

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І
ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ**

Факультет інформаційних технологій

ПОГОДЖЕНО
Декан факультету

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ
Завідувач кафедри інформаційних
систем і технологій

(підпис) **Ігор БОЛБОТ**

(підпис) **Михайло ШВИДЕНКО**

«__» _____ 2026 р.

«__» _____ 2026 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

**на тему: «Веборієнтована інформаційна система біржі як багатосторонньої
цифрової платформи для взаємодії користувачів»**

Спеціальність «Комп'ютерні науки»

Освітньо-професійна програма «Комп'ютерні науки»

Гарант освітньої програми
Професор, доктор економічних наук

Роман РУДЕНСЬКИЙ

**Керівник бакалаврської
кваліфікаційної роботи**
Доцент, доктор філософії

Валентина КОРОЛЬЧУК

Виконав

Олександр ХАРЧЕНКО

Київ 2026

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ**

Факультет інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри інформаційних систем і
технологій

к.е.н., доцент _____ Михайло ШВИДЕНКО
(підпис)

« ____ » _____ 2026 р.

ЗАВДАННЯ
ДО ВИКОНАННЯ БАКАЛАВРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ
ЗДОБУВАЧУ

Харченко Олександр Сергійовичу

Спеціальність «122 Комп'ютерні науки»

Освітньо-професійна програма «Комп'ютерні науки»

Тема бакалаврської кваліфікаційної роботи

«Веборієнтована інформаційна система біржі як багатосторонньої цифрової платформи для взаємодії користувачів»

затверджена наказом від «06» січня 2026 р. № 46 «С»

Термін подання завершеної роботи на кафедру « ____ » _____ 2026 р.

Вихідні дані до бакалаврської кваліфікаційної роботи: інформація про користувачів, біржові пропозиції та укладені угоди, результати пошуку і фільтрації даних, аналітичні звіти, статистика активності користувачів, а також візуалізовані результати моніторингу.

Перелік питань, що підлягають дослідженню:

Огляд багатосторонніх цифрових платформ та існуючих біржових систем.
Формулювання функціональних, нефункціональних та безпекових вимог;
обґрунтування технологічного стеку. Проектування архітектури системи.
Реалізація ключових модулів. Забезпечення безпеки. Тестування системи.
Економіко-експлуатаційне обґрунтування.

Дата видачі завдання «18» січня 2026 р.

**Керівник бакалаврської кваліфікаційної
роботи**

Валентина КОРОЛЬЧУК

Завдання взяв до виконання

Олександр ХАРЧЕНКО

ЗМІСТ

ЗМІСТ.....	
ВСТУП.....	
РОЗДІЛ 1 ТЕОРЕТИЧНІ ЗАСАДИ ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ.....	
1.1. Концепція багатосторонніх цифрових платформ у сучасній економіці та їх роль у цифровізації ринків віртуальних активів.....	
1.2. Аналіз існуючих рішень: біржі, маркетплейси та P2P-майданчики для обміну віртуальними товарами, акаунтами та цифровими послугами.....	
1.3. Вимоги до веборієнтованої інформаційної системи біржового типу: функціональні, нефункціональні, безпекові та нормативно-правові аспекти.....	
1.4. Обґрунтування вибору технологічного стеку та методології розробки.....	
РОЗДІЛ 2 ПРОЕКТУВАННЯ АРХІТЕКТУРИ ТА ФУНКЦІОНАЛУ СИСТЕМИ.....	
2.1. Концептуальна та логічна архітектура платформи: модель взаємодії учасників, API-шар, структура застосунку.....	
2.2. Проектування бази даних та інформаційних моделей: сутності, зв'язки, індекси.....	
2.3. Розробка ключових модулів системи: автентифікація, каталог активів з двома моделями ціноутворення, торговий модуль, управління інвентарем, портфель, новини та сповіщення.....	
2.4. Забезпечення безпеки та довіри на платформі.....	
РОЗДІЛ 3 РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ ПЛАТФОРМИ.....	
3.1. Програмна реалізація системи: інтерфейс, серверна логіка, інтеграції.....	
3.2. Тестування працездатності та навантажувальне дослідження.....	
3.3. Економіко-експлуатаційне обґрунтування: модель монетизації, собівартість, рентабельність.....	
ВИСНОВКИ.....	

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	
ДОДАТКИ.....	
Додаток А.....	
Додаток Б.....	
Додаток В.....	
Додаток И.....	
Додаток Е.....	

ВСТУП

Під час вибору теми кваліфікаційної роботи мене зацікавило, як насправді влаштовані біржі та торгові майданчики зсередини. Більшість сучасних трейдерів щодня користуються онлайн-платформами, проте мало хто уявляє, скільки систем працює "під капотом": черги ордерів, механізми зіставлення, real-time трансляція цін, перевірка балансу. Тож виникло бажання не просто описати тему на папері, а побудувати робочу модель, яку можна запустити. У сучасній термінології подібні системи називають багатосторонніми цифровими платформами (MDP), де в одному середовищі взаємодіють різні категорії користувачів з різними ролями та інтересами [1]. Ринок таких платформ зростає швидко: за даними [2], обсяг електронної торгівлі цінними паперами у 2023 році перевищив 30 трлн доларів США, а кількість роздрібних трейдерів у світі збільшується приблизно на 18% щороку [3]. За цією статистикою стоїть тренд, помітний і в Україні, що видно хоча б з популярності криптовалютних бірж серед молодих користувачів.

Огляд готових рішень показав їхні слабкі місця. Великі біржові платформи (Interactive Brokers, Binance, eToro) мають потужний функціонал, але водночас закриті, дорогі та орієнтовані на великих гравців. Open-source альтернативи на кшталт OpenCBDC чи академічні проєкти типу CSE-Exchange теж існують, проте часто реалізовані як набір розрізнених скриптів без цілісного інтерфейсу користувача. Дослідження [4] підтверджує таке спостереження: для малого та середнього бізнесу, а тим паче для навчальних цілей, більшість доступних рішень або складні в розгортанні, або не забезпечують необхідного рівня захисту коштів. Саме це стало поштовхом до розробки власної платформи з ясною та задокументованою архітектурою. Окрім суто технічного інтересу, робота має й практичний бік: в Україні діє стратегія цифрової трансформації [5], яка

передбачає розвиток вітчизняної фінтех-інфраструктури, а багато іноземних платформ для українських користувачів обмежені або взагалі недоступні, особливо після 2022 року. Власні розробки, нехай і навчального рівня, частково знімають цю залежність та формують базу знань, на якій можна будувати комерційні рішення.

Мета роботи з огляду на зазначені проблеми полягає у проектуванні та реалізації веборієнтованої інформаційної системи ExchangeUA, що моделює роботу багатосторонньої цифрової платформи біржового типу. У межах однієї системи мають співіснувати трейдери (звичайні користувачі, які купують і продають), емітенти (відповідальні за котирування "своїх" активів) та адміністратори (які керують ролями та призначеннями). Окремою метою став реальний механізм зіставлення ордерів (matching engine) замість демонстраційної імітації, щоб ціни змінювалися від дій користувачів, а не випадково.

Для досягнення поставленої мети визначено такі завдання: дослідити принципи функціонування багатосторонніх цифрових платформ та виконати огляд існуючих біржових систем, як комерційних, так і open-source, щоб з'ясувати, які підходи себе зарекомендували, а які стали джерелом типових проблем; на основі цього аналізу сформулювати функціональні, нефункціональні та безпекові вимоги до системи й обґрунтувати вибір технологічного стеку (зокрема, чому обрано Node.js, а не Python чи Go; MySQL, а не PostgreSQL; рішення без React); спроектувати архітектуру системи, як логічну, так і фізичну, з розробкою структури бази даних (14 таблиць у третій нормальній формі), розподілу функцій по контролерах і API-маршрутах, схеми взаємодії між клієнтом і сервером; реалізувати програмний код модуля автентифікації з JWT та bcrypt, торгового модуля зі стаканом і matching engine, управління балансом, портфеля з підрахунком прибутків і збитків, системи сповіщень та адміністративної панелі; впровадити багаторівневу безпеку (bcrypt для паролів з cost-factor 12, HMAC-SHA256 для JWT, рольову модель доступу RBAC, параметризовані SQL-запити для захисту від ін'єкцій, екранування

виводу для запобігання XSS); перевірити роботу системи в ручному та автоматизованому режимах через вбудований Node.js Test Runner без використання зовнішніх фреймворків; виконати техніко-економічне обґрунтування з оцінкою витрат, очікуваних надходжень від комісій та строку окупності.

Об'єктом дослідження є процеси організації електронної торгівлі фінансовими та цифровими активами в межах багатосторонніх онлайн-платформ. Предмет дослідження становлять архітектурні рішення, алгоритми зіставлення угод та програмна реалізація веборієнтованої біржової інформаційної системи на стеку Node.js + Express + MySQL + WebSocket. У роботі застосовано системний аналіз для опрацювання предметної області, порівняльний аналіз під час огляду аналогів, реляційне моделювання при проєктуванні бази даних, метод прототипування з поетапним нарощуванням функціоналу, автоматизоване тестування через вбудований Node.js Test Runner, а також техніко-економічний аналіз для розрахунку окупності.

