МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

перший (бакалаврський)
(рівень вищої освіти)

на тему **Web застосунок для менеджменту та моніторингу криптовалютних рахунків**

Виконав: студент 4 курсу, групи 6.1211-пзс
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Програмне забезпечення систем
(код і назва освітньої програми)
О. В. Семенюк
(ініціали та прізвище)
Керівник асистент О.О.Жуков
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ «Дискус»
Р.О. Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріж
жя 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра електроніки, інформаційних систем та програмного забезпечення
Рівень вищої освіти перший (бакалаврський)
Спеціальність 121 Інженерія програмного забезпечення
(код та назва)
Освітня програма Програмне забезпечення систем
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри Тетяна КРИТСЬКА
"_01_" березня 2025_року

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Семенюку Олександру Васильовичу
(прізвище, ім'я, по батькові)

1.Тема роботи Web застосунок для менеджменту та моніторингу криптовалютних рахунків
керівник роботи Жуков Олександр Олексійович, асистент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом ЗНУ від 27.12.2024 №2029-с

2. Строк подання студентом кваліфікаційної роботи 02.06.2025

3. Вихідні дані бакалаврської роботи
- комплект нормативних документів ;
- технічне завдання до роботи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
- огляд та збір літератури стосовно теми кваліфікаційної роботи;
- огляд та аналіз існуючих рішень та аналогів;
- створення програмного продукту та його опис;
- дослідження поставленої проблеми та розробка висновків.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
слайдів презентації

6. Консультанти розділів бакалаврської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата Завдання прийняв

7. Дата видачі завдання 01.03.2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів бакалаврської роботи	Примітка
1	Аналіз предметної області	15.03.25	виконано
2	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником	17.03.25	виконано
3	Аналіз існуючих методів рішення	18.03.25	виконано
4	Дослідження засобів реалізації гри	19.03-25.03.25	виконано
5	Узгодження подальших дій з науковим керівником	30.03.25	виконано
6	Проектування архітектури	01.04-15.04.25	виконано
7	Програмна реалізація	16.04-15.05.25	виконано
8	Представлення отриманих результатів науковому керівнику та узгодження плану подальшого дослідження	25.05.25	виконано
9	Реалізація користувацького інтерфейсу	26.05-30.05.25	виконано
10	Перевірка працездатності проекту	02.06-04.06.25	виконано
11	Оформлення звіту	05.06-12.06.25	виконано
12	Оформлення презентації. Отримання рецензій від опонентів.	13.06-15.06.25	виконано

Студент _____ Семенюк О.В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Жуков О.О.
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено
Нормоконтролер _____ Жуков О.О.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Сторінок 64

Рисунків 11

Джерел 11

Семенюк О.В. Розробка web-застосунку для менеджменту та моніторингу криптовалютних рахунків: кваліфікаційна робота бакалавра спеціальності 121 «Інженерія програмного забезпечення» / наук. керівник О.О Жуков. Запоріжжя : ЗНУ, 2025. 64 с.

Метою даного проєкту є створення web-застосунку для управління криптовалютними рахунками користувачів, що дозволяє здійснювати моніторинг балансу, відкритих позицій, історії операцій та керувати API-ключами бірж. Особливу увагу приділено безпечному зберіганню конфіденційної інформації, зручному інтерфейсу користувача та інтеграції з API провідних криптовалютних бірж.

У процесі розробки було реалізовано багаторівневу архітектуру застосунку з використанням ASP.NET Core для серверної частини, React для клієнтської частини та PostgreSQL як основи для зберігання даних.

У результаті було створено функціональний веб-застосунок, що дозволяє користувачам централізовано моніторити свої криптовалютні активи на різних біржах.

Ключові слова: *web-застосунок*, *криптовалюта*, *API*, *ASP.NET Core*, *моніторинг*, *безпека*.

ABSTRACT

Pages 64

Drawings 11

Sources 11

Semeniuk O.V. Development of a Web Application for Management and Monitoring of Cryptocurrency Accounts: bachelor's thesis in specialty 121 "Software Engineering" / Scientific adviser O.O Zhukov. Zaporizhzhia: ZNU, 2025. 64 p.

The objective of this project is to develop a web application for managing users' cryptocurrency accounts, allowing them to monitor balances, open positions, transaction history, and manage API keys for different exchanges. Special attention is paid to secure handling of sensitive data, a user-friendly interface, and integration with the APIs of major crypto exchanges.

During the development process, a multi-layered architecture was implemented using ASP.NET Core for the backend, React for the frontend, and PostgreSQL for data storage.

As a result, a functional web application was created, enabling users to centrally monitor their crypto assets across multiple exchanges.

Keywords: *web application*, *cryptocurrency*, *API*, *ASP.NET Core*, *monitoring*, *security*.

ЗМІСТ

ВСТУП ... 8
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ... 12
1.1 Огляд літературних джерел ... 12
1.2 Аналіз існуючих аналогів ... 13
1.2.1 CoinStats ... 13
1.2.2 Koinly ... 14
1.2.3 CoinMarketCap Portfolio ... 14
1.2.4 Додаткові сервіси (Blockfolio, Delta, Kubera) ... 15
1.3 Постановка завдання ... 15
1.4 Висновки до розділу 1 ... 17
2 ДОСЛІДЖЕННЯ ПРОГРАМНИХ ЗАСОБІВ РЕАЛІЗАЦІЇ ... 18
2.1 Огляд сучасних рішень для веб-розробки ... 18
2.1.1 React ... 19
2.1.2 ASP.NET Core Web API ... 20
2.1.3 Взаємодія з біржами через REST і WebSocket API ... 22
2.2 Вибір фронтенд фреймворку ... 25
2.2.1 Аналіз React ... 25
2.2.2 Аналіз Vue.js ... 26
2.2.3 Аналіз Angular ... 27
2.2.4 Обґрунтування вибору React ... 27
2.3 Аналіз вимог до технічної інфраструктури ... 28
2.3.1 Серверна інфраструктура ... 28
2.3.2 Хмарні рішення ... 29
2.3.3 Мережеві вимоги ... 30
2.3.4 Безпека ... 30
2.3 Висновки до розділу 2 ... 31
3 РОЗРОБКА WEB ЗАСТОСУНОК ДЛЯ МЕНЕДЖМЕТУ ТА МОНІТОРИНГУ КРИПТОВАЛЮТНИХ РАХУНКІВ ... 32
3.1 Опис предметної області ... 32

3.2 Вимоги до системи 33
3.3 Архітектура системи 34
3.3.1 Огляд бази даних 35
3.3.2 Огляд серверної частини (шлюз API Gateway) 36
3.3.3 Огляд клієнтської частини 43
3.4 Розробка мікросервісної архітектури 46
3.5 Програмна реалізація серверної частини 48
3.5.1 Програмна реалізація сервісу автентифікації (AuthService) 48
3.5.2 Програмна реалізація DataService 50
3.5.3 Програмна реалізація ProcessorService 53
3.6 Розробка дизайну застосунку 58
ВИСНОВКИ 62
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 64

ВСТУП

Актуальність теми

Сучасний ринок криптовалют є однією з найдинамічніших сфер фінансових технологій, що характеризується швидким збільшенням кількості трейдерів, інвесторів та користувачів цифрових активів. Це обумовлює потребу в дієвих інструментах для централізованого управління криптовалютними рахунками. Користувачі часто взаємодіють із кількома біржами, на зразок Binance, Gate.io чи Bybit, що ускладнює облік активів, відкритих позицій, балансу та фінансових результатів, зокрема прибутку чи збитків. Більшість наявних сервісів мають обмежений функціонал або вимагають платної підписки, що породжує попит на рішення, котре консолідує всі необхідні дані в єдиному інтерфейсі [1]. Розробка веб-додатку на основі сучасних технологій, зокрема фреймворку ASP.NET Core для серверної частини та React для клієнтської, дозволяє створити ефективне, масштабоване та зручне рішення. Таке рішення відповідає потребам як початківців, так і досвідчених трейдерів завдяки зрозумілому інтерфейсу та можливостям аналізу фінансових даних. Крім того, створення такого додатку надає можливість практичного застосування знань із програмної інженерії, зокрема в роботі з базами даних, таких як PostgreSQL, інтеграції сторонніх сервісів, обробці даних в реальному часі та забезпеченні безпеки ключів API [5, 8].

Мета дослідження

Мета дослідження полягає в розробці та аналізі веб-застосунку для управління та моніторингу криптовалютних акаунтів з використанням фреймворку ASP.NET Core для серверної частини та React для клієнтської. Вивчення зосереджене на дослідженні технологій, архітектурних підходів та інструментів, необхідних для створення безпечних, надійних і масштабованих веб-систем, які взаємодіють із API криптовалютних бірж. Головна увага приділяється обробці та захисту ключів API, реалізації логіки авторизації користувачів, розробці REST API, моніторингу фінансових показників, як-от

баланс і відкриті позиції, а також обчисленню прибутку чи збитків. Підсумки дослідження можуть бути корисними для розробників фінансових додатків і студентів, що вивчають сучасні підходи до створення клієнт-серверних систем [3, 4].

Завдання дослідження

Дослідження передбачає виконання комплексу задач, спрямованих на створення функціонального веб-додатку. Спочатку здійснюється аналіз наукових і технічних джерел, що описують архітектуру та реалізацію фінансових веб-систем, а також вивчення наявних застосунків для моніторингу криптовалютних портфелів з визначенням їхніх переваг, вад та можливостей для вдосконалення. На основі цього обґрунтовується вибір технологічного стеку, зокрема ASP.NET Core, React, PostgreSQL і JWT, а також інших інструментів, що забезпечують ефективну реалізацію проєкту. Важливим щаблем є розробка архітектури додатку з урахуванням клієнт-серверної взаємодії, проєктування моделі даних і API-контрактів. Для гарантування безпеки реалізується модуль авторизації, що включає реєстрацію, вхід і перевірку прав доступу з використанням JWT. Крім того, розробляється модуль інтеграції з API криптовалютних бірж, таких як Binance, із забезпеченням захисту ключів API. Додаток передбачає моніторинг фінансових показників, зокрема відображення балансу, відкритих позицій, історії транзакцій і розрахунку прибутку чи збитків. На завершальному етапі проводиться модульне та інтеграційне тестування, оптимізація продуктивності та безпеки, а також створення адаптивного клієнтського інтерфейсу з використанням React для зручної навігації та візуалізації даних [4, 5, 8, 10].

Об’єкт дослідження

Об’єктом дослідження є процес розробки веб-додатку для фінансового менеджменту, що інтегрується з API криптовалютних бірж.

Предмет дослідження

Предметом дослідження є архітектура клієнт-серверних веб-систем,

принципи створення безпечних API-сервісів, методи інтеграції з криптовалютними біржами, захист даних користувачів, а також сучасні технології фронтенд- і бекенд-розробки.

Методи дослідження

Для досягнення задуму дослідження використовувалися різні способи. Розбір наукових статей, технічної документації й літературних джерел дозволив вивчити засади проєктування фінансових веб-систем, інтеграції API та захисту секретних даних. Порівняльний аналіз наявних застосунків для моніторингу криптоактивів посприяв визначити їх функціональні риси. Архітектура системи розроблялася з застосуванням UML-діаграм. У процесі створення додатку використовувалися інженерні способи розробки програмного забезпечення, зокрема модульна побудова, тестування та логування. Результати апробовано через створення MVP-прототипу системи [3, 7, 9].

Практичне значення одержаних результатів

Практична вартість дослідження полягає в розробці програмного рішення для централізованого моніторингу криптовалютних активів, яке має практичне значення в кількох напрямах. По-перше, праця демонструє підходи до створення API для обробки фінансових даних із надійним зберіганням ключів і можливістю масштабування. По-друге, результати можуть бути використані в освітніх курсах з програмної інженерії, інформаційної безпеки та веб-розробки. По-третє, додаток оптимізує взаємодію з криптовалютними біржами завдяки кешуванню, паралельній обробці запитів і використанню планувальників. По-четверте, рішення підвищує безпеку обробки конфіденційних даних через використання JWT, валідацію запитів і контроль доступу. Наостанок, модульна архітектура додатку забезпечує можливість інтеграції нових бірж без значних змін у коді, що скорочує витрати часу та ресурсів. Таким чином, дослідження створює підґрунтя для розвитку складніших фінансових технологій, сприяє автоматизації аналізу активів і підвищує рівень цифрової безпеки для

користувачів [1, 4, 11].

Глосарій

Криптовалюта: цифрова валюта, що використовує криптографічні методи для захисту транзакцій і контролю створення нових одиниць [2].

Ринкова капіталізація: загальна вартість усіх монет певної криптовалюти, розрахована як ціна за монету, помножена на кількість монет у обігу.

Торгова пара: дві валюти, що торгуються одна відносно іншої на біржі, наприклад, BTC/USDT.

Волатильність: міра коливань ціни фінансового інструменту за певний період, що характеризує ризик інвестицій.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Огляд літературних джерел

Розробка веб-додатків для управління криптовалютними активами є важливим напрямом сучасних фінансових технологій, що спричинено стрімким розвитком ринку цифрових активів і потребою в ефективних інструментах для їхнього моніторингу, аналізу та керування. Аналіз наукової та технічної літератури дозволяє окреслити ключові принципи створення таких систем, зокрема щодо безпеки, масштабованості, інтеграції із зовнішніми сервісами та зручності користувацького інтерфейсу.

У літературі особливу увагу приділено структурі блокчейн-технологій, механізмам функціонування криптовалют і засадам роботи децентралізованих і централізованих бірж. Зокрема, дослідники наголошують на необхідності забезпечення високого рівня конфіденційності та надійності під час обробки фінансових даних, що є основою для створення безпечних серверних модулів [3]. Для реалізації серверної частини веб-додатків, які взаємодіють із криптовалютними біржами, рекомендується використання фреймворку ASP.NET Core, який підтримує побудову RESTful-сервісів, модульну архітектуру та застосування контейнерів залежностей (Dependency Injection). Ці принципи дозволяють створювати масштабовані системи, здатні ефективно обробляти запити через HTTP або WebSocket, що є критично важливим для інтеграції з API бірж, таких як Binance [4, 5]. У контексті безпеки автори підкреслюють важливість сучасних методів авторизації, як-от JSON Web Tokens (JWT) і OAuth2, які забезпечують захист даних користувачів і контроль доступу до API [3].

На клієнтському рівні значну роль відіграють технології фронтенд-розробки, зокрема фреймворк React, який дає змогу створювати інтерактивні, адаптивні та зручні інтерфейси для відображення фінансових даних, як-от

баланс, історія транзакцій чи прибутковість активів. Технічна література описує засади компонентного підходу в React, що сприяє створенню модульних інтерфейсів із можливістю повторного використання коду [3]. Крім того важливим джерелом інформації є офіційна документація та бібліотеки, як-от RestSharp, які надають практичні рекомендації щодо інтеграції з API криптовалютних бірж. Наприклад, документація Binance API детально описує методи для отримання даних про баланс, позиції та транзакції, а також вимоги до безпечного зберігання ключів API за допомогою інструментів, як-от Secret Manager [4, 6].

Додаткові ресурси, зокрема онлайн-документація PostgreSQL і Swagger/OpenAPI, висвітлюють аспекти проєктування баз даних і документування API, що необхідне для створення структурованих і прозорих веб-систем [8, 9]. Таким чином, проаналізована література формує теоретичну базу для розробки веб-додатків, які відповідають сучасним вимогам до безпеки, продуктивності та зручності, а також надає практичні інструменти для реалізації таких проєктів.

1.2 Аналіз існуючих аналогів

Ринок фінансових технологій пропонує ряд програмних рішень для керування криптовалютними активами, які частково задовольняють потреби користувачів, але мають певні обмеження в функціональності, безпеці чи інтеграційних можливостях. Проведений аналіз популярних сервісів дає змогу оцінити їхні технічні характеристики, архітектуру, інтерфейс користувача, рівень безпеки та потенціал для удосконалення, що є важливим для обґрунтування розробки нового веб-додатка.

1.2.1 CoinStats

Сервіс CoinStats є одним з найвідоміших інструментів для моніторингу криптовалютних портфелів, котрий підтримує інтеграцію з численними

централізованими біржами, такими як Binance чи Gate.io, а також із DeFi-гаманцями. Система дозволяє користувачам синхронізувати дані через API, відображати баланс, історію транзакцій і порівнювати прибутковість активів у реальному часі. Зручний інтерфейс і гнучкі налаштування роблять CoinStats популярним серед широкого кола користувачів. Проте сервіс має суттєві обмеження, які знижують його ефективність для професійного використання. Зокрема, відсутність підтримки торгівлі ф'ючерсами через WebSocket ускладнює моніторинг динамічних ринкових позицій. Крім того, безкоштовна версія має обмежену кількість підключених джерел, а інтеграція нових децентралізованих платформ (DEX) часто відбувається із затримками. Ці недоліки роблять CoinStats менш придатним для трейдерів, які потребують розширених аналітичних і торгових функцій [4].

1.2.2 Koinly

Koinly спеціалізується на аналізі історичних даних і підготовці податкової звітності для користувачів криптовалют. Сервіс підтримує імпорт даних із гаманців, CSV-файлів і API бірж, дозволяючи розраховувати прибуток за транзакціями за методами FIFO, LIFO чи іншими підходами, що є важливим для виконання податкових зобов'язань. Аналітичний бекенд Koinly забезпечує високу точність обробки великих обсягів даних. Однак сервіс не призначений для активного моніторингу позицій у реальному часі чи торгівлі на ф'ючерсних ринках. Інтерфейс і функціонал орієнтовані на ретроспективний аналіз, а не на оперативну взаємодію з біржами, що обмежує його застосування для трейдерів, котрі працюють із динамічними ринковими сценаріями [3].

1.2.3 CoinMarketCap Portfolio

CoinMarketCap Portfolio є базовим інструментом для ручного відстеження криптовалютних активів, інтегрованим у популярний сервіс моніторингу ринкової капіталізації. Користувачі можуть додавати монети,

вказувати дату й ціну купівлі та відстежувати їхню поточну вартість залежно від ринкових змін. Простота інтерфейсу робить сервіс доступним для початківців. Однак повна відсутність API-інтеграції з біржами, неможливість автоматичного оновлення даних, аналізу відкритих позицій чи роботи з деривативами значно знижують його функціональність. CoinMarketCap Portfolio підходить лише для базового моніторингу без розширених аналітичних чи торгових можливостей [1].

1.2.4 Додаткові сервіси (Blockfolio, Delta, Kubera)

До інших рішень належать Blockfolio (тепер FTX App), Delta та Kubera, які переважно орієнтовані на мобільні платформи та відображення вартості криптовалютних активів. Наприклад, Kubera дозволяє об’єднувати дані про криптовалюти, банківські рахунки та інші активи в єдиному портфелі, що зручно для комплексного керування фінансами. Проте ці сервіси мають обмеження в інтеграційних можливостях: вони не надають відкритих SDK для розробки кастомного функціоналу, а їхня функціональність часто обмежується відображенням ринкової вартості токенів без підтримки активної торгівлі чи аналізу позицій. Такі обмеження підкреслюють потребу створення нового веб-додатка, який поєднує розширений функціонал, гнучку інтеграцію з біржами та високий рівень безпеки [4, 10].

1.3 Постановка завдання

Розробка веб-додатку для моніторингу та керування криптовалютними рахунками потребує чіткої специфікації вимог до його структури, технологічного стеку, функціональності та якості, аби забезпечити зручність використання, безпеку та масштабованість. Основним завданням є створення сучасного програмного рішення, яке дозволяє користувачам централізовано відслідковувати активи на кількох криптовалютних біржах, таких як Binance, Gate.io чи Bybit, а також виконувати операції управління рахунками. Цільовою

аудиторією додатку є трейдери, інвестори та власники криптовалют, які працюють із кількома біржами та потребують єдиного інтерфейсу для аналізу балансу, відкритих позицій і прибутковості (PnL) [4].

Функціональні можливості додатку охоплюють декілька ключових аспектів. Система має підтримувати авторизацію та аутентифікацію користувачів, забезпечуючи безпечний доступ до їхніх даних. Користувачі зможуть підключати API-ключі від різних бірж, що дозволить отримувати та відображати актуальну інформацію про баланс на всіх підключених рахунках, а також аналізувати прибутковість за окремими біржами чи активами. Додаток передбачає можливість виконання внутрішніх переказів між рахунками на одній біржі, що спрощує керування активами. Для підвищення зручності інформація буде візуалізована за допомогою графіків, таблиць та інформаційних панелей, які забезпечують інтуїтивне відображення фінансових даних. Крім того, система сповіщатиме користувачів про помилки, зміни статусу операцій, а також, за потреби, про критичні події, такі як ліквідація позицій чи значні зміни прибутковості [3, 9].

З технічного погляду, додаток розроблятиметься з використанням фреймворку React для клієнтської частини, що забезпечить створення адаптивного та інтерактивного інтерфейсу, придатного для використання на десктопних і мобільних пристроях. Серверна частина базуватиметься на ASP.NET Core Web API, який підтримує ефективну обробку запитів і взаємодію з біржами через REST і WebSocket API [5]. Безпека ключів API користувачів реалізовуватиметься за допомогою спеціалізованих інструментів, таких як Azure Key Vault, що гарантує їхнє захищене зберігання [3]. Для забезпечення високої продуктивності додаток розгортатиметься в хмарному середовищі, наприклад, Azure, Netlify або Vercel, що дозволить масштабувати систему в залежності від навантаження [11].

Очікуваним результатом є створення мінімального життєздатного продукту (MVP), який надасть користувачам можливість централізованого моніторингу криптовалютних рахунків через єдиний інтерфейс. Додаток

матиме зручний інтерфейс, високий рівень безпеки даних і продуктивності, а також модульну архітектуру, що дозволить у майбутньому додавати нові біржі, типи рахунків чи аналітичні функції. Терміни реалізації проєкту визначатимуться на етапі планування, з урахуванням складності інтеграції та тестування [5, 10].

1.4 Висновки до розділу 1

Аналіз проблемної сфери демонструє, що теперішні рішення для моніторингу криптовалютних активів, такі як CoinStats, Koinly чи CoinMarketCap Portfolio, мають обмеження у функціональності, гнучкості та можливостях інтеграції з біржами. Ці сервіси часто не відповідають потребам активних трейдерів чи користувачів з декількома акаунтами, оскільки підтримують обмежену кількість API, мають недостатню підтримку торгівлі деривативами чи не забезпечують моніторинг позицій в реальному часі [1, 4]. Здійснений огляд літератури підтверджує актуальність розробки веб-додатків для управління криптовалютними активами, акцентуючи на необхідності використання сучасних технологій, таких як ASP.NET Core, React та безпечних методів авторизації (JWT, OAuth2), для створення ефективних та захищених систем [3, 5].

Метою дослідження є створення веб-додатку, що забезпечує єдину точку доступу для управління криптовалютними акаунтами, з підтримкою API-ключів, безпечним зберіганням даних і розширеним функціоналом для аналізу фінансових показників. Постановка задачі передбачає розробку модульної системи з адаптивним інтерфейсом, інтеграцією з біржами через REST і WebSocket, а також високим рівнем безпеки та продуктивності. Отримані результати створюють теоретичну базу для реалізації проєкту, що відповідає сучасним вимогам фінансових технологій [4, 10].

2 ДОСЛІДЖЕННЯ ПРОГРАМНИХ ЗАСОБІВ РЕАЛІЗАЦІЇ

2.1 Огляд сучасних рішень для веб-розробки

Сучасні фінансові технології, що висувають підвищені вимоги до обробки цифрових активів, глобалізації ринків і необхідності моніторингу в реальному часі, формують потребу у програмних рішеннях для управління криптовалютними рахунками. Розробка таких веб-додатків потребує не тільки вміння програмувати, а й глибокого розуміння веб-технологій, архітектурних підходів, протоколів обміну даними та інструментів автоматизації. У контексті створення системи для моніторингу криптовалют критичними є технології, які забезпечують швидке створення інтерактивних інтерфейсів, безпечну інтеграцію з API бірж, надійну обробку даних та масштабування.

Сучасний ландшафт веб-розробки пропонує широкий вибір інструментів, які відповідають цим потребам. Для клієнтської частини широко використовуються бібліотеки та фреймворки, що підтримують компонентну архітектуру й реактивне програмування, що дозволяє створювати адаптивні інтерфейси з високою продуктивністю. На серверному рівні застосовуються фреймворки, які забезпечують розробку масштабованих API для взаємодії з зовнішніми сервісами через HTTP або WebSocket. Захист конфіденційних даних, таких як ключі API, реалізується за допомогою спеціалізованих інструментів, що гарантують їх безпечне зберігання. Для керування даними використовуються реляційні бази даних, що підтримують складні запити та забезпечують надійність. Хмарні платформи спрощують розгортання та масштабування додатків, що є важливим для систем з високим навантаженням. Цей огляд зосереджений на ключових технологіях, зокрема на бібліотеці React, яка є основою для створення клієнтської частини веб-додатку для моніторингу криптовалют.

2.1.1 React

React є однією з найпопулярніших бібліотек для створення інтерфейсів користувача, яку часто використовують для розробки односторінкових веб-застосунків (SPA) з компонентною архітектурою. Завдяки розвиненій екосистемі та підтримці спільноти, ця бібліотека дозволяє створювати модульні, продуктивні та масштабовані інтерфейси, що робить її оптимальним вибором для веб-застосунку, який моніторить криптовалютні рахунки.

Ключовою особливістю React є компонентна структура, яка дозволяє розробникам створювати ізольовані, повторно використовувані елементи інтерфейсу з власною логікою та станом. Цей підхід сприяє чіткому розподілу функціональних обов'язків, спрощує підтримку коду та покращує його модульність. Наприклад, у застосунку для моніторингу криптовалют окремі компоненти можуть відображати баланс, графіки прибутковості чи таблиці транзакцій, забезпечуючи гнучкість та легкість модифікації системи.

React використовує механізм віртуального DOM, який оптимізує оновлення інтерфейсу шляхом порівняння змін у віртуальному представленні DOM із попереднім станом та застосування лише необхідних змін до реального DOM. Цей підхід суттєво покращує продуктивність, особливо для застосунків з динамічними даними, наприклад, відображення ринкових цін чи оновлення позицій у реальному часі. Реактивність у React реалізується через хуки, зокрема useState для керування станом, useEffect для обробки побічних ефектів та useContext для роботи з глобальними даними. Ці інструменти дають можливість створювати функціональні компоненти, які швидко реагують на дії користувача, наприклад, підключення нового API-ключа чи зміну фільтрів у таблиці транзакцій.

Для покращення взаємодії з користувачем React підтримує серверний рендеринг (SSR) і статичну генерацію сторінок (SSG) через фреймворки, як-от Next.js або Remix. SSR забезпечує генерацію HTML на сервері, що підвищує швидкість завантаження та індексацію пошуковими системами, тоді як SSG створює статичні сторінки на етапі збірки, що ідеально для статичних даних,

наприклад, документації чи інформаційних сторінок застосунку. Підтримка TypeScript в React забезпечує статичну типізацію, що зменшує кількість помилок у великих проєктах, покращує читабельність коду та спрощує рефакторинг.

Інтеграція з UI-фреймворками, такими як Material UI, Ant Design чи Tailwind CSS, дозволяє швидко створювати візуально привабливі інтерфейси, які відповідають сучасним стандартам дизайну. Ці інструменти надають готові компоненти та системи стилізації, що прискорюють розробку. React також підтримує створення прогресивних веб-застосунків (PWA), які забезпечують офлайн-режим, push-сповіщення та нативний користувацький досвід. PWA особливо корисні для фінансових застосунків, де стабільність мережі може бути обмеженою.

Екосистема React містить потужні інструменти для розробки, тестування та оптимізації. Наприклад, React Developer Tools полегшують дебагінг компонентів та аналіз продуктивності, тоді як інструменти збірки, такі як Vite або Webpack, забезпечують швидке налаштування середовища з підтримкою hot-reload та оптимізацією коду. Ці можливості роблять React ідеальним вибором для створення клієнтської частини веб-застосунку, що потребує високої продуктивності, зручності та масштабованості.

2.1.2 ASP.NET Core Web API

ASP.NET Core — це багатоплатформний фреймворк від Microsoft, розроблений для створення високопродуктивних серверних застосунків, що характеризуються гнучкою архітектурою. Цей фреймворк відповідає сучасним вимогам до масштабованості, надійності та ефективності. Він чудово підходить для розробки веб-додатків для моніторингу криптовалютних рахунків, оскільки підтримує створення RESTful API для взаємодії з біржами, обробку даних в реальному часі та забезпечує захист конфіденційної інформації, такої як API-ключі. Завдяки модульній структурі та вбудованим інструментам, ASP.NET Core забезпечує ефективну розробку

бекенду, що відповідає потребам фінансових систем з високим рівнем навантаження.

Однією з ключових переваг ASP.NET Core є підтримка мікросервісної архітектури. Це дозволяє розподіляти функціонал на окремі незалежні сервіси, кожен з яких можна розробляти, розгортати та масштабувати незалежно. Такий підхід сприяє паралельній роботі команд, спрощує впровадження CI/CD-процесів та дозволяє використовувати різні технології для окремих компонентів системи. Наприклад, у додатку для моніторингу криптовалют, один мікросервіс може відповідати за інтеграцію з API бірж, інший — за обробку фінансових даних. Це підвищує гнучкість та стійкість системи в цілому.

Фреймворк включає вбудовану систему впровадження залежностей (Dependency Injection), яка забезпечує слабке зв'язування компонентів, що полегшує тестування та підтримку коду. DI-контейнер підтримує різні життєві цикли об’єктів (Singleton, Scoped, Transient), що дає змогу гнучко налаштовувати поведінку сервісів залежно від потреб додатку, наприклад, для управління підключеннями до бази даних або зовнішніх API. Гнучка система middleware дозволяє налаштувати конвеєр обробки HTTP-запитів, додаючи функціонал аутентифікації, логування, обробки помилок чи кешування. Кожен middleware може модифікувати запит або відповідь, що забезпечує високу адаптивність системи до конкретних вимог, таких як валідація API-запитів до бірж.

ASP.NET Core підтримує розширені можливості аутентифікації та авторизації через інтеграцію з ASP.NET Core Identity, що дозволяє реалізовувати управління користувачами, ролями, JWT-токенами та мультифакторною аутентифікацією. Це є надзвичайно важливим для фінансових додатків, де безпека доступу до даних користувачів є пріоритетом. Фреймворк також забезпечує захист від поширених загроз, таких як CSRF-атаки, за допомогою налаштування CORS, HTTPS-редиректів та Content Security Policy. Для реального часу комунікації ASP.NET Core

підтримує SignalR, що використовує WebSocket, Server-Sent Events або Long Polling для створення інформаційних панелей з живими оновленнями, наприклад, для відображення змін балансу чи ринкових подій. Крім того, підтримка gRPC забезпечує високопродуктивну комунікацію між сервісами, що є корисним для мікросервісних систем з великою кількістю запитів.

Для документування API фреймворк пропонує інтеграцію з інструментами Swagger/OpenAPI, які дозволяють автоматично генерувати документацію та створювати інтерактивний інтерфейс для тестування ендпоінтів. Це спрощує розробку та підтримку API, особливо при інтеграції з біржами, де потрібна чітка специфікація запитів і відповідей. Вбудовані механізми Health Checks дозволяють відстежувати стан додатку та його залежностей, таких як бази даних чи зовнішні сервіси, що забезпечує надійність системи.

З точки зору продуктивності та розгортання, ASP.NET Core оптимізований для роботи в хмарних середовищах, підтримує контейнеризацію через Docker та оркестрацію через Kubernetes. Це дозволяє ефективно масштабувати додаток в залежності від навантаження, що є критичним для фінансових систем з великою кількістю користувачів. Механізми кешування (in-memory або розподіленого), rate limiting та response compression оптимізують продуктивність і захищають від надмірного навантаження, наприклад, при обробці великої кількості API-запитів до бірж. Ці особливості роблять ASP.NET Core відмінним вибором для створення бекенду веб-додатку для моніторингу криптовалют.

2.1.3 Взаємодія з біржами через REST і WebSocket API

Інтеграція з криптовалютними біржами є ключовим елементом веб-застосувань для відстеження та керування активами. Вона надає доступ до фінансової інформації та функцій, як-от аналіз балансу, позицій, транзакцій і виконання торговельних операцій. Сучасні біржі пропонують API, що дозволяють отримувати ринкові дані, управляти акаунтами та

автоматизувати торгові процеси, роблячи їх незамінними для розробки аналітичних панелей та систем керування активами. Взаємодія з біржами здійснюється за допомогою двох основних протоколів: REST API для статичних запитів та WebSocket API для оновлень у режимі реального часу, кожен з яких має власні особливості та вимоги до впровадження.

REST API діляться на публічні та приватні. Публічні API забезпечують доступ до ринкових даних, як-от котирування, глибина книги ордерів, історичні свічки, обсяги торгів та метадані про торгові пари, без необхідності аутентифікації. Ці дані використовуються для відображення поточних цін, аналізу ринкової активності чи побудови графіків. Приватні API, навпаки, потребують аутентифікації та дозволяють виконувати операції, пов’язані з управлінням рахунком: перегляд балансу, історії транзакцій, відкритих ордерів, а також створення або скасування ордерів та переказів між рахунками. Для ефективної роботи REST API підтримують пагінацію та фільтрацію даних, що дозволяє обмежувати кількість результатів, сортувати їх за часовими проміжками або типами ордерів, оптимізуючи обробку великих обсягів інформації.

WebSocket API забезпечують двосторонній зв'язок в реальному часі, що критично важливо для моніторингу динамічних ринкових змін, таких як зміни цін, виконання угод чи оновлення статусу ордерів. Користувачі мають змогу підписуватися на потоки даних, наприклад, для конкретних торгових пар, та отримувати оновлення з мінімальною затримкою. Управління підписками включає динамічне підключення та відключення від потоків, групування даних за типами або активами, а також обробку повідомлень про стан з'єднання. Для забезпечення стабільної роботи WebSocket-з'єднань реалізується автоматичне перепідключення при втраті зв'язку та буферизація даних для згладжування пікового навантаження. Механізми черг і backpressure допомагають уникнути переповнення пам'яті при високій частоті оновлень, що особливо важливо для бірж з великим обсягом операцій.

Для забезпечення стабільної та безпечної взаємодії з API важливо

враховувати обмеження бірж, такі як ліміти на кількість запитів (rate limits). Це вимагає реалізації механізмів контролю швидкості, обробки помилок HTTP 429 (Too Many Requests) та застосування алгоритмів exponential backoff для повторних спроб. Безпека забезпечується шляхом аутентифікації запитів, зокрема, з використанням HMAC SHA256 підписів, OAuth2 або білого списку IP-адрес. API ключі зберігаються в захищених системах, як-от Key Vault, для запобігання їх компрометації. Моніторинг та ведення логів відіграють ключову роль у відстеженні статистики запитів, виявленні помилок, аналізі латентності та оптимізації продуктивності системи.

Агрегація даних з декількох бірж є складним завданням через відмінності у форматах даних, назвах торгових пар та точності цін. Для цього створюється єдиний адаптаційний шар, який транслює запити в стандартизований формат, кешує результати та нормалізує дані для уніфікованої обробки. Наприклад, система нормалізації може приводити ціни до однакової кількості знаків після коми або уніфікувати назви торгових пар (наприклад, BTC/USDT). Це дає змогу створювати універсальні інтерфейси, незалежні від специфіки конкретної біржі, що спрощує розробку та використання додатку.

Управління з’єднаннями передбачає створення пулу для REST API для оптимізації повторного використання з’єднань та логіку graceful shutdown для WebSocket, щоб уникнути втрати даних у разі розриву з’єднання. Обробка помилок включає мережеві таймаути, біржові помилки та механізми повторних спроб із exponential backoff. Валідація даних забезпечує перевірку відповідності очікуваному формату, узгодженості часових міток та виявлення аномалій, що гарантує цілісність даних. Оптимізація продуктивності досягається за рахунок стиснення даних, ефективного парсингу JSON, повторного використання з’єднань та оптимізації пам’яті для обробки високої частоти оновлень. Ці аспекти забезпечують надійний та ефективний потік даних, що є основою для створення веб-додатку для моніторингу криптовалют.

2.2 Вибір фронтенд фреймворку

Вибір фронтенд-фреймворку для розробки веб-застосунку з моніторингу криптовалютних рахунків є ключовим рішенням, оскільки від нього залежать продуктивність, зручність користування, масштабованість та швидкість розробки. Сучасні фронтенд-технології пропонують низку рішень, кожне з яких має свої переваги та недоліки, які залежать від вимог конкретного проєкту. Для застосунку, що оперує фінансовими даними у реальному часі, потребує взаємодії з API бірж та повинен мати адаптивний інтерфейс, необхідно враховувати такі аспекти, як підтримка реактивного програмування, модульність, продуктивність та можливість інтеграції з бібліотеками для візуалізації даних. Проведений аналіз трьох найпопулярніших фреймворків - React, Vue.js та Angular - дозволяє визначити найоптимальніше рішення для реалізації клієнтської частини веб-застосунку.

2.2.1 Аналіз React

React є провідною бібліотекою для створення інтерфейсів користувача, що вирізняється компонентною архітектурою, яка сприяє розробці модульних та повторно використовуваних елементів інтерфейсу. Цей підхід особливо цінний для фінансових застосунків, де інтерфейс включає складні компоненти, такі як дашборди, графіки прибутків чи таблиці транзакцій. Компонентна структура дозволяє ізолювати логіку кожного елемента, що полегшує підтримку та модифікацію коду. Механізм віртуального DOM у React оптимізує оновлення інтерфейсу, що критично важливо для відображення даних у реальному часі, наприклад, ринкових цін або змін балансу. React підтримує реактивне програмування через хуки, такі як useState та useEffect, які забезпечують ефективне управління станом та швидку реакцію на дії користувача, наприклад, підключення API-ключа.

Екосистема React пропонує широкий вибір бібліотек для роботи з фінансовими даними, зокрема Chart.js, D3.js та Recharts для створення

графіків, а також Redux та Zustand для централізованого управління станом. Інтеграція з TypeScript забезпечує статичну типізацію, що підвищує надійність при обробці складних фінансових даних та API. React також підтримує серверний рендеринг (SSR) через фреймворки, такі як Next.js, що покращує швидкість завантаження сторінок та оптимізацію для пошукових систем, що може бути корисним для публічних частин застосунку, наприклад, інформаційних сторінок. Крім того, React дозволяє створювати прогресивні веб-застосунки (PWA), які забезпечують офлайн-режим та нативний користувацький досвід, що важливо для користувачів із нестабільним мережевим з'єднанням.

Проте React має певні обмеження. Налаштування проєкту може бути складним через необхідність використання додаткових інструментів, таких як бандлери (Webpack, Vite) та транспілери (Babel). Швидка еволюція екосистеми може призводити до застарівання залежностей, що вимагає регулярного оновлення бібліотек. Незважаючи на це, велика спільнота розробників та розвинена екосистема інструментів компенсують ці недоліки, забезпечуючи швидке вирішення технічних проблем.

2.2.2 Аналіз Vue.js

Vue.js є легким та гнучким фреймворком, який вирізняється низьким порогом входження завдяки простому синтаксису та поступовому підходу до освоєння. Це дозволяє розробникам швидко розпочати створення інтерфейсів, що є перевагою для проєктів із обмеженим часом. Двосторонній зв'язок даних у Vue.js спрощує синхронізацію інтерфейсу з даними, що зручно для реалізації форм введення торгових параметрів, наприклад, для налаштування API-ключів або фільтрів транзакцій. Composition API забезпечує гнучку організацію логіки компонентів, що полегшує розробку складних фінансових розрахунків, таких як аналіз прибутковості. Вбудовані інструменти управління станом, такі як Vuex або Pinia, дозволяють централізовано керувати даними застосунку, що корисно для роботи з кількома біржами.

Проте Vue.js має меншу екосистему порівняно з React, що обмежує вибір спеціалізованих бібліотек для фінансових застосунків, таких як інструменти для побудови складних графіків або реального часу комунікації. Менша популярність фреймворку може ускладнити пошук розробників із відповідним досвідом або рішень для специфічних проблем, що є недоліком для великих проєктів із довгостроковою підтримкою.

2.2.3 Аналіз Angular

Angular є повноцінним фреймворком, який включає широкий набір інструментів для розробки, таких як маршрутизація, HTTP-клієнт, форми та тестування, що робить його придатним для великих корпоративних проєктів. TypeScript використовується за замовчуванням, що забезпечує строгу типізацію та надійність при інтеграції з фінансовими API. Вбудована система впровадження залежностей спрощує тестування та підтримку коду, що є важливим для складних бізнес-процесів у фінансових застосунках. Інтеграція з RxJS дозволяє ефективно працювати з асинхронними потоками даних, наприклад, WebSocket-потоками для оновлення ринкових даних у реальному часі. Angular підходить для проєктів із високими вимогами до структури та масштабованості.

Однак Angular має високий поріг входження через складну архітектуру та концепції, що може сповільнити старт розробки. Розмір фінального бандлу застосунку зазвичай більший порівняно з React або Vue.js, що може вплинути на швидкість завантаження, особливо на мобільних пристроях із обмеженим ресурсом. Ці фактори роблять Angular менш оптимальним для проєктів, де потрібна швидка розробка та легкий інтерфейс.

2.2.4 Обґрунтування вибору React

Для розробки веб-застосунку з моніторингу криптовалютних рахунків було обрано React завдяки його оптимальному поєднанню продуктивності, гнучкості та екосистеми. Механізм віртуального DOM забезпечує високу

швидкість оновлення інтерфейсу, що критично для відображення даних у реальному часі, таких як ринкові ціни або зміни балансу. Широка екосистема бібліотек, включаючи Chart.js, D3.js та TradingView для графіків, а також Socket.io для реального часу комунікації, дозволяє створювати спеціалізовані фінансові інтерфейси. Гнучка архітектура React підтримує поступову міграцію від простого односторінкового застосунку до серверного рендерингу через Next.js, що забезпечує гнучкість у розвитку проєкту.

Інтеграція з TypeScript підвищує надійність при роботі з фінансовими API, де важлива точність типів даних. Велика спільнота розробників полегшує пошук рішень та підбір команди, що є важливим для довгострокової підтримки проєкту. Можливість створення прогресивних веб-застосунків дозволяє забезпечити мобільний досвід без необхідності розробки нативних застосунків, що знижує витрати. Порівняно з Vue.js, React пропонує ширший вибір бібліотек та більшу популярність, а порівняно з Angular - менший поріг входження та легший бандл. Таким чином, React є оптимальним вибором для створення клієнтської частини веб-застосунку, що відповідає вимогам продуктивності, інтеграції з API та зручності користувача.

2.3 Аналіз вимог до технічної інфраструктури

Веб-застосунки для моніторингу криптовалют мають специфічні вимоги до інфраструктури через необхідність обробки великих обсягів real-time даних, забезпечення високої доступності та безпеки фінансової інформації.

2.3.1 Серверна інфраструктура

Рекомендовані характеристики серверного обладнання:

Веб-сервер (Frontend хостинг):

- CPU: 4+ vCPU (Intel Xeon або AMD EPYC)
- RAM: 8GB DDR4

- Пам’ять: 100GB SSD
- Мережа: 1Gbps пропускна здатність
- CDN: CloudFlare або AWS CloudFront для глобального розподілу контенту

Сервер API (Backend-сервіси):

- CPU: 8+ vCPU з високою частотою (3.0GHz+)
- RAM: 16GB DDR4 (32GB для середовища з високим навантаженням)
- Storage: 200GB NVMe SSD
- Network: 10Gbps для обробки множинних WebSocket з'єднань
- Load Balancer: NGINX або HAProxy для розподілу навантаження

Сервер бази даних:

- CPU: 12+ vCPU оптимізованих для баз даних
- RAM: 32GB DDR4 (64GB для великих датасетів)
- Storage: 500GB NVMe SSD з RAID 10 для надійності
- IOPS: 10,000+ для швидких запитів до даних
- Backup: Автоматизовані щоденні резервні копії.

2.3.2 Хмарні рішення

Azure/AWS Конфігурація:

- Frontend: Azure App Service або AWS Elastic Beanstalk (B2 tier мінімум)
- Backend API: Azure Container Instances або AWS ECS (2-4 екземплярів)
- Database: Azure Database for PostgreSQL або AWS RDS (Загальний рівень)
- Cache: Azure Redis Cache або AWS ElastiCache (1GB мінімум)

Networking:

- CDN: Azure Front Door або AWS CloudFront
- Load Balancer: Azure Load Balancer або AWS Application Load

Balancer

- API Gateway: Azure API Management або AWS API Gateway
- VPN: Site-to-site VPN для доступу до панелі

Моніторинг та безпека:

- Моніторинг: Azure Monitor або AWS CloudWatch
- Логування: Azure Log Analytics або AWS CloudTrail
- Безпека: Azure Key Vault або AWS Secrets Manager
- WAF: Web Application Firewall для захисту від атак

2.3.3 Мережеві вимоги

Вимоги до пропускної здатності:

- WebSocket connections: 1-10 Mbps на 1000 одночасних ористувачів
- REST API calls: 100-500 запитів/секунду базове навантаження
- Static content: CDN з периферійним розташуванням для мінімізації затримок
- Запити до бази даних: Виділена мережа для внутрішнього зв'язку

Вимоги до затримки:

- Оновлення в реальному часі: <100мс від біржі до користувача
- Час відповіді API: <200мс для 95% запитів
- Час завантаження сторінки: <2 секунди
- WebSocket затримка: <50мс для оновлення цін

2.3.4 Безпека

Інфраструктура безпеки:

- SSL/TLS: Сертифікати з автоматичним оновленням
- DDoS Protection: CloudFlare або AWS Shield
- Firewall: Web Application Firewall
- Authentication: Аутентифікація до адмін доступу

2.3 Висновки до розділу 2

Аналіз сучасних методів веб-розробки вказує на те, що для розробки системи моніторингу криптовалют, React для клієнтської сторони та ASP.NET Core Web API для серверної частини є оптимальним рішенням. React вирізняється високою продуктивністю через віртуальний DOM і компонентну архітектуру, що дає змогу будувати модульні інтерфейси для візуалізації ринкової інформації та дашбордів. Інтеграція з TypeScript і бібліотеками, як-от Chart.js, гарантує стабільну роботу з фінансовою інформацією. ASP.NET Core характеризується кросплатформністю, підтримкою мікросервісів та безпечним обробленням запитів, що чудово підходить для взаємодії з API бірж.

Взаємодія з криптовалютними біржами за допомогою REST і WebSocket API забезпечує доступ як до статичних, так і до динамічних даних, на кшталт балансу чи котирувань. Механізми аутентифікації (HMAC SHA256) та управління rate-limiting гарантують безпеку та стабільність роботи системи. PostgreSQL обрано для транзакційних даних, Redis — для кешування, а бази даних time-series використовуються для зберігання котирувань. Хмарна інфраструктура, що базується на Docker і Kubernetes, гарантує масштабованість, а захист інформації забезпечується через Key Vault і HTTPS. Використання цих технологій створює фундамент для ефективного та безпечного веб-додатку.

3 РОЗРОБКА WEB ЗАСТОСУНОК ДЛЯ МЕНЕДЖМЕТУ ТА МОНІТОРИНГУ КРИПТОВАЛЮТНИХ РАХУНКІВ

3.1 Опис предметної області

Сучасна цифрова економіка визначається дедалі більшою вагомістю криптовалютних активів у портфелях приватних і корпоративних інвесторів. Ринок цифрових валют відзначається високою волатильністю, постійними змінами та різноманітністю платформ, таких як Binance, Gate.io, OKX та Bybit, кожна з яких має власні API, моделі комісій, методи аутентифікації та інтерфейси. Це зумовлює необхідність у єдиному інструменті для централізованого управління активами, усуваючи потребу безпосереднього використання інтерфейсів окремих бірж.

Предметна область включає розробку багатокомпонентної веб-системи, яка забезпечує комплексне управління криптовалютними акаунтами. Система підтримує авторизацію користувачів, безпечне збереження API-ключів, отримання даних про баланси, відкриті позиції, прибутки та збитки (PnL), а також виконання операцій, наприклад, розміщення ордерів, перекази між акаунтами та перегляд історії транзакцій. Додатково передбачено обробку платіжних реквізитів, зокрема адрес для депозитів та виведення коштів, що спрощує фінансові операції.

Особливу увагу приділено інформаційній безпеці, оскільки API-ключі забезпечують повний доступ до акаунтів на біржах. Для цього реалізовано шифрування ключів, їх зберігання у захищеному сховищі, наприклад Azure Key Vault, а також обмеження доступу через авторизовані мікросервіси та сегментування користувачів. Система також має аналітичну складову, що включає агрегацію даних з різних бірж, обчислення вартості портфеля, аналіз прибутковості та формування рекомендацій. Це закладає основу для подальшого розвитку в напрямках автоматизованої торгівлі, прогнозування

ринкових тенденцій або підготовки фінансової звітності.

3.2 Вимоги до системи

Розробка веб-додатку для централізованого керування крипто-акаунтами потребує окреслення функціональних і нефункціональних потреб, що гарантують зручність, захищеність та стабільність роботи. Функціональність системи передбачає авторизацію користувачів за допомогою надійних методів, додавання API-ключів з метаданими (режим доступу, дозволені операції), а також управління обліковими записами на різних біржах. Користувачі зможуть переглядати баланси спотових та ф’ючерсних гаманців, відкривати й закривати позиції, розміщувати ринкові та лімітні ордери, виконувати перекази між акаунтами та запускати виведення коштів. Система надає підтримку декількох API-ключів для однієї біржі, що дає можливість керувати кількома акаунтами одночасно, наприклад, для арбітражних стратегій.

Нефункціональні вимоги приділяють увагу безпеці та продуктивності. API-ключі шифруються та зберігаються у зовнішньому сховищі, такому як Azure Key Vault, доступ до якого можливий тільки через авторизовані мікросервіси, що зменшує ймовірність витоку даних. Система розрахована на горизонтальне масштабування, забезпечуючи підтримку зростаючої кількості користувачів та акаунтів без втрати швидкодії. Для забезпечення стійкості до збоїв реалізовано логування всіх запитів, моніторинг стану сервісів та резервне копіювання даних.

Веб-інтерфейс розроблено з урахуванням адаптивності та зручності, забезпечуючи коректну роботу на різних пристроях, включаючи настільні та мобільні, у всіх основних браузерах. Користувачам надається інтерактивний перегляд портфелю, інструменти для аналізу прибутковості та проста навігація, що підвищує ефективність використання системи.

3.3 Архітектура системи

Архітектура системи управління крипторахунками побудована на мікросервісному підході, який гарантує модульність, масштабованість та незалежність складових. Така структура дозволяє відокремити бізнес-логіку, зменшити залежності між модулями, а також спростити тестування та підтримку. Взаємодія між мікросервісами реалізована за допомогою gRPC-протоколу, що вирізняється високою продуктивністю та суворим типізуванням даних, що скорочує мережеві витрати та забезпечує надійну взаємодію.

Система складається з чотирьох ключових мікросервісів: API шлюз, сервіс аутентифікації, сервіс даних та сервіс обробки. API шлюз є єдиною точкою входу для клієнтського інтерфейсу, приймаючи запити користувачів та розподіляючи їх між внутрішніми сервісами. Це забезпечує централізовану обробку запитів та полегшує інтеграцію з фронтендом.

Сервіс аутентифікації відповідає за авторизацію та управління користувачами. Він обробляє реєстрацію, вхід в систему, створення токенів доступу та безпечне зберігання API-ключів для взаємодії з біржами. Цей сервіс гарантує захист доступу до конфіденційної інформації, застосовуючи шифрування та обмеження прав.

Сервіс даних виконує роль централізованого сховища, зберігаючи інформацію про користувачів, їхні API-ключі та метадані. Він забезпечує доступ до даних для інших мікросервісів, підтримуючи синхронізацію та цілісність даних. Такий підхід дозволяє ефективно керувати даними та забезпечує їхню доступність для обробки.

Сервіс обробки втілює основну бізнес-логіку, пов’язану з взаємодією з криптобіржами. Він відповідає за отримання балансів, виконання ринкових та лімітних ордерів, створення адрес для депозитів, перекази між акаунтами та виведення коштів. Цей сервіс є найбільше навантаженим, оскільки обробляє запити до зовнішніх API бірж, що вимагає високої продуктивності та стійкості.

Взаємодія між мікросервісами відбувається через gRPC, що забезпечує швидку та типізовану передачу даних. Кожен сервіс розроблений з урахуванням принципів розділення обов’язків, що сприяє повторному використанню компонентів і дає змогу масштабувати систему горизонтально, залежно від навантаження. Така архітектура створює надійний фундамент для управління крипторахунками, забезпечуючи ефективність та гнучкість.

3.3.1 Огляд бази даних

У мікросервісній архітектурі системи управління криптовалютними рахунками сховище даних зосереджено у мікросервісі DataService. Цей сервіс відповідає за збереження та обробку важливих бізнес-сутностей, таких як користувачі та їхні API-ключі, що необхідні для доступу до криптовалютних бірж. Для роботи з структурованою інформацією використовується реляційна база даних PostgreSQL у поєднанні з бібліотекою Entity Framework Core. Остання забезпечує об’єктно-реляційне відображення (ORM), що дозволяє ефективно працювати з даними.

Модель даних була розроблена відповідно до принципів нормалізації, з підтримкою зв’язків між різними сутностями та забезпеченням цілісності даних. Ключовими сутностями є об’єкти, що описують користувачів та їх API-ключі, які відображаються на відповідні таблиці в базі даних. Сутність користувача містить ідентифікаційні дані, наприклад, унікальний ідентифікатор, адресу електронної пошти та хеш пароля. Сутність API-ключа включає інформацію про біржу, власне ключ, секретний ключ і, за потреби, додатковий пароль. Зв’язок між цими сутностями побудовано за принципом "один-до-багатьох". Це дозволяє одному користувачу мати декілька ключів, як для різних бірж, так і для однієї й тієї ж біржі.

Лістинг *1 – Опис моделей даних*

```
public class UserEntity
{
```

```
    public Guid Id { get; set; } = Guid.NewGuid();
    public string Email { get; set; }
    public string PasswordHash { get; set; }

    public ICollection<ApiKeyEntity> ExchangeKeys { get; set;
} = new List<ApiKeyEntity>();
}

public class ApiKeyEntity
{
    public Guid Id { get; set; } = Guid.NewGuid();
    public Guid UserId { get; set; }
    public Exchanges Exchange { get; set; }
    public string ApiKey { get; set; }
    public string ApiSecret { get; set; }
    public string? ApiPassword { get; set; }
}
```

Для гарантування цілісності даних в базі використовуються зовнішні ключі та індексація за полями, що зв’язують користувачів і API-ключи, а також за видом біржі. Безпека даних досягається через зберігання паролів у вигляді хешів без можливості зворотного дешифрування та шифрування API-ключів за допомогою засобів захисту, таких як IDataProtector в ASP.NET Core. Така модель даних є гнучкою для розширення, підтримує транзакції та забезпечує безпечний доступ до конфіденційних даних, що є критично важливим для керування криптовалютними акаунтами.

3.3.2 Огляд серверної частини (шлюз API Gateway)

Проєкт APIGateway структуровано за принципами модульності та відповідності функціональним зонам, що спрощує підтримку та масштабування системи. На зображенні нижче представлено структуру проєкту:

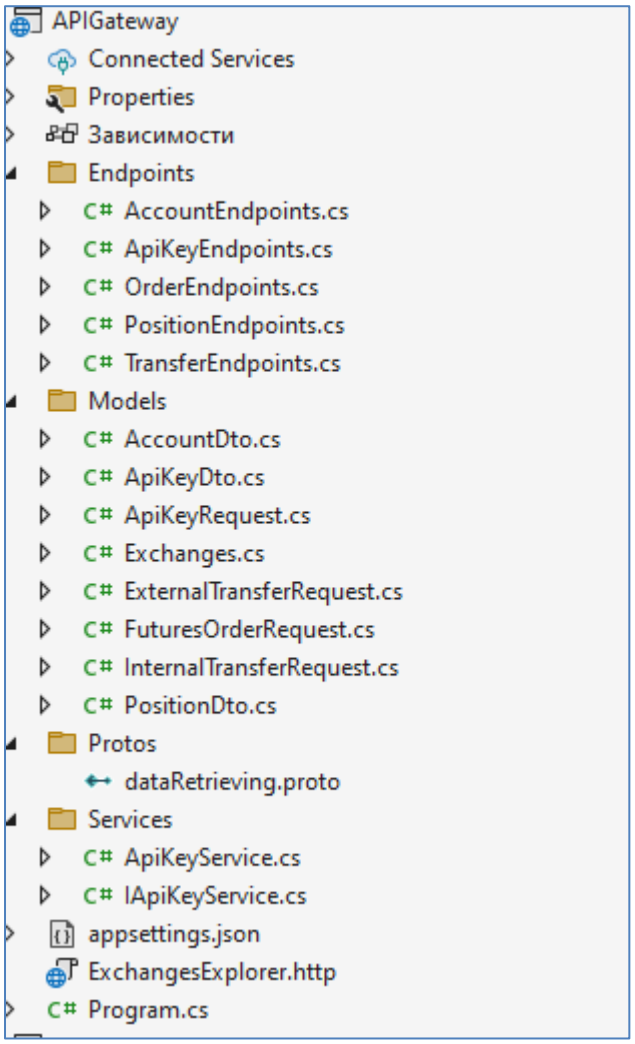


Рисунок *1 – Структура API Gateway*

До основних директорій належать:

- Endpoints/ — містить класи з HTTP-ендпоінтами, згрупованими за функціональними сценаріями: обробка акаунтів (AccountEndpoints.cs), API-ключів (ApiKeyEndpoints.cs), ордерів (OrderEndpoints.cs), позицій та трансферів. Кожен файл реалізує відповідні маршрути, які звертаються до внутрішніх сервісів через gRPC-клієнтів.
- Models/ — DTO-моделі, що використовуються для передачі даних між клієнтом і API. Зокрема: AccountDto, ApiKeyRequest, FuturesOrderRequest, PositionDto тощо. Вони відображають структуру HTTP-запитів та відповідають контрактам сервісів.
- Services/ — класи, що реалізують прикладну логіку шлюзу.

Наприклад, ApiKeyService.cs містить методи для взаємодії з AuthService та DataService з метою додавання, видалення або отримання ключів.

- Protos/ — каталог, у якому розміщено .proto-файли для визначення gRPC-контрактів. У файлі dataRetrieving.proto оголошено структури запитів і відповідей, що використовуються в комунікації з ProcessorService.
- Program.cs та appsettings.json — основні конфігураційні файли, відповідальні за ініціалізацію вебзастосунку, підключення gRPC-клієнтів, реєстрацію залежностей, CORS-політик та запуск служби.

Компонент ApiKeyService в API Gateway реалізує прикладну логіку взаємодії з мікросервісом DataService, через gRPC-інтерфейс DataRetrieving.DataRetrievingClient. Його основне призначення — обробка запитів на додавання нових API-ключів користувача, а також отримання існуючих ключів із бази даних. Нижче представлено фрагмент реалізації відповідного сервісу:

Лістинг *2 – Сервіс API*

```
public class ApiKeyService : IApiKeyService
{
    private readonly DataRetrieving.DataRetrievingClient
_grpcClient;

    public
ApiKeyService(DataRetrieving.DataRetrievingClient
grpcClient)
    {
        _grpcClient = grpcClient;
    }
```

```
    public async Task<bool> AddApiKey(ApiKeyRequest
request)
    {
        var grpcRequest = new AddExchangeKeyRequest
        {
            UserId = Guid.NewGuid().ToString(),
            Exchange =
(int)Enum.Parse<Exchanges>(request.Exchange, true),
            ApiKey = request.ApiKey,
            ApiSecret = request.SecretKey,
            ApiPass = request.ApiPass ?? string.Empty
        };

        var response = await
_grpcClient.AddExchangeKeyAsync(grpcRequest);
        return response.Success;
    }

    public async Task<IEnumerable<ApiKeyDto>>
GetApiKeysAsync()
    {
        var result = await
_grpcClient.GetExchangeKeysAsync(new GetExchangeKeysRequest
        {
            UserId = "123"
        });

        return result.Keys.SelectMany(kvp =>
kvp.Value.Entries.Select(entry =>
            new ApiKeyDto(
                Guid.NewGuid().ToString(),
                "AccountName",
                kvp.Key,
```

```
                entry.ApiKey,
                entry.ApiSecret
            ))).ToList();
    }
}
```

У методі AddApiKey формується gRPC-запит AddExchangeKeyRequest, який містить параметри API-доступу до біржі, після чого цей запит передається до сервісу DataService. У методі GetApiKeysAsync відбувається виклик GetExchangeKeys, результат якого повертається у вигляді колекції ApiKeyDto. Варто зауважити, що для демонстраційних цілей UserId у методі отримання ключів жорстко закодовано як "123", що у реальному середовищі має бути замінено на ідентифікатор автентифікованого користувача.

Комунікація між шлюзом і сервісом даних базується на контрактах, визначених у файлі dataRetrieving.proto. Ось приклад оголошення відповідного RPC-сервісу:

Лістинг 3 – *Визначення контрактів*

```
service DataRetrieving {
  rpc AddExchangeKey(AddExchangeKeyRequest) returns
(AddExchangeKeyResponse);
  rpc GetExchangeKeys(GetExchangeKeysRequest) returns
(ExchangesKeysResponse);
}

message AddExchangeKeyRequest {
  string user_id = 1;
  int32 exchange = 2;
  string apiKey = 3;
  string apiSecret = 4;
  string apiPass = 5;
}

message ExchangesKeysResponse {
```

```
  bool success = 1;
  string message = 2;
  map<string, ExchangeKeysList> keys = 3;
}
```

Такий підхід дозволяє чітко описати типи даних та інтерфейс запитів/відповідей між сервісами, що спрощує підтримку та масштабування проєкту.

Взаємодія з користувачем реалізується через HTTP-ендпоінти, визначені у файлі ApiKeyEndpoints.cs. Вони дозволяють здійснювати POST-запити для додавання ключів, GET-запити для отримання списку, та потенційно DELETE-запити для видалення, як наведено на лістингу 3:

Лістинг *3 – Точки входу взаємодії з користувачем*

```
public static class ApiKeyEndpoints
{
    public static IEndpointRouteBuilder MapApiKeyEndpoints(this
IEndpointRouteBuilder app)
    {
        app.MapPost("/api/keys", async (ApiKeyRequest request,
IApiKeyService service) =>
        {
            var result = await service.AddApiKey(request);
            return Results.Ok(result);
        });

        app.MapGet("/api/keys", async (IApiKeyService service)
=>
        {
            var result = await service.GetApiKeysAsync();
            return Results.Ok(result);
        });

        app.MapDelete("/api/keys/{id}", async (Guid id) =>
        {
```

```
            // TODO: реалізувати видалення
            return Results.Ok(new { success = true });
        });

        return app;
    }
}
```

Ініціалізація застосунку, підключення gRPC-клієнтів та реєстрація служб відбувається у Program.cs. Основні кроки включають реєстрацію сервісу IApiKeyService, підключення gRPC-клієнта до DataService, увімкнення Swagger для тестування API, а також мапінг усіх маршрутів:

Лістинг *4 – Ініціалізація застосунку*

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
builder.Services.AddScoped<IApiKeyService,
ApiKeyService>();

builder.Services.AddGrpcClient<DataRetrievingGrpc.DataRetri
eving.DataRetrievingClient>(o =>
{
    o.Address = new Uri("http://localhost:5186");
});

var app = builder.Build();

app.MapApiKeyEndpoints();
// Додаткові маршрути:
app.MapAccountEndpoints();
app.MapPositionEndpoints();
app.MapTransferEndpoints();
```

```
app.MapOrderEndpoints();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();
app.Run();
```

Таким чином, компоненти ApiKeyService, ApiKeyEndpoints та конфігураційна логіка в Program.cs утворюють повноцінний шар шлюзу для роботи з API-ключами. Ця архітектура дозволяє централізувати взаємодію з користувачем, забезпечуючи при цьому надійне розділення відповідальності між фронтендом, шлюзом і внутрішніми мікросервісами.

3.3.3 Огляд клієнтської частини

Клієнтський інтерфейс спроектовано для забезпечення комфортної та інтуїтивної взаємодії з системою, що дає користувачам можливість ефективно вирішувати ключові завдання: управління акаунтами, аналіз ринкових даних та здійснення торгових операцій. Навігація та візуальні елементи сприяють швидкому доступу до потрібних функцій. Приміром, вбудований торговий графік показує динаміку вартості BTC/USDT на біржі Binance у режимі реального часу, надаючи інструмент для дослідження ринкових тенденцій (див. Рисунок 2).

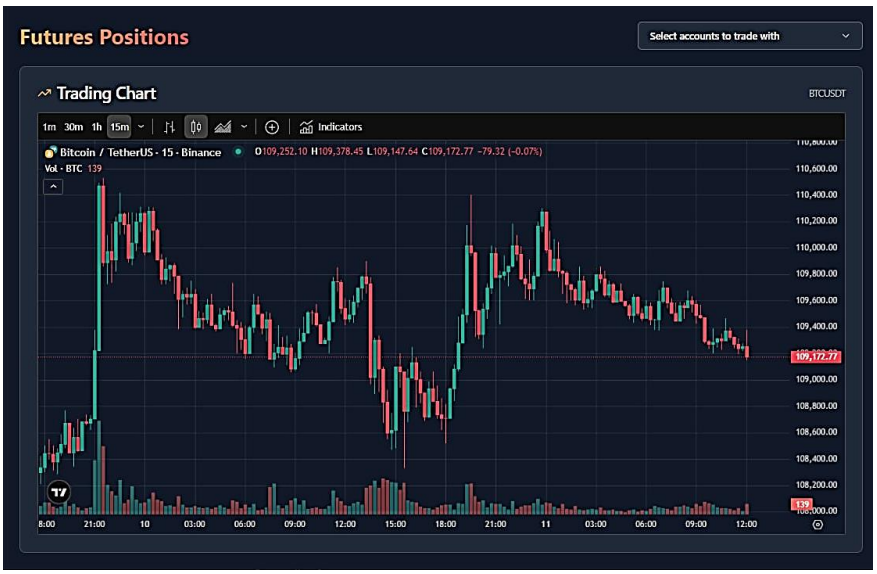


Рисунок *2 — Графік ціни BTC/USDT з біржі Binance*

Функціонал додавання API-ключів забезпечує підключення до криптовалютних бірж, таких як Binance, Kraken чи Coinbase. Користувачі вводять ключі через форму з автоматичною перевіркою даних та вибором біржі зі списку, що робить процес налаштування простішим (див. Рисунок 3).

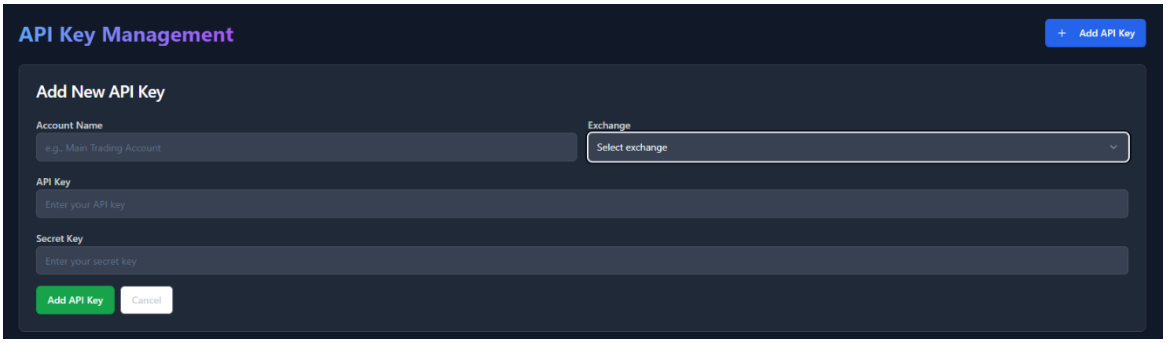


Рисунок *3 — Інтерфейс додавання нового API-ключа*

Панель управління акаунтами та торговими операціями дає змогу миттєво перемикатися між акаунтами на різних біржах та виконувати операції, зокрема перекази коштів чи створення ф’ючерсних ордерів. Цей модуль підтримує різні типи операцій, гарантуючи гнучкість в управлінні активами (див. Рисунок 4).

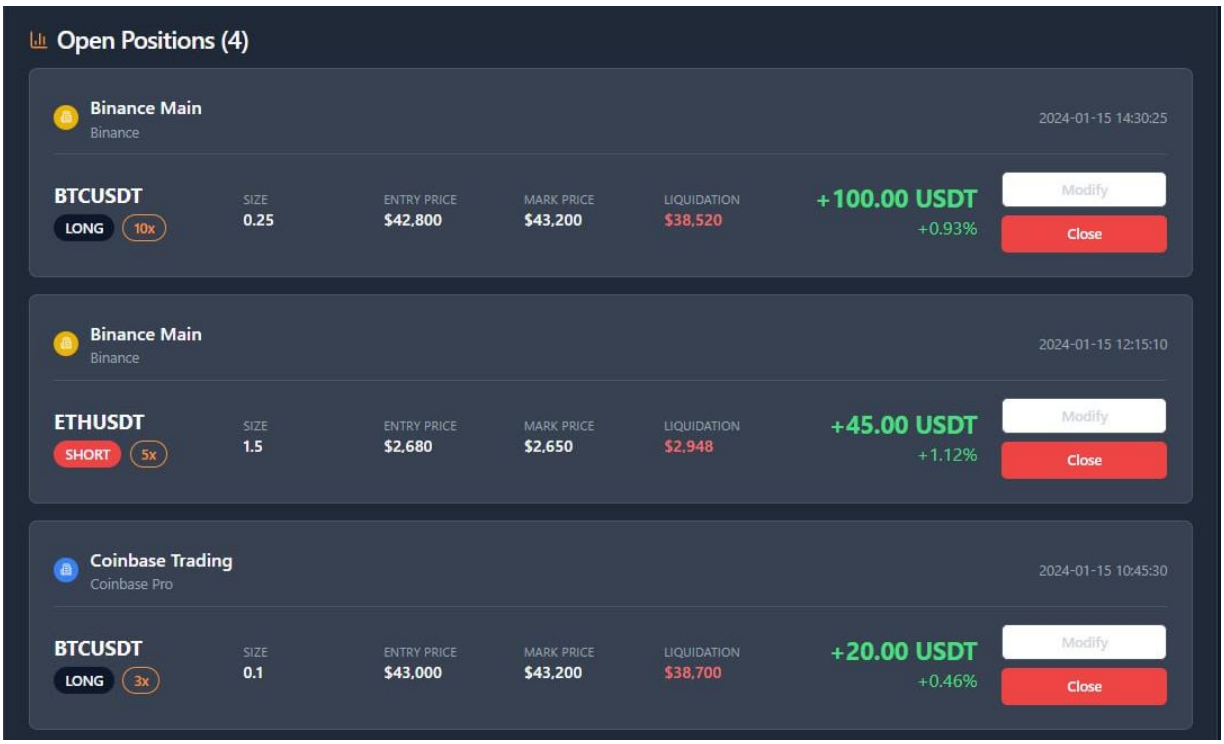


Рисунок *4 — Панель торгових дій і перемикання акаунтів*

Для відстеження торгової активності створено інтерактивну панель відкритих позицій, яка відображає всі активні ф’ючерсні позиції користувача на різних біржах. Панель містить детальну інформацію про обсяг позиції, кредитне плече, ціну входу, поточну ринкову ціну, ціну ліквідації, а також обчислений прибуток чи збиток. Користувачі мають можливість змінювати або закривати позиції безпосередньо через інтерфейс, що збільшує ефективність управління (див. Рисунок 5).

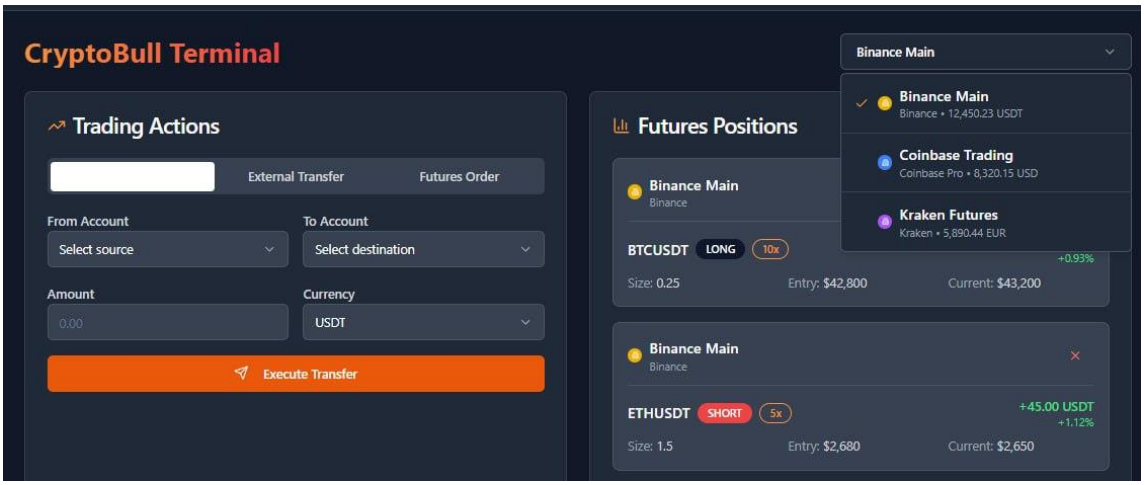


Рисунок 5 — *Перелік відкритих позицій на ф’ючерсному ринку*

Така структура клієнтського інтерфейсу забезпечує повноцінну взаємодію з системою, від керування акаунтами до прийняття обґрунтованих торгових рішень. Завдяки модульній архітектурі інтерфейс легко адаптується до нових вимог, дозволяє інтегрувати сторонні API та масштабувати функціональність без значних змін у базовій логіці.

3.4 Розробка мікросервісної архітектури

Інформаційна система управління криптографічними рахунками побудована на мікросервісній архітектурі, яка передбачає поділ функціоналу на окремі сервіси. Кожен сервіс виконує конкретні задачі та комунікує з іншими через стандартизовані інтерфейси, що забезпечує гнучкість, масштабованість та спрощує технічне обслуговування системи в процесі експлуатації.

В основі системи знаходяться чотири ключові мікросервіси, які спільно гарантують повноцінну роботу застосунку. API Gateway виступає центральним вхідним вузлом, обробляючи HTTP-запити від клієнтської сторони, перетворюючи їх на gRPC-виклики та перенаправляючи до відповідних сервісів. Цей компонент відповідає за маршрутизацію запитів, базову авторизацію, агрегацію результатів та формування стандартизованих відповідей, приховуючи внутрішню структуру системи від зовнішніх користувачів, що збільшує безпеку та гнучкість.

AuthService відповідає за безпечне керування обліковими даними користувачів. Він обробляє процеси реєстрації, авторизації та керування API-ключами для підключення до криптовалютних бірж, використовуючи хешування паролів та токени JWT. Виокремлення аутентифікації в окремий сервіс дає змогу масштабувати ці процеси незалежно від інших компонентів і забезпечує централізовану перевірку прав доступу.

DataService виконує роль централізованого сховища даних, обробляючи

операції створення, оновлення, зчитування та видалення (CRUD) для даних користувачів, API-ключів та облікових записів. Використання gRPC-інтерфейсу забезпечує єдиний підхід до взаємодії з реляційною базою даних PostgreSQL, оптимізуючи індексацію, транзакції та міграції. Відсутність бізнес-логіки в цьому сервісі сприяє чіткому розподілу обов'язків між компонентами.

ProcessorService забезпечує інтеграцію з API криптовалютних бірж, таких як Binance, Gate.io та Bybit, через протоколи REST та WebSocket. Цей сервіс обробляє ключові операції, зокрема отримання балансів, створення ринкових та лімітних ордерів, виведення коштів, генерацію депозитних адрес та внутрішні перекази між рахунками. Стандартизований підхід до виконання операцій на різних платформах забезпечує структуровані відповіді для інших модулів системи.

Взаємодія між мікросервісами відбувається за допомогою протоколу gRPC, який застосовує бінарний формат Protocol Buffers для швидкого та ефективного обміну даними. Висока швидкість обробки, автоматична генерація клієнтського коду та підтримка стрімінгових запитів роблять gRPC оптимальним вибором для системи. Наприклад, під час додавання API-ключа користувач надсилає запит до API Gateway, який перевіряє токен через AuthService, зберігає ключ у DataService, а для здійснення торгових операцій звертається до ProcessorService, який повертає результат.

Мікросервісна архітектура надає ряд переваг. Незалежне розгортання та масштабування сервісів підвищує стійкість системи до збоїв, оскільки відмова одного компонента не впливає на функціонування інших. Чіткий розподіл обов’язків спрощує розробку, тестування та підтримку. Архітектура дозволяє оперативно адаптувати систему до нових вимог, наприклад, додавати сервіси для аналітики чи керування ризиками, без значних змін наявної структури.

3.5 Програмна реалізація серверної частини

3.5.1 Програмна реалізація сервісу автентифікації (AuthService)

Сервіс автентифікації AuthService – це наріжний камінь мікросервісної архітектури системи. Він відповідає за обробку запитів на реєстрацію й авторизацію користувачів, генерування токенів доступу та управління API-ключами для взаємодії з криптовалютними біржами. Розроблений на платформі ASP.NET Core з використанням протоколу gRPC для міжсервісної комунікації, AuthService гарантує надійне та продуктивне опрацювання облікових даних.

Структура проєкту AuthService вибудувана логічно, з поділом на компоненти, що чітко розділяють функціональні обов'язки та полегшують супровід коду (див. Рисунок 6).

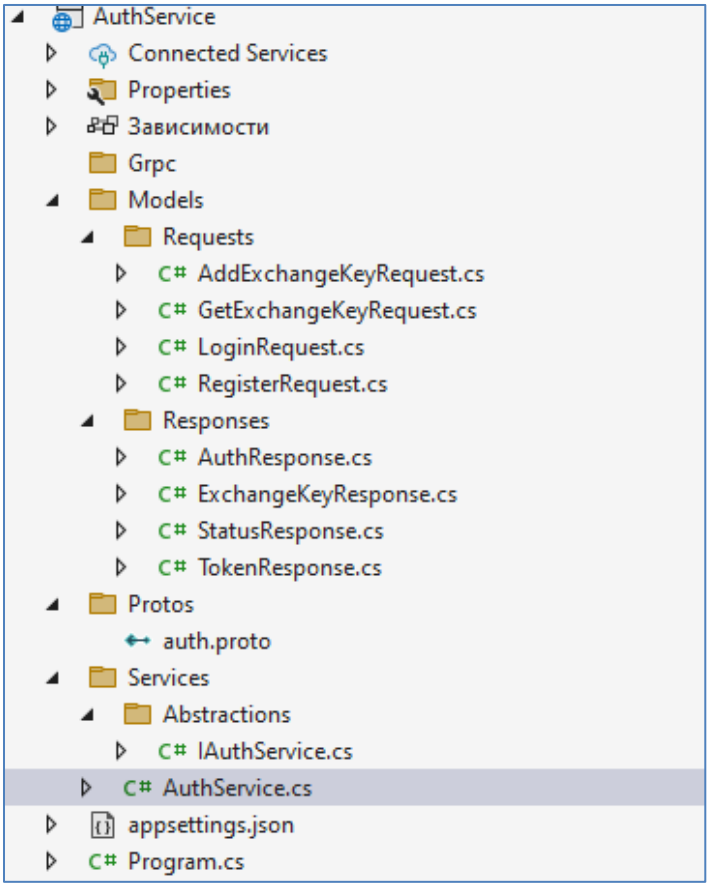


Рисунок *6 — Структура проєкту AuthService*

Основні компоненти включають:

- Grpc/ — містить автозгенеровані класи з файлу auth.proto, які визначають контракти для взаємодії AuthService з іншими мікросервісами через gRPC.
- Models/ — включає моделі даних для типізації запитів і відповідей, такі як RegisterRequest, LoginRequest, AddExchangeKeyRequest, а також відповідні моделі відповідей (AuthResponse, TokenResponse, StatusResponse, ExchangeKeyResponse).
- Protos/ — містить файл auth.proto, де описано структуру gRPC-запитів і відповідей.
- Services/ — реалізує бізнес-логіку через інтерфейс IAuthService і клас AuthService.cs, який відповідає за виконання методів, визначених у gRPC-контрактах.

Файл auth.proto визначає ключові методи для реалізації основних сценаріїв використання:

Лістинг 5 – *Сервіс авторизації*

```
service AuthService {
  rpc Register(RegisterRequest) returns (AuthResponse);
  rpc Login(LoginRequest) returns (TokenResponse);
  rpc AddExchangeKey(AddExchangeKeyRequest) returns
(StatusResponse);
  rpc GetExchangeKey(GetExchangeKeyRequest) returns
(ExchangeKeyResponse);
}
```

Ці підходи забезпечують реєстрацію нового користувача, авторизацію, додавання API-ключів для бірж та отримання збережених даних. Строга типізація даних у контрактах сприяє взаємодії між мікросервісами та полегшує масштабування системи.

Бізнес-логіка сервісу реалізована в класі AuthService.cs, котрий обробляє методи, визначені в gRPC-контрактах. Приміром, метод Login перевіряє

облікові дані, звертаючись до DataService через gRPC для пошуку користувача в базі даних. У разі успіху генерується JWT-токен доступу за допомогою бібліотеки System.IdentityModel.Tokens.Jwt, котрий повертається клієнту у вигляді TokenResponse. Метод AddExchangeKey валідує запит, нормалізує дані та передає API-ключ до DataService для збереження.

Обробка запитів у AuthService базується на наступних принципах:

- Сувора валідація вхідних даних задля забезпечення безпеки.
- Використання інтерфейсів (IAuthService) для управління залежностями.
- Взаємодія з DataService через gRPC для здійснення CRUD-операцій.
- Захист даних, зокрема хешування паролів перед збереженням та безпечна робота з токенами.

Реєстрація користувача здійснюється через обробку RegisterRequest, котрий передається до DataService для збереження даних. Під час авторизації система порівнює хеш пароля з LoginRequest із збереженим значенням у базі, забезпечуючи надійну перевірку.

Генерація токенів доступу виконується після успішної авторизації. AuthService генерує JWT-токен, що повертається клієнту через API Gateway у форматі TokenResponse. Цей механізм забезпечує безпечний доступ до системи та її ресурсів.

3.5.2 Програмна реалізація DataService

Сервіс DataService виконує роль централізованого сховища даних у розподіленій мікросервісній архітектурі. Його основне призначення полягає в наданні доступу до персистентного зберігання інформації про користувачів, API-ключі та пов’язані біржові акаунти. Сервіс ізольований від клієнтського інтерфейсу та взаємодіє з іншими мікросервісами винятково через gRPC-контракти.

Архітектурно DataService є CRUD-сервісом, побудованим на основі ASP.NET Core, який використовує Entity Framework Core для взаємодії з реляційною СУБД PostgreSQL. Усі дані зберігаються у вигляді зв’язаних

сутностей з чітко визначеними зовнішніми ключами, що дозволяє зберігати структуровану інформацію з дотриманням цілісності.

У структурі моделі даних передбачено дві ключові сутності: UserEntity та ApiKeyEntity. Вони мають зв’язок "один до багатьох", що відображає можливість одного користувача мати декілька API-ключів до різних бірж.

Лістинг *6 –Сутності користувача та API ключа*

```
public class UserEntity
{
    public Guid Id { get; set; } = Guid.NewGuid();
    public string Email { get; set; }
    public string PasswordHash { get; set; }

    public ICollection<ApiKeyEntity> ExchangeKeys { get;
set; } = new List<ApiKeyEntity>();
}

public class ApiKeyEntity
{
    public Guid Id { get; set; } = Guid.NewGuid();
    public Guid UserId { get; set; }
    public Exchanges Exchange { get; set; }
    public string ApiKey { get; set; }
    public string ApiSecret { get; set; }
    public string? ApiPassword { get; set; }
}
```

Ці класи описують структуру таблиць, які створюються за допомогою механізмів міграцій EF Core. Зв’язок між UserEntity та ApiKeyEntity реалізовано через зовнішній ключ UserId.

gRPC-інтерфейс DataService визначено у файлі dataRetrieving.proto. Він містить типові запити для запису й зчитування даних:

Лістинг 7 – *Сервіс взаємодії gRPC*

```
service DataRetrieving {
  rpc AddUser(AddUserToDbRequest) returns
(AddUserToDbResponse);
  rpc GetUserByEmail(GetUserByEmailRequest) returns
(GetUserByEmailResponse);
  rpc GetExchangeKeys(GetExchangeKeysRequest) returns
(ExchangesKeysResponse);
  rpc AddExchangeKey(AddExchangeKeyRequest) returns
(AddExchangeKeyResponse);
}
```

Усі запити типізовано через Protocol Buffers, що забезпечує ефективну серіалізацію та десеріалізацію даних.

Сервіс надає метод AddUser, який перевіряє наявність користувача за email і в разі відсутності створює новий обліковий запис у базі. Аналогічно реалізовано метод GetUserByEmail, що виконує пошук користувача й повертає хешований пароль для подальшої перевірки в AuthService.

Метод AddExchangeKey створює запис API-ключа, пов’язаний із відповідним користувачем, а GetExchangeKeys повертає повну карту ключів, згруповану за біржею, що дає змогу обробляти ці дані в інтерфейсі користувача.

У сервісі використано патерн DbContext для інкапсуляції доступу до таблиць:

Лістинг *8* – *Контекст бази даних*

```
public class ApplicationDbContext : DbContext
{
    public DbSet<UserEntity> Users { get; set; }
    public DbSet<ApiKeyEntity> ApiKeys { get; set; }

    public
```

```
    ApplicationDbContext(DbContextOptions<ApplicationDbCont
    ext> options)
        : base(options) { }
}
```

Завдяки інтеграції з EF Core забезпечується автоматична генерація міграцій, lazy loading сутностей, валідація зв’язків, а також підтримка LINQ для зручного виконання запитів.

Усі операції запису реалізовано з використанням транзакцій, що гарантує консистентність даних у випадку одночасного звернення з боку кількох сервісів. Виняткові ситуації (наприклад, дублювання email) обробляються з поверненням відповідного статусу та повідомлення для виклику.

Таким чином, DataService виконує роль надійного централізованого сховища, з яким взаємодіють інші мікросервіси, не маючи прямого доступу до бази даних. Це дозволяє гарантувати цілісність інформації, логічну ізоляцію шарів і спрощення подальшого масштабування та супроводу системи.

3.5.3 Програмна реалізація ProcessorService

Сервіс ProcessorService є центральною обчислювальною компонентою системи, відповідальною за виконання операцій, пов’язаних із криптовалютною торгівлею, зокрема обробку запитів користувачів щодо розміщення ордерів, отримання залишків, переказів між рахунками, виведення коштів тощо. Він безпосередньо взаємодіє з API криптобірж (наприклад, Binance, Gate.io), інкапсулюючи низькорівневу логіку в спеціалізовані сервіси.

Архітектура ProcessorService побудована за принципами модульності та абстракції. Кожна біржа, з якою працює система, має окремий сервіс — наприклад, BinanceService, який реалізує спільний інтерфейс IExchangeService. Це дозволяє інкапсулювати специфіку API кожної платформи та спростити масштабування сервісу під нові торгові майданчики.

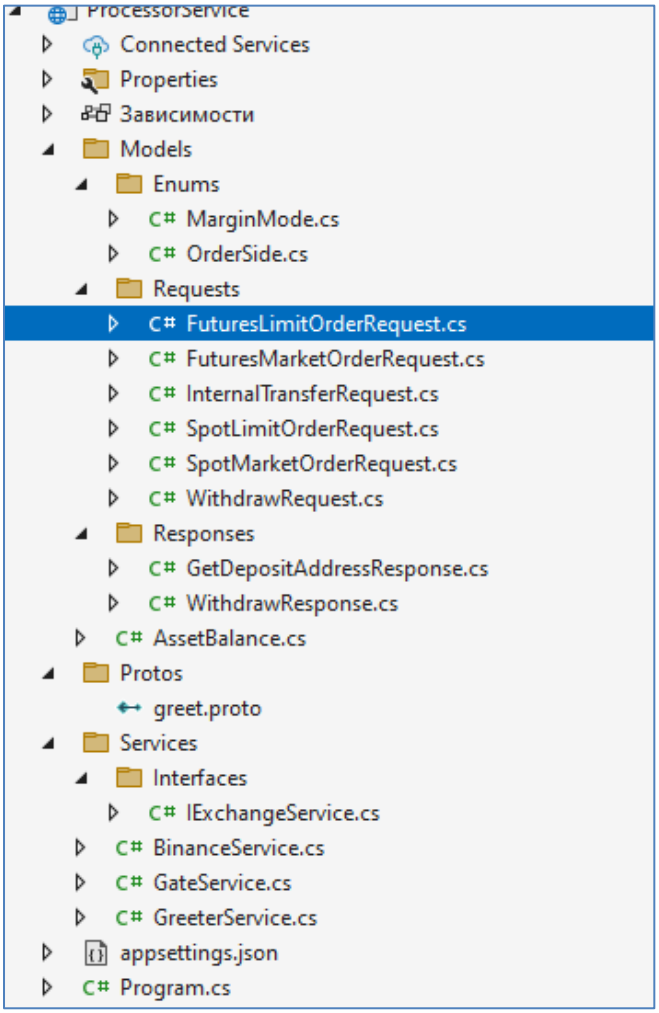


Рисунок 7 — *Структура проєкту ProcessorService*

Всі реалізації біржових сервісів повинні реалізовувати інтерфейс IExchangeService. Він описує набір методів для взаємодії зі спотовими та ф’ючерсними ринками, включаючи запити балансів, розміщення маркет- та ліміт-ордерів, отримання адрес депозиту тощо.

Лістинг 9 –*Інтерфейс взаємодії з біржовими сервісами*

```
public interface IExchangeService
{
    Task<Dictionary<string, List<AssetBalance>>>
    GetBalances();
    Task PlaceMarketSpotOrderAsync(SpotMarketOrderRequest
    request);
    Task PlaceLimitSpotOrderAsync(SpotLimitOrderRequest
```

```
    request);
    Task PlaceMarketOrderAsync(FuturesMarketOrderRequest
    request);
    Task PlaceLimitOrderAsync(FuturesLimitOrderRequest
    request);
    Task<WithdrawResponse> Withdraw(WithdrawRequest
    request);
    Task<GetDepositAddressResponse>
    GetDepositAddress(string symbol);
    Task
    TransferBetweenSpotAndFutures(InternalTransferRequest
    request);
}
```

Клас BinanceService є прикладом реалізації клієнта для платформи Binance. У конструкторі сервісу ініціалізуються REST- і WebSocket-клієнти з відповідними API-ключами.

Лістинг *10 – Сервіс взаємодії з платформої Binance*

```
public BinanceService(string apiKey, string secret)
{
    _binanceRestClient.SetApiCredentials(new
    ApiCredentials(apiKey, secret));
    _socketClient.SetApiCredentials(new
    ApiCredentials(apiKey, secret));
    _restClient = new
    RestClient("https://api.binance.com");
}
```

Один з ключових методів — GetBalances, який повертає поточні залишки по гаманцях користувача:

Лістинг 11 – *Отримання балансу*

```
public async Task<Dictionary<string, List<AssetBalance>>>
```

```
    GetBalances()
{
    var result = new Dictionary<string,
    List<AssetBalance>>();
    var perpBalanceRequest = await
    _binanceRestClient.UsdFuturesApi.Account.GetBalancesAsyn
    c();
    var spotBalanceRequest = await
    _binanceRestClient.SpotApi.Account.GetBalancesAsync();

    var futuresBalances = perpBalanceRequest.Success
        ? perpBalanceRequest.Data
            .Where(b => b.WalletBalance > 0)
            .Select(b => new AssetBalance(b.Asset,
    b.WalletBalance))
            .ToList()
        : new List<AssetBalance>();

    var spotBalances = spotBalanceRequest.Success
        ? spotBalanceRequest.Data
            .Where(b => b.Total > 0)
            .Select(b => new AssetBalance(b.Asset,
    b.Total))
            .ToList()
        : new List<AssetBalance>();

    result["spot"] = spotBalances;
    result["futures"] = futuresBalances;

    return result;
}
```

Для передачі параметрів на створення ордерів використовуються типи запитів (DTO). Наприклад, SpotMarketOrderRequest:

Лістинг *12 - Створення замовлення*

```
public record SpotMarketOrderRequest(string Pair,
    decimal? TokenQuantity,
    decimal? QuoteQuantity,
    OrderSide Side)
{
    public bool IsValid => TokenQuantity > 0 ||
    QuoteQuantity > 0;
}
```

Розміщення маркет-ордера на спотовому ринку реалізовано наступним чином:

Лістинг *13 – Розміщення маркет замовлення*

```
public async Task
    PlaceMarketSpotOrderAsync(SpotMarketOrderRequest
    request)
{
    var orderSide = request.Side == OrderSide.Buy
        ? Binance.Net.Enums.OrderSide.Buy
        : Binance.Net.Enums.OrderSide.Sell;

    var orderResult = await
    _binanceRestClient.SpotApi.Trading.PlaceOrderAsync(
        request.Pair,
        orderSide,
        SpotOrderType.Market,
        quantity: request.QuoteQuantity);

    if (orderResult.Success)
        Log.Information("✅ Spot order placed: {Id}",
```

```
    orderResult.Data.Id);
    else
        Log.Error("❌ Failed to place spot order: {Error}",
    orderResult.Error?.Message);
}
```

У випадку помилки вона фіксується через систему логування Serilog.

Файл greet.proto слугує шаблоном для демонстрації реалізації gRPC-взаємодії. Він визначає базовий контракт, який може бути розширений для всіх доступних функцій:

Лістинг *14 – Приклад базового контракту*

```
service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}
```

У практичній реалізації створено окремий ProcessorServiceGrpc, який надаватиме методи:

- GetBalances
- PlaceMarketOrder
- TransferBetweenSpotAndFutures
- Withdraw тощо.

Таким чином, ProcessorService є стратегічним компонентом системи, що забезпечує виконання основних торгових операцій із використанням реальних API-бірж. Його розширювана архітектура, сувора типізація та модульність сприяють створенню стабільної, масштабованої та безпечної системи криптовалютного трейдингу.

3.6 Розробка дизайну застосунку

Користувацький інтерфейс є критично важливим елементом вебзастосунку для моніторингу та управління криптовалютними рахунками, оскільки саме він забезпечує зручність, доступність і ефективну взаємодію

користувача з системою. У процесі розробки інтерфейсу було дотримано принципів UX/UI-дизайну, що передбачають зрозумілу структуру, адаптивність, візуальну ієрархію та мінімізацію когнітивного навантаження.

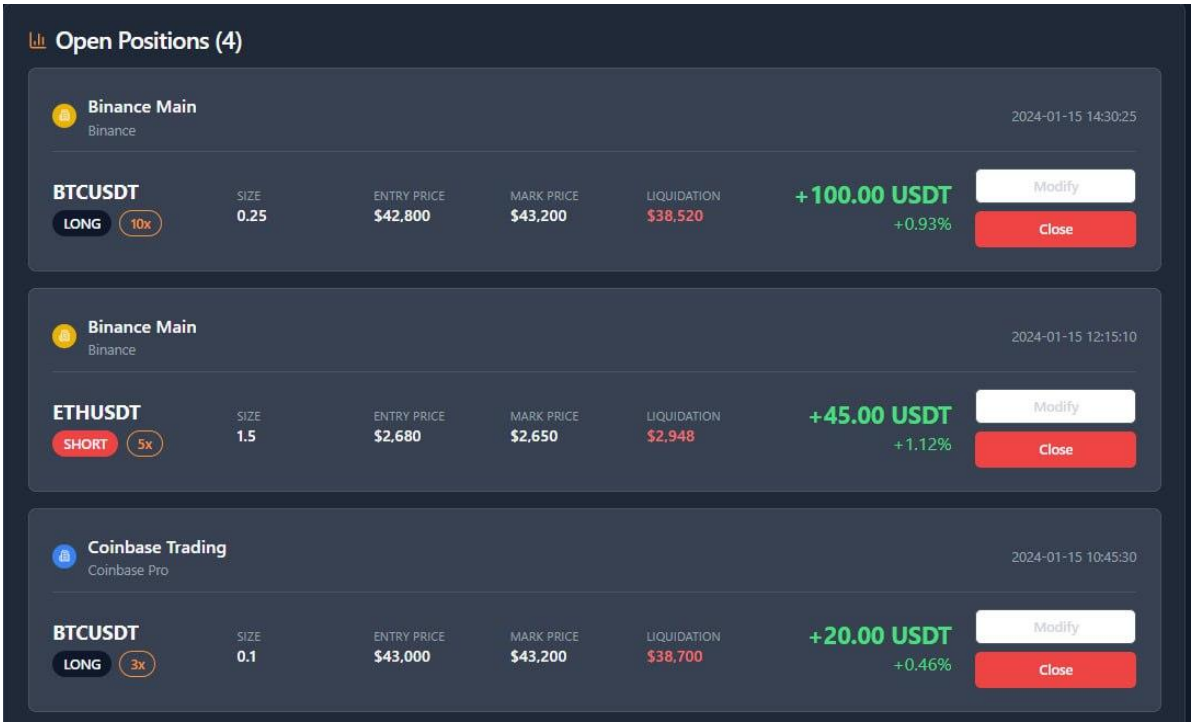


Рисунок *8 — Відкриті позиції користувача*

Інтерфейс реалізований як окремий вебдодаток із сучасним фреймворком для frontend-розробки. Основну увагу приділено побудові логічно впорядкованої панелі керування, яка відображає дані в реальному часі, підтримує виконання транзакцій та дозволяє керувати налаштуваннями акаунтів.

Однією з ключових сторінок є панель управління відкритими позиціями, що візуалізує інформацію про кожну активну угоду користувача: валютну пару, тип позиції (LONG/SHORT), обсяг, плече, вхідну та поточну ціну, рівень ліквідації та прибуток/збиток.

Окрему роль відіграє блок графічного відображення ринку, який інтегрує сторонній компонент для побудови торгового графіка. Користувач може перемикати таймфрейми, додавати індикатори та оцінювати ринкову ситуацію безпосередньо у застосунку.

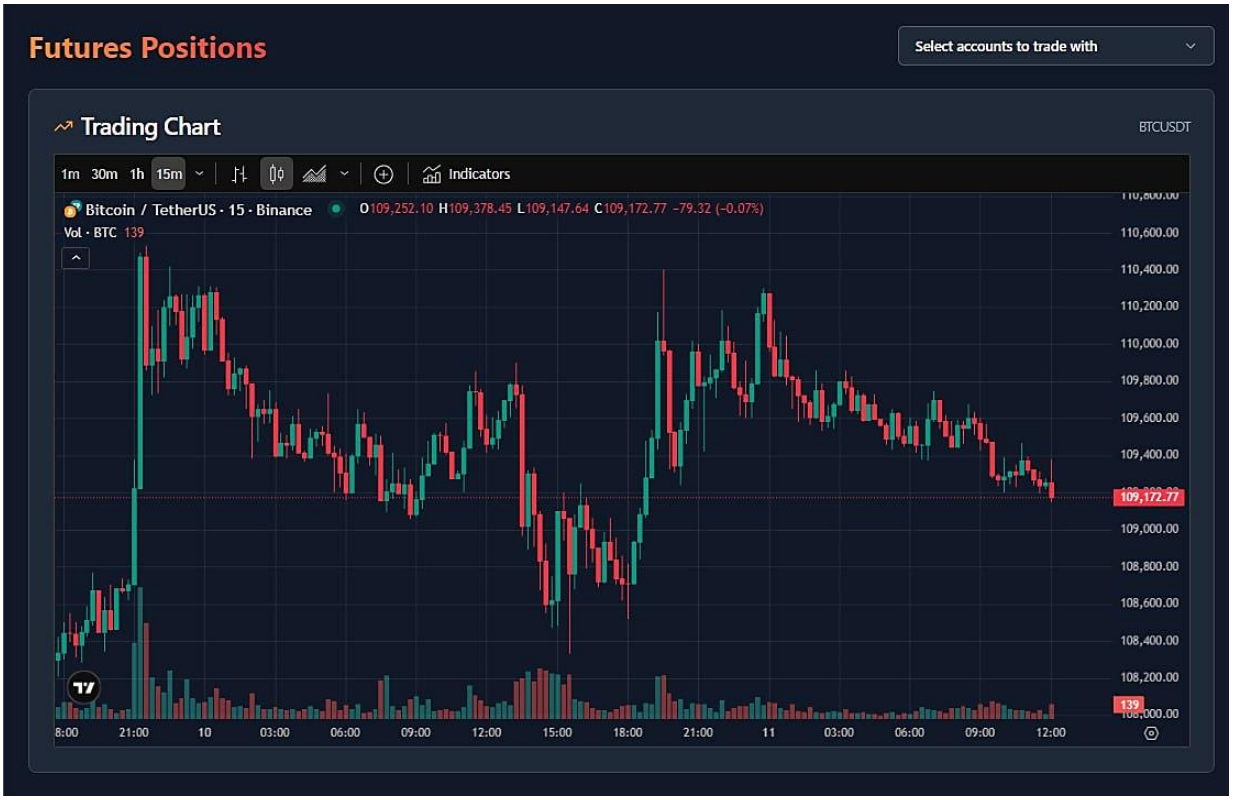


Рисунок *9 — Графік динаміки ціни торгової пари*

У секції торгових дій реалізовано два основних режими — створення ф’ючерсного ордера та переказ коштів між рахунками. Поля введення супроводжуються валідацією та підказками, що підвищує точність введення.

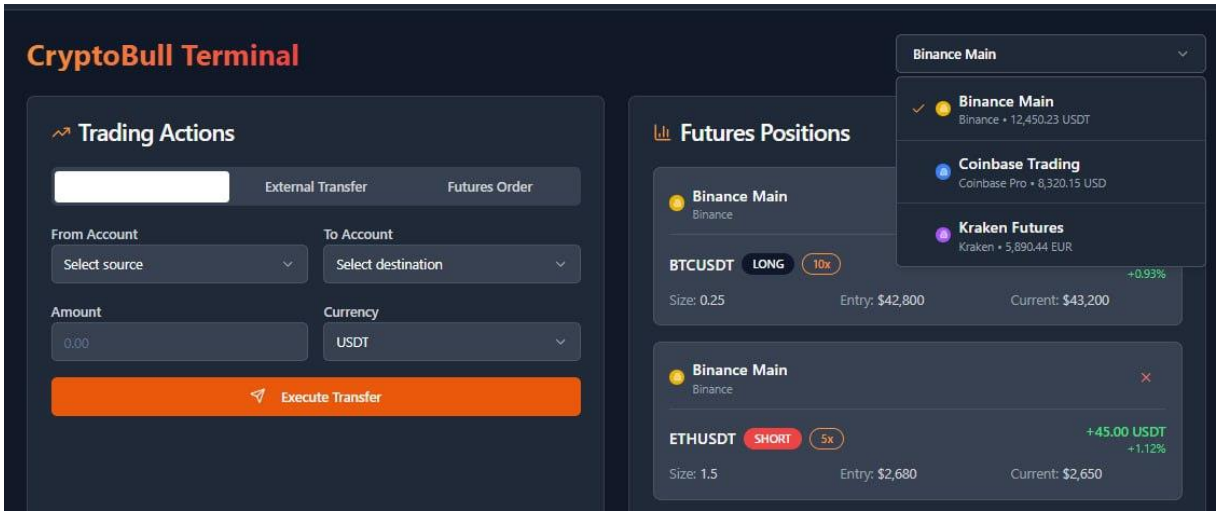


Рисунок *10 — Трансфер та відкриття нової позиції*

Інтерфейс для додавання API-ключів реалізований з підтримкою перевірки введених даних, автозаповнення та вибору біржі зі списку.

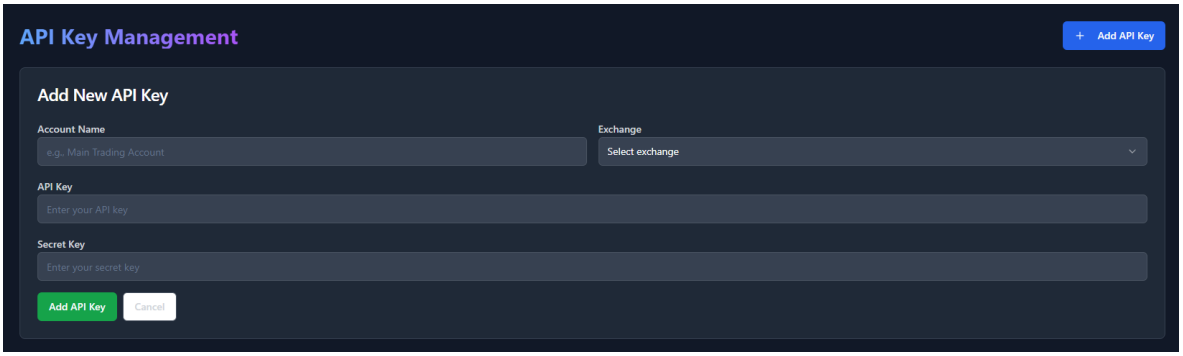


Рисунок *11 — Додавання нового API-ключа*

Загальне оформлення застосунку витримане в темній кольоровій гамі, що відповідає сучасним тенденціям UI-дизайну для фінансових систем. Для підсвічування важливих елементів використано контрастні кольори: зелений — для прибуткових позицій, червоний — для збиткових, помаранчевий — для кнопок дій.

Шрифти, відступи, інтерактивні компоненти та анімації забезпечують інтуїтивну навігацію без необхідності вивчення інструкцій.

Інтерфейс підтримує адаптивну верстку та може використовуватися як браузері на ПК, так і на мобільних пристроях. Це досягається за рахунок гнучкої сітки, масштабованих компонентів і відповідного CSS-дизайну.

ВИСНОВКИ

Основною метою кваліфікаційної роботи було створення комплексного веб-додатку з сучасною мікросервісною архітектурою для відстеження та управління криптовалютними рахунками, в результаті виконання роботи можно зробити наступні висновки:

1. Було продемонстровано, що використання стратегії мікросервісів для створення масштабованих та гнучких систем є можливим:

Функціонал був розділений на окремі сервіси (AuthService, DataService, ProcessorService та APIGateway) для забезпечення модульності, незалежного розгортання та легкої підтримки.

2. Реалізовано безпечну взаємодію з криптовалютними біржами:

Сервіс ProcessorService використовує протоколи WebSocket і REST API для обробки ключових торгових завдань (відкриття/закриття позицій, переказ коштів, запит балансу тощо), які забезпечують швидкість і надійність.

3. Забезпечено централізоване зберігання користувацьких даних:

У DataService реалізовано логіку додавання, пошуку та отримання ключів API, які пов’язано з окремими користувачами системи. Схема даних проєктувалась з урахуванням вимог до безпеки, масштабованості та цілісності.

4. Створено клієнтський інтерфейс, орієнтований на кінцевого користувача: Інтерфейс реалізовано як окремий фронтенд-застосунок із зручною навігацією, можливістю додавання ключів, перегляду позицій та управління торговими операціями. Завдяки використанню REST API через APIGateway, забезпечено синхронізацію з внутрішніми сервісами системи.

5. Проведено повний цикл програмної реалізації:

Від аналізу предметної області до тестування й демонстрації працездатності системи. Було використано такі технології: ASP.NET 8, gRPC, PostgreSQL, C#, Binance.Net, RestSharp, Docker, Swagger.

Загалом, розроблений програмний продукт підтвердив ефективність обраного технологічного стеку для побудови інформаційної системи управління криптовалютними рахунками. Рішення може бути масштабовано та розширено для підтримки нових бірж, інтеграції аналітики та автоматизації торгівлі, що відкриває нові перспективи його розвитку.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Poster POS. URL: https://joinposter.com/ua. (дата звернення: 25.04.2025).
2. R-Keeper. URL: https://ucs.ua/r-keeper/. (дата звернення: 25.04.2025).
3. IEEE Xplore Digital Library. Статті з веброзробки та безпеки. URL: https://ieeexplore.ieee.org. (дата звернення: 25.04.2025).
4. Binance API Documentation. URL: https://binance-docs.github.io/apidocs/spot/en/. (дата звернення: 25.04.2025).
5. .NET 8 Documentation — Microsoft Learn. URL: https://learn.microsoft.com/en-us/dotnet/core/dotnet-eight. (дата звернення: 25.04.2025).
6. RestSharp GitHub. URL: https://github.com/restsharp/RestSharp. (дата звернення: 25.04.2025).
7. gRPC Documentation. URL: https://grpc.io/docs/. (дата звернення: 25.04.2025).
8. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/. (дата звернення: 25.04.2025).
9. Swagger/OpenAPI Tools. URL: https://swagger.io/tools/. (дата звернення: 25.04.2025).
10. Binance.Net — .NET API Wrapper. URL: https://github.com/JKorf/Binance.Net. (дата звернення: 25.04.2025).
11. Docker Documentation. URL: https://docs.docker.com/. (дата звернення: 25.04.2025).

Декларація
академічної доброчесності
здобувача ступеня вищої освіти ЗНУ

Я, Семенюк Олександр Васильович, студент 4 курсу, форми навчання денної, Інженерного навчально-наукового інституту ім. Ю.М. Потебні ЗНУ, спеціальність 121 Інженерія програмного забезпечення освітньої програми "Програмне забезпечення систем", адреса електронної пошти se121-18@stu.zsea.edu.ua, — підтверджую, що написана мною кваліфікаційна робота на тему «**Web застосунок для менеджмету та моніторингу кріптовалютних рахунків**» відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений.

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

- згоден на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-системи, а також на архівування моєї роботи в базі даних цієї системи.

Дата 02.06.2025 _____ О. В. Семенюк
(підпис) (прізвище та ініціали) (студент)

Дата 02.06.2025 _____ О.О. Жуков
(підпис) (прізвище та ініціали) (керівник)