, http_method, headers,
query_params)
        return UserTargets(**response)

    async def get_user_targets_closed(self, limit: str = '100', order_dir: str = 'desc') -
> ClosedTargets:
        http_method = 'GET'
        path = '/marketplace-api/v1/user-targets/closed'
        query_params = {'Limit': limit, 'OrderDir': order_dir}
        headers = self.generate_request_headers(http_method, path, query_params)
        base_url = API_URL
        request_url = base_url + path
        response = await self.perform_api_call(request_url, http_method, headers,
query_params)
        return ClosedTargets(**response)

    async def create_user_targets(self, request_body: CreateTargets):
        http_method = 'POST'
        path = '/marketplace-api/v1/user-targets/create'
        headers = self.generate_request_headers(http_method, path,
request_body=request_body.dict())
        base_url = API_URL
        request_url = base_url + path
        response = await self.perform_api_call(request_url, http_method, headers,
request_body=request_body.dict())
        return response

    async def delete_user_targets(self, targets: List[Target]):
        http_method = 'POST'
        path = '/marketplace-api/v1/user-targets/delete'
        if len(targets) > 150:
            additional_skins = await self.delete_user_targets(targets[150:])
        else:
```

```
            additional_skins = []
        targets = [{'TargetID': i.TargetID} for i in targets[:150]]
        request_body = {"Targets": targets}
        headers = self.generate_request_headers(http_method, path,
request_body=request_body)
        base_url = API_URL
        request_url = base_url + path
        response = await self.perform_api_call(request_url, http_method, headers,
request_body=request_body)
        return response['Result'] + additional_skins

    async def get_cumulative_price_levels(self, title: str, game: str):
        http_method = 'GET'
        path = '/marketplace-api/v1/cumulative-price-levels'
        query_params = {'Title': title, 'GameID': game}
        headers = self.generate_request_headers(http_method, path, query_params)
        base_url = API_URL
        request_url = base_url + path
        response = await self.perform_api_call(request_url, http_method, headers,
query_params)
        return CumulativePrices(**response)

    async def get_user_inventory(self, game: Games = Games.CS, in_market: bool
= 'true', limit: str = '100') -> UserItems:
        http_method = 'GET'
        path = '/marketplace-api/v1/user-inventory'
        query_params = {'GameID': game.value, 'BasicFilters.InMarket': in_market,
'Limit': limit}
        headers = self.generate_request_headers(http_method, path,
query_params=query_params)
        base_url = API_URL
        request_url = base_url + path
        response = await self.perform_api_call(request_url, http_method, headers,
params=query_params)
        return UserItems(**response)

    async def get_user_items(self, game: Games = Games.CS) -> MarketOffers:
        http_method = 'GET'
        path = '/exchange/v1/user/items'
```

```
        query_params = {'GameId': game.value, 'currency': 'USD', 'limit': '50'}
        headers = self.generate_request_headers(http_method, path, query_params)
        base_url = API_URL
        request_url = base_url + path
        response = await self.perform_api_call(request_url, http_method, headers,
params=query_params)
        return MarketOffers(**response)

    async def get_user_offers(self, game: Games = Games.CS, status: str =
'OfferStatusDefault',
                          sort_type: str = 'UserOffersSortTypeDateNewestFirst', limit: str =
'20'):
        http_method = 'GET'
        path = '/marketplace-api/v1/user-offers'
        query_params = {'GameId': game.value, 'Status': status, 'Limit': limit,
'SortType': sort_type}
        headers = self.generate_request_headers(http_method, path, query_params)
        base_url = API_URL
        request_url = base_url + path
        response = await self.perform_api_call(request_url, http_method, headers,
params=query_params)
        return UserItems(**response)

    async def create_user_offers(self, request_body: CreateOffers):
        http_method = 'POST'
        path = '/marketplace-api/v1/user-offers/create'
        request_body = request_body.dict()
        headers = self.generate_request_headers(http_method, path,
request_body=request_body)
        base_url = API_URL
        request_url = base_url + path
        response = await self.perform_api_call(request_url, http_method, headers,
request_body=request_body)
        return CreateOffersResponse(**response)

    async def edit_user_offers(self, request_body: EditOffers):
        http_method = 'POST'
        path = '/marketplace-api/v1/user-offers/edit'
        request_body = request_body.dict()
```

```
        headers = self.generate_request_headers(http_method, path,
request_body=request_body)
        base_url = API_URL
        request_url = base_url + path
        response = await self.perform_api_call(request_url, http_method, headers,
request_body=request_body)
        return EditOffersResponse(**response)

    async def delete_user_offers(self, request_body: DeleteOffers):
        http_method = 'DELETE'
        path = '/exchange/v1/offers'
        request_body = request_body.dict()
        headers = self.generate_request_headers(http_method, path,
request_body=request_body)
        base_url = API_URL
        request_url = base_url + path
        response = await self.perform_api_call(request_url, http_method, headers,
request_body=request_body)
        return response

    async def details_user_offers(self, request_body: OfferDetails):
        http_method = 'POST'
        path = '/exchange/v1/offers/details'
        request_body = request_body.dict()
        headers = self.generate_request_headers(http_method, path,
request_body=request_body)
        base_url = API_URL
        request_url = base_url + path
        response = await self.perform_api_call(request_url, http_method, headers,
request_body=request_body)
        return OfferDetailsResponse(**response)
```

models.py

```
import datetime
import json
from playhouse.sqlite_ext import Model, CharField, FloatField, TextField, DateTimeField, IntegerField

from db.database import db


def default(o):
    if isinstance(o, (datetime.date, datetime.datetime)):
        return o.isoformat()


class JSONField(TextField):
    def db_value(self, value):
        return json.dumps(value, default=default)

    def python_value(self, value):
        if value is not None:
            return json.loads(value)


class BaseModel(Model):
    class Meta:
        database = db


class Skin(BaseModel):
    title = CharField()
```

```
    game = CharField()
    LastSales = JSONField()
    avg_price = FloatField()
    update_time = DateTimeField()



class SkinOffer(BaseModel):
    title = CharField(null=True)
    AssetID = CharField()
    game = CharField(null=True)
    buyPrice = FloatField(null=True)
    buyTime = DateTimeField(null=True)
    OfferID = CharField(null=True)
    sellTime = DateTimeField(null=True)
    sellPrice = FloatField(null=True)
    fee = IntegerField(default=7)
```

errors. Py

```
from config import logger

__all__ = ['CustomError', 'InvalidAPIKeyException', 'InvalidRequestException',
'NotEnoughFundsException', 'GatewayErrorException',
           'UnknownException', 'InvalidResponseException', 'RequestLimitException']

class CustomError(Exception):
    pass

class InvalidAPIKeyException(CustomError):
    def __init__(self):
        logger.error('Unauthorized or invalid API key was used')

class InvalidRequestException(CustomError):
    pass

class NotEnoughFundsException(CustomError):
```

```
    pass

class GatewayErrorException(CustomError):
    def __init__(self, msg: str = ''):
        if msg == '':
            logger.error('Gateway error exception')
        else:
            logger.error(msg)
        self.response = msg

class UnknownException(CustomError):
    def __init__(self, msg: str):
        logger.error('Unknown exception')
        logger.debug(msg)
        self.response = msg

class InvalidResponseException(CustomError):
    def __init__(self, response_msg: str):
        logger.error(f'Invalid response was received {response_msg}')
        self.response = response_msg

class RequestLimitException(CustomError):
    pass
```