Серверну частину побудовано на платформі Node.js з використанням фреймворку Express для маршрутизації та обробки HTTP-запитів. Для зберігання даних обрано реляційну СУБД MySQL з підтримкою транзакцій InnoDB. Двосторонній обмін даними в реальному часі реалізовано через протокол WebSocket. Безпека забезпечується бібліотеками bcrypt (хешування паролів) та jsonwebtoken (видача й перевірка JWT-токенів). Клієнтська частина виконана на чистому HTML5, CSS3 та JavaScript без використання SPA-фреймворків, що зменшує час завантаження сторінок і спрощує підтримку коду. Для проєктування бази даних застосовано MySQL Workbench, для контролю версій – Git.

Наукова новизна полягає в поєднанні в межах однієї монолітної архітектури кількох механізмів, які зазвичай реалізують окремо: ринкового й лімітного режимів виконання ордерів з матчингом, атомарних транзакцій MySQL, моделі автоматичного впливу обсягу торгівлі на ціну (price impact) та RBAC-моделі безпеки з диференціацією прав між адміністратором, емітентом і трейдером. Окремої уваги заслуговує відмова від типової для навчальних робіт

"симуляції цін" на користь повноцінного процесу котирування з аудитом усіх змін. Розроблена система може використовуватись у трьох сценаріях: як навчальний майданчик для підготовки трейдерів, де ризик грошових втрат відсутній; як готова основа для фінтех-стартапу з власною документацією, тестами та інструкцією з розгортання; як ядро внутрішньої корпоративної біржі для торгівлі цифровими товарами всередині компанії.

Робота складається зі вступу, трьох основних розділів, висновків до кожного з них, загальних висновків та списку використаних джерел (45 позицій). Перший розділ присвячено аналізу предметної області: розглянуто принципи функціонування багатосторонніх цифрових платформ, проведено огляд існуючих біржових систем, сформульовано вимоги до власної розробки. Другий розділ містить опис архітектури системи, проєктування бази даних, обґрунтування технологічних рішень та схему взаємодії компонентів. У третьому розділі описано програмну реалізацію ключових модулів, результати тестування, заходи інформаційної безпеки та техніко-економічне обґрунтування. У додатках наведено код ключових модулів, перелік API-ендпоінтів, покрокову інструкцію з розгортання системи а також рисунки. Загальний обсяг основного тексту – 71 сторінка.

РОЗДІЛ 1 ТЕОРЕТИЧНІ ЗАСАДИ ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Концепція багатосторонніх цифрових платформ у сучасній економіці та їх роль у цифровізації ринків віртуальних активів

Теорія багатосторонніх ринків – порівняно молодий напрям в економіці. Її основи заклали Роше і Тіроль [6], а пізніше суттєвий внесок зробили Паркер і ван Елстайн [7]. Головна теза проста: коли платформа з'єднує кілька різних груп користувачів, цінність для кожної з них залежить від того, скільки учасників на інших боках. Чим більше продавців, тим цікавіше покупцям, і навпаки. Це і називають мережевим ефектом.

Варто наголосити на одному важливому термінологічному моменті. У класичній роботі Роше і Тіроля [6] поняття "двостороннього" (two-sided) і "багатостороннього" (multi-sided) ринку трактуються як один клас платформ із спільними економічними властивостями: мережевими ефектами, диференційованим ціноутворенням для різних сторін, ефектом критичної маси. Тобто, наявність двох або більше категорій користувачів з різними ролями вже є достатньою умовою для класифікації платформи як багатосторонньої. Хрестоматійні приклади – VISA (картки і продавці), eBay (продавці і покупці), YouTube (творці контенту і глядачі) – формально є двосторонніми, але в літературі їх беззаперечно зараховують до категорії multi-sided platforms. У моїй системі реалізовано три основні категорії учасників: трейдери (купують і продають активи), емітенти (заводять активи на платформу, котирують ціни, керують ринковою пропозицією) та адміністратори (керують ролями та призначеннями). Окремо існує механізм тимчасового заморожування акаунтів (статус frozen) – це дозволяє адміністратору обмежити дії конкретного користувача без зміни його ролі. Кожна категорія має власний набір прав, інтересів і вносить власну цінність до екосистеми: емітенти забезпечують пропозицію активів, трейдери – попит і ліквідність, адміністратори –

інфраструктурну стабільність. Така диференціація ролей повністю відповідає критеріям багатосторонньої платформи в розумінні класичної теорії.

У контексті взаємодії слід окремо звернути увагу на те, що на біржових платформах ця взаємодія відрізняється від соціальних мереж чи маркетплейсів. Тут немає прямого спілкування між учасниками, відгуків чи рейтингів – і це не обмеження, а особливість фінансової галузі. Взаємодія опосередкована через об'єктивні цінові сигнали, книгу ордерів і matching engine. Коли один трейдер виставляє ордер на купівлю, а інший – на продаж за тією ж ціною, система автоматично зводить їх в угоду. Це і є основна форма user-to-user взаємодії на біржах. Додатково реалізована опосередкована взаємодія між емітентами і трейдерами – через котирування цін, управління пропозицією та публікацію аналітичних новин про власні активи. Така модель відповідає реальній практиці провідних бірж (NYSE, NASDAQ, Binance), де анонімна ораque взаємодія через накази є нормою, обумовленою вимогами KYC та правилами анонімності торгів.

Біржі – хрестоматійний приклад такої системи. З одного боку є ті, хто хоче продати актив, з іншого – ті, хто шукає, що купити. Чим більше учасників з обох боків, тим вузчий спред і тим краща виконуваність ордерів. Сама платформа, по суті, виступає інфраструктурою для зустрічі покупця з продавцем і знижує витрати кожного з них на пошук контрагента [8].

Перехід традиційних ринків у цифрову форму відбувається не одночасно, а кількома паралельними напрямками. Якщо узагальнити роботу [9], їх можна звести до п'яти основних:

Таблиця 1.1. Вектори цифровізації фінансових ринків

Вектор цифровізації	Прояв на біржових платформах	Технологічний базис
Автоматизація торгівлі	Алгоритмічний трейдинг, HFT	REST API, WebSocket, FIX-протокол
Децентралізація	DEX, DeFi-протоколи	Blockchain, смарт-контракти
продовження табл. 1.1.		
Вектор цифровізації	Прояв на біржових платформах	Технологічний базис

Мобілізація	Торгівля зі смартфону	Progressive Web Apps, нативні застосунки
Персоналізація	AI-рекомендації, адаптивний UI	Machine Learning, A/B-тестування
Відкритість	Open banking, публічні API	REST, GraphQL, OAuth 2.0

За способом організації торгів цифрові платформи можна поділити на кілька типів. Найпоширеніші – централізовані біржі (CEX). До них належать NYSE, NASDAQ, Binance. Вони підтримують класичну книгу ордерів, забезпечують кліринг та беруть на себе юридичну відповідальність за виконання угод [10]. Для користувача це зручно: інтерфейс простий, ліквідність висока, кошти захищені правилами регулятора.

Альтернативний підхід – децентралізовані біржі (DEX) на кшталт Uniswap чи dYdX. Тут торгівля відбувається через смарт-контракти, без проміжного оператора, який зберігав би активи користувачів. Це підвищує безпеку від експлуатації біржі, але водночас ускладнює онбординг і знижує ліквідність [11]. Гібридні рішення намагаються поєднати краще з обох світів – централізовану взаємодію з користувачем і блокчейн на розрахунках.

Окремий клас – маркетплейси цифрових товарів: Steam Market, G2G, PlayerAuctions. У них торгують не фінансовими активами, а скоріше ігровими предметами, акаунтами чи підписками. Заробляє платформа на комісіях з угод, а довіру між незнайомими учасниками забезпечує через ескроу-послуги [12].

Цікаво, що в економічній літературі такі системи часто розглядають у ширшому контексті платформної економіки [13]. На відміну від традиційної моделі, де цінність нарощується по виробничому ланцюгу, тут цінність народжується у момент взаємодії. Для біржі це означає, що головними активами стають три речі: ліквідність (скільки і яких ордерів у книзі), довіра (хто кому і чому вірить) та сама технологічна інфраструктура – те, наскільки швидко й надійно виконуються розрахунки.

Найкраще мережевий ефект на біржі видно за глибиною книги ордерів. Чим більше людей виставляють заявки, тим менший спред між найвищою ціною купівлі та найнижчою ціною продажу. Тонкий спред притягує нових учасників –

і цикл повторюється. Саме тому новій біржі так складно стартувати: вона повинна якось набрати критичну масу ліквідності, маючи проти себе вже зрілих гравців [14].

Окремо варто торкнутися регулювання. У ЄС діє MiFID II, у США – правила SEC, в Україні – нормативи НБУ та НКЦПФР. Усі вони так чи інакше зобов'язують біржі ідентифікувати клієнтів (KYC/AML), захищати персональні дані (а в межах ЄС – ще й дотримуватися GDPR) та звітувати про підозрілі операції [15]. Для розробника платформи це означає, що питання audit log і disposal персональних даних треба закладати ще на етапі архітектури.

Ринок віртуальних активів давно перестав бути нішевим. На середину 2024 року сукупна капіталізація криптоактивів коливалася навколо 2,5 трлн доларів [16], а сумарний річний обсяг торгів на топ-15 централізованих біржах перевищив 18 трлн доларів – це більш ніж удвічі вищий рівень, ніж у 2023 році. У пікові періоди (наприклад, у грудні 2024-го, коли BTC перетнув позначку 100 000 доларів) денний обсяг торгів на топ-біржах сягав 190 млрд доларів за добу. Це вже сформований клас фінансових інструментів, який має свою інфраструктуру і свою категорію регулювання.

Підсумовуючи цей огляд, біржова платформа – це не просто веб-застосунок, а складна соціотехнічна система, де переплелися економіка ціноутворення, технологія виконання угод та юридичні зобов'язання. Перш ніж пропонувати власну архітектуру, варто подивитися, як аналогічні задачі вирішують у наявних рішеннях – цьому присвячено наступний підрозділ.

1.2. Аналіз існуючих рішень: біржі, маркетплейси та P2P-майданчики для обміну віртуальними товарами, акаунтами та цифровими послугами

Для аналізу обрано систематичний огляд аналогів за методологією, описаною в [17]. Критеріїв порівняння було чотири: функціональність,

архітектурні рішення, рівень безпеки та модель монетизації. До огляду потрапили шість платформ – кожна представляє свій окремий підклас рішень, від великих криптобірж до нішевих P2P-маркетплейсів.

Binance. Найбільша у світі централізована криптобіржа за обсягом торгів [18]. Архітектура – мікросервісна, написана переважно на Java і Go. Власний *matching engine* здатен пропускати до 1,4 млн ордерів за секунду, ціни транслуються через *WebSocket*. Підтримується весь типовий набір ордерів – ринкові, лімітні, стоп-лімітні, OCO. Слабкі місця: вся модель кастодіальна (приватні ключі зберігає сама біржа), інтерфейс перевантажений для новачків, для частини країн є географічні обмеження.

Interactive Brokers. Американський брокер з виходом на понад 150 ринків у 33 країнах [19]. Має свій десктопний термінал TWS (*Trader Workstation*) з потужною аналітикою. API доступне через REST і FIX-протокол. Сильна сторона – глибока інтеграція з реальними ринками цінних паперів, не лише з криптою. Втім, для роздрібного трейдера платформа дуже громіздка: мінімальний депозит, складна реєстрація, інтерфейс розрахований на досвідченого користувача.

Steam Community Market. Внутрішня площадка Steam для торгівлі ігровими предметами між гравцями [20]. Це класичний приклад замкнутої екосистеми: гроші, зароблені на Steam, зі Steam і не виходять. Можна продати скін зі CS:GO, але вивести виручку через банк – у більшості випадків ні. Користувачі виставляють лоти за фіксованою ціною або через аукціон. Платформа корисна для розуміння того, як працює модель “закритого гаманця” та яким повинен бути API маркетплейсу всередині великої екосистеми.

G2G. P2P-маркетплейс, вузько спеціалізований на ігровій валюті, акаунтах та супутніх послугах [21]. Цікава деталь – ескроу: гроші покупця заморожуються до моменту, коли він підтвердить, що отримав те, за що заплатив. Між учасниками працює репутаційна система з відгуками. Мінуси – досить висока комісія (від 4,9%) та ризики при торгівлі чисто нематеріальними послугами (бустинг, прокачка), де довести “невиконання” складніше.

LocalBitcoins. P2P-сервіс для купівлі та продажу Bitcoin за готівку чи банківські перекази, закrywся у 2023 році [22]. У свій час був піонером у застосуванні ескроу та репутаційної системи саме для P2P-угод. Сам факт закриття показав, наскільки уразливими залишаються централізовані P2P-сервіси, коли посилюється регуляторний тиск з боку фінмоніторингу.

OpenSea та подібні NFT-маркетплейси заслуговують окремої уваги [23]. Вони продемонстрували, що поняття “цифрового активу” можна розширити на унікальні неподільні об’єкти. З технічної точки зору, цікаві саме рішення для відображення метаданих активу, аукціонного ціноутворення та верифікації автентичності – це знадобиться будь-якій платформі, що працює з нестандартними типами активів.

Таблиця 1.2. Порівняльний аналіз існуючих платформ

Критерій	Binance	Interactive Brokers	Steam Market	G2G	ExchangeUA (проект)
Типи активів	Крипто	Акції, ETF	Ігрові предмети	Ігр. акаунти	Акції, крипто, форекс, товари
Matching engine	Власний	Власний	Відсутній	Відсутній	Вбудований
Real-time ціни	WebSocket	WebSocket + FIX	Так	Ні	WebSocket
Ескроу	Ні	Кліринг	Steam Wallet	Так	Транзакційний
Open Source	Ні	Ні	Ні	Ні	Так (MIT)
Вартість входу	Безкоштовно	Від \$10 000	Безкоштовно	Безкоштовно	Безкоштовно
Комісія	0.1%	0.005\$/акція	15%	4.9%	0.1%

Проведений огляд показав, що більшість існуючих комерційних платформ є закритими системами, адаптація яких потребує значних фінансових витрат та обмежується умовами ліцензування. Водночас доступні open-source рішення часто є застарілими або потребують складного налаштування та доопрацювання для практичного використання. Крім того, у наявних системах недостатньо реалізовано комплексну підтримку багатосторонньої взаємодії користувачів,

аналітичних інструментів, засобів моніторингу та сучасного веб-інтерфейсу в межах єдиної платформи [24]. Саме усунення зазначених недоліків і є основною метою даного дипломного проєкту.

Підсумовуючи аналіз, можу виділити перелік критичних вимог, які виявилися універсальними для всіх розглянутих систем: атомарність фінансових транзакцій (інакше неминучі ситуації подвійного витрачання), гнучка модель ролей і прав, ціни в реальному часі, інтерфейс, зрозумілий і трейдеру-початківцю, і досвідченому користувачу. Ці пункти стали відправною точкою для формулювання вимог до власної системи у наступному підрозділі.

1.3. Вимоги до веборієнтованої інформаційної системи біржового типу: функціональні, нефункціональні, безпекові та нормативно-правові аспекти

1.3.1. Функціональні вимоги

Вимоги до системи я формував, спираючись на метод варіантів використання (Use Case Analysis) [25] та стандарт IEEE 830 для специфікації програмних вимог. Усе те, що отримав на виході, природно розділилося на чотири логічні групи: функціональні, нефункціональні, безпекові та нормативно-правові.

Передбачено три основні ролі учасників: trader (повний торговий функціонал – основна роль), issuer (доступ до інструментів котирування на призначених активах та керування ринковою пропозицією), admin (керування платформою). Для тимчасового обмеження дій конкретного користувача використовується не окрема роль, а статус frozen, що дозволяє адміну заморозити акаунт без зміни ролі – користувач продовжує бачити свої дані, але не може виконувати активних дій. Кожна роль бачить тільки ті функції, на які має право – це класична реалізація принципу найменших привілеїв [26].

Таблиця 1.3. Функціональні вимоги до платформи

Функціональна група	Вимога	Пріоритет
---------------------	--------	-----------

Автентифікація	Реєстрація з валідацією, вхід, вихід, оновлення профілю	Критичний
Ринок	Перегляд активів у реальному часі, пошук, фільтрація	Критичний
Торгівля	Чотири типи ордерів (ринкові, лімітні, стоп, стоп-лімітні), скасування, стакан	Критичний
Портфель	Позиції, P&L, середня ціна	Високий
Гаманець	Поповнення, виведення, повна історія	Високий
Новини	Стрічка новин з категоріями (market, company, crypto, forex, commodity, economy)	Середній
Сповіщення	Внутрішні сповіщення (notifications) про виконання ордерів, поллінг кожні 30 сек	Середній
Адміністрування	Управління користувачами, статистика платформи	Критичний

1.3.2. Нефункціональні вимоги

Нефункціональні вимоги визначають якісні характеристики системи та описують не лише її функціональні можливості, а й ефективність, надійність і зручність використання [27]. Для розроблюваної веборієнтованої інформаційної системи було сформовано такі нефункціональні вимоги:

- Продуктивність – забезпечення часу відповіді API не більше 200 мс для 95 % запитів при одночасній роботі щонайменше 100 користувачів;
- Доступність – забезпечення безперервної роботи системи з показником uptime не менше 99,5 %, що передбачає мінімальний допустимий простій;
- Масштабованість – підтримка горизонтального масштабування для можливості збільшення кількості користувачів та обсягів даних без втрати продуктивності;
- Надійність – гарантування цілісності даних і коректного виконання транзакцій, включаючи захист від дублювання операцій та втрати інформації;

- Зручність використання – реалізація сучасного SPA-інтерфейсу із часом початкового завантаження до 2 секунд та інтуїтивно зрозумілою навігацією;
- Сумісність – коректна робота системи у сучасних веббраузерах, зокрема в останніх версіях Google Chrome, Mozilla Firefox та Microsoft Edge;

1.3.3. Вимоги безпеки

Безпекові вимоги формував із прицілом на стандарт OWASP Top 10 [28] та рекомендації NIST щодо ідентифікації й аутентифікації [29]. Конкретно:

- хешування паролів bcrypt із коефіцієнтом складності не менше 10;
- захист від SQL-ін'єкцій через параметризовані запити;
- CSRF-захист через httpOnly session cookies;
- налаштування HTTP-заголовків безпеки через Helmet.js (X-Frame-Options, X-Content-Type-Options, HSTS; CSP вимкнено через специфіку SPA з inline-обробниками і винесено у напрямки подальшого розвитку);
- JWT-токени з коротким TTL (1 година) та обов'язковою перевіркою підпису;
- логування всіх фінансових операцій для аудиту.

1.3.4. Нормативно-правові вимоги

У правовому полі України платформи для торгівлі фінансовими інструментами регулюються передусім Законом "Про ринки капіталу та організовані товарні ринки" [30] і нормативами НКЦПФР, які регламентують ліцензування. Зрозуміло, що для дипломної навчальної системи, де реальні гроші не задіяні, більшість цих вимог опціональна. А ось вимоги до захисту персональних даних (відповідний профільний закон [31]) актуальні в будь-якому разі – це базовий мінімум.

1.4. Обґрунтування вибору технологічного стеку та методології розробки

Вибір технологій – одне з тих рішень, які важко переглянути на пізніх етапах. Воно багато в чому визначає і продуктивність, і зручність підтримки, і можливості масштабування [32]. Я підходив до цього питання послідовно: на кожну з технологій складав короткий список альтернатив, проводив порівняння за критеріями з попереднього підрозділу і вже після цього обирав найкращий, як на мене, варіант.

Серверна частина: Node.js + Express.js. Серверним середовищем було обрано Node.js. Причин кілька. По-перше, Event Loop-архітектура добре пасує до сценарію з великою кількістю довготривалих з'єднань – а саме така ситуація на біржі, де WebSocket-канали тримаються одночасно для всіх активних клієнтів [33]. По-друге, єдина мова – JavaScript – і на сервері, і в браузері. Це спрощує обмін кодом і прискорює перемикання контексту під час розробки. По-третє, npm-екосистема – мабуть, найбільша зі всіх. Готову бібліотеку можна знайти майже для будь-якої рутинної задачі.

Веб-фреймворком став Express.js. У ньому мені сподобалося, що він мінімалістичний (так звана "unopinionated" модель – фреймворк не нав'язує своїх правил), гнучко працює з middleware і має максимально велику спільноту [34]. На альтернативи я теж дивився – Fastify працює помітно швидше, Коа виглядає елегантніше – але для дипломної роботи кількість прикладів і готових туторіалів в Express переважила.

Базу даних довго обирав між MySQL і PostgreSQL. Зупинився на MySQL, ось чому. Будь-яка фінансова операція має бути атомарною – або вона виконалася повністю, або не виконалася взагалі. ACID-гарантії дає InnoDB, основний рушій MySQL [35]. Сутності, з якими я працюю – користувачі, ордери, угоди, транзакції – добре кладуться в реляційну модель. У MySQL 8.0 з'явилися JSON-колони, що зручно для метаданих сповіщень. PostgreSQL – теж чудовий варіант, особливо в плані типів даних і повнотекстового пошуку [36], але для

мого випадку MySQL виявилася простішою в розгортанні і кращою з готовими прикладами саме для біржової логіки.

Real-time трансляція цін. Тут було три кандидати: polling (періодичні HTTP-запити), Server-Sent Events і WebSocket. Polling неефективний – створює зайвий трафік і затримки. SSE – добре, але односторонній. WebSocket – повноцінний двосторонній канал з мінімальними накладними витратами [37], і саме він став моїм вибором. Серед бібліотек ws – найшвидша нативна реалізація для Node.js, що підтверджується незалежними бенчмарками [38].

Окремо варто сказати про аутентифікацію. Я зробив гібридну схему: сесійні HTTP Only-куки – для звичайних користувачів через браузер, JWT з Bearer-токенами – для API-клієнтів та інтеграцій. Це дає CSRF-захист "з коробки" і водночас гнучкість для зовнішніх клієнтів. Бібліотеку для JWT не використовував – реалізував свою через crypto.createHmac. Це 30 рядків коду, повністю прозоро, і одна зайва залежність уже не зможе додати в проєкт нових вразливостей [39].

Що стосується клієнтської частини – тут я свідомо відмовився від React, Vue, Angular на користь чистого ванільного JavaScript. Аргументи: менший розмір зібраного клієнтського файлу, відсутність зовнішніх залежностей (а отже – менша поверхня для атак), простіше розгортання та необхідність глибше розібратися в самій платформі веб [40]. Це водночас і обмеження, і виклик – мусиш сам писати маршрутизатор і керування станом. SPA-підхід я при цьому зберіг – навігація між сторінками відбувається без перезавантаження.

Таблиця 1.4. Обґрунтування технологічного стеку

Технологія	Альтернатива	Причина вибору
Node.js	Python/Django, Java/Spring	Event Loop для WebSocket, єдина мова стеку
Express.js	Fastify, Koa	Зрілість, документація, middleware-екосистема
MySQL 8.0	PostgreSQL, MongoDB	ACID-транзакції, реляційна модель, простота
ws (WebSocket)	Socket.io, SSE	Мінімальні накладні витрати, нативний протокол
Vanilla JS	React, Vue, Angular	Нульовий overhead, простота розгортання
bcryptjs	argon2, scrypt	Стандарт де-факто, cost-factor

		конфігурований
Helmet.js	Власна реалізація	Готовий комплект HTTP-заголовків безпеки

Методологія розробки. Працював у власному темпі за принципом тижневих ітерацій: на початку тижня визначав, які модулі повинен закрити, в кінці тижня – підбирав, що вийшло, а що довелося переробити. Це не класичний Scrum (бо я був один), але сам підхід "малі ітерації + перегляд результату" виявився зручним. Структура проєкту відповідає типовому розподілу MVC: бізнес-логіка винесена в контролери, маршрутизація – в окремий файл, доступ до бази – у config/db, а представлення (HTML/CSS/JS) лежить у public. Такий поділ полегшує тестування – кожен частину можна перевірити окремо, не запускаючи цілу систему [41].

Висновки до розділу 1. Огляд теоретичних засад та аналіз існуючих рішень підтвердили актуальність розробки власної біржової платформи. Жодне з розглянутих рішень не поєднує водночас усіх ключових властивостей – підтримки різних класів активів, вбудованого matching engine, real-time трансляції цін і відкритого коду. Комерційні системи закриті, open-source альтернативи або застаріли, або потребують значних доопрацювань. Сформовано систему функціональних, нефункціональних, безпекових та нормативно-правових вимог, що стане основою для проєктування на наступному етапі. Обраний технологічний стек – Node.js, Express, MySQL та WebSocket – обґрунтовано як такий, що максимально відповідає визначеним вимогам.

РОЗДІЛ 2 ПРОЕКТУВАННЯ АРХІТЕКТУРИ ТА ФУНКЦІОНАЛУ СИСТЕМИ

2.1. Концептуальна та логічна архітектура платформи: модель взаємодії учасників, API-шар, структура застосунку

Архітектура – це фундамент, на якому далі тримається все. Помилишся з нею – переробляти доведеться все [42]. Для ExchangeUA обрав підхід Modular Monolith: на рівні розгортання це монолітний застосунок, який розгортається одним процесом, але всередині поділений на чіткі модулі за відповідальністю. Цей компроміс дає простоту монолітного підходу і водночас зручність окремого тестування та подальшої модульної рефакторизації.

Платформа розрізняє чотири категорії учасників взаємодії: трьох автентифікованих користувачів із власними ролями у системі (Trader, Issuer, Admin) та неавтентифікованого відвідувача (Guest), який представляє стан користувача до входу в систему. Ролі автентифікованих користувачів зберігаються у полі role таблиці users і визначають рівень доступу до функціоналу платформи:

1. Guest (Гість) – неавтентифікований відвідувач платформи. Не є окремою роллю на рівні бази даних: запис у таблиці users для нього не створюється, оскільки гість представляє стан користувача до проходження автентифікації. Розмежування реалізовано на рівні middleware застосунку – публічні ендпоінти (перегляд переліку активів та поточних цін, перегляд стрічки новин, форми реєстрації та входу) доступні без токена авторизації, тоді як решта маршрутів потребує валідної сесії з присвоєною роллю. Після успішної реєстрації користувач за замовчуванням отримує роль Trader.

2. Trader(Трейдер) – це основна "робоча" роль платформи. Трейдер розміщує ринкові, лімітні та стоп-ордери, скасовує ще не виконані, поповнює/виводить кошти, бачить свій портфель і журнал угод.

3. Issuer(Емітент) – емітент активу, ключова роль для забезпечення ліквідності платформи. Емітент виступає офіційним представником певного активу або групи активів: має юридичні документи на представництво (для класичних інструментів – проспект емісії, для крипто-токенів – listing-угода з платформою), створює нові активи на біржі (при створенні автоматично отримує право їх котирувати – запис у issuer_assignments), видаляє власні призначені активи, котирує ціни активів, на які його призначив адміністратор або які він створив сам. Окремою функцією емітента є управління ринковою пропозицією – він може поповнювати запас активу на платформу (action=add) або відкликати з обігу (action=withdraw, з захистом активів, уже придбаних трейдерами). Емітент може публікувати ринкові новини та аналітичні звіти за своїми активами. Не має доступу до загальної адмін-панелі та приватних даних інших користувачів.

4. Admin(Адміністратор) – повний доступ до всіх функцій, включаючи управління користувачами (зміна ролей, статусу, балансу), додавання нових активів, призначення емітентів на конкретні активи, котирування будь-якого активу, перегляд агрегованої статистики платформи та повного журналу котирувань.

На рис. 2.1. можна побачити взаємодію всіх описаних ролей з функціоналом платформи.

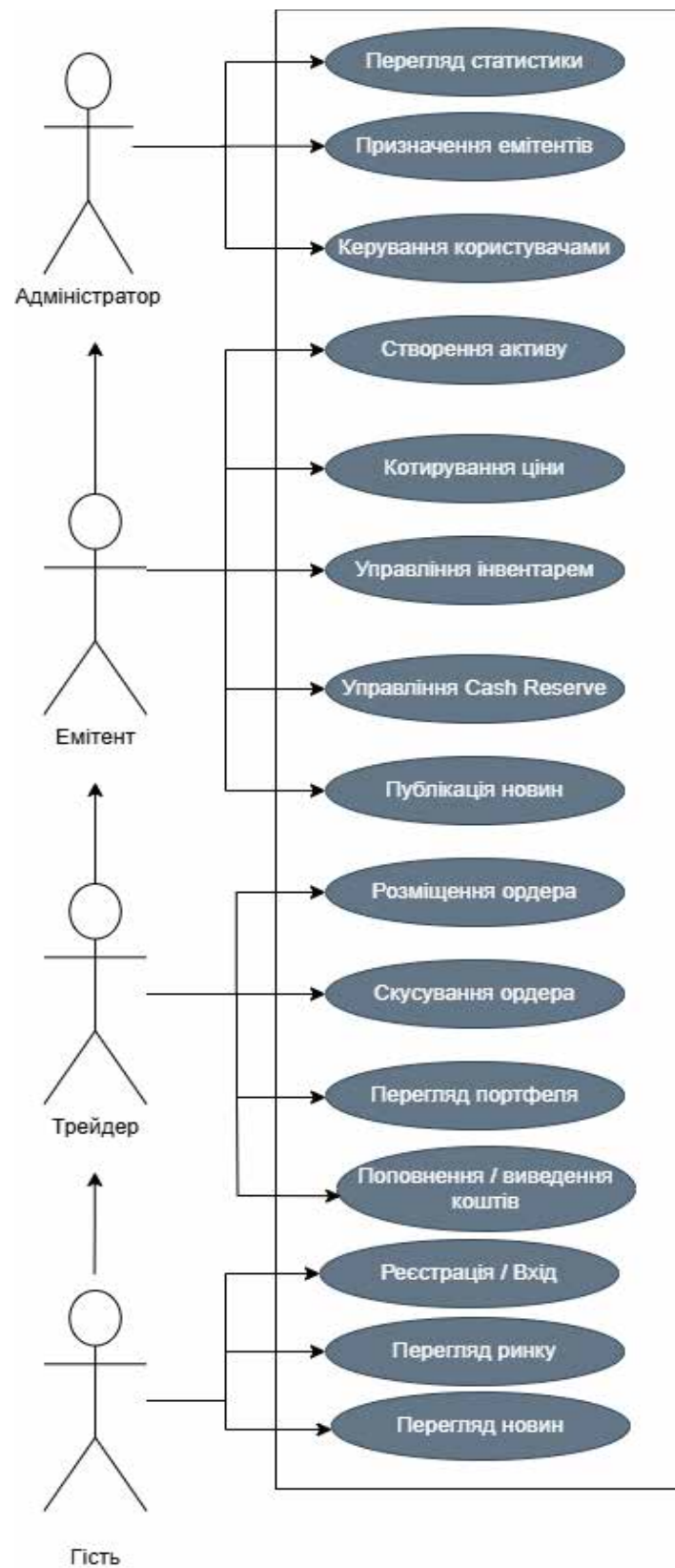


Рисунок 2.1. Діаграма варіантів використання

Класичний тришаровий шаблон [43] добре зайшов до моєї задачі. У підсумку отримав таку структуру:

Таблиця 2.1. Тришарова архітектура ExchangeUA

Шар	Компоненти	Технологія	Відповідальність
Представлення	index.html, style.css, app.js	Vanilla JS SPA	Рендеринг UI, клієнтська маршрутизація, WebSocket-клієнт
Бізнес-логіка	controllers/ middleware/	Node.js/Express	Обробка запитів, валідація, виконання ордерів, RBAC
Доступу до даних	config/db.js, sql/schema.sql	MySQL 8.0 (InnoDB)	Зберігання даних, транзакції, індексування
Транспортний	utils/ websocket.js, routes/api.js	WebSocket + REST	Комунікація між шарами

API спроектоване в RESTful-стилі [44]: стейтлес-запити, стандартні HTTP-методи (GET/POST/PUT/DELETE), правильні коди відповіді (200/201/400/401/403/404/500), ієрархічні URL ресурсів. Перед кожним обробником стоїть ланцюжок middleware у такій послідовності: requireAuth (перевіряє, чи користувач залогінений) , requireRole (перевіряє, чи у нього потрібна роль) і вже потім сам обробник. Далі на рис. 2.2. показано загальну структуру серверної частини з усіма шарами та потоками даних.

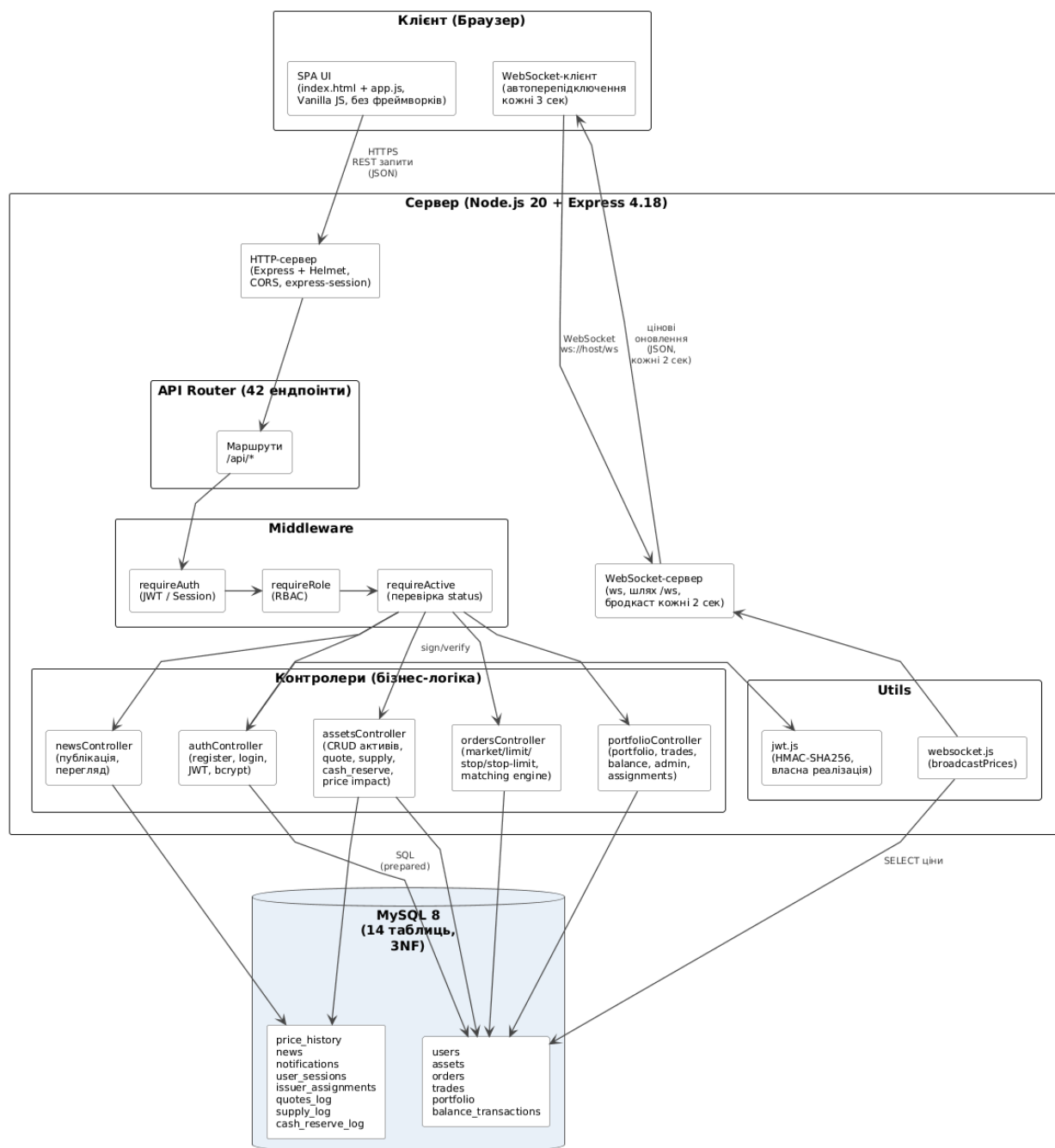


Рисунок 2.2. Компонентна діаграма архітектури ExchangeUA з відображенням потоків даних

Маршрутизатор `api.js` визначає 42 ендпоінти, згрупованих у десять ресурсних груп: `/auth` (6 маршрутів – реєстрація, вхід, вихід, профіль, оновлення профілю, зміна пароля), `/assets` (13 – список, деталі, історія, журнал котирувань, стакан, створення, редагування, видалення, котирування, управління інвентарем, журнал інвентарю, управління cash reserve і журнал cash reserve), `/orders` (3 – створення, перегляд, скасування), `/portfolio` (1) та `/balance` (3 – баланс,

поповнення, виведення), /trades (1), /notifications (2), /news (4 – список, деталі, створення, видалення), /admin (6 – управління користувачами, статистика, призначення емітентів), /issuer (1 – мої активи) та допоміжні /health, /debug (2). Кожна група відповідає окремому контролеру, що реалізує логіку CRUD-операцій із відповідними транзакційними гарантіями.

WebSocket-сервер живе в тому самому HTTP-сервері Express і підключається через подію "upgrade". Завдяки цьому не потрібно ні окремого порту, ні CORS-налаштувань – клієнт підключається до того самого хосту. Сервер обробляє три типи повідомлень: subscribe (клієнт каже, на оновлення яких символів він підписується), ping (keepalive, щоб з'єднання не закрилося), prices (зворотна трансляція – сервер кожні 2 секунди надсилає всім підписаним поточні ціни).

Окремо хочу зупинитися на тому, як саме формуються ціни. У роботі я відмовився від "симулятора цін" з рандомом. Замість нього передбачено три реальні механізми. Перший – котирування активу емітентом або адміністратором через POST /api/assets/:id/quote. Емітент котирує лише ті активи, на які його призначив адмін (зв'язок через таблицю issuer_assignments). Адмін може котирувати все. Другий механізм – last trade price: після кожної виконаної угоди current_price автоматично оновлюється до ціни цієї угоди. Третій – опціональний price impact: якщо для активу його увімкнено, купівля автоматично трохи піднімає ціну, продаж – опускає, пропорційно обсягу. Усі зміни ціни (від кого б вони не походили) фіксуються в таблиці quotes_log з полем source. Це дає повний аудит і прозорість ціноутворення.

2.2. Проектування бази даних та інформаційних моделей: сутності, зв'язки, індекси

Проектування БД проходило в три класичні етапи: концептуальне моделювання (ER-діаграма), логічне (нормалізація до третьої нормальної форми) і фізичне (DDL-скрипти з індексами, зовнішніми ключами та обмеженнями) [45]. Загальну структуру даних з усіма таблицями та зв'язками між ними наведено на рис. 2.3.

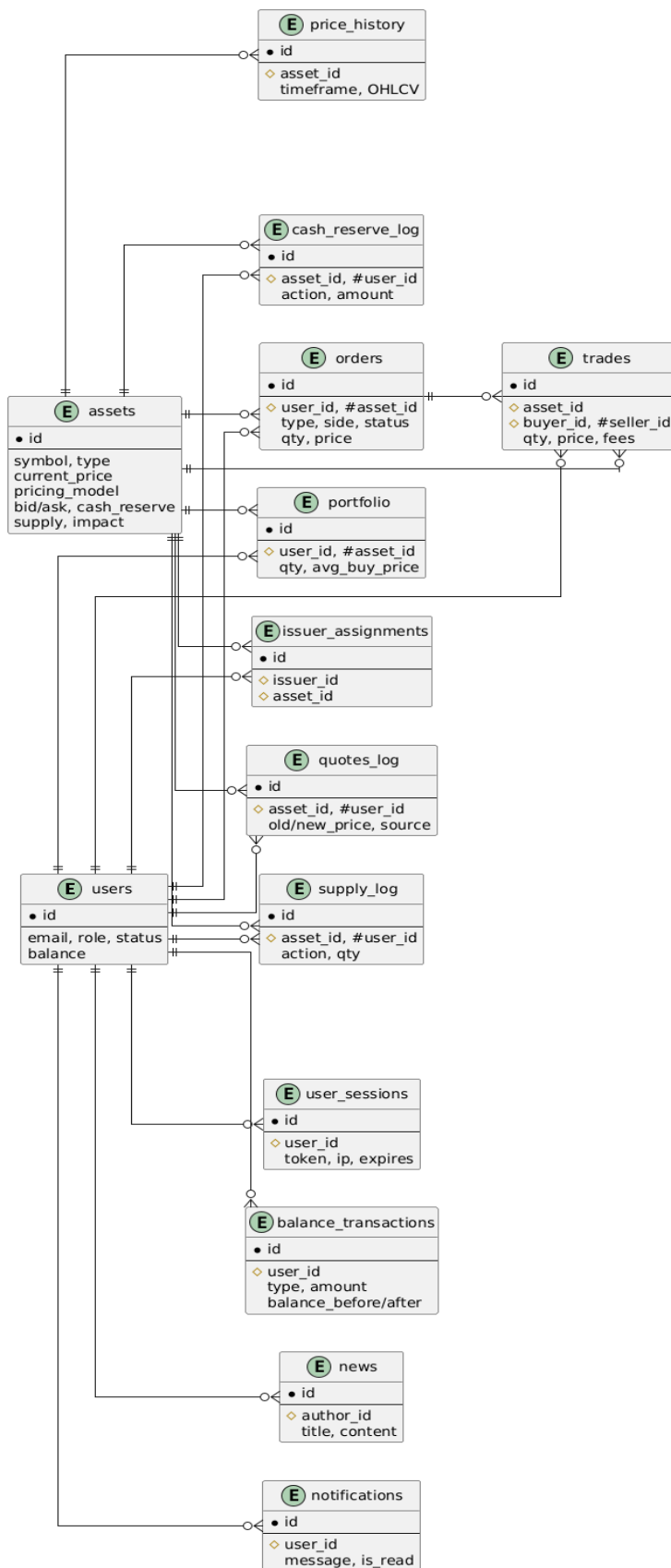


Рисунок 2.3. ER-діаграма бази даних exchange_platform

Центральна таблиця системи – users. У ній зберігається все, що стосується самого користувача: ідентифікаційні дані (uuid + email + password_hash), роль (ENUM з трьох можливих значень), статус (active/suspended/frozen) та поточний баланс. UUID поряд зі звичайним id дозволяє посилатися на користувача "зовні" (наприклад, у URL) без розкриття автоінкрементного номера. Баланс зберігаю в DECIMAL(20,8) – це з запасом покриває навіть найдрібніший крок криптовалют (сатоші).

Окремої уваги заслуговує статус "frozen" – це проміжний стан між повністю активним і повністю заблокованим акаунтом. Frozen користувач залишає доступ до перегляду свого портфеля, історії угод, балансу та активів на ринку, але втрачає можливість виконувати будь-які активні дії: створювати ордери, поповнювати чи виводити кошти, редагувати дані профілю, котирувати ціни (для емітентів). Технічно це реалізовано через окремий middleware requireActive, що ставиться поверх requireAuth на маршрутах CRUD-операцій і повертає HTTP 403 з кодом ACCOUNT_FROZEN для frozen користувачів. Read-only маршрути (GET-запити на портфель, угоди, баланс, активи) залишаються доступними без обмежень. Frontend паралельно показує жовтий банер у верхній частині сторінки і блокує відповідні кнопки інтерфейсу через CSS-атрибут body[data-frozen]. Така архітектура важлива з кількох причин: вона дозволяє адміністратору тимчасово "поставити на паузу" підозрілий акаунт без втрати користувачем доступу до власних даних (на відміну від suspended, що повністю забороняє вхід); забезпечує compliance-сценарій "AML/KYC review" – коли регулятор зобов'язує платформу тимчасово припинити торгівлю для конкретного користувача до завершення перевірки; зменшує негативний UX-вплив тимчасової блокади, бо користувач бачить чітку причину обмеження і знає що його дані залишаються в безпеці. Адміністратор може перевести користувача в будь-який з трьох статусів через адмін-панель: розморозити (active), заморозити (frozen) або повністю заблокувати (suspended).

assets – це довідник торгових інструментів. Поле type (ENUM: stock, crypto, forex, commodity) розділяє активи на чотири класи й використовується для фільтрування на головній сторінці. Поточна ціна (current_price) та ціна закриття

попереднього дня (prev_close) – ось два поля, з яких будується відсоток зміни на головній сторінці. Оновлюються вони не якимось окремим скриптом-завданням, а двома реальними механізмами: котируванням емітента/адміна через POST /api/assets/:id/quote і автоматично після кожної угоди в matching engine. volume_24h рахує реальний обсяг торгів за останні 24 години.

Найбільш насичена логікою – таблиця orders. Вона підтримує всі чотири типи ордерів – market, limit, stop, stop_limit – і два напрямки – buy і sell. Статусна машина ордеру складається з п'яти станів: open – filled / partial / cancelled / rejected. Окремі поля filled_qty та avg_fill_price дають змогу обробляти часткове виконання (коли ордер виконався не одразу, а кількома порціями). Комісія фіксується в момент виконання – це важливо для аудиту.

Таблиця trades – журнал виконаних угод. Це append-only сховище: записи туди тільки додаються, не редагуються і не видаляються. Для кожної угоди зберігаються пари ордерів (buy/sell), сторони (покупець і продавець), ціна, кількість, окремо комісія покупця і продавця. Останнє дає змогу реалізувати асиметричну тарифну сітку (наприклад, нижчі комісії для maker'ів), якщо знадобиться у майбутньому.

Таблиця 2.2. Характеристики таблиць бази даних

Таблиця	Рядків (прогноз)	Ключові індекси	Примітки
users	~10 000	email, role, status	UUID для зовнішніх посилань
assets	~100	symbol, type	Seed: 12 активів
orders	~1 000 000	user_id, asset_id, status, side, created_at	Атомарне виконання
trades	~500 000	asset_id, buyer_id, seller_id, executed_at	Append-only журнал
продовження табл. 2.2.			
Таблиця	Рядків(прогноз)	Ключові індекси	Примітки

portfolio	~50 000	UNIQUE(user_id, asset_id)	Агрегований стан позиції
balance_transactions	~2 000 000	user_id, type, created_at	Повна аудит-стежка
price_history	~5 000 000	UNIQUE(asset_id, timeframe, candle_time)	OHLCV свічки
news	~500	is_published, published_at, category	Новини та аналітика
notifications	~500 000	user_id, is_read, created_at	Внутрішні сповіщення
user_sessions	~200 000	user_id	Аналітика активності

Схема приведена до 3НФ – транзитивних залежностей немає, кожна таблиця відповідає одній сутності. Для дочірніх записів налаштовано каскадні видалення: коли видаляється користувач, разом з ним з БД підуть його ордери, портфельні позиції, сповіщення (відповідні ON DELETE CASCADE). Це позбавляє від "сирітських" записів та підтримує консистентність. Усі зв'язки оформлені як FOREIGN KEY на рівні самої СУБД – будь-яка спроба порушити цілісність буде заблокована ще на етапі INSERT/UPDATE.

Індекси проєктував окремо під найчастіші реальні запити. Найгарячіші сценарії: пошук відкритих ордерів за активом, отримання непрочитаних сповіщень користувача, вибірка свічок за активом і таймфреймом. Під кожен такий патерн зробив комбінований індекс – наприклад, (user_id, is_read) або (asset_id, candle_time). Запити такі індекси прискорюють кратно, а накладні витрати на запис залишаються прийнятними.

2.3. Розробка ключових модулів системи: автентифікація, каталог активів з двома моделями ціноутворення, торговий модуль, управління інвентарем, портфель, новини та сповіщення

Модуль автентифікації живе у двох файлах: `authController.js` (логіка) та `middleware/auth.js` (перевірки на вхідних запитах). Реєстрація, яка зображена на рис. 2.4. проходить у п'ять кроків: валідація даних, перевірка унікальності email, генерація UUID, хешування пароля через `bcryptjs` з `cost=12` та запис у БД з нульовим стартовим балансом. Поповнити рахунок користувач може окремо – через сторінку "Гаманець" або `POST /api/balance/deposit`. Такий підхід відповідає поведінці реальних бірж і одночасно прибирає ризик зловживань через множинні акаунти. Паралельно генерується JWT-токен і встановлюється сесійний cookie – щоб користувач відразу залогінився, без необхідності окремого входу після реєстрації.

Валідація email-адреси заслуговує окремої уваги, бо реалізована тришаровою системою для захисту як від простих помилок користувача, так і від спроб обходу. Перший шар – браузерна валідація через атрибут `type="email"` HTML5-форми; вона ловить найочевидніші помилки на стороні клієнта ще до відправки запиту, але її легко обійти через прямий API-виклик. Другий шар – JavaScript-перевірка на фронтенді перед відправкою через регулярний вираз `/^[^\s@]+@[^\s@]+\.[^\s@]+$/` , що вимагає наявності символу `@` і крапки після нього (мінімально достатня структура email). Третій і головний шар – серверна перевірка у `authController.register`: той самий регулярний вираз плюс обмеження довжини у 254 символи (відповідає RFC 5321, який встановлює максимум для адреси листа). Тільки після проходження всіх трьох рівнів email потрапляє до БД. Така дублююча архітектура – практика OWASP "defense in depth": один шар може бути обійдений, але не всі одночасно. У тестовому наборі `business.test.js` передбачено десять окремих перевірок: валідні випадки (`user@example.com`, `user.name@subdomain.co.uk`, `user+tag@example.com`), невалідні (порожній рядок, без `@`, без домену, з пробілом, з декількома `@`, дуже довгий понад 254 символи). Усі тести проходять успішно, що гарантує стійкість фільтра до типових кейсів атак на форму реєстрації.

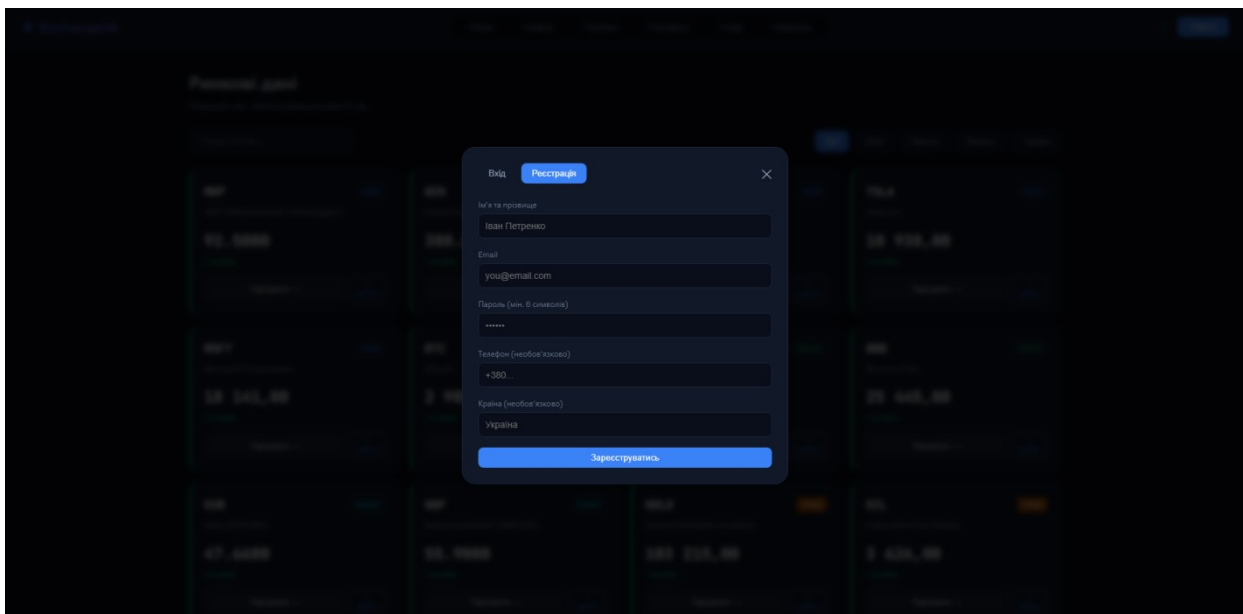


Рисунок 2.4. Сторінка реєстрації нового трейдера

Middleware `requireAuth` перевіряє входження двома способами поспіль. Спочатку дивиться, чи є `userId` у сесії (це для браузерних клієнтів). Якщо немає – дивиться, чи переданий Bearer-токен у заголовок `Authorization` (це для API-клієнтів). Така "двоканальна" схема дозволяє одному й тому ж застосунку обслуговувати і web-фронт, і автоматизовані інтеграції. Функція `requireRole` відповідає за RBAC: приймає список дозволених ролей і повертає HTTP 403, якщо роль поточного користувача не входить до переліку.

`assetsController.js` реалізує тринадцять функцій. `GET /assets` повертає повний список активних активів із розрахованим відсотком зміни ціни, що і зображено на рис. 2.5. `GET /assets/:id` повертає детальну інформацію про конкретний актив разом зі списком призначених емітентів. `GET /assets/:id/history` повертає дані з таблиці `price_history` для побудови графіка ціни активу, що продемонстровано на рис. 2.11.

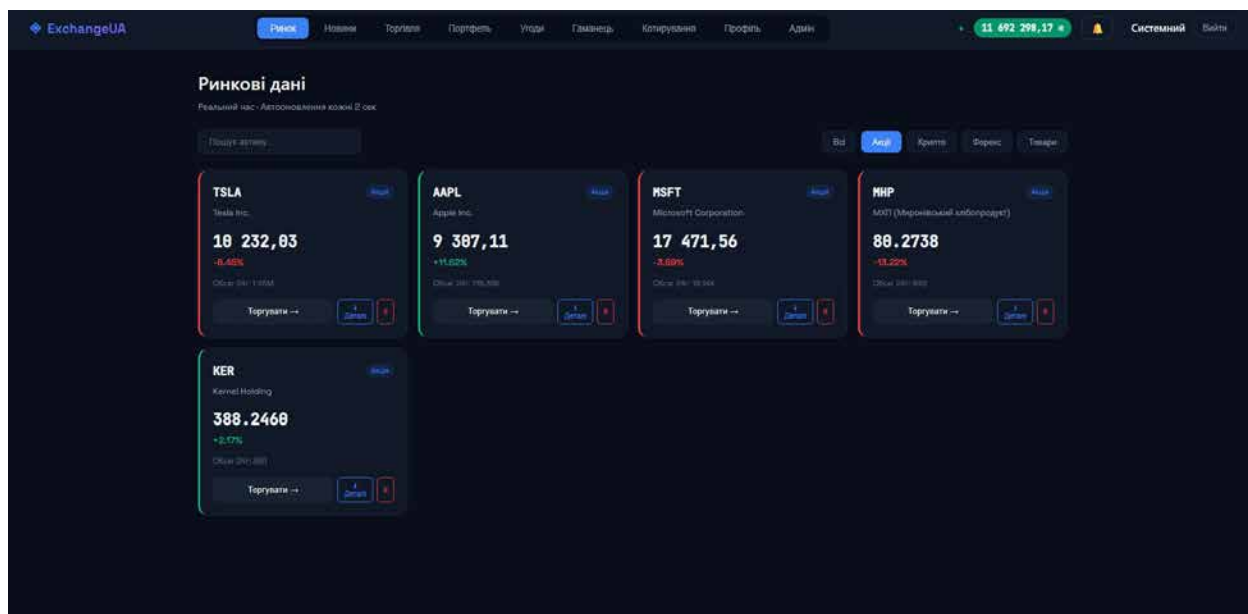


Рисунок 2.5. Головний екран ринку з реальним часом оновлення котирувань

GET /assets/:id/orderbook формує стакан ордерів, агрегуючи відкриті лімітні ордери за рівнями цін, що зображено на рис. 2.12. POST /assets – створення нового активу (доступно адміністратору і емітенту; при створенні емітентом запис автоматично додається до issuer_assignments). PUT /assets/:id – редагування активу (назва, опис, налаштування авторегулювання ціни); адміністратор редагує будь-який, емітент – лише призначені. DELETE /assets/:id – видалення активу (м'яке is_active=0, якщо є залежності). POST /assets/:id/quote – встановлення нової ціни емітентом або адміном з обмеженням стрибка $\pm 50\%$. GET /assets/:id/quotes-log повертає історію котирувань. POST /assets/:id/supply – управління ринковою пропозицією активу (емітент поповнює або відкликає запас). GET /assets/:id/supply-log повертає журнал змін інвентарю. Для активів у режимі active_mm додатково реалізовано пару endpoint-ів: POST /assets/:id/cash-reserve (поповнення або виведення грошового резерву емітента) і GET /assets/:id/cash-reserve-log (журнал змін резерву). Окремою функцією реалізовано авто-регулювання ціни активу під впливом торгових операцій (price impact model): якщо для активу увімкнено цей режим і задано коефіцієнт впливу, то після кожної угоди ціна автоматично коригується пропорційно обсягу – купівля підвищує ціну, продаж знижує. Усі зміни ціни (емітентом, адміном, через угоду,

через автокорекцію) журналюються у `quotes_log` з полем `source`, що забезпечує повну прозорість ціноутворення.

Декілька слів про модуль управління ринковою пропозицією, який реалізовано як невелику, але важливу частину `assetsController.js`. Кожен актив у таблиці `assets` має дві числові колонки: `total_supply` – загальну кількість, що коли-небудь була заведена емітентом на платформу, та `available_supply` – кількість, доступну зараз для купівлі трейдерами. Початково на старті проєкту всі дванадцять активів отримують значення 10 000 одиниць (це задано через `DEFAULT` у схемі). Далі емітент, якого призначено на конкретний актив, може через інтерфейс сторінки «Котирування» яку зображено на рис. 2.6. або `POST /api/assets/:id/supply` виконати дві дії – додати запас (`action='add'`), що збільшує обидві колонки на задану кількість, або відкликати запас (`action='withdraw'`), що зменшує їх. Останнє дозволено тільки якщо `available_supply` більше або дорівнює запитуваній кількості – відкликати з активів, які вже придбали трейдери, заборонено на рівні логіки контролера. Таким чином, модель моделює реальну роль емітента-маркетмейкера, який заводить актив на ринок, але не може забрати в трейдерів те, що вже їм продане.

Початковий набір (`seed`) з дванадцяти активів сформовано спеціально для української аудиторії – всі ціни виражено у гривнях, що відповідає валюті балансу системи. До нього увійшли українські акції (МХП, Kernel Holding), американські акції (Apple, Tesla, Microsoft), криптовалюти (Bitcoin, Ethereum, Binance Coin), валютні пари у форматі грн за одиницю іноземної валюти (євро і британський фунт), а також товарні активи (золото за унцію та нафта Brent за барель). На відміну від класичних USD-крос-пар, форекс представлено як гривневий курс – це логічніше для української платформи, бо користувач відразу бачить, скільки грошей потрібно для купівлі однієї одиниці іноземної валюти. Стартові ціни беруться з реальних ринкових даних на момент розгортання системи і можуть оновлюватися; конкретні значення зберігаються у файлі `sql/schema.sql` і легко модифікуються. Початкові баланси демо-користувачів узгоджені з типовим діапазоном цін, щоб після першого входу можна було відразу провести повний цикл операцій – купівлю акцій, торгівлю

криптовалютою, поповнення/виведення коштів. Окремо у скрипті `setupUsers.js` при першому запуску автоматично створюється демонстраційний актив у режимі `active_mm` з налаштованими `bid/ask` і `cash reserve`, призначений на тестового емітента. Це дозволяє одразу після розгортання продемонструвати другу модель ціноутворення без додаткових налаштувань.

Інтерфейс сторінки «Котирування» побудовано як двопанельний макет `master-detail` з вкладеною системою табів. Ліва частина – список активів, на які поточний користувач має право котирувати: для емітента це його `assignments`, для адміністратора – всі активи системи. Кожен рядок містить тікер, повну назву, поточну ціну, бейдж моделі ціноутворення (зелений «ММ» для `active_mm`, синій «Listing» для `passive`) і кнопку «Керувати». При виборі активу права частина перетворюється на панель керування зі заголовком (тікер і назва) та трьома кнопками-діями у заголовку – налаштування активу, видалення, повторний вибір. Нижче розташовується перемикач табів: вкладка «Ціна» (для `passive` – одне поле, для `active_mm` – два поля `bid/ask` з автоматичним розрахунком спреда), вкладка «Інвентар» (управління `total_supply` і `available_supply` через дії `add/withdraw`) і вкладка «Cash Reserve», яка показується лише для активів `active_mm` (поповнення та виведення грошового резерву емітента). Така архітектура виправдана з кількох міркувань: по-перше, вона зменшує когнітивне навантаження (емітент бачить тільки той розділ, який потрібен зараз); по-друге, природньо розділяє різні рівні операцій (зміна ціни – це повсякденна дія, керування інвентарем – рідша, `cash reserve` – рідкісна для більшості активів); по-третє, дозволяє приховати непотрібний функціонал – наприклад, для `passive` активу таб «Cash Reserve» взагалі не показується, бо це поняття не застосовне. Внизу панелі завжди видно журнал змін ціни – це повна історія котирувань з тимчасовою позначкою, джерелом і ким зроблено, що відповідає принципу аудиту фінансових операцій.

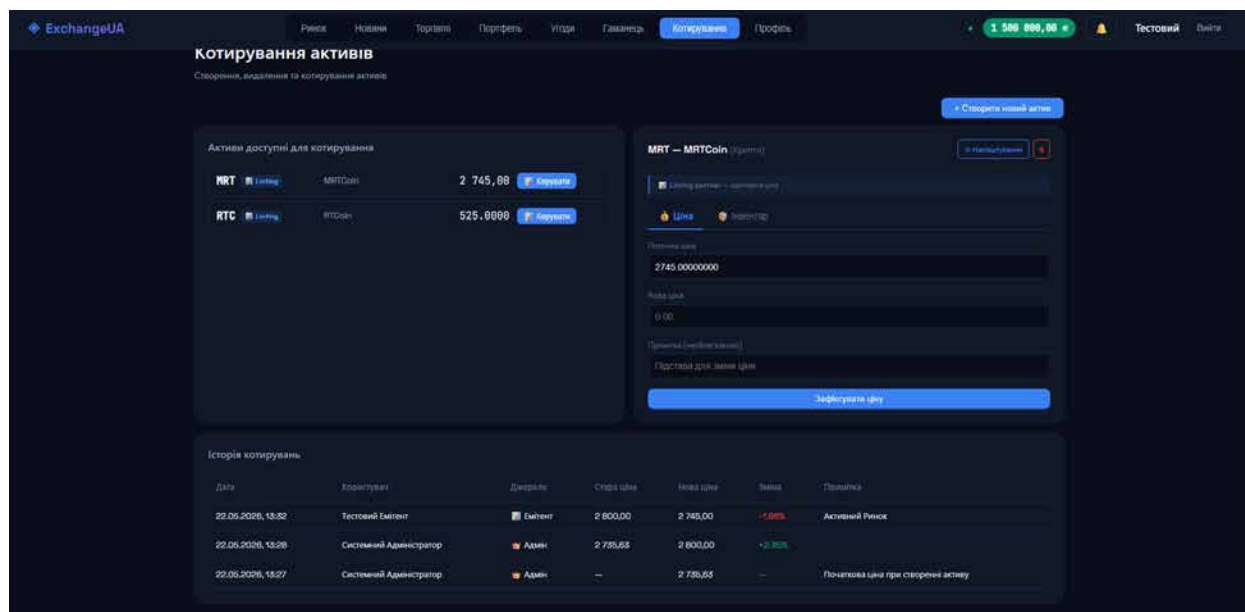


Рисунок 2.6. Сторінка котирування активів емітентом

Коли трейдер купує актив ринковим ордером, виконується перевірка $available_supply \geq qty$ – якщо запасу недостатньо, ордер відхиляється з повідомленням «Зверніться до емітента для поповнення запасу». При успішній купівлі $available_supply$ зменшується на куплену кількість ($total_supply$ залишається сталою – актив фізично не зник, він перейшов у портфель трейдера). При продажі трейдером – навпаки, $available_supply$ збільшується (актив повертається в обіг біржі). Усі дії – і емітента, і автоматичні від купівлі/продажу – журналюються в окрему таблицю `supply_log` з полями `action` (`add/withdraw/trade_buy/trade_sell/init`), `available_before/after`, `total_before/after` та посиланням на ордер чи користувача. Це забезпечує повний аудит інвентарю в будь-який момент часу.

Окремий блок функціональності – управління особистими даними користувача. Сторінка «Профіль» продемонстрована на рис. 2.7. дозволяє користувачу будь-якої ролі переглянути свої облікові дані (повне ім'я, email, поточну роль, дату реєстрації, дату останнього входу) і відредагувати їх через модальне вікно. Сам ендпоінт `PUT /api/auth/profile` дозволяє оновити ім'я та email, при цьому валідується унