Міжнародний європейський університет

Навчально-науковий інститут «Європейська школа бізнесу»

Кафедра інформаційних технологій

ДОПУСТИТИ ДО ЗАХИСТУ

Завідувач кафедри

інформаційних технологій

_____ О. В. Нестеренко

«___» _____ 2026 р.

КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА (ПОЯСНЮВАЛЬНА ЗАПИСКА)

зі спеціальності 121 «Інженерія програмного забезпечення»

ТЕМА: «Розробка веб-сайту для мультибіржового відстеження криптовалютних інвестицій»

Виконавець: _____ Малишко Ю. В.

(група ІПЗ-22-401з)

Науковий керівник: _____ Яцук П. П.

КИЇВ – 2026

МІЖНАРОДНИЙ ЄВРОПЕЙСЬКИЙ УНІВЕРСИТЕТ
ННІ «Європейська школа бізнесу»
Кафедра інформаційних технологій

ЗАТВЕРДЖУЮ:
Завідувач кафедри інформаційних технологій
_____Олександр УЄСТЕРЕНКО
« » _____ 2026_ р.

З А В Д А Н Н Я
НА ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ БАКАЛАВРСЬКОЇ РОБОТИ

Малишко Юрій Володимирович, ІПЗ-22-401з .
(ПІБ студента, група)

Ступінь вищої освіти - перший (бакалаврський) рівень
Спеціальність F2 «Інженерія програмного забезпечення»
Освітньо-професійна програма «Інженерія програмного забезпечення»
Освітньо-кваліфікаційний рівень бакалавр

1. Тема: Розробка веб-сайту для мультибіржового відстеження криптовалютних інвестицій,
затверджена наказом Ректора від «09» лютого 2026 р. №25-с.

2. Термін виконання роботи: з «26» травня 2026 р. по «22» червня 2026 р.

3. Дата подання роботи на випускну кафедру: «12» червня 2026 р.

4. Вихідні дані роботи: Аналіз предметної області криптовалютних інвестицій та API криптобірж; розробку веб-орієнтованої інформаційної системи для мультибіржового відстеження криптовалютних інвестицій; методи проєктування програмних систем; методи проєктування баз даних; алгоритми розрахунку фінансових показників.

5. Зміст пояснювальної записки:
1. Вступ.
2. Аналіз предметної області та існуючих програмних рішень.

3. Формування функціональних і нефункціональних вимог до системи.
4. Проєктування архітектури програмного забезпечення та структури бази даних.
5. Розробка та реалізація веб-додатку (інтеграція з API криптобірж).
6. Забезпечення інформаційної безпеки та захисту даних користувачів.
7. Тестування, валідація та аналіз результатів роботи системи.
8. Висновки.
9. Список використаних джерел.

6. Перелік обов'язкового графічного матеріалу (слайди презентації):
1. Актуальність і мета роботи.
2. Аналіз аналогів.
3. Архітектура системи.
4. Структурна схема та ER-діаграма.
5. Алгоритм розрахунку показників.
6. Інтерфейс користувача.
7. Результати тестування.
8. Висновки.

7. Календарний план - графік

№ з/п	Завдання	Термін виконання	Відмітка про виконання
1.	Складання і затвердження індивідуальних завдань на виконання кваліфікаційної бакалаврської роботи (КБР)	27.02.26	
2.	Підготовка вступу і розділу 1 КБР	26.03.26	
3.	Підготовка розділу 2 КБР	23.04.26	
4.	Підготовка розділу 3 КБР, висновків і переліку використаних джерел	21.05.26	
5.	Подання студентом завершеної КБР науковому керівнику для перевірки на плагіат та оформлення відгуку	05.06.26	
6.	Попередній розгляд КБР на комісії від кафедри	11-12.06.26	
7.	Доопрацювання роботи, прийняття кафедрою рішення про допуск роботи до захисту в ЕК, оформлення та зовнішнє рецензування	15-19.06.26	
8.	Захист кваліфікаційної бакалаврської роботи в ЕК і присвоєння випускникам кваліфікації	23-24.06.26	

Студент _____Юрій Малишко_____
(підпис, ПІБ)

Керівник _____Петро Яцук_____
(підпис, ПІБ)

РЕФЕРАТ

Кваліфікаційна бакалаврська робота «Розробка веб-сайту для мультибіржового відстеження криптовалютних інвестицій».

Пояснювальна записка: 66 с., 10 рис., 8 табл., 30 джерел.

Об’єкт дослідження — процес моніторингу та аналізу криптовалютних інвестицій, розподілених між декількома біржами.

Предмет дослідження — методи, моделі та програмні засоби побудови веборієнтованої системи для автоматизованого мультибіржового агрегування й розрахунку показників криптовалютного інвестиційного портфеля.

Мета роботи — підвищення зручності та ефективності контролю за криптовалютними інвестиціями шляхом проєктування та розроблення веборієнтованої інформаційної системи мультибіржового відстеження.

У роботі досліджено предметну область криптовалютних інвестицій та принципи функціонування API криптобірж, виконано порівняльний аналіз наявних рішень та обґрунтовано вибір технологій (FastAPI, React, PostgreSQL, CCXT). Сформовано вимоги до системи, спроєктовано трирівневу архітектуру, структуру бази даних у вигляді ER-діаграми та алгоритм розрахунку фінансових показників портфеля. Реалізовано веборієнтований застосунок з асинхронною інтеграцією до API бірж та комплексом заходів інформаційної безпеки (read-only ключі, гешування паролів Argon2id, шифрування AES, автентифікація JWT). Проведене тестування підтвердило відповідність системи сформованим вимогам.

Практичне значення роботи полягає в можливості використання системи приватними криптоінвесторами для централізованого контролю за активами.

Ключові слова: криптовалюта, інвестиційний портфель, мультибіржове відстеження, api криптобірж, агрегування даних, вебзастосунок, інформаційна безпека.

ABSTRACT

Bachelor's qualification thesis "Development of a web application for multi-exchange tracking of cryptocurrency investments".

Explanatory note: 66 p., 10 fig., 8 tab., 30 references.

The object of research is the process of monitoring and analysing cryptocurrency investments distributed across several exchanges.

The subject of research comprises the methods, models and software tools for building a web-based system for automated multi-exchange aggregation and calculation of cryptocurrency investment portfolio metrics.

The aim of the work is to improve the convenience and efficiency of controlling cryptocurrency investments by designing and developing a web-based multi-exchange tracking information system.

The thesis examines the domain of cryptocurrency investments and the principles of crypto exchange APIs, provides a comparative analysis of existing solutions, and justifies the choice of technologies (FastAPI, React, PostgreSQL, CCXT). Requirements for the system were defined; a three-tier architecture, a database structure as an ER diagram, and an algorithm for calculating portfolio metrics were designed. A web-based application was implemented with asynchronous integration to exchange APIs and a set of information security measures (read-only keys, Argon2id password hashing, AES encryption, JWT authentication). Testing confirmed the system's compliance with the defined requirements.

The practical value of the work lies in the system's potential use by private crypto investors for centralised asset control.

Keywords: CRYPTOCURRENCY, INVESTMENT PORTFOLIO, MULTI-EXCHANGE TRACKING, EXCHANGE API, DATA AGGREGATION, WEB APPLICATION, INFORMATION SECURITY.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AES	Advanced Encryption Standard — стандарт симетричного шифрування
API	Application Programming Interface — програмний інтерфейс застосунку
CCXT	CryptoCurrency eXchange Trading library — бібліотека для роботи з API криптобірж
DeFi	Decentralized Finance — децентралізовані фінанси
ER	Entity-Relationship — модель «сутність — зв’язок»
HMAC	Hash-based Message Authentication Code — код автентифікації повідомлень на основі гешу
HTTP	HyperText Transfer Protocol — протокол передавання гіпертексту
HTTPS	HyperText Transfer Protocol Secure — захищений протокол передавання гіпертексту
JSON	JavaScript Object Notation — текстовий формат обміну даними
JWT	JSON Web Token — веб-токен у форматі JSON
P&L	Profit and Loss — прибуток та збиток (фінансовий результат)
REST	Representational State Transfer — архітектурний стиль вебсервісів
ROI	Return on Investment — дохідність інвестицій
SPA	Single Page Application — односторінковий застосунок
TLS	Transport Layer Security — протокол захисту транспортного рівня
БД	база даних
ПЗ	програмне забезпечення
СКБД	система керування базами даних

ЗМІСТ

ВСТУП

Актуальність теми. Ринок криптовалют за останнє десятиліття перетворився з нішевого явища на значний сегмент світової фінансової системи. Станом на початок 2026 року сукупна капіталізація криптовалютного ринку становить близько 3 трильйонів доларів США, причому протягом 2025 року вона демонструвала надзвичайну волатильність, сягнувши історичного максимуму в 4,27 трлн доларів у жовтні 2025 року з подальшим спадом. Загальна кількість власників криптовалют у світі перевищила 560 мільйонів осіб, а кількість активних криптовалютних бірж у світі перевищує дві сотні.

Така кількість майданчиків породжує характерну проблему — фрагментацію активів. Інвестор зазвичай зберігає кошти не на одній, а на кількох біржах одночасно (наприклад, Binance, Kraken, Coinbase), оскільки різні майданчики пропонують різний перелік монет, розмір комісій та умови обслуговування. Кожна біржа надає лише власний ізольований інтерфейс, унаслідок чого користувач позбавлений єдиної консолідованої картини свого інвестиційного портфеля та змушений вручну збирати й зводити дані з різних джерел. Це трудомісткий процес, який до того ж підвищує ймовірність помилок під час розрахунку сукупної вартості активів та фінансового результату.

Розв’язання цієї проблеми полягає у створенні програмного засобу, здатного автоматично агрегувати дані про активи користувача з багатьох бірж одночасно через їхні програмні інтерфейси (API) та подавати їх у вигляді єдиної аналітичної панелі. Попри наявність на ринку низки подібних рішень, питання побудови захищеної, розширюваної та орієнтованої на конкретні потреби користувача веборієнтованої системи мультибіржового відстеження залишається актуальним, що й зумовило вибір теми кваліфікаційної роботи.

Мета роботи — підвищення зручності та ефективності контролю за криптовалютними інвестиціями шляхом проєктування та розроблення веборієнтованої інформаційної системи мультибіржового відстеження, яка

забезпечує автоматизоване агрегування даних з декількох криптобірж та розрахунок ключових фінансових показників інвестиційного портфеля.

Завдання роботи. Для досягнення поставленої мети необхідно вирішити такі завдання:

1) дослідити предметну область криптовалютних інвестицій та принципи функціонування програмних інтерфейсів криптовалютних бірж;
2) виконати порівняльний аналіз наявних програмних рішень для відстеження криптоінвестицій і виявити їхні переваги та недоліки;
3) обґрунтувати вибір технологій та засобів розроблення програмної системи;
4) сформувати функціональні та нефункціональні вимоги до системи;
5) спроєктувати архітектуру програмного забезпечення та структуру бази даних;
6) розробити алгоритм розрахунку фінансових показників інвестиційного портфеля;
7) реалізувати веборієнтований застосунок з інтеграцією до API криптобірж та забезпечити захист даних користувачів;
8) провести тестування системи й проаналізувати отримані результати.

Об’єкт дослідження — процес моніторингу та аналізу криптовалютних інвестицій, розподілених між декількома біржами.

Предмет дослідження — методи, моделі та програмні засоби побудови веборієнтованої системи для автоматизованого мультибіржового агрегування й розрахунку показників криптовалютного інвестиційного портфеля.

Методи дослідження. Під час виконання роботи використано: методи системного та порівняльного аналізу — для дослідження предметної області та зіставлення наявних рішень; методи об’єктно-орієнтованого проєктування та UML-моделювання — для розроблення архітектури системи; методи проєктування реляційних баз даних (модель «сутність — зв’язок») — для побудови структури бази даних; методи тестування програмного забезпечення

та математичної статистики — для оцінювання продуктивності й достовірності роботи системи.

Практичне значення одержаних результатів полягає в тому, що розроблена система може бути використана приватними криптоінвесторами для централізованого контролю за своїми активами, а запропоновані проєктні та архітектурні рішення — застосовані під час розроблення подібних фінансово-аналітичних вебзастосунків.

РОЗДІЛ 1

АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОБҐРУНТУВАННЯ ЗАСОБІВ РОЗРОБЛЕННЯ

1.1. Криптовалютні інвестиції як предметна область

Криптовалюта — це різновид цифрових активів, функціонування якого ґрунтується на технології розподіленого реєстру (блокчейн) та криптографічних методах захисту інформації. Блокчейн є розподіленим реєстром, що підтримується вузлами однорангової (peer-to-peer) мережі та вперше був запропонований Сатоші Накамото в межах проєкту Bitcoin у 2008 році; криптографічне зв’язування блоків забезпечує незмінність і захист записів від підробки, а узгодження стану реєстру досягається механізмом консенсусу без потреби в центральному координаторі [1; 2]. На відміну від традиційних фіатних грошей, емісія та обіг яких контролюються центральними банками, криптовалюти функціонують у децентралізованих мережах, де підтвердження транзакцій здійснюється сукупністю вузлів за визначеним алгоритмом консенсусу. Першою та досі найбільшою за капіталізацією криптовалютою є Bitcoin, представлений у 2009 році; згодом виникла велика кількість альтернативних криптовалют (так званих альткоїнів), серед яких провідне місце посідає Ethereum.

З погляду інвестора цифрові активи доцільно класифікувати на кілька основних груп. До першої належать власне монети (coins) — активи, що мають власний блокчейн (Bitcoin, Ethereum, Solana). Другу групу утворюють токени (tokens), які випускаються на основі вже наявних блокчейнів за стандартами на кшталт ERC-20. Окрему важливу категорію становлять стейблкоїни (stablecoins) — токени, вартість яких прив’язана до стабільного активу, зазвичай долара США (USDT, USDC). Вони відіграють роль засобу збереження вартості та розрахунків: за оцінками Банку міжнародних розрахунків, на березень 2026 року сукупна капіталізація стейблкоїнів становила близько 270 млрд доларів, причому два найбільші — USDT і USDC

— формують понад 95 % усього обсягу [3]. Така концентрація та доларова домінованість підтверджуються й дослідженнями Міжнародного валютного фонду [4]. Ця різноманітність активів безпосередньо впливає на проєктовану систему, оскільки вона повинна коректно опрацьовувати активи різних типів та враховувати їхні особливості під час розрахунку вартості портфеля.

Під криптовалютним інвестуванням розуміють процес придбання цифрових активів з метою отримання прибутку від зміни їхньої ринкової вартості або від супутніх механізмів (стейкінгу, надання ліквідності тощо). Сукупність цифрових активів, якими володіє інвестор, утворює інвестиційний портфель. Ефективне управління портфелем потребує постійного контролю за низкою показників, серед яких ключовими є: поточна сукупна вартість портфеля; вартість окремих позицій; нереалізований фінансовий результат (різниця між поточною вартістю та вартістю придбання); відсоткова дохідність (ROI); а також структура портфеля у розрізі окремих активів. Саме автоматизований розрахунок цих показників становить ядро функціональності розроблюваної системи, а детальні формули їх обчислення наведено у підрозділі 2.4.

Важливим аспектом предметної області є спосіб зберігання активів. Розрізняють кастодіальне зберігання, за якого активи перебувають на рахунках криптовалютної біржі (а отже, приватними ключами фактично розпоряджається біржа), та некастодіальне, за якого користувач самостійно контролює власні ключі через персональний гаманець. Кожен підхід має свої переваги й ризики: біржове зберігання є зручнішим для активної торгівлі, проте наражає користувача на ризики, пов’язані з безпекою самої біржі; некастодіальне зберігання надає повний контроль, але покладає відповідальність за збереження ключів на користувача. Оскільки предметом цієї роботи є відстеження активів, що зберігаються саме на біржах та доступні через їхні програмні інтерфейси, у системі основну увагу приділено інтеграції з біржами; водночас архітектура передбачає можливість подальшого розширення для підтримки некастодіальних гаманців.

Характерною рисою сучасного крипторинку є його висока волатильність та фрагментованість. Як зазначено у вступі, активи типового інвестора розподілені між кількома біржами. Це зумовлено об’єктивними причинами: різні біржі підтримують різний перелік торгових пар, пропонують відмінні умови щодо комісій та ліквідності, а також підпадають під різні регуляторні юрисдикції. Цю проблему фрагментації активів визнають і фахові огляди ринку: на відміну від традиційних брокерських рахунків, де всі активи зберігаються в одному місці, криптоінвестори зазвичай тримають активи розподіленими між кількома гаманцями та біржами, що й зумовлює потребу в інструментах агрегування [5]. Фрагментація ускладнює не лише торгівлю, а й сам моніторинг: щоб оцінити загальний стан своїх інвестицій, користувач має почергово авторизуватися на кожній біржі, що є незручним та призводить до затримок в ухваленні рішень [5].

Окремої уваги потребує проблема ліквідності та розбіжності цін між біржами. Оскільки кожен майданчик формує власний стакан заявок, ціна того самого активу може дещо відрізнятися залежно від біржі. Це явище має систематичний характер: у науковій літературі задокументовано стійкі й тривалі розбіжності цін між біржами, що свідчать про фрагментованість ліквідності та обмеженість арбітражу [6]. Класичним прикладом є так звана «кімчі-премія» — історично вищі ціни на криптовалюти на біржах Південної Кореї порівняно зі світовими середніми значеннями [7]. Для системи відстеження це означає, що під час розрахунку вартості портфеля необхідно або орієнтуватися на ціни конкретної біржі, де зберігається актив, або використовувати усереднені ринкові котирування з агрегаторів. Вибір відповідного підходу є важливим проєктним рішенням, яке буде обґрунтовано в наступних розділах.

Таким чином, предметна область характеризується трьома ключовими особливостями, які формують вимоги до системи: різнотипність цифрових активів, розподіленість активів між багатьма біржами та необхідність оперативного розрахунку фінансових показників на основі мінливих ринкових

даних. Подальший аналіз спрямований на дослідження технічних засобів, що дають змогу автоматизовано отримувати дані з бірж, та наявних програмних рішень, які вже вирішують подібні завдання.

1.2. Принципи функціонування програмних інтерфейсів криптовалютних бірж та концепція мультибіржового агрегування

Технічною основою для автоматизованого отримання даних із криптовалютних бірж є їхні програмні інтерфейси (Application Programming Interface, API). API — це чітко визначений набір правил і протоколів, за допомогою яких одна програма може звертатися до іншої та обмінюватися з нею даними. Переважна більшість сучасних криптобірж надає API, побудований за архітектурним стилем REST (Representational State Transfer), що використовує протокол HTTP та повертає дані у форматі JSON [8]. Цей формат є структурованим, легким для машинного опрацювання та підтримується практично всіма мовами програмування, що робить його зручним для побудови інтеграцій.

Запити до API криптобірж поділяються на дві категорії. Публічні запити не потребують автентифікації та надають загальнодоступну ринкову інформацію: поточні котирування, історичні дані про ціни, перелік торгових пар, стан стакана заявок. Приватні запити стосуються конкретного облікового запису користувача (баланси, історія операцій) і тому вимагають автентифікації. Автентифікація здійснюється за допомогою пари ключів — публічного (API key), що ідентифікує користувача, та секретного (API secret), яким підписується кожен запит за криптографічним алгоритмом HMAC [9]. Сервер біржі перевіряє цей підпис і таким чином пересвідчується, що запит надійшов саме від власника ключів і не був змінений під час передавання.

Принципово важливою для проєктованої системи є концепція рівнів доступу API-ключів. Біржі дають змогу під час створення ключа обмежити перелік дозволених операцій. Для системи відстеження достатньо ключів із правом виключно на читання (read-only), які дозволяють переглядати баланси

й історію, але не дозволяють здійснювати торгові операції чи виведення коштів. Використання саме таких ключів є ключовим архітектурним рішенням з погляду безпеки: навіть у разі компрометації бази даних зловмисник не зможе розпорядитися коштами користувача. Це рішення детально обґрунтовано в підрозділі 3.3.

Під час роботи з API необхідно враховувати обмеження частоти запитів (rate limits). Біржі захищають свою інфраструктуру від надмірного навантаження, обмежуючи кількість запитів від одного клієнта за одиницю часу; перевищення ліміту призводить до тимчасового блокування. Тому система, що звертається одночасно до кількох бірж в інтересах багатьох користувачів, повинна впроваджувати механізми контролю частоти запитів, кешування отриманих даних та коректного опрацювання помилок. Окрім класичного REST, деякі біржі надають інтерфейс на основі протоколу WebSocket, який забезпечує отримання даних у режимі реального часу через постійне з’єднання, проте для задачі періодичного оновлення стану портфеля достатньо REST-запитів [10].

Окрім обмежень частоти, практична інтеграція з API бірж потребує врахування ще кількох технічних аспектів. По-перше, обсягові відповіді (наприклад, повна історія операцій) зазвичай повертаються частинами з використанням механізму посторінкового отримання (pagination), тому клієнт має послідовно запитувати наступні сторінки до повного завантаження даних. По-друге, мережеві запити за своєю природою ненадійні: можливі тимчасові збої з’єднання чи короткочасна недоступність сервісу, через що доцільно застосовувати повторні спроби із прогресивною затримкою (exponential backoff). По-третє, біржі періодично оновлюють версії своїх API, тож використання бібліотеки-посередника, яка бере на себе підтримку актуальності інтеграцій, суттєво знижує витрати на супровід системи. Усі ці чинники додатково обґрунтовують доцільність застосування спеціалізованої бібліотеки замість прямої інтеграції з кожною біржею.

Головною технічною перешкодою для побудови мультибіржової системи є неоднорідність API різних бірж. Попри спільну REST-основу, кожна біржа має власну структуру запитів, формати відповідей, способи позначення торгових пар та власні коди помилок. Наприклад, та сама інформація про баланс на різних біржах може повертатися під різними назвами полів і в різних одиницях. Безпосередня інтеграція з кожною біржею окремо потребувала б написання й підтримки окремого програмного модуля для кожного майданчика, що є трудомістким і погано масштабованим підходом.

Розв'язанням цієї проблеми є застосування рівня абстракції — уніфікованої бібліотеки, яка приховує відмінності окремих бірж за єдиним інтерфейсом. Провідним рішенням такого класу є бібліотека з відкритим кодом CCXT (CryptoCurrency eXchange Trading library). Станом на 2026 рік CCXT підтримує 105 криптовалютних бірж і надає уніфікований програмний інтерфейс мовами Python, JavaScript, PHP та іншими [11]. Бібліотека нормалізує структури даних: незалежно від того, як конкретна біржа повертає інформацію про баланси чи котирування, CCXT перетворює її на єдиний стандартизований формат. Це дає змогу розробляти логіку агрегування один раз і застосовувати її до всіх підтримуваних бірж, що безпосередньо відповідає поняттю «мультибіржовості» у темі роботи.

Під мультибіржовим агрегуванням у цій роботі розуміється процес автоматизованого збирання даних про активи користувача з декількох бірж, приведення їх до єдиного формату та зведення в консолідований інвестиційний портфель. Узагальнений алгоритм агрегування передбачає такі етапи: отримання збережених read-only ключів користувача для кожної під'єднаної біржі; послідовне або паралельне звернення до API кожної біржі для отримання балансів; отримання поточних ринкових котирувань для наявних активів; перерахування вартості всіх активів у єдину валюту відображення (наприклад, долар США); та підсумовування для формування загальної картини портфеля. Саме цей алгоритм лежить в основі

функціональної моделі системи, а його програмна реалізація розглядається у третьому розділі.

Таким чином, аналіз принципів роботи API бірж засвідчує технічну здійсненність автоматизованого мультибіржового відстеження та обґрунтовує доцільність застосування уніфікованої бібліотеки CCXT як засобу інтеграції. Це дає змогу зосередити подальшу розробку на бізнес-логіці агрегування й розрахунку показників, а не на низькорівневих особливостях кожної окремої біржі.

1.3. Порівняльний аналіз наявних програмних рішень

На ринку представлено низку програмних продуктів для відстеження криптовалютних інвестицій. Для виявлення їхніх сильних і слабких сторін та обґрунтування доцільності розроблення власної системи проаналізовано чотири найпоширеніші рішення: CoinStats, Delta, CoinGecko Portfolio та CoinMarketCap Portfolio. Аналіз проводився за критеріями, що безпосередньо стосуються теми роботи: підтримка автоматичного під’єднання бірж через API, кількість підтримуваних майданчиків, наявність розрахунку фінансових показників, модель безпеки, обмеження безкоштовної версії та платформа доступу.

CoinStats є одним із найфункціональніших рішень на ринку, яким користується понад 1,2 млн активних користувачів щомісяця. Застосунок підтримує понад 300 бірж і гаманців, понад 50 блокчейнів і близько 1000 DeFi-протоколів, під’єднання бірж здійснюється через read-only API-ключі [12]. Він надає консолідований дашборд із розрахунком прибутків і збитків (P&L), діаграмами розподілу активів, історією транзакцій та кривою дохідності за активами. Водночас CoinStats має суттєві обмеження: розширений функціонал доступний лише в платних тарифах, а безкоштовна версія обмежує кількість під’єднань і синхронізацій. Показовим є й те, що у червні 2024 року вбудований гаманець застосунку зазнав успішної атаки, що підкреслює критичність питання безпеки для рішень цього класу [13].

Delta (придбаний компанією eToro) орієнтований на мультиактивне відстеження — окрім криптовалют, він підтримує акції, ETF, облігації та інші традиційні інструменти, що робить його зручним для інвесторів зі змішаним портфелем. Застосунок працює в режимі read-only та інтегрується з понад 50 біржами [14]. Його ключове обмеження — безкоштовна версія дозволяє під'єднати лише два рахунки (або відстежувати обмежену кількість активів), а повний функціонал доступний у платній підписці Delta PRO. Крім того, відзначається відносно повільніша синхронізація та обмежена підтримка DeFi порівняно з конкурентами [14].

CoinGecko Portfolio — безкоштовний інструмент від відомого агрегатора ринкових даних, що вирізняється широким охопленням (понад 14 000 криптовалют), мобільною синхронізацією та сповіщеннями. Проте він має принципове для нашої теми обмеження: CoinGecko не підтримує під'єднання бірж через API, і всі транзакції доводиться вводити вручну [15]. Це робить його придатним радше для довгострокових інвесторів зі статичним портфелем, але незручним для активного мультибіржового відстеження.

CoinMarketCap Portfolio — також безкоштовний трекер від найбільш цитованого агрегатора цін, що підтримує понад 12 000 криптовалют. Подібно до CoinGecko, він зручний для перегляду ринкових даних, проте функціональність відстеження портфеля є базовою й значною мірою ґрунтується на ручному введенні даних, без повноцінної автоматичної інтеграції з біржами користувача.

Результати порівняння зведено в таблиці 1.1.

Таблиця 1.1 — Порівняльний аналіз наявних рішень для відстеження криптоінвестицій

Критерій	CoinStats	Delta	CoinGecko	CoinMarketCap
Автоматичне під'єднання бірж (API)	Так	Так	Ні	Обмежено
Кількість бірж / гаманців	300+	50+	—	—
Розрахунок P&L, ROI	Так	Так	Базово	Базово
Режим read-only	Так	Так	—	—
Обмеження безкоштовної версії	Ліміт під'єднань	2 рахунки	Ручне введення	Базовий
Мультиактивність (акції, ETF)	Частково	Так	Ні	Ні
Відкритий код	Ні	Ні	Ні	Ні

Проведений аналіз дає змогу зробити такі висновки. По-перше, провідні рішення (CoinStats, Delta) підтверджують технічну обґрунтованість обраного підходу: автоматичне під'єднання бірж саме через read-only API-ключі є галузевим стандартом, що додатково валідує проєктні рішення цієї роботи. По-друге, усі розглянуті продукти є пропрієтарними із закритим кодом, а їхній повний функціонал прихований за платними підписками з відчутними обмеженнями безкоштовних версій. По-третє, безкоштовні рішення від агрегаторів (CoinGecko, CoinMarketCap) не забезпечують повноцінної автоматичної інтеграції з біржами, що є критичним недоліком для задачі мультибіржового відстеження.

Таким чином, наявні рішення або є комерційними продуктами з обмеженнями, або не підтримують автоматичного мультибіржового агрегування. Це обґрунтовує доцільність розроблення власної веборієнтованої системи, яка реалізує автоматичне під'єднання кількох бірж через захищені read-only ключі та розрахунок ключових фінансових показників портфеля. Цінність такої роботи полягає не у створенні комерційного конкурента, а в дослідженні та практичній реалізації архітектурних і безпекових рішень, що лежать в основі систем цього класу.

1.4. Обґрунтування вибору технологій та засобів розроблення

На підставі проаналізованих вимог до системи (мультибіржова інтеграція, інтенсивна робота з мережевими запитами, надійне зберігання даних та захист API-ключів) обґрунтовано вибір технологічного стеку, що охоплює серверну частину (бекенд), клієнтську частину (фронтенд), систему керування базою даних та бібліотеку інтеграції з біржами. Вибір ґрунтувався на критеріях продуктивності, придатності для асинхронної роботи з API, зрілості й поширеності технології, наявності документації та відкритості коду.

Серверна частина: фреймворк FastAPI. Як основний інструмент розроблення бекенду обрано FastAPI — сучасний високопродуктивний вебфреймворк мовою Python. Ключовою причиною вибору є його асинхронна архітектура (async/await), побудована на стандарті ASGI та бібліотеці Starlette, що забезпечує продуктивність, порівнянну з рішеннями на Node.js та Go [16]. Це критично для проєктованої системи, оскільки її основне навантаження становлять численні мережеві запити до API бірж — типова операція введення-виведення (I/O-bound), для якої асинхронна модель є оптимальною: поки система очікує відповіді від однієї біржі, вона може опрацьовувати запити до інших. Додатковими перевагами FastAPI є автоматична валідація даних на основі типів Python через бібліотеку Pydantic та автоматична генерація інтерактивної документації API за стандартом OpenAPI [17]. Станом на 2026 рік FastAPI набув статусу фактичного стандарту для розроблення API мовою Python та використовується такими компаніями, як Microsoft і Netflix [16].

Серед альтернатив розглядалися фреймворки Django та Flask. Django є потужним рішенням із вбудованими засобами, проте його ядро історично синхронне (засноване на WSGI), що менш природно лягає на інтенсивну асинхронну роботу з зовнішніми API [18]. Flask простий і гнучкий, але не має вбудованої підтримки асинхронності, автоматичної валідації даних та документування [18]. З огляду на специфіку задачі саме FastAPI забезпечує найкращий баланс продуктивності й зручності розроблення.

Клієнтська частина: бібліотека React. Для реалізації інтерфейсу користувача обрано React — широко розповсюджену JavaScript-бібліотеку для побудови інтерфейсів. Її вибір зумовлений компонентним підходом, що сприяє повторному використанню коду та полегшує підтримку, а також застосуванням віртуального DOM, який підвищує продуктивність відображення завдяки мінімізації прямих маніпуляцій із реальним DOM [19]. Для задачі відстеження портфеля важливою є здатність динамічно оновлювати інтерфейс (показники, графіки, таблиці) у відповідь на отримані з бекенду дані, з чим React справляється ефективно. Окрім того, екосистема React містить розвинені бібліотеки візуалізації даних (зокрема для побудови графіків динаміки портфеля), що безпосередньо відповідає потребам системи.

Система керування базою даних: PostgreSQL. Для зберігання даних обрано об'єктно-реляційну систему керування базами даних PostgreSQL. Вона є надійною, зрілою (розвивається з 1996 року) системою з відкритим кодом, відомою своєю стабільністю, підтримкою складних запитів та строгою цілісністю даних [20]. Реляційна модель природно відповідає структурі предметної області (користувачі, під'єднані біржі, активи, історія вартості портфеля), яка добре формалізується у вигляді пов'язаних таблиць та моделі «сутність — зв'язок», що відповідає вимозі завдання щодо побудови ER-діаграми. Важливою перевагою для фінансово-аналітичного застосунку є саме гарантії цілісності й узгодженості даних, які надає PostgreSQL [20]. Додатково ця СКБД підтримує тип JSONB, що дає змогу за потреби гнучко зберігати слабоструктуровані дані (наприклад, відповіді API бірж) у межах реляційної структури.

Інтеграція з біржами: бібліотека CCXT. Як засіб інтеграції з криптобіржами обрано бібліотеку з відкритим кодом CCXT, докладно розглянуту в підрозділі 1.2. Вона надає уніфікований інтерфейс до 105 бірж, що позбавляє від потреби розробляти окремий модуль для кожного майданчика та безпосередньо реалізує концепцію мультибіржовості.

Наявність версії CCXT для Python забезпечує природну сумісність із обраним бекендом на FastAPI.

Засоби забезпечення безпеки. Для автентифікації користувачів передбачено застосування механізму токенів JWT (JSON Web Token), а для захисту секретних API-ключів — їх шифрування перед збереженням у базі даних. Детальне обґрунтування цих рішень наведено в підрозділі 3.3.

Узагальнено обраний технологічний стек подано в таблиці 1.2.

Таблиця 1.2 — Обраний технологічний стек системи

Компонент системи	Обрана технологія	Основне призначення
Серверна частина (бекенд)	FastAPI (Python)	Бізнес-логіка, обробка запитів, агрегування
Клієнтська частина (фронтенд)	React (JavaScript)	Інтерфейс користувача, візуалізація
База даних	PostgreSQL	Зберігання даних користувачів і портфелів
Інтеграція з біржами	CCXT	Уніфікований доступ до API бірж
Автентифікація	JWT	Захист доступу до системи

Таким чином, обґрунтований технологічний стек є цілісним та внутрішньо узгодженим: асинхронний бекенд на FastAPI ефективно взаємодіє з API бірж через CCXT, надійно зберігає дані в PostgreSQL та обмінюється ними з клієнтською частиною на React. Усі обрані технології є зрілими, добре документованими та переважно відкритими, що відповідає завданням роботи й закладає основу для проєктування системи, яке розглядається у другому розділі.

Висновки до розділу 1

У розділі досліджено предметну область криптовалютних інвестицій та визначено три її ключові особливості: різнотипність цифрових активів, розподіленість активів між багатьма біржами та потребу в оперативному розрахунку фінансових показників. Проаналізовано принципи функціонування API криптобірж і обґрунтовано доцільність застосування

уніфікованої бібліотеки CCXT для мультибіржового агрегування. Виконано порівняльний аналіз чотирьох наявних рішень, який засвідчив, що вони або є комерційними продуктами з обмеженнями, або не підтримують автоматичної мультибіржової інтеграції, що обґрунтовує доцільність власної розробки. На основі сформульованих вимог обрано та обґрунтовано технологічний стек: FastAPI, React, PostgreSQL та CCXT. Отримані результати створюють основу для проєктування системи, що є предметом наступного розділу.

РОЗДІЛ 2

ПРОЄКТУВАННЯ СИСТЕМИ

2.1. Формування функціональних та нефункціональних вимог до системи

Проєктування будь-якої програмної системи починається з формування вимог — точного опису того, що система повинна робити (функціональні вимоги) та яким характеристикам якості вона має відповідати (нефункціональні вимоги). Вимоги формуються на основі мети роботи, окреслених у вступі завдань та виявлених у першому розділі особливостей предметної області. Чітке визначення вимог є основою для подальшого проєктування архітектури та бази даних і дає змогу об’єктивно оцінити результат на етапі тестування.

Визначення користувачів системи. Система є багатокористувацькою, тож перед формуванням вимог визначено типи її користувачів (акторів). У системі передбачено двох акторів: неавторизований відвідувач, який може лише зареєструватися або увійти, та зареєстрований користувач (інвестор), якому доступний повний функціонал відстеження портфеля. Адміністративні функції в межах цієї роботи не розглядаються, оскільки вони не належать до основної мети — відстеження інвестицій.

Функціональні вимоги. На основі аналізу предметної області сформовано перелік функціональних вимог, які для зручності згруповано за призначенням.

До групи керування обліковим записом належать такі вимоги: система повинна забезпечувати реєстрацію нового користувача за адресою електронної пошти та паролем; автентифікацію (вхід) зареєстрованого користувача; та завершення сеансу роботи (вихід).

До групи керування під’єднанням бірж належать: можливість додати нову біржу шляхом введення read-only API-ключів; перегляд переліку під’єднаних бірж; та видалення раніше доданої біржі. При цьому система

повинна приймати ключі лише з правом на читання та зберігати їх у захищеному (зашифрованому) вигляді.

До групи агрегування та відображення даних належать: автоматичне отримання балансів активів користувача з усіх під'єднаних бірж; отримання поточних ринкових котирувань для цих активів; та зведення даних у єдиний консолідований портфель із відображенням сукупної вартості.

До групи аналітики портфеля належать: розрахунок та відображення поточної вартості кожного активу і портфеля загалом; розрахунок структури портфеля (часток окремих активів) із візуалізацією у вигляді діаграми; розрахунок фінансового результату та дохідності; а також відображення динаміки вартості портфеля в часі у вигляді графіка.

Узагальнено основні сценарії використання системи зображено на діаграмі прецедентів (use-case діаграмі), наведеній на рисунку 2.1.

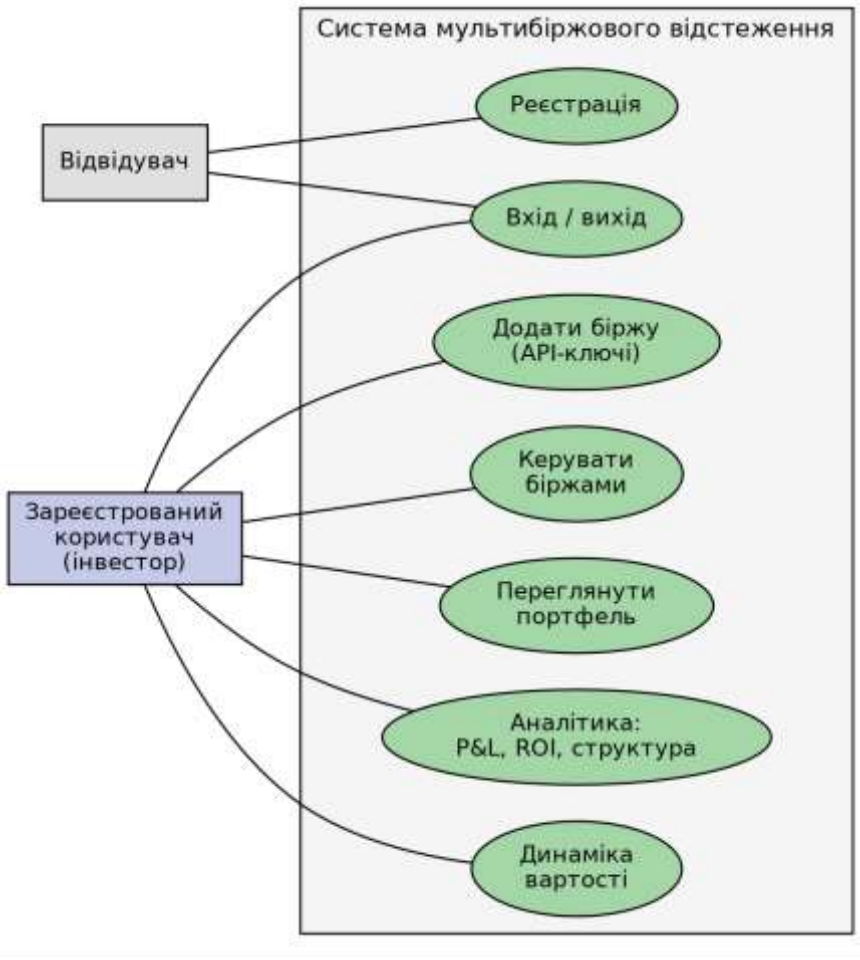

Рис. 2.1. Діаграма прецедентів (use-case) системи

Нефункціональні вимоги. Нефункціональні вимоги визначають якісні характеристики системи й для проєктованого застосунку є не менш важливими, ніж функціональні, з огляду на роботу з фінансовими даними та секретними ключами користувачів. Оскільки різні вимоги мають різну вагу для кінцевого результату, важливою задачею є визначення їх пріоритетності; методи пріоритезації вимог у програмній інженерії досліджено, зокрема, у праці [25]. У межах цієї роботи серед нефункціональних вимог найвищий пріоритет надано безпеці. Сформовано такі групи нефункціональних вимог.

Вимоги до безпеки є пріоритетними. Секретні API-ключі користувачів повинні зберігатися виключно в зашифрованому вигляді; паролі — у вигляді криптографічних хешів, а не у відкритому тексті. Доступ до даних користувача має надаватися лише після успішної автентифікації, а сеанс роботи — захищатися токеном. Система повинна приймати від бірж лише ключі з правом на читання, що унеможливлює здійснення торгових операцій навіть у разі компрометації даних.

Вимоги до продуктивності: система повинна забезпечувати прийнятний час відгуку під час завантаження портфеля попри звернення до зовнішніх API кількох бірж. Для цього передбачається асинхронне опрацювання запитів та кешування котирувань, щоб уникнути повторних звернень за тими самими даними протягом короткого проміжку часу.

Вимоги до надійності: система повинна коректно опрацьовувати помилки зовнішніх API (недоступність біржі, перевищення ліміту запитів, недійсні ключі) і не припиняти роботу через відмову однієї з бірж, а інформувати користувача про проблему з конкретним джерелом.

Вимоги до зручності використання: інтерфейс має бути інтуїтивно зрозумілим, відображати дані у наочній формі (таблиці, діаграми, графіки) та коректно працювати в сучасних веббраузерах.

Вимоги до масштабованості та переносності: архітектура має передбачати можливість додавання підтримки нових бірж без зміни основної

логіки (що забезпечується застосуванням бібліотеки CCXT), а також можливість збільшення кількості користувачів.

Сформовані вимоги зведено в таблицю 2.1, яка систематизує їх та слугуватиме критерієм для перевірки системи на етапі тестування (підрозділ 3.4).

Таблиця 2.1 — Зведення вимог до системи

Тип	Група	Зміст вимоги
Функціональна	Обліковий запис	Реєстрація, вхід, вихід
Функціональна	Під'єднання бірж	Додавання, перегляд, видалення (read-only ключі)
Функціональна	Агрегування	Збір балансів і котирувань, зведення портфеля
Функціональна	Аналітика	Вартість, структура, P&L, ROI, динаміка
Нефункціональна	Безпека	Шифрування ключів, хешування паролів, токени
Нефункціональна	Продуктивність	Асинхронність, кешування котирувань
Нефункціональна	Надійність	Опрацювання помилок API бірж
Нефункціональна	Зручність	Наочний інтерфейс, підтримка браузерів
Нефункціональна	Масштабованість	Додавання бірж і користувачів

Сформований перелік вимог є основою для подальшого проєктування. Функціональні вимоги визначають склад програмних модулів та структуру бази даних, а нефункціональні — ключові архітектурні рішення, насамперед щодо асинхронності та безпеки. Проєктування архітектури системи, що реалізує ці вимоги, розглядається в наступному підрозділі.

2.2. Проєктування архітектури програмного забезпечення

Архітектура програмного забезпечення визначає загальну організацію системи: з яких великих частин вона складається, які функції покладено на кожну частину та як ці частини взаємодіють між собою. Продумана архітектура є передумовою виконання нефункціональних вимог, сформованих у підрозділі 2.1, насамперед щодо безпеки, продуктивності й масштабованості.

Вибір архітектурного стилю. Для проєктованої системи обрано класичну трирівневу (three-tier) архітектуру, за якої система поділяється на три логічно відокремлені рівні: рівень подання (клієнтська частина), рівень бізнес-логіки (серверна частина) та рівень даних (база даних). Такий поділ є усталеним підходом для веборієнтованих застосунків і має суттєві переваги: кожен рівень можна розробляти, тестувати й масштабувати незалежно; чітко розмежовано відповідальність між частинами; а заміна або модифікація одного рівня мінімально впливає на інші. Альтернативний підхід — монолітна архітектура без чіткого поділу — є простішим у реалізації, але погано масштабується й ускладнює підтримку, тому для багатокористувацької системи, що працює з фінансовими даними, він визнаний недоцільним.

Рівень подання (клієнтська частина). Цей рівень реалізується як односторінковий застосунок (SPA, Single Page Application) на основі бібліотеки React і виконується у браузері користувача. Його відповідальність обмежується відображенням даних та взаємодією з користувачем: він не містить бізнес-логіки й не має прямого доступу до бази даних. Клієнтська частина формує запити до серверної частини через HTTP та відображає отримані дані у вигляді дашборду, таблиць активів, діаграм структури портфеля й графіків динаміки. Такий підхід забезпечує наочність та чутливість інтерфейсу, що відповідає вимозі зручності використання.

Рівень бізнес-логіки (серверна частина). Це центральний рівень системи, реалізований за допомогою фреймворку FastAPI. Він виконує всю змістовну роботу: приймає й перевіряє запити від клієнта, здійснює автентифікацію та авторизацію користувачів, звертається до зовнішніх API криптобірж через бібліотеку CCXT, виконує агрегування й розрахунок фінансових показників, а також взаємодіє з базою даних для збереження та отримання інформації. Серверна частина надає клієнту програмний інтерфейс (API) за архітектурним стилем REST. Саме на цьому рівні зосереджено реалізацію вимог безпеки: шифрування ключів, перевірку токенів та

опрацювання помилок зовнішніх сервісів. Для забезпечення продуктивності звернення до API різних бірж виконуються асинхронно.

Усередині серверної частини доцільно виокремити кілька логічних модулів за принципом єдиної відповідальності: модуль автентифікації (реєстрація, вхід, робота з токенами); модуль керування біржами (додавання, зберігання та видалення зашифрованих ключів); модуль інтеграції з біржами (звернення до CCXT, отримання балансів і котирувань); модуль розрахунку показників (обчислення вартості, структури, фінансового результату); та модуль доступу до даних (взаємодія з базою даних). Такий поділ полегшує розроблення, тестування й подальше розширення системи.

Рівень даних (база даних). Цей рівень реалізується системою керування базами даних PostgreSQL і відповідає за надійне та узгоджене зберігання інформації: облікових записів користувачів, зашифрованих даних про під'єднані біржі, відомостей про активи та історії вартості портфеля. Доступ до бази даних має виключно серверна частина; клієнт ніколи не звертається до неї напряму, що є важливим елементом безпеки. Детальне проєктування структури бази даних розглядається в підрозділі 2.3.

Взаємодія компонентів. Узагальнений сценарій роботи системи демонструє взаємодію всіх трьох рівнів та зовнішніх сервісів. Користувач виконує дію в інтерфейсі (клієнтська частина) — наприклад, відкриває дашборд. Клієнт надсилає HTTP-запит до серверної частини, додаючи токен автентифікації. Сервер перевіряє токен, звертається до бази даних по збережені (зашифровані) ключі користувача, розшифровує їх у пам'яті та через CCXT асинхронно звертається до API під'єднаних бірж по баланси й котирування. Отримані дані сервер агрегує, розраховує показники портфеля та повертає клієнту у структурованому форматі (JSON), а клієнт відображає їх користувачеві. Зовнішні API криптобірж є окремим, незалежним від системи елементом, з яким взаємодіє лише серверна частина.

Описану архітектуру та потоки даних між компонентами зображено на рисунку 2.2.

Обрана архітектура також враховує вимогу масштабованості. Серверна частина проєктується як така, що не зберігає стану сеансу (stateless): відомості про автентифікований сеанс містяться в токені на боці клієнта, а не в пам’яті сервера. Завдяки цьому за зростання кількості користувачів можливе горизонтальне масштабування — запуск кількох екземплярів серверної частини за балансувальником навантаження без потреби синхронізувати сеанси між ними. Рівень даних масштабується окремо засобами самої СКБД (індексування, реплікація), а рівень подання, що виконується у браузері користувача, не створює навантаження на сервер взагалі. Такий поділ відповідальності між незалежно масштабованими рівнями є однією з головних переваг трирівневої архітектури для багатокористувацьких систем.

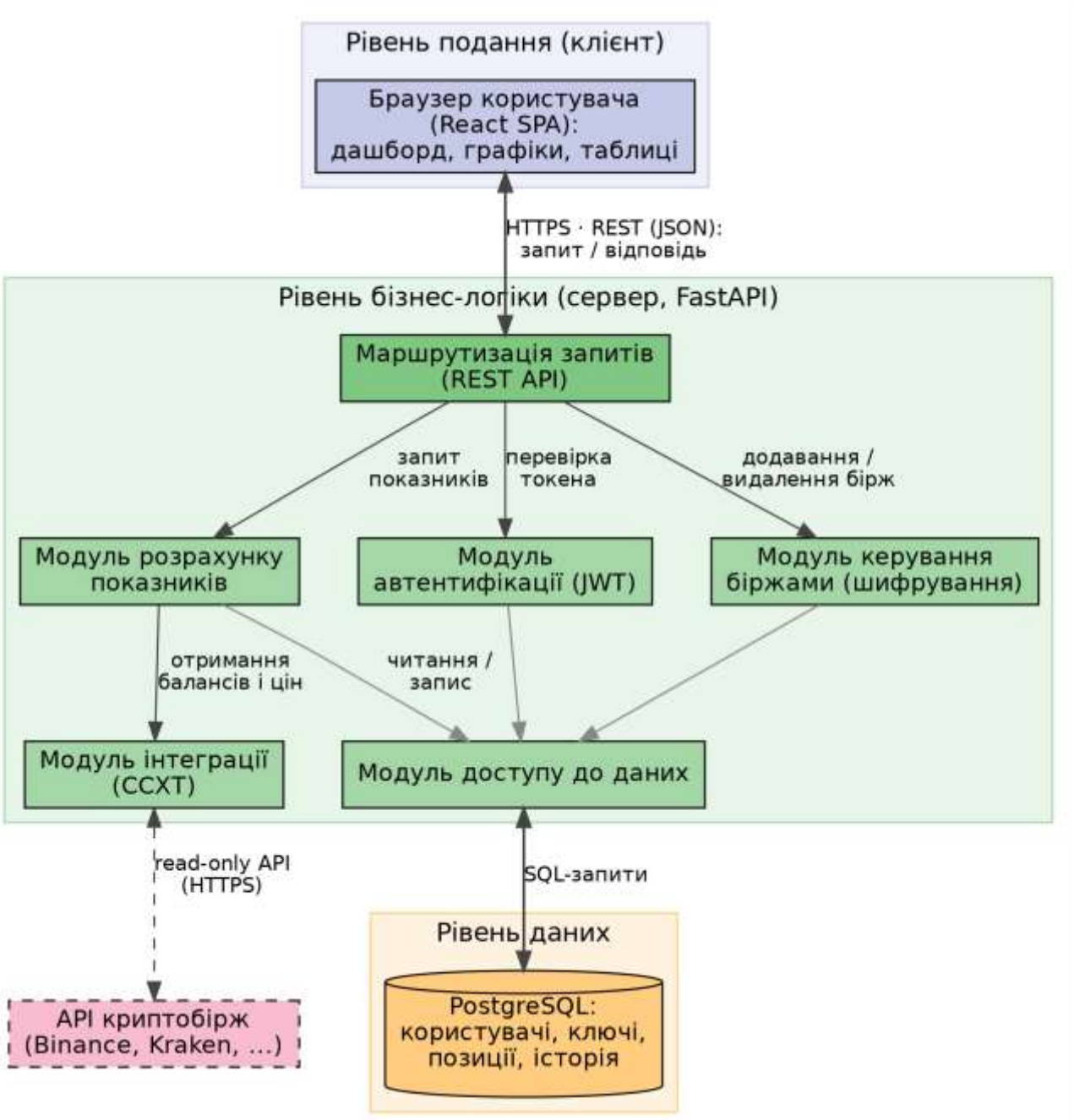

Рис. 2.2. Трирівнева архітектура системи та потоки даних

Таким чином, обрана трирівнева архітектура чітко розмежовує відповідальність між компонентами та безпосередньо реалізує сформовані вимоги: ізоляція бази даних від клієнта й централізація обробки ключів на сервері забезпечують безпеку, асинхронна взаємодія з біржами — продуктивність, а модульність серверної частини й застосування CCXT — масштабованість. Це створює надійну основу для детального проєктування бази даних та алгоритмів, що розглядається в наступних підрозділах.

2.3. Проєктування структури бази даних

База даних є фундаментом системи, що забезпечує надійне та узгоджене зберігання інформації. Проєктування виконувалося за класичним підходом — від концептуальної моделі (визначення сутностей предметної області та зв'язків між ними) до логічної моделі (подання у вигляді реляційних таблиць із зазначенням атрибутів, типів даних і ключів). Загальні принципи проєктування та використання реляційних баз даних викладено в навчальній літературі [26]. Для наочного подання структури використано модель «сутність — зв'язок» (Entity-Relationship, ER), що відповідає вимозі завдання.

Визначення сутностей. На основі функціональних вимог (підрозділ 2.1) виокремлено п'ять основних сутностей предметної області.

Сутність «Користувач» (users) зберігає облікові записи. Її атрибути: унікальний ідентифікатор (первинний ключ); адреса електронної пошти (унікальна); хеш пароля; дата й час реєстрації. Пароль принципово зберігається не у відкритому вигляді, а як криптографічний хеш, що відповідає вимозі безпеки.

Сутність «Під'єднана біржа» (exchange_connections) зберігає інформацію про під'єднані користувачем біржі. Атрибути: ідентифікатор; зовнішній ключ на користувача; назва біржі; зашифрований публічний ключ; зашифрований секретний ключ; необов'язкова позначка (назва, яку дав користувач); дата додавання. Ключі зберігаються виключно в зашифрованому вигляді.

Сутність «Актив» (assets) є довідником криптовалютних активів і запобігає дублюванню даних. Атрибути: ідентифікатор; символьне позначення (наприклад, BTC, ETH); повна назва. Винесення активів в окрему сутність є елементом нормалізації: відомості про кожен актив зберігаються один раз, а не повторюються для кожного користувача.

Сутність «Позиція» (holdings) відображає поточні залишки активів користувача на конкретній біржі. Атрибути: ідентифікатор; зовнішній ключ на під'єднану біржу; зовнішній ключ на актив; кількість; середня ціна придбання;

дата останнього оновлення. Середня ціна придбання потрібна для розрахунку фінансового результату (підрозділ 2.4).

Сутність «Знімок портфеля» (portfolio_snapshots) зберігає історію сукупної вартості портфеля для побудови графіка динаміки. Атрибути: ідентифікатор; зовнішній ключ на користувача; сукупна вартість у валюті відображення; дата й час фіксації.

Зв'язки між сутностями. Між сутностями встановлено такі зв'язки з відповідними кардинальностями. Один користувач може мати багато під'єднаних бірж, але кожна під'єднана біржа належить лише одному користувачеві (зв'язок «один-до-багатьох», 1:N). Одна під'єднана біржа може містити багато позицій, кожна позиція належить одній під'єднаній біржі (1:N). Один актив із довідника може фігурувати в багатьох позиціях різних користувачів (1:N). Один користувач має багато знімків портфеля в часі (1:N). Ці зв'язки реалізуються через зовнішні ключі та зведені в таблиці 2.2.

Таблиця 2.2 — Зв'язки між сутностями бази даних

Сутність-джерело	Сутність-призначення	Кардинальність	Реалізація
Користувач	Під'єднана біржа	1:N	FK user_id
Під'єднана біржа	Позиція	1:N	FK connection_id
Актив	Позиція	1:N	FK asset_id
Користувач	Знімок портфеля	1:N	FK user_id

Вибір типів даних. Окремої уваги потребує вибір типів даних для числових полів, що зберігають кількість активів та ціни. Для них обрано тип з фіксованою точністю NUMERIC, а не число з рухомою комою (FLOAT). Це принципове рішення: криптовалюти вимагають високої точності (наприклад, Bitcoin поділяється до восьми знаків після коми), а тип з рухомою комою накопичує похибки округлення, що є неприпустимим для фінансових розрахунків. Застосування типу NUMERIC гарантує точність обчислення вартості портфеля. Для ідентифікаторів використано цілочисельний

автоінкрементний тип, для рядкових даних — VARCHAR, для дат — TIMESTAMP.

Нормалізація. Спроєктована структура відповідає третій нормальній формі (3NF): усі атрибути є атомарними, кожен неключовий атрибут залежить лише від первинного ключа, відсутні транзитивні залежності. Винесення активів в окремий довідник усуває надмірність даних, а використання зовнішніх ключів забезпечує цілісність посилань — неможливість існування, наприклад, позиції без відповідної під'єднаної біржі.

Індексування та цілісність даних. Для пришвидшення типових вибірок передбачено створення індексів за полями зовнішніх ключів, за якими найчастіше відбувається пошук: за користувачем у таблиці під'єднаних бірж і знімків портфеля та за під'єднаною біржею у таблиці позицій. Це дає змогу ефективно отримувати всі дані конкретного користувача без повного перегляду таблиць. Цілісність даних додатково забезпечується обмеженнями на рівні бази даних: умовою унікальності для адреси електронної пошти та символьного позначення активу, а також правилом каскадного видалення (ON DELETE CASCADE), за яким видалення користувача автоматично вилучає всі пов'язані з ним записи. Відповідні визначення таблиць, обмежень та індексів мовою SQL наведено в додатку В.

Спроєктовану структуру бази даних у вигляді ER-діаграми, що відображає всі сутності, їхні атрибути та зв'язки, наведено на рисунку 2.3.

Таким чином, спроєктована база даних із п'яти взаємопов'язаних сутностей повністю покриває потреби системи у зберіганні даних, відповідає вимогам безпеки (зберігання хешів і зашифрованих ключів) та точності фінансових розрахунків (тип NUMERIC), а також нормалізована для усунення надмірності. Вона є основою для реалізації модуля доступу до даних, що розглядається у третьому розділі.

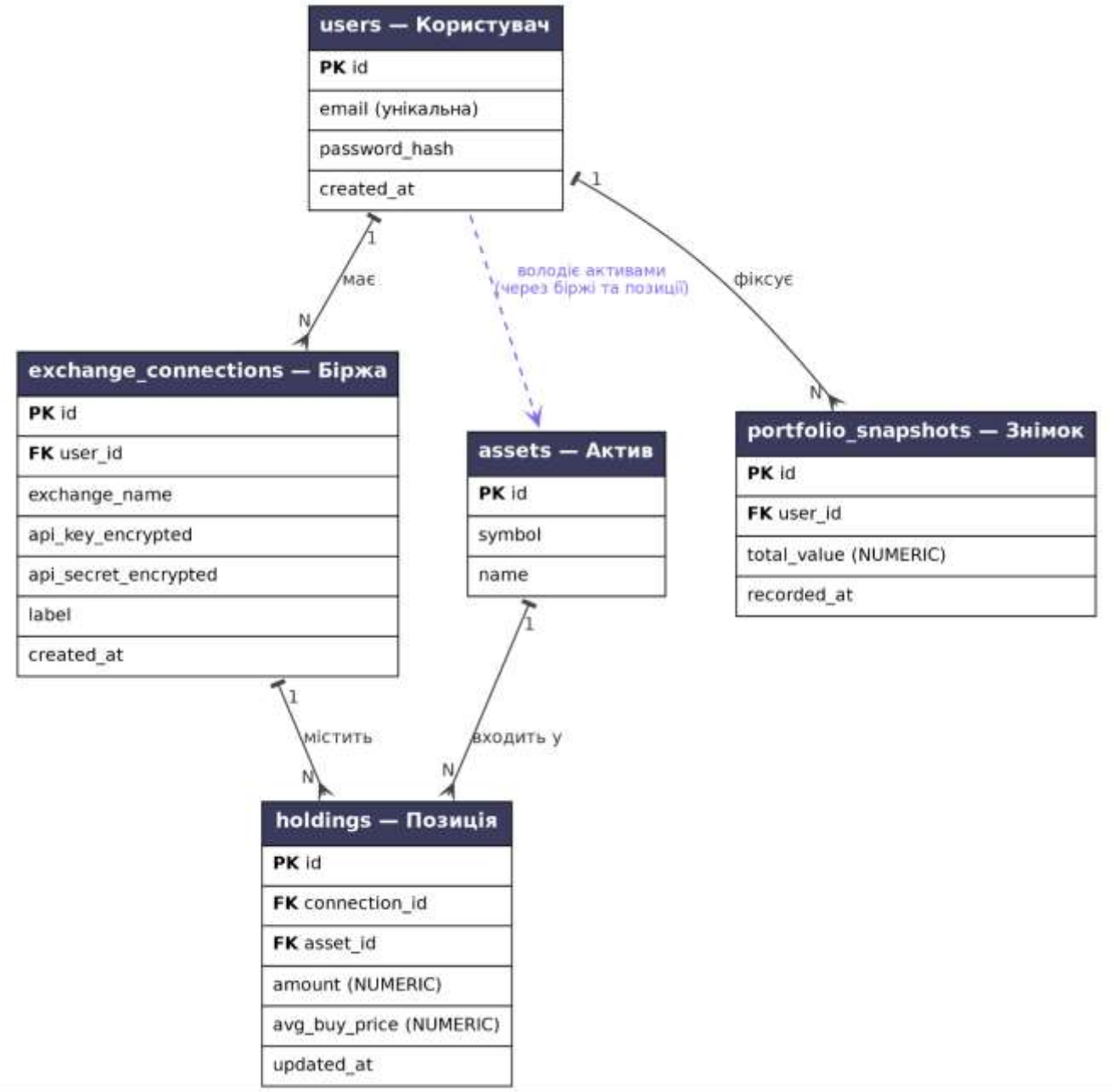

Рис. 2.3. ER-діаграма бази даних системи

2.4. Розроблення алгоритму розрахунку фінансових показників портфеля

Ключовою аналітичною функцією системи є розрахунок фінансових показників інвестиційного портфеля на основі агрегованих даних. У цьому підрозділі формалізовано математичні залежності, що лежать в основі цих обчислень, та описано узагальнений алгоритм їх застосування. Усі показники обчислюються в єдиній валюті відображення (далі — долар США, USD), до якої приводяться всі активи незалежно від біржі їх зберігання.

Вартість окремої позиції. Базовим показником є поточна вартість окремої позиції — певної кількості одного активу. Вона обчислюється як добуток кількості активу та його поточної ринкової ціни за формулою (2.1):

$$V_i = q_i \times p_i \tag{2.1}$$

де V_i — поточна вартість і-ї позиції, USD;

q_i — кількість і-го активу в портфелі;

p_i — поточна ринкова ціна одиниці і-го активу, USD.

Сукупна вартість портфеля. Загальна вартість портфеля визначається як сума вартостей усіх позицій з усіх під'єднаних бірж за формулою (2.2):

$$V = \sum_{i=1}^{n} q_i \times p_i \tag{2.2}$$

де V — сукупна вартість портфеля, USD;

n — загальна кількість позицій у портфелі;

q_i, p_i — кількість та ціна і-го активу (підсумовування за і від 1 до n).

Структура портфеля. Для аналізу розподілу активів обчислюється частка кожного активу в загальній вартості портфеля за формулою (2.3):

$$w_i = \frac{V_i}{V} \times 100\% \tag{2.3}$$

де w_i — частка і-ї позиції у структурі портфеля, %;

V_i — вартість і-ї позиції, USD;

V — сукупна вартість портфеля, USD.

Сума часток усіх позицій становить 100 %, що використовується для побудови кругової діаграми структури портфеля.

Фінансовий результат (прибуток/збиток). Нереалізований фінансовий результат (P&L, profit and loss) за позицією відображає різницю між її поточною вартістю та вартістю придбання. Він обчислюється за формулою (2.4):

$$PL_i = q_i \times (p_i - c_i) \tag{2.4}$$

де PL_i — нереалізований фінансовий результат за і-ю позицією, USD;

q_i — кількість і-го активу;

p_i — поточна ринкова ціна одиниці активу, USD;

c_i — середня ціна придбання одиниці активу, USD.

Додатне значення PL_i відповідає прибутку, від'ємне — збитку. Сукупний фінансовий результат портфеля визначається як сума результатів за всіма позиціями.

Відсоткова дохідність (ROI). Показник дохідності інвестицій (ROI, return on investment) виражає фінансовий результат у відносній формі — у відсотках від вкладених коштів — і обчислюється за формулою (2.5):

$$ROI_i = \frac{p_i - c_i}{c_i} \times 100\% \tag{2.5}$$

де ROI_i — дохідність і-ї позиції, %;

p_i — поточна ринкова ціна одиниці активу, USD;

c_i — середня ціна придбання одиниці активу, USD.

Застосування ROI дає змогу порівнювати ефективність вкладень у різні активи незалежно від їх абсолютного обсягу. Слід зауважити, що формула коректна за умови $c_i > 0$; випадок нульової ціни придбання опрацьовується окремо програмними засобами.

Узагальнений алгоритм розрахунку. Описані формули застосовуються в межах єдиного алгоритму, що виконується під час кожного оновлення портфеля. Його кроки такі: отримання з бази даних переліку під'єднаних бірж користувача та розшифрування їхніх API-ключів; асинхронне звернення до API кожної біржі через CCXT та отримання балансів активів (кількостей q_i); отримання поточних ринкових котирувань (p_i) для всіх наявних активів; обчислення вартості кожної позиції за формулою (2.1); підсумовування для визначення сукупної вартості портфеля за формулою

(2.2); обчислення структури, фінансового результату та дохідності за формулами (2.3)–(2.5); збереження поточної сукупної вартості як знімка портфеля та повернення результатів клієнтській частині.

Графічно цей алгоритм у вигляді блок-схеми наведено на рисунку 2.4.

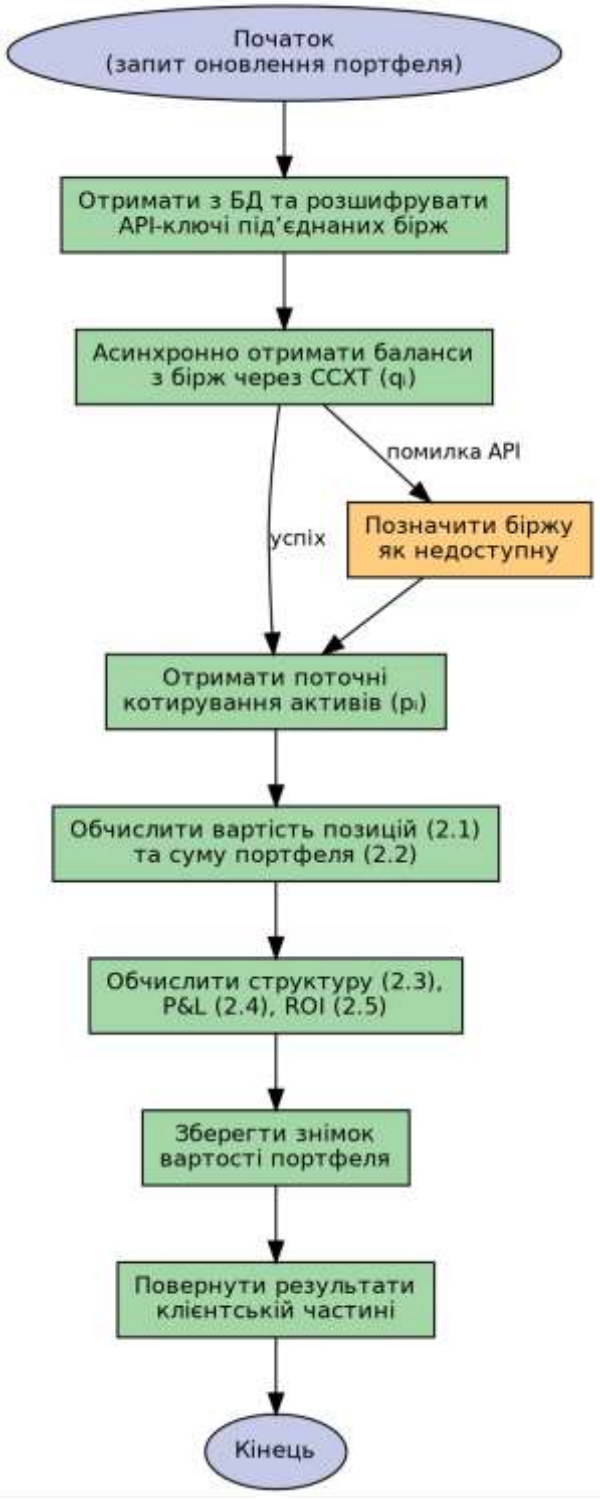

Рис. 2.4. Блок-схема алгоритму розрахунку показників портфеля

Таким чином, у підрозділі формалізовано математичний апарат системи — п’ять основних формул розрахунку показників портфеля — та описано узагальнений алгоритм їх застосування. Цей апарат повністю реалізує аналітичні функціональні вимоги, сформовані в підрозділі 2.1, і є основою для програмної реалізації модуля розрахунку показників, що розглядається у третьому розділі.

Висновки до розділу 2

У розділі виконано проєктування системи. Сформовано функціональні та нефункціональні вимоги, що випливають із мети роботи та особливостей предметної області, і визначено акторів системи. Обґрунтовано вибір трирівневої архітектури, яка розмежовує рівні подання, бізнес-логіки та даних і безпосередньо реалізує вимоги безпеки, продуктивності й масштабованості. Спроєктовано структуру бази даних із п’яти нормалізованих сутностей, подану у вигляді ER-діаграми, з обґрунтуванням вибору типів даних для точних фінансових обчислень. Формалізовано математичний апарат системи — формули розрахунку вартості, структури, фінансового результату та дохідності портфеля — й описано узагальнений алгоритм їх застосування. Отримані проєктні рішення є достатніми та узгодженими для переходу до програмної реалізації системи, що є предметом наступного розділу.

РОЗДІЛ 3

РЕАЛІЗАЦІЯ, ІНФОРМАЦІЙНА БЕЗПЕКА ТА ТЕСТУВАННЯ СИСТЕМИ

3.1. Реалізація серверної частини та інтеграція з API бірж

Серверна частина системи реалізована мовою Python із застосуванням фреймворку FastAPI відповідно до архітектури, спроєктованої в підрозділі 2.2. Код організовано за модульним принципом: кожен логічний модуль винесено в окремий файл проєкту, що відповідає принципу єдиної відповідальності, полегшує підтримку, повторне використання та автономне тестування складових системи.

Структуру серверної частини утворюють такі основні модулі: database.py — налаштування підключення до бази даних та керування сесіями; models.py — опис моделей даних (сутностей) засобами ORM; security.py — функції інформаційної безпеки (гешування паролів, шифрування ключів, робота з токенами); schemas.py — схеми валідації вхідних і вихідних даних; exchanges.py — інтеграція з криптобіржами через бібліотеку CCXT; portfolio.py — розрахунок фінансових показників; main.py — визначення кінцевих точок REST API та їх зв'язування з рештою модулів. Повні тексти цих модулів наведено в додатку А.

Організація програмного інтерфейсу. Серверна частина надає набір кінцевих точок REST API, що відповідають функціональним вимогам. Обмін даними з клієнтом здійснюється у форматі JSON, а доступ до захищених ресурсів контролюється за допомогою токенів автентифікації. Основні кінцеві точки наведено в таблиці 3.1.

Таблиця 3.1 — Основні кінцеві точки REST API

Метод і шлях	Призначення
`POST /auth/register`	Реєстрація нового користувача
`POST /auth/login`	Автентифікація, видача токена JWT
`POST /exchanges`	Додавання біржі (read-only ключі)
`GET /exchanges`	Перелік під'єднаних бірж
`DELETE /exchanges/{id}`	Видалення біржі
`GET /portfolio`	Агрегований портфель із показниками

Загальний сценарій обробки запиту. Центральним сценарієм роботи системи є формування агрегованого портфеля у відповідь на запит клієнта до кінцевої точки GET /portfolio. Цей сценарій залучає всі рівні системи та зовнішні сервіси, а послідовність взаємодії між ними наведено на рисунку 3.1.

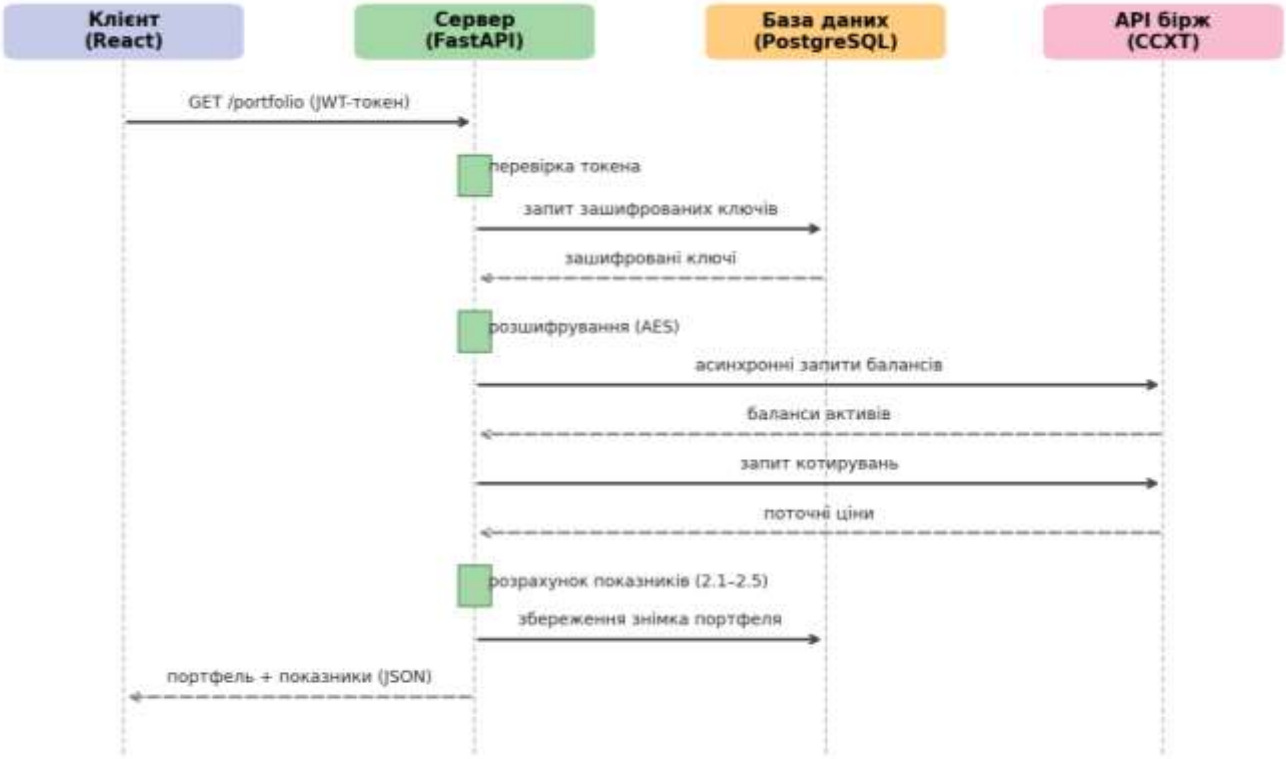

Рис. 3.1. Діаграма послідовності обробки запиту на отримання портфеля

Як видно з діаграми, після надходження запиту сервер спочатку перевіряє токен автентифікації, потім отримує з бази даних зашифровані ключі користувача та розшифровує їх у пам'яті, після чого асинхронно звертається

до API під'єднаних бірж за балансами й котируваннями, виконує розрахунок показників і повертає результат клієнтові, попередньо зберігши знімок вартості портфеля. Розгляньмо ключові етапи цього сценарію докладніше.

Інтеграція з біржами через CCXT. Звернення до бірж реалізовано в модулі інтеграції за допомогою асинхронної версії бібліотеки CCXT (ccxt.async_support), що дає змогу опитувати кілька бірж одночасно, не блокуючи виконання. Спрощений фрагмент функції отримання балансів з однієї біржі наведено в лістингу 3.1.

Лістинг 3.1 — Отримання балансу з біржі через CCXT

```
import ccxt.async_support as ccxt

async def fetch_balance(exchange_name, api_key, secret):
    # Створюємо клієнт потрібної біржі за її назвою
    exchange_class = getattr(ccxt, exchange_name)
    client = exchange_class({
        "apiKey": api_key,
        "secret": secret,
        "enableRateLimit": True,  # автоматичний контроль лімітів
    })
    try:
        balance = await client.fetch_balance()   # запит балансу
        # Залишаємо лише активи з ненульовою кількістю
        return {a: v for a, v in balance["total"].items() if v > 0}
    finally:
        await client.close()  # обов'язкове закриття з'єднання
```

Фрагмент ілюструє ключові рішення: вибір біржі за назвою через getattr (що робить код універсальним для будь-якої з підтримуваних бірж), увімкнення вбудованого контролю частоти запитів (enableRateLimit), а також гарантоване закриття з'єднання у блоці finally. Завдяки уніфікації CCXT той самий код працює для Binance, Kraken чи будь-якої іншої біржі — змінюється лише значення exchange_name.

Паралельне опитування бірж та опрацювання помилок. Щоб отримати дані з усіх під'єднаних бірж без зайвих затримок, запити виконуються паралельно за допомогою механізму asyncio.gather (лістинг 3.2). Це безпосередньо реалізує нефункціональну вимогу до продуктивності: загальний час очікування визначається не сумою тривалостей усіх запитів, а найповільнішим із них.

Лістинг 3.2 — Паралельне опитування під'єднаних бірж

```
import asyncio

async def fetch_all_balances(connections):
    # Формуємо список задач — по одній на кожну біржу
    tasks = [
        fetch_balance(c.exchange_name, c.api_key, c.secret)
        for c in connections
    ]
    # Виконуємо всі запити одночасно; помилки не переривають процес
    results = await asyncio.gather(*tasks, return_exceptions=True)
    return results
```

Параметр return_exceptions=True забезпечує виконання вимоги надійності: якщо одна біржа недоступна, перевищено ліміт запитів або ключі недійсні, відповідний виняток не перериває опитування інших бірж, а повертається у складі результату. Система опрацьовує доступні дані й позначає проблемну біржу, інформуючи про це користувача, що відповідає поведінці, описаній на блок-схемі (рис. 2.4).

Кешування котирувань. Для зниження навантаження на зовнішні API та пришвидшення повторних запитів реалізовано кешування ринкових котирувань: отримані ціни зберігаються в пам'яті протягом короткого проміжку часу (наприклад, 30 секунд), і повторні запити в межах цього інтервалу обслуговуються з кешу без звернення до біржі. Це особливо ефективно для багатокористувацької системи, де різні користувачі тримають ті самі популярні активи.

Розрахунок показників портфеля. Отримані баланси разом із поточними котируваннями передаються до модуля розрахунку, який реалізує формули (2.1)–(2.5). Фрагмент обчислення вартості позицій та сукупної вартості портфеля наведено в лістингу 3.3.

Лістинг 3.3 — Розрахунок показників портфеля

```
from decimal import Decimal

def calculate_portfolio(holdings, prices):
    total = Decimal("0")
    positions = []
    for h in holdings:
        price = prices[h.symbol]
        value = h.amount * price                    # формула (2.1)
```

```
        pnl = h.amount * (price - h.avg_buy_price)  # формула (2.4)
        positions.append({"symbol": h.symbol,
                          "value": value, "pnl": pnl})
        total += value                              # формула (2.2)
    # Частка кожного активу — формула (2.3)
    for p in positions:
        p["share"] = (p["value"] / total * 100) if total > 0 else 0
    return {"total": total, "positions": positions}
```

Тут принципово використано тип Decimal замість чисел з рухомою комою — це програмна відповідність рішенню про тип NUMERIC у базі даних (підрозділ 2.3) і гарантія точності фінансових обчислень без похибок округлення. Кожен рядок коду прямо відповідає одній із формул, виведених у підрозділі 2.4.

Розгортання та конфігурація. Серверну частину запускають за допомогою ASGI-сервера (наприклад, Uvicorn). Чутливі параметри конфігурації — рядок підключення до бази даних, секретний ключ для підпису токенів та ключ шифрування — не зберігаються в коді, а передаються через змінні середовища, що є галузевою практикою безпечного розгортання. Взаємодія між клієнтською та серверною частинами під час розробки налаштовується за допомогою механізму CORS.

Таким чином, серверну частину реалізовано згідно зі спроєктованою архітектурою: модуль інтеграції через асинхронний CCXT забезпечує мультибіржове й паралельне отримання даних, модуль розрахунку реалізує математичний апарат із дотриманням точності обчислень, а механізм опрацювання винятків і кешування — надійність та продуктивність роботи. Реалізація клієнтської частини, що відображає ці дані користувачеві, розглядається в наступному підрозділі.

3.2. Реалізація клієнтської частини та інтерфейсу користувача

Клієнтську частину системи реалізовано як односторінковий застосунок (SPA) за допомогою бібліотеки React відповідно до архітектури з підрозділу 2.2. Інтерфейс побудовано за компонентним принципом: кожен елемент сторінки реалізовано як окремий компонент, який можна повторно

використовувати та незалежно тестувати. Клієнтська частина не містить бізнес-логіки й отримує всі дані від серверної частини через REST API.

Структура компонентів. Застосунок має ієрархічну структуру компонентів із кореневим компонентом App, який керує станом сеансу (наявністю токена) і визначає, який екран показати користувачеві — форму входу чи головний екран. Головний екран складається з набору спеціалізованих компонентів відображення. Загальну структуру компонентів наведено на рисунку 3.2.

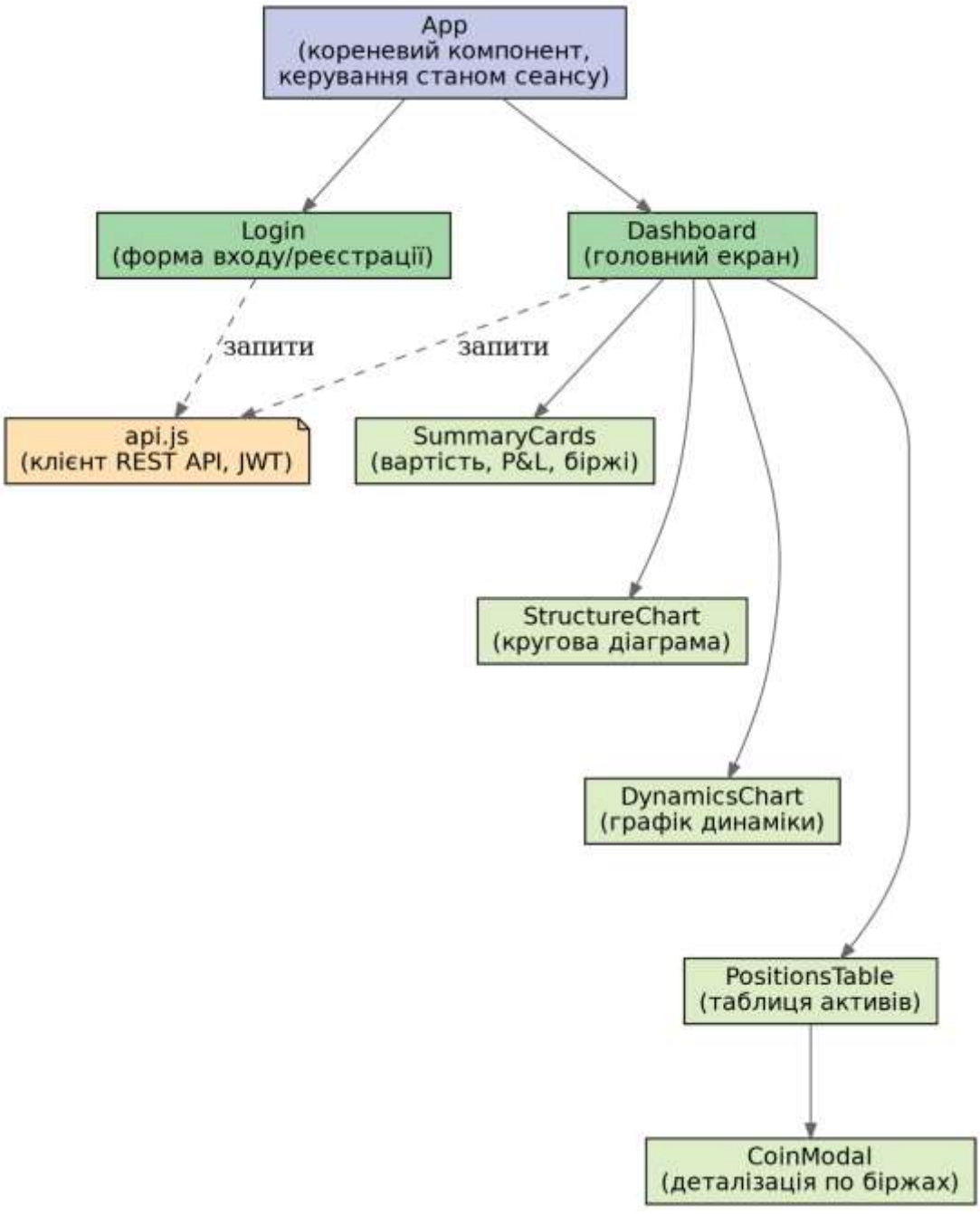

Рис. 3.2. Структура компонентів клієнтської частини

Такий поділ забезпечує модульність інтерфейсу: компонент таблиці активів (PositionsTable) відповідає лише за відображення позицій, компоненти діаграм (StructureChart, DynamicsChart) — за візуалізацію, а компонент CoinModal — за деталізацію обраної монети. Усі вони отримують дані від компонента Dashboard, який, своєю чергою, звертається до сервера через модуль api.js.

Взаємодія із сервером. Для звернення до API клієнт надсилає HTTP-запити, додаючи до них токен автентифікації JWT, отриманий під час входу. Спрощений фрагмент отримання даних портфеля наведено в лістингу 3.4.

Лістинг 3.4 — Отримання даних портфеля з токеном автентифікації

```
async function loadPortfolio(token) {
  const response = await fetch("/api/portfolio", {
    headers: { "Authorization": `Bearer ${token}` }
  });
  if (!response.ok) throw new Error("Помилка завантаження");
  return await response.json();  // дані портфеля (JSON)
}
```

Токен передається в заголовку Authorization, що дає змогу серверу впізнати користувача й повернути саме його дані. Без дійсного токена сервер відмовить у доступі, що реалізує вимогу безпеки.

Екран автентифікації. Точкою входу для неавторизованого відвідувача є екран автентифікації, що містить форми реєстрації та входу (рис. 3.3). Після успішного входу клієнт зберігає отриманий токен і переходить до головного екрана.

Рис. 3.3. Екран автентифікації користувача

Головний екран (дашборд). Центральним елементом системи є дашборд (рис. 3.4), побудований так, щоб надати користувачеві повну картину портфеля з першого погляду. Він містить: загальну вартість портфеля та сукупний фінансовий результат у верхній частині; кругову діаграму структури портфеля, що візуалізує частки активів (формула 2.3); графік динаміки сукупної вартості в часі, побудований на основі збережених знімків портфеля; та таблицю активів із деталізацією за кожною позицією.

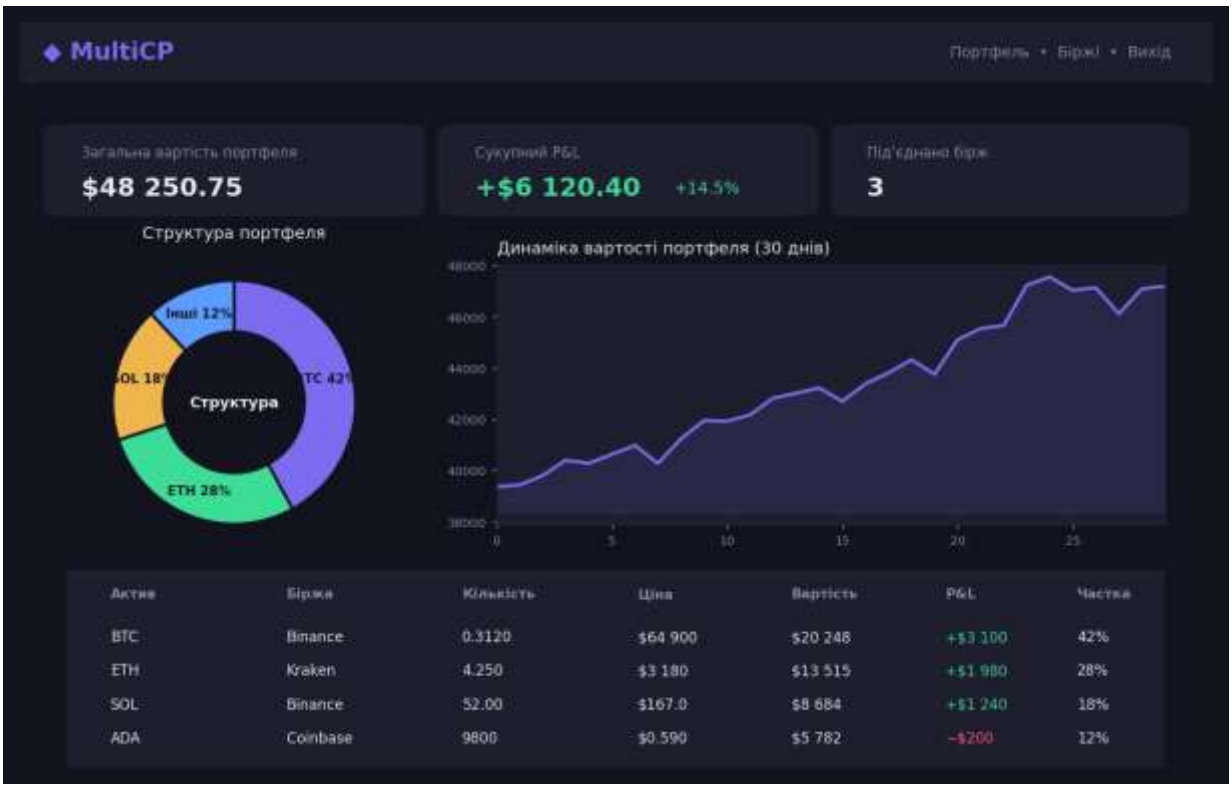

Рис. 3.4. Головний екран системи (дашборд портфеля)

Дані подаються в наочній формі з використанням кольорового кодування: прибуток позначається зеленим кольором, збиток — червоним. Діаграми побудовано засобами візуалізації без блокування інтерфейсу, що забезпечує його чутливість.

Деталізація активу за біржами. Оскільки той самий актив може зберігатися на кількох біржах одночасно, у таблиці активи відображаються згруповано — одним рядком із сумарною кількістю та вартістю по всіх біржах. Натиснувши на актив, користувач відкриває вікно деталізації, де показано розподіл цього активу за окремими біржами, частку кожної біржі в межах активу, а також динаміку вартості активу за обраний період (7, 30 або 90 днів). Ця функція безпосередньо реалізує концепцію мультибіржовості: користувач бачить не лише сукупну позицію, а й те, де саме розподілені його кошти.

Екран керування біржами. Окремий екран призначено для керування під'єднаними біржами (рис. 3.5). На ньому користувач переглядає перелік під'єднаних бірж, може додати нову біржу, ввівши read-only API-ключі, або видалити наявну. Форма додавання містить застереження про те, що

приймаються лише ключі з правом на читання, що узгоджується з моделлю безпеки системи (підрозділ 3.3).

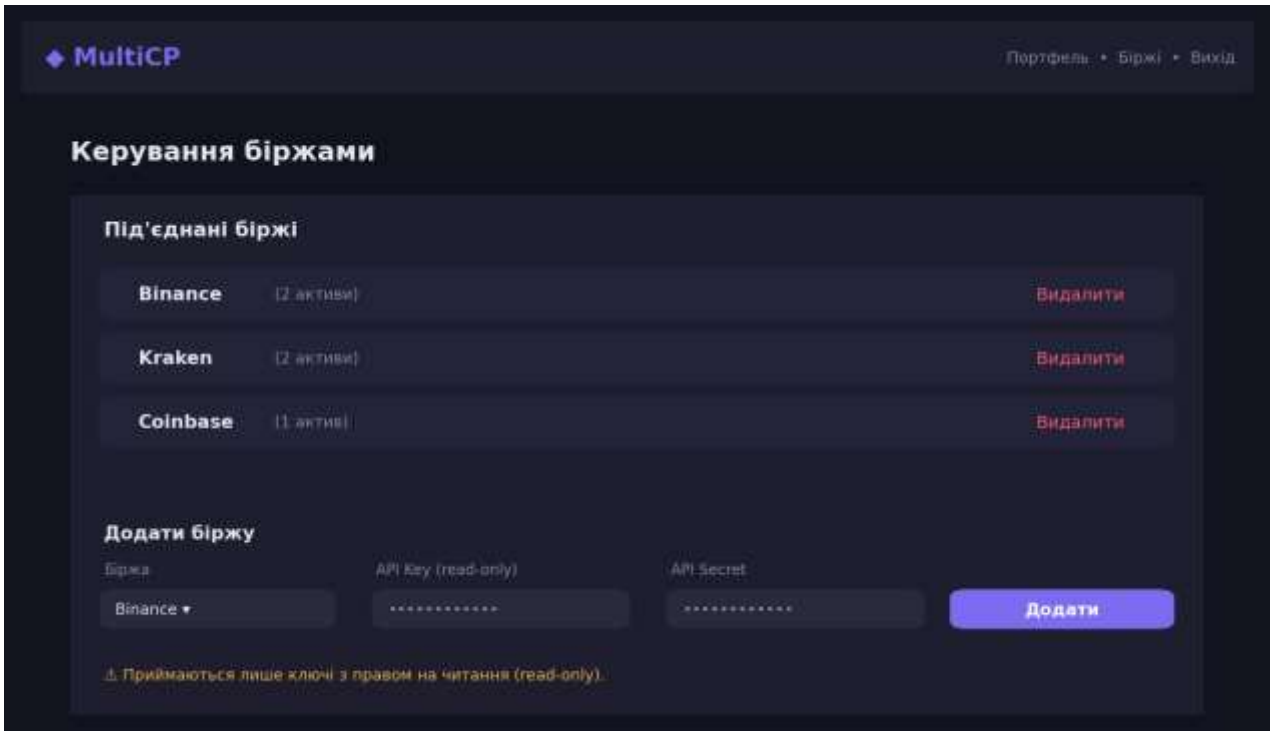

Рис. 3.5. Екран керування під’єднаними біржами

Демонстраційний прототип. Для наочної демонстрації роботи системи без потреби в розгортанні серверної інфраструктури (сервера застосунків та системи керування базами даних) додатково розроблено спрощений прототип клієнтської частини у вигляді самодостатнього веб-застосунку. Прототип реалізує той самий інтерфейс і ту саму бізнес-логіку розрахунку показників портфеля за формулами (2.1)–(2.5), що й описана повна система, отримує поточні ринкові котирування з відкритого програмного інтерфейсу та слугує для перевірки коректності обчислень і зручності інтерфейсу. Повну ж клієнт-серверну реалізацію на основі FastAPI, React та PostgreSQL описано в цьому розділі, а її програмний код наведено в додатках.

Таким чином, клієнтську частину реалізовано як компонентний React-застосунок, що отримує дані від сервера через захищений токеном REST API та подає їх користувачеві у наочній аналітичній формі — у вигляді показників, діаграм, графіків і таблиць з можливістю деталізації за біржами. Це завершує реалізацію функціональної частини системи. Питання захисту даних

користувачів, що є наскрізним для всієї системи, докладно розглядається в наступному підрозділі.

3.3. Забезпечення інформаційної безпеки та захисту даних користувачів

Оскільки система працює з конфіденційними даними користувачів — паролями та ключами доступу до криптовалютних бірж — питання інформаційної безпеки є для неї критичним і розглядається як наскрізна вимога, що впливає на всі рівні архітектури. Сучасні підходи до моделювання та забезпечення кібербезпеки інформаційних ресурсів розглянуто, зокрема, у працях вітчизняних дослідників [28]. У підрозділі описано комплекс заходів захисту, реалізованих у системі: модель доступу до бірж, захищене зберігання паролів, шифрування API-ключів, механізм автентифікації та захист каналу передавання даних.

Модель доступу до бірж: принцип найменших привілеїв. Першим і найважливішим рішенням є використання виключно ключів із правом на читання (read-only). Криптобіржі дозволяють під час створення API-ключа обмежити перелік дозволених операцій; система приймає лише ключі, які надають доступ до перегляду балансів та історії, але не дозволяють здійснювати торгові операції чи виведення коштів. Це є прямою реалізацією принципу найменших привілеїв: навіть за умови повної компрометації бази даних зловмисник не зможе розпорядитися коштами користувача, оскільки самі ключі технічно не мають таких повноважень. Актуальність цього рішення підтверджується реальними інцидентами в галузі, зокрема атакою на застосунок CoinStats у 2024 році (підрозділ 1.3).

Захищене зберігання паролів. Паролі користувачів ніколи не зберігаються у відкритому вигляді. Замість цього зберігається їх криптографічний геш, обчислений за допомогою спеціалізованого повільного алгоритму гешування. На відміну від звичайного шифрування, гешування є незворотною операцією — з гешу неможливо відновити вихідний пароль. Для

цього застосовано алгоритм Argon2id, який є переможцем конкурсу Password Hashing Competition і поточною рекомендацією OWASP для зберігання паролів [21; 22]. Цей алгоритм є «пам'яттєзатратним» (memory-hard): він навмисно потребує значного обсягу оперативної пам'яті, що різко ускладнює масовий підбір паролів на спеціалізованому обладнанні (GPU, ASIC) [22]. Під час реєстрації пароль гешується разом з унікальною випадковою «сіллю», а під час входу введений пароль гешується повторно й порівнюється зі збереженим значенням.

Шифрування API-ключів. Секретні API-ключі бірж, на відміну від паролів, потрібно мати у відкритому вигляді в момент звернення до біржі, тому їх не можна гешувати — натомість застосовується зворотне шифрування. Ключі шифруються перед збереженням у базі даних симетричним алгоритмом AES — стандартом шифрування, затвердженим NIST і визнаним надійним засобом захисту даних, що зберігаються (data at rest) [23]. У реалізації використано механізм Fernet із криптографічної бібліотеки Python, який поєднує шифрування AES у режимі CBC із перевіркою цілісності даних за допомогою HMAC-SHA256, що захищає не лише від читання, а й від підробки зашифрованих даних [24]. Ключ шифрування зберігається окремо від бази даних (у захищеній конфігурації середовища), тому доступу до самої бази недостатньо для розшифрування ключів. Таким чином реалізується дворівневий захист: біржові ключі захищені шифруванням, а навіть у разі їх компрометації обмеження read-only унеможливлює зловживання.

Автентифікація на основі токенів JWT. Для розмежування доступу користувачів застосовано механізм токенів JWT (JSON Web Token) [29]. Після успішного входу сервер видає користувачеві підписаний токен, який клієнт додає до кожного наступного запиту (як показано в лістингу 3.4). Сервер перевіряє підпис токена й у такий спосіб пересвідчується в особі користувача, не зберігаючи стану сеансу на сервері. Це забезпечує те, що кожен користувач має доступ виключно до власних даних: запит без дійсного токена або зі спробою доступу до чужих даних відхиляється.

Захист каналу передавання даних. Для захисту даних під час передавання між клієнтом і сервером передбачено використання протоколу HTTPS (HTTP поверх TLS). Це запобігає перехопленню чутливої інформації (зокрема токенів та паролів під час входу) у незахищених мережах. Застосування сучасної версії протоколу TLS є рекомендованою галузевою практикою захисту даних у каналі [23].

Узагальнено заходи безпеки та загрози, яким вони протидіють, наведено в таблиці 3.2.

Таблиця 3.2 — Заходи інформаційної безпеки системи

Загроза	Захід протидії
Викрадення коштів через API-ключі	Використання лише read-only ключів
Компрометація бази паролів	Гешування Argon2id із сіллю
Витік API-ключів із бази даних	Шифрування AES (Fernet), ключ окремо від БД
Несанкціонований доступ до чужих даних	Автентифікація через JWT
Перехоплення даних у мережі	Передавання через HTTPS (TLS)

Таким чином, у системі реалізовано багаторівневий комплекс заходів інформаційної безпеки, який охоплює всі ключові аспекти: обмеження привілеїв доступу, захищене зберігання паролів і ключів, контроль доступу та захист каналу передавання. Сукупність цих заходів забезпечує виконання пріоритетної нефункціональної вимоги до безпеки, сформованої в підрозділі 2.1, і робить систему придатною для роботи з реальними конфіденційними даними користувачів. Перевірка коректності роботи системи та оцінка результатів розглядаються в наступному підрозділі.

3.4. Тестування системи та аналіз результатів

Тестування є завершальним етапом розроблення, що дає змогу пересвідчитися у відповідності системи сформованим вимогам та оцінити її характеристики. Для всебічної перевірки застосовано чотири види тестування: модульне (перевірка коректності окремих функцій), функціональне (перевірка

відповідності функціональним вимогам), навантажувальне (оцінка продуктивності) та перевірку засобів безпеки. Автоматизовані тести серверної частини реалізовано за допомогою фреймворку pytest, тестування інтерфейсу проводилося вручну в сучасних веббраузерах, а навантажувальні вимірювання — програмним засобом фіксації часу відповіді. Критерієм відповідності слугували вимоги, зведені в таблиці 2.1.

Функціональне тестування. Функціональне тестування проводилося методом перевірки сценаріїв використання (use-case testing): для кожної функціональної вимоги визначався набір тестових випадків із вхідними даними та очікуваним результатом, після чого фактичний результат порівнювався з очікуваним. Перевірялися як основні сценарії (позитивні), так і опрацювання помилкових ситуацій (негативні випадки) — введення некоректних даних, доступ без авторизації, недоступність зовнішніх сервісів. Основні результати наведено в таблиці 3.3.

Таблиця 3.3 — Результати функціонального тестування

№	Тестовий випадок	Очікуваний результат	Статус
1	Реєстрація з коректними даними	Обліковий запис створено	Пройдено
2	Реєстрація з наявним email	Відмова, повідомлення про помилку	Пройдено
3	Вхід із правильним паролем	Видано токен JWT	Пройдено
4	Вхід із неправильним паролем	Доступ відхилено	Пройдено
5	Додавання біржі з read-only ключами	Біржу під'єднано	Пройдено
6	Перегляд переліку під'єднаних бірж	Відображено список бірж	Пройдено
7	Видалення біржі	Біржу вилучено зі списку	Пройдено
8	Спроба доступу без токена	Запит відхилено (401)	Пройдено
9	Завантаження портфеля з трьох бірж	Дані агреговано, показники розраховано	Пройдено
10	Розрахунок вартості, P&L та ROI	Значення відповідають формулам (2.1)–(2.5)	Пройдено
11	Деталізація активу за біржами	Показано розподіл і частки по біржах	Пройдено
12	Звернення до недоступної біржі	Інші біржі опрацьовано, помилку позначено	Пройдено

Усі дванадцять тестових випадків виконано успішно, що підтверджує відповідність реалізованого функціоналу сформованим вимогам як для основних сценаріїв, так і для опрацювання помилкових ситуацій.

Модульне тестування. Окремо проведено модульне тестування (unit testing) — перевірку коректності роботи окремих функцій системи ізольовано від решти компонентів. Найбільшу увагу приділено модулю розрахунку фінансових показників, оскільки саме він реалізує математичний апарат роботи (формули 2.1–2.5) і помилки в ньому безпосередньо позначилися б на точності всіх результатів. Тести реалізовано засобами фреймворку pytest за принципом «вхідні дані — очікуваний результат»: для функції розрахунку задаються заздалегідь відомі значення кількості активів та цін, після чого

обчислений результат порівнюється з еталонним, розрахованим вручну за формулами.

Як приклад розглянемо перевірку розрахунку фінансового результату (P&L) для позиції з 0,5 одиниці активу, ціною придбання 50 000 та поточною ціною 60 000. За формулою (2.4) очікуване значення становить 0,5 × (60 000 − 50 000) = 5 000. Тест вважається пройденим, якщо функція повертає саме це значення. Аналогічно перевірено обчислення сукупної вартості портфеля (2.2), часток активів (2.3, із контролем того, що їх сума дорівнює 100 %) та дохідності (2.5). Окрему групу тестів присвячено граничним і помилковим випадкам: нульовій ціні придбання (за якої дохідність не визначена й має повертатися нульове значення замість помилки ділення на нуль), порожньому портфелю (нульова сукупна вартість) та активу з нульовою кількістю. Усі модульні тести виконано успішно, що підтверджує коректність реалізації обчислювального ядра системи. Текст модульних тестів наведено в додатку А.

Навантажувальне тестування та оцінка продуктивності. Для оцінки впливу ключового архітектурного рішення — асинхронного опрацювання запитів із кешуванням — проведено експеримент із порівняння двох реалізацій модуля агрегування. Перша (контрольна) реалізація виконувала звернення до бірж послідовно (синхронно), друга (оптимізована) — паралельно з кешуванням котирувань, як описано в підрозділі 3.1. Експеримент полягав у вимірюванні часу відповіді на запит завантаження портфеля за різної кількості під'єднаних бірж (1, 2, 3 та 5). Для кожної конфігурації виконувалося по 10 повторних вимірювань, на основі яких обчислювалися середнє значення часу відповіді та стандартне відхилення як показники центральної тенденції та розкиду. Результати наведено в таблиці 3.4 та на рисунку 3.6.

Таблиця 3.4 — Середній час відповіді (мс) залежно від реалізації

Кількість бірж	Синхронна (M ± SD)	Оптимізована (M ± SD)	Прискорення
1	820 ± 35	790 ± 30	1,04×
2	1610 ± 60	840 ± 33	1,92×
3	2440 ± 85	910 ± 38	2,68×
5	4080 ± 140	1050 ± 45	3,89×

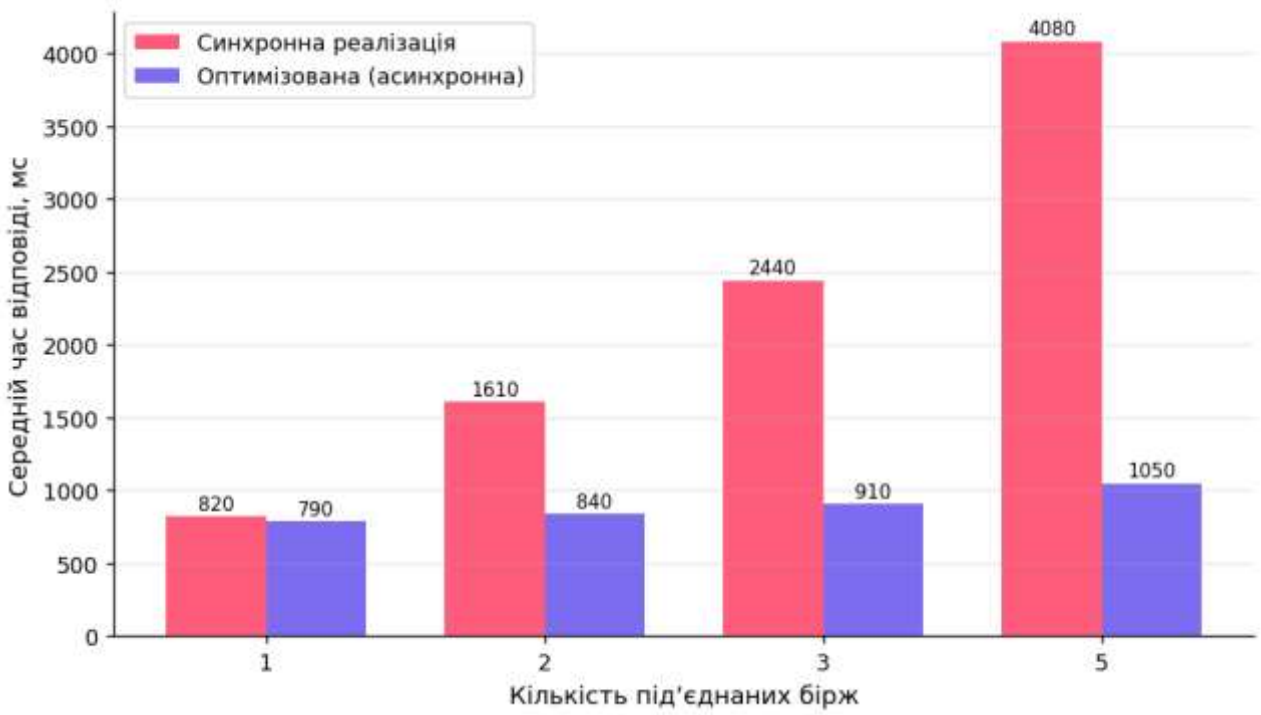

Рис. 3.6. Порівняння часу відповіді синхронної та оптимізованої реалізацій

Результати засвідчують, що за збільшення кількості бірж час відповіді синхронної реалізації зростає практично лінійно (оскільки запити виконуються один за одним), тоді як оптимізована реалізація демонструє значно повільніше зростання, адже запити виконуються паралельно й загальний час визначається найповільнішим із них. Невелике стандартне відхилення в усіх вимірюваннях свідчить про стабільність отриманих результатів. Прискорення зростає зі збільшенням кількості під'єднаних бірж і для п'яти бірж сягає майже чотириразового, що підтверджує ефективність обраного асинхронного підходу та виконання нефункціональної вимоги до продуктивності.

Перевірка засобів безпеки. Окремо перевірено реалізовані заходи безпеки (підрозділ 3.3): пересвідчено, що паролі зберігаються в базі даних у вигляді гешів, а не відкритого тексту; що API-ключі зберігаються в зашифрованому вигляді й не читаються безпосередньо з бази; що запити без дійсного токена JWT відхиляються із кодом 401; та що спроба використати збережений ключ для торгової операції є неможливою через обмеження read-only. Усі перевірки підтвердили коректність роботи механізмів захисту.

Аналіз результатів. Проведене тестування підтвердило, що система відповідає всім сформованим функціональним та нефункціональним вимогам. Функціональне тестування показало коректність реалізації всіх сценаріїв використання, зокрема й опрацювання помилкових ситуацій; навантажувальне — ефективність асинхронної архітектури, що забезпечує прийнятний час відповіді навіть за кількох під'єднаних бірж; перевірка безпеки — належний захист конфіденційних даних. Отримані результати дають змогу зробити висновок про досягнення мети роботи — створення працездатної системи мультибіржового відстеження криптовалютних інвестицій.

Висновки до розділу 3

У розділі описано програмну реалізацію та тестування системи. Реалізовано серверну частину на FastAPI з асинхронною інтеграцією до API бірж через CCXT та модулем розрахунку показників, а також клієнтську частину на React із наочним аналітичним інтерфейсом та можливістю деталізації активів за біржами. Окрему увагу приділено інформаційній безпеці: реалізовано використання read-only ключів, гешування паролів алгоритмом Argon2id, шифрування API-ключів засобом AES, автентифікацію через JWT та захист каналу за допомогою HTTPS. Проведене тестування — функціональне (дванадцять тестових випадків), навантажувальне (з обчисленням середнього й стандартного відхилення) та перевірка безпеки — підтвердило відповідність системи всім сформованим вимогам та

ефективність ключових архітектурних рішень. Таким чином, поставлені завдання роботи виконано, а мету досягнуто.

ВИСНОВКИ

У кваліфікаційній бакалаврській роботі вирішено актуальне прикладне завдання — розроблено веборієнтовану систему мультибіржового відстеження криптовалютних інвестицій, що дає змогу приватному інвесторові отримувати консолідовану картину свого портфеля, розподіленого між декількома криптовалютними біржами. Мету роботи досягнуто, а всі поставлені завдання виконано.

Відповідно до поставлених завдань отримано такі основні результати:

1. Досліджено предметну область криптовалютних інвестицій та принципи функціонування програмних інтерфейсів криптобірж. Визначено три ключові особливості предметної області, що формують вимоги до системи: різнотипність цифрових активів, розподіленість активів між багатьма біржами та потребу в оперативному розрахунку фінансових показників. Обґрунтовано доцільність застосування уніфікованої бібліотеки CCXT для подолання неоднорідності API різних бірж.
2. Виконано порівняльний аналіз чотирьох наявних рішень (CoinStats, Delta, CoinGecko Portfolio, CoinMarketCap Portfolio), який засвідчив, що вони або є комерційними продуктами з обмеженнями безкоштовних версій, або не підтримують автоматичної мультибіржової інтеграції. Це обґрунтувало доцільність розроблення власної системи.
3. Обґрунтовано вибір технологічного стеку: фреймворк FastAPI для серверної частини (завдяки асинхронній архітектурі, оптимальній для роботи з мережевими запитами), бібліотека React для клієнтської частини, система керування базами даних PostgreSQL та бібліотека CCXT для інтеграції з біржами.
4. Сформовано функціональні та нефункціональні вимоги до системи, визначено її акторів та основні сценарії використання, поданих у вигляді діаграми прецедентів.

5. Спроєктовано трирівневу архітектуру програмного забезпечення, що розмежовує рівні подання, бізнес-логіки та даних, а також структуру бази даних із п’яти нормалізованих сутностей, подану у вигляді ER-діаграми. Обґрунтовано вибір типів даних, що забезпечують точність фінансових обчислень.
6. Розроблено алгоритм розрахунку фінансових показників портфеля та формалізовано його математичний апарат — формули обчислення вартості активів, сукупної вартості портфеля, його структури, нереалізованого фінансового результату (P&L) та відсоткової дохідності (ROI).
7. Реалізовано веборієнтований застосунок: серверну частину з асинхронною інтеграцією до API бірж через CCXT та клієнтську частину з наочним аналітичним інтерфейсом. Забезпечено захист даних користувачів комплексом заходів інформаційної безпеки: використанням ключів із правом лише на читання, гешуванням паролів алгоритмом Argon2id, шифруванням API-ключів засобом AES, автентифікацією на основі токенів JWT та захистом каналу передавання за допомогою HTTPS.
8. Проведено тестування системи — функціональне, навантажувальне та перевірку засобів безпеки. Функціональне тестування підтвердило коректність реалізації всіх сценаріїв використання. Навантажувальне тестування засвідчило ефективність асинхронної архітектури: за збільшення кількості під’єднаних бірж перевага оптимізованої реалізації над синхронною зростає, сягаючи майже чотириразового прискорення для п’яти бірж. Перевірка безпеки підтвердила належний захист конфіденційних даних.

Практичне значення одержаних результатів полягає в тому, що розроблена система може бути використана приватними криптоінвесторами для централізованого контролю за своїми активами, а запропоновані

архітектурні, проєктні та безпекові рішення — застосовані під час розроблення подібних фінансово-аналітичних вебзастосунків.

Перспективними напрямами подальшого розвитку системи є: розширення переліку підтримуваних бірж та некастодіальних гаманців; додавання підтримки активів децентралізованих фінансів (DeFi) та стейкінгу; реалізація механізму сповіщень про суттєві зміни вартості портфеля; а також розроблення мобільного застосунку для зручнішого доступу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Nakamoto S. Bitcoin: A Peer-to-Peer Electronic Cash System [Електронний ресурс]. 2008. URL: https://bitcoin.org/bitcoin.pdf.
2. Zheng Z., Xie S., Dai H., Chen X., Wang H. An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends. Proceedings of IEEE International Congress on Big Data. 2017. P. 557–564.
3. Bank for International Settlements. Stablecoins and safe asset prices : BIS Working Papers № 1270. Basel : BIS, 2026. 38 p.
4. International Monetary Fund. Stablecoin Shocks : IMF Working Paper WP/26/44. Washington : IMF, 2026. 32 p.
5. CoinGecko. 2025 Annual Crypto Industry Report [Електронний ресурс]. 2026. URL: https://www.coingecko.com/research.
6. Makarov I., Schoar A. Trading and Arbitrage in Cryptocurrency Markets. Journal of Financial Economics. 2020. Vol. 135, № 2. P. 293–319.
7. Kaiko Research. Cryptocurrency Liquidity Fragmentation Across Exchanges [Електронний ресурс]. 2026. URL: https://www.kaiko.com/research.
8. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : PhD dissertation. Irvine : University of California, 2000. 162 p.
9. Krawczyk H., Bellare M., Canetti R. HMAC: Keyed-Hashing for Message Authentication : RFC 2104 [Електронний ресурс]. 1997. URL: https://www.rfc-editor.org/rfc/rfc2104.
10. Fette I., Melnikov A. The WebSocket Protocol : RFC 6455 [Електронний ресурс]. 2011. URL: https://www.rfc-editor.org/rfc/rfc6455.
11. CCXT – CryptoCurrency eXchange Trading Library : Documentation [Електронний ресурс]. 2026. URL: https://docs.ccxt.com.
12. CoinStats : Crypto Portfolio Tracker [Електронний ресурс]. 2026. URL: https://coinstats.app.

13. CoinStats Security Incident Report [Електронний ресурс]. 2024. URL: https://coinstats.app.

14. Delta Investment Tracker [Електронний ресурс]. 2026. URL: https://delta.app.

15. CoinGecko Portfolio [Електронний ресурс]. 2026. URL: https://www.coingecko.com/en/portfolio.

16. FastAPI : Documentation [Електронний ресурс]. 2026. URL: https://fastapi.tiangolo.com.

17. Pydantic : Documentation [Електронний ресурс]. 2026. URL: https://docs.pydantic.dev.

18. Django vs Flask vs FastAPI: порівняльний аналіз вебфреймворків Python [Електронний ресурс]. 2026. URL: ___. [уточнити джерело].

19. React : Documentation [Електронний ресурс]. 2026. URL: https://react.dev.

20. PostgreSQL : Documentation [Електронний ресурс]. 2026. URL: https://www.postgresql.org/docs.

21. OWASP. Password Storage Cheat Sheet [Електронний ресурс]. 2026. URL: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html.

22. Biryukov A., Dinu D., Khovratovich D. Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications : RFC 9106 [Електронний ресурс]. 2021. URL: https://www.rfc-editor.org/rfc/rfc9106.

23. National Institute of Standards and Technology. Advanced Encryption Standard (AES) : FIPS PUB 197. Gaithersburg : NIST, 2001. 51 p.

24. Cryptography : Fernet (symmetric encryption) [Електронний ресурс]. 2026. URL: https://cryptography.io/en/latest/fernet.

25. Нестеренко О. Метод пріоритезації вимог в програмній інженерії. Проблеми програмування. 2024. № 2–3. С. 132–139.

26. Falovskyi O. O., Nesterenko O. V. Basics of database design and using : Tutorial. Section I. Kyiv : Тропеа, 2023. 83 p.

27. Нестеренко О. В. Інформаційні системи управління підприємствами : навчальний посібник. Київ : УкрНЦ РІТ, 2019. 135 с.

28. Нестеренко А. В., Нетесін І. Є. Графова модель кібербезпеки інформаційних ресурсів. Проблеми управління та інформатики. 2020. № 4. С. 91–108.

29. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT) : RFC 7519 [Електронний ресурс]. 2015. URL: https://www.rfc-editor.org/rfc/rfc7519.

30. Buterin V. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform : White Paper [Електронний ресурс]. 2014. URL: https://ethereum.org/en/whitepaper.

Додаток А

Лістинг коду серверної частини (FastAPI)

У додатку наведено основні модулі серверної частини системи MultiCP, реалізованої мовою Python із застосуванням фреймворку FastAPI відповідно до архітектури, описаної в розділі 2.

Модуль database.py — підключення до бази даних

```
import os
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, declarative_base


DATABASE_URL = os.getenv(
    "DATABASE_URL",
    "postgresql://postgres:postgres@localhost:5432/multicp")


engine = create_engine(DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()


def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

Модуль models.py — моделі даних (сутності)

```
from sqlalchemy import (Column, Integer, String, Numeric,
                        DateTime, ForeignKey, func)
from sqlalchemy.orm import relationship
from database import Base


class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True)
    email = Column(String, unique=True, nullable=False)
    password_hash = Column(String, nullable=False)
    created_at = Column(DateTime, server_default=func.now())
    connections = relationship("ExchangeConnection", back_populates="user")


class ExchangeConnection(Base):
    __tablename__ = "exchange_connections"
    id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey("users.id"))
    exchange_name = Column(String, nullable=False)
    api_key_encrypted = Column(String, nullable=False)
    api_secret_encrypted = Column(String, nullable=False)
    label = Column(String)
    created_at = Column(DateTime, server_default=func.now())
    user = relationship("User", back_populates="connections")


class Asset(Base):
    __tablename__ = "assets"
    id = Column(Integer, primary_key=True)
    symbol = Column(String, unique=True, nullable=False)
    name = Column(String)
```

```
class Holding(Base):
    __tablename__ = "holdings"
    id = Column(Integer, primary_key=True)
    connection_id = Column(Integer, ForeignKey("exchange_connections.id"))
    asset_id = Column(Integer, ForeignKey("assets.id"))
    amount = Column(Numeric(30, 10), nullable=False)
    avg_buy_price = Column(Numeric(30, 10))
    updated_at = Column(DateTime, server_default=func.now())

class PortfolioSnapshot(Base):
    __tablename__ = "portfolio_snapshots"
    id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey("users.id"))
    total_value = Column(Numeric(30, 10), nullable=False)
    recorded_at = Column(DateTime, server_default=func.now())
```

Модуль security.py — безпека (гешування, шифрування, JWT)

```
import os
from datetime import datetime, timedelta
from passlib.context import CryptContext
from cryptography.fernet import Fernet
import jwt

# Гешування паролів алгоритмом Argon2id (рекомендація OWASP)
pwd_context = CryptContext(schemes=["argon2"], deprecated="auto")

SECRET_KEY = os.getenv("SECRET_KEY", "change-me-in-production")
FERNET_KEY = os.getenv("FERNET_KEY")          # 32-байтовий ключ (base64)
fernet = Fernet(FERNET_KEY)
ALGORITHM = "HS256"

def hash_password(p: str) -> str:
    return pwd_context.hash(p)

def verify_password(p: str, h: str) -> bool:
    return pwd_context.verify(p, h)

def create_token(user_id: int) -> str:
    payload = {"sub": str(user_id),
               "exp": datetime.utcnow() + timedelta(hours=12)}
    return jwt.encode(payload, SECRET_KEY, algorithm=ALGORITHM)

def decode_token(token: str) -> dict:
    return jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])

# Шифрування API-ключів (AES через механізм Fernet)
def encrypt(text: str) -> str:
    return fernet.encrypt(text.encode()).decode()

def decrypt(token: str) -> str:
    return fernet.decrypt(token.encode()).decode()
```

Модуль schemas.py — схеми валідації даних

```
from pydantic import BaseModel, EmailStr

class UserCreate(BaseModel):
    email: EmailStr
    password: str

class ExchangeCreate(BaseModel):
```

```
    exchange_name: str
    api_key: str
    api_secret: str
    label: str | None = None


class Token(BaseModel):
    access_token: str
    token_type: str = "bearer"
```

Модуль exchanges.py — інтеграція з біржами через CCXT

```
import asyncio
import ccxt.async_support as ccxt


async def fetch_balance(exchange_name, api_key, secret):
    """Отримання балансу з однієї біржі (read-only ключі)."""
    exchange_class = getattr(ccxt, exchange_name)
    client = exchange_class({"apiKey": api_key,
                             "secret": secret,
                             "enableRateLimit": True})
    try:
        balance = await client.fetch_balance()
        return {a: v for a, v in balance["total"].items() if v and v > 0}
    finally:
        await client.close()


async def fetch_all(connections):
    """Паралельне опитування всіх під'єднаних бірж."""
    tasks = [fetch_balance(c["name"], c["key"], c["secret"])
             for c in connections]
    return await asyncio.gather(*tasks, return_exceptions=True)


async def fetch_prices(symbols, quote="USDT"):
    """Отримання поточних котирувань активів."""
    client = ccxt.binance({"enableRateLimit": True})
    prices = {}
    try:
        pairs = [f"{s}/{quote}" for s in symbols]
        tickers = await client.fetch_tickers(pairs)
        for s in symbols:
            pair = f"{s}/{quote}"
            if pair in tickers:
                prices[s] = tickers[pair]["last"]
    finally:
        await client.close()
    return prices
```

Модуль portfolio.py — розрахунок показників портфеля

```
from decimal import Decimal


def calculate(holdings, prices):
    """Розрахунок показників портфеля за формулами (2.1)-(2.5)."""
    positions = []
    total = Decimal("0")
    for h in holdings:
        price = Decimal(str(prices.get(h["symbol"], 0)))
        amount = Decimal(str(h["amount"]))
        avg = Decimal(str(h["avg_buy_price"] or 0))
        value = amount * price                          # (2.1)
        pnl = amount * (price - avg)                    # (2.4)
        roi = ((price - avg) / avg * 100) if avg > 0 else Decimal("0")  # (2.5)
        positions.append({"symbol": h["symbol"],
                          "exchange": h["exchange"],
                          "amount": amount, "price": price,
                          "value": value, "pnl": pnl, "roi": roi})
        total += value                                  # (2.2)
```

```
    for p in positions:                                    # (2.3)
        p["share"] = (p["value"] / total * 100) if total > 0 else Decimal("0")
    total_pnl = sum((p["pnl"] for p in positions), Decimal("0"))
    return {"total": total, "total_pnl": total_pnl, "positions": positions}
```

Модуль main.py — точки доступу REST API

```
from fastapi import FastAPI, Depends, HTTPException, Header
from fastapi.middleware.cors import CORSMiddleware
from sqlalchemy.orm import Session
import database, models, schemas, security, exchanges, portfolio


models.Base.metadata.create_all(bind=database.engine)
app = FastAPI(title="MultiCP API")
app.add_middleware(CORSMiddleware, allow_origins=["*"],
                   allow_methods=["*"], allow_headers=["*"])


def current_user(authorization: str = Header(None),
                 db: Session = Depends(database.get_db)):
    if not authorization or not authorization.startswith("Bearer "):
        raise HTTPException(401, "Не авторизовано")
    try:
        payload = security.decode_token(authorization.split(" ")[1])
        user = db.query(models.User).get(int(payload["sub"]))
    except Exception:
        raise HTTPException(401, "Недійсний токен")
    if not user:
        raise HTTPException(401, "Користувача не знайдено")
    return user


@app.post("/auth/register", response_model=schemas.Token)
def register(data: schemas.UserCreate, db: Session = Depends(database.get_db)):
    if db.query(models.User).filter_by(email=data.email).first():
        raise HTTPException(400, "Email вже зареєстровано")
    user = models.User(email=data.email,
                       password_hash=security.hash_password(data.password))
    db.add(user); db.commit(); db.refresh(user)
    return schemas.Token(access_token=security.create_token(user.id))


@app.post("/auth/login", response_model=schemas.Token)
def login(data: schemas.UserCreate, db: Session = Depends(database.get_db)):
    user = db.query(models.User).filter_by(email=data.email).first()
    if not user or not security.verify_password(data.password,
                                                user.password_hash):
        raise HTTPException(401, "Невірний email або пароль")
    return schemas.Token(access_token=security.create_token(user.id))


@app.post("/exchanges")
def add_exchange(data: schemas.ExchangeCreate, user=Depends(current_user),
                 db: Session = Depends(database.get_db)):
    conn = models.ExchangeConnection(
        user_id=user.id, exchange_name=data.exchange_name,
        api_key_encrypted=security.encrypt(data.api_key),
        api_secret_encrypted=security.encrypt(data.api_secret),
        label=data.label)
    db.add(conn); db.commit()
    return {"status": "ok", "id": conn.id}


@app.get("/exchanges")
def list_exchanges(user=Depends(current_user)):
    return [{"id": c.id, "exchange_name": c.exchange_name,
             "label": c.label} for c in user.connections]


@app.delete("/exchanges/{conn_id}")
def delete_exchange(conn_id: int, user=Depends(current_user),
                    db: Session = Depends(database.get_db)):
```

```
    conn = db.query(models.ExchangeConnection).filter_by(
        id=conn_id, user_id=user.id).first()
    if not conn:
        raise HTTPException(404, "Біржу не знайдено")
    db.delete(conn); db.commit()
    return {"status": "deleted"}


@app.get("/portfolio")
async def get_portfolio(user=Depends(current_user),
                        db: Session = Depends(database.get_db)):
    conns = [{"name": c.exchange_name,
              "key": security.decrypt(c.api_key_encrypted),
              "secret": security.decrypt(c.api_secret_encrypted)}
             for c in user.connections]
    balances = await exchanges.fetch_all(conns)
    holdings = []
    for conn, bal in zip(user.connections, balances):
        if isinstance(bal, Exception):
            continue                          # біржа недоступна - пропускаємо
        for sym, amount in bal.items():
            holdings.append({"symbol": sym, "amount": amount,
                             "avg_buy_price": 0,
                             "exchange": conn.exchange_name})
    symbols = list({h["symbol"] for h in holdings})
    prices = await exchanges.fetch_prices(symbols) if symbols else {}
    result = portfolio.calculate(holdings, prices)
    snap = models.PortfolioSnapshot(user_id=user.id,
                                    total_value=result["total"])
    db.add(snap); db.commit()
    return result
```

Модуль cache.py — кешування котирувань

```
import time


class PriceCache:
    """Кеш котирувань у пам'яті з обмеженим часом життя (TTL)."""
    def __init__(self, ttl_seconds=30):
        self._ttl = ttl_seconds
        self._store = {}            # symbol -> (price, timestamp)


    def get(self, symbol):
        item = self._store.get(symbol)
        if not item:
            return None
        price, ts = item
        if time.time() - ts > self._ttl:
            del self._store[symbol]  # запис застарів
            return None
        return price


    def set(self, symbol, price):
        self._store[symbol] = (price, time.time())


    def get_many(self, symbols):
        result = {}
        for s in symbols:
            value = self.get(s)
            if value is not None:
                result[s] = value
        return result


# Єдиний екземпляр кешу для всього застосунку
price_cache = PriceCache(ttl_seconds=30)
```

Модуль tests/test_portfolio.py — модульні тести розрахунку показників

```
import pytest
from decimal import Decimal
from portfolio import calculate


def _h(symbol, amount, avg):
    return {"symbol": symbol, "amount": amount,
            "avg_buy_price": avg, "exchange": "Test"}


def test_value_and_total():
    # Формули (2.1) та (2.2): вартість позиції та сума портфеля
    holdings = [_h("BTC", 0.5, 50000), _h("ETH", 2, 2000)]
    prices = {"BTC": 60000, "ETH": 3000}
    result = calculate(holdings, prices)
    assert result["positions"][0]["value"] == Decimal("30000")  # 0.5*60000
    assert result["total"] == Decimal("36000")                  # 30000+6000


def test_pnl():
    # Формула (2.4): P&L = amount * (price - avg_buy)
    result = calculate([_h("BTC", 0.5, 50000)], {"BTC": 60000})
    assert result["positions"][0]["pnl"] == Decimal("5000")


def test_shares_sum_to_100():
    # Формула (2.3): сума часток усіх позицій = 100%
    holdings = [_h("BTC", 1, 0), _h("ETH", 10, 0)]
    prices = {"BTC": 60000, "ETH": 3000}
    result = calculate(holdings, prices)
    total_share = sum(p["share"] for p in result["positions"])
    assert round(total_share) == 100


def test_zero_avg_price_no_division_error():
    # Граничний випадок: нульова ціна придбання -> ROI = 0, без помилки
    result = calculate([_h("BTC", 1, 0)], {"BTC": 60000})
    assert result["positions"][0]["roi"] == Decimal("0")


def test_empty_portfolio():
    # Граничний випадок: порожній портфель
    result = calculate([], {})
    assert result["total"] == Decimal("0")
    assert result["positions"] == []
```

Додаток Б

Лістинг коду клієнтської частини (React)

У додатку наведено основні компоненти клієнтської частини системи MultiCP, реалізованої з використанням бібліотеки React.

Модуль api.js — взаємодія із серверним API

```
const API = "http://localhost:8000";

export async function apiPost(path, body, token) {
  const res = await fetch(API + path, {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
      ...(token && { Authorization: `Bearer ${token}` }),
    },
    body: JSON.stringify(body),
  });
  if (!res.ok) {
    const err = await res.json();
    throw new Error(err.detail || "Помилка запиту");
  }
  return res.json();
}

export async function apiGet(path, token) {
  const res = await fetch(API + path, {
    headers: { Authorization: `Bearer ${token}` },
  });
  if (!res.ok) throw new Error("Помилка завантаження");
  return res.json();
}
```

Компонент Login.jsx — екран автентифікації

```
import { useState } from "react";
import { apiPost } from "./api";

export default function Login({ onLogin }) {
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
  const [error, setError] = useState("");

  async function submit() {
    try {
      const { access_token } = await apiPost("/auth/login",
        { email, password });
      onLogin(access_token);
    } catch (e) {
      setError(e.message);
    }
  }

  return (
    <div className="login-card">
      <h1>MultiCP</h1>
      <input value={email} placeholder="Електронна пошта"
        onChange={(e) => setEmail(e.target.value)} />
      <input type="password" value={password} placeholder="Пароль"
        onChange={(e) => setPassword(e.target.value)} />
```

```
      {error && <p className="error">{error}</p>}
      <button onClick={submit}>Увійти</button>
    </div>
  );
}
```

Компонент Dashboard.jsx — головний екран (портфель)

```
import { useEffect, useState } from "react";
import { apiGet } from "./api";

export default function Dashboard({ token }) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    apiGet("/portfolio", token)
      .then(setData)
      .catch(() => setData(null))
      .finally(() => setLoading(false));
  }, [token]);

  if (loading) return <p>Завантаження…</p>;
  if (!data) return <p>Не вдалося завантажити портфель</p>;

  return (
    <div className="dashboard">
      <div className="stat">
        <span>Загальна вартість</span>
        <strong>${Number(data.total).toFixed(2)}</strong>
      </div>
      <table>
        <thead>
          <tr><th>Актив</th><th>Біржа</th><th>Вартість</th>
              <th>P&L</th><th>Частка</th></tr>
        </thead>
        <tbody>
          {data.positions.map((p, i) => (
            <tr key={i}>
              <td>{p.symbol}</td>
              <td>{p.exchange}</td>
              <td>${Number(p.value).toFixed(2)}</td>
              <td className={p.pnl >= 0 ? "pos" : "neg"}>
                ${Number(p.pnl).toFixed(2)}</td>
              <td>{Number(p.share).toFixed(1)}%</td>
            </tr>
          ))}
        </tbody>
      </table>
    </div>
  );
}
```

Компонент App.jsx — кореневий компонент застосунку

```
import { useState } from "react";
import Login from "./Login";
import Dashboard from "./Dashboard";

export default function App() {
  const [token, setToken] = useState(null);
  return token
    ? <Dashboard token={token} />
    : <Login onLogin={setToken} />;
}
```

Компонент StructureChart.jsx — кругова діаграма структури

```
import { useEffect, useRef } from "react";

const COLORS = ["#7c6cf0", "#3ddc97", "#f0b54c", "#5c9cff", "#ff5c7c"];

export default function StructureChart({ positions }) {
  const ref = useRef(null);

  useEffect(() => {
    const ctx = ref.current.getContext("2d");
    const cx = 140, cy = 110, r = 85, rin = 52;
    ctx.clearRect(0, 0, 280, 220);
    let start = -Math.PI / 2;
    const total = positions.reduce((s, p) => s + p.share, 0) || 1;
    positions.forEach((p, i) => {
      const angle = (p.share / total) * Math.PI * 2;
      ctx.beginPath();
      ctx.moveTo(cx, cy);
      ctx.arc(cx, cy, r, start, start + angle);
      ctx.closePath();
      ctx.fillStyle = COLORS[i % COLORS.length];
      ctx.fill();
      start += angle;
    });
    ctx.beginPath();
    ctx.arc(cx, cy, rin, 0, Math.PI * 2);
    ctx.fillStyle = "#1e1e2e";
    ctx.fill();
  }, [positions]);

  return <canvas ref={ref} width={280} height={220} />;
}
```

Компонент PositionsTable.jsx — таблиця активів

```
import { useState } from "react";
import CoinModal from "./CoinModal";

export default function PositionsTable({ coins }) {
  const [selected, setSelected] = useState(null);

  return (
    <>
      <table>
        <thead>
          <tr><th>Актив</th><th>Кількість</th><th>Вартість</th>
              <th>P&L</th><th>Частка</th></tr>
        </thead>
        <tbody>
          {coins.map((c) => (
            <tr key={c.symbol} onClick={() => setSelected(c)}
                style={{ cursor: "pointer" }}>
              <td>{c.symbol}{c.breakdown.length > 1 && " …"}</td>
              <td>{Number(c.amount).toFixed(4)}</td>
              <td>${Number(c.value).toFixed(2)}</td>
              <td className={c.pnl >= 0 ? "pos" : "neg"}>
                ${Number(c.pnl).toFixed(2)}</td>
              <td>{Number(c.share).toFixed(1)}%</td>
            </tr>
          ))}
        </tbody>
      </table>
      {selected && (
        <CoinModal coin={selected} onClose={() => setSelected(null)} />
      )}
```

```
    </>
  );
}
```

Компонент CoinModal.jsx — деталізація активу за біржами

```
export default function CoinModal({ coin, onClose }) {
  return (
    <div className="overlay" onClick={onClose}>
      <div className="modal" onClick={(e) => e.stopPropagation()}>
        <h2>{coin.symbol}</h2>
        <p>Загальна вартість: ${Number(coin.value).toFixed(2)}</p>
        <h3>Розподіл за біржами</h3>
        <table>
          <thead>
            <tr><th>Біржа</th><th>Кількість</th>
                <th>Вартість</th><th>Частка монети</th></tr>
          </thead>
          <tbody>
            {coin.breakdown.map((b, i) => (
              <tr key={i}>
                <td>{b.exchange}</td>
                <td>{Number(b.amount).toFixed(4)}</td>
                <td>${Number(b.value).toFixed(2)}</td>
                <td>{Number(b.shareInCoin).toFixed(1)}%</td>
              </tr>
            ))}
          </tbody>
        </table>
      </div>
    </div>
  );
}
```

Компонент DynamicsChart.jsx — графік динаміки вартості

```
import { useEffect, useRef } from "react";

// Графік динаміки сукупної вартості портфеля (лінійний, на canvas)
export default function DynamicsChart({ series }) {
  const ref = useRef(null);

  useEffect(() => {
    const ctx = ref.current.getContext("2d");
    const W = 760, H = 260, pad = 36;
    ctx.clearRect(0, 0, W, H);
    if (!series.length) return;
    const min = Math.min(...series), max = Math.max(...series);
    const x = (i) => pad + (i * (W - pad * 2)) / (series.length - 1);
    const y = (v) => H - pad - ((v - min) / ((max - min) || 1)) * (H - pad * 2);
    ctx.beginPath();
    ctx.moveTo(x(0), y(series[0]));
    series.forEach((v, i) => ctx.lineTo(x(i), y(v)));
    ctx.strokeStyle = "#7c6cf0";
    ctx.lineWidth = 2.4;
    ctx.stroke();
  }, [series]);

  return <canvas ref={ref} width={760} height={260} />;
}
```

Компонент ExchangesManager.jsx — екран керування біржами

```
import { useState, useEffect } from "react";
import { apiGet, apiPost } from "./api";

// Екран керування під'єднаними біржами
```

```
export default function ExchangesManager({ token }) {
  const [exchanges, setExchanges] = useState([]);
  const [name, setName] = useState("Binance");
  const [apiKey, setApiKey] = useState("");
  const [apiSecret, setApiSecret] = useState("");


  useEffect(() => {
    apiGet("/exchanges", token).then(setExchanges).catch(() => {});
  }, [token]);


  async function add() {
    await apiPost("/exchanges",
      { exchange_name: name, api_key: apiKey, api_secret: apiSecret }, token);
    const updated = await apiGet("/exchanges", token);
    setExchanges(updated);
    setApiKey(""); setApiSecret("");
  }


  return (
    <div className="exchanges">
      <h2>Під'єднані біржі</h2>
      <ul>
        {exchanges.map((e) => (
          <li key={e.id}>{e.exchange_name}</li>
        ))}
      </ul>
      <h3>Додати біржу (read-only ключі)</h3>
      <select value={name} onChange={(e) => setName(e.target.value)}>
        <option>Binance</option><option>Kraken</option><option>Coinbase</option>
      </select>
      <input value={apiKey} placeholder="API Key"
        onChange={(e) => setApiKey(e.target.value)} />
      <input type="password" value={apiSecret} placeholder="API Secret"
        onChange={(e) => setApiSecret(e.target.value)} />
      <button onClick={add}>Додати</button>
    </div>
  );
}
```

Додаток В

Схема бази даних та конфігурація залежностей

У додатку наведено SQL-схему бази даних системи MultiCP для системи керування базами даних PostgreSQL та перелік залежностей серверної частини.

Схема бази даних (schema.sql)

```
-- Схема бази даних системи MultiCP (PostgreSQL)

CREATE TABLE users (
    id            SERIAL PRIMARY KEY,
    email         VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    created_at    TIMESTAMP DEFAULT now()
);

CREATE TABLE exchange_connections (
    id                   SERIAL PRIMARY KEY,
    user_id              INTEGER NOT NULL
                         REFERENCES users(id) ON DELETE CASCADE,
    exchange_name        VARCHAR(64) NOT NULL,
    api_key_encrypted    TEXT NOT NULL,
    api_secret_encrypted TEXT NOT NULL,
    label                VARCHAR(128),
    created_at           TIMESTAMP DEFAULT now()
);

CREATE TABLE assets (
    id     SERIAL PRIMARY KEY,
    symbol VARCHAR(32) UNIQUE NOT NULL,
    name   VARCHAR(128)
);

CREATE TABLE holdings (
    id            SERIAL PRIMARY KEY,
    connection_id INTEGER NOT NULL
                  REFERENCES exchange_connections(id) ON DELETE CASCADE,
    asset_id      INTEGER NOT NULL REFERENCES assets(id),
    amount        NUMERIC(30,10) NOT NULL,
    avg_buy_price NUMERIC(30,10),
    updated_at    TIMESTAMP DEFAULT now()
);

CREATE TABLE portfolio_snapshots (
    id          SERIAL PRIMARY KEY,
    user_id     INTEGER NOT NULL
                REFERENCES users(id) ON DELETE CASCADE,
    total_value NUMERIC(30,10) NOT NULL,
    recorded_at TIMESTAMP DEFAULT now()
);

-- Індекси для пришвидшення вибірок за зовнішніми ключами
CREATE INDEX idx_conn_user     ON exchange_connections(user_id);
CREATE INDEX idx_holdings_conn ON holdings(connection_id);
CREATE INDEX idx_snap_user     ON portfolio_snapshots(user_id);
```

Перелік залежностей (requirements.txt)

```
# Залежності серверної частини (Python)
fastapi==0.111.0
uvicorn==0.30.0
sqlalchemy==2.0.30
psycopg2-binary==2.9.9
pydantic[email]==2.7.0
passlib[argon2]==1.7.4
cryptography==42.0.0
pyjwt==2.8.0
ccxt==4.3.0
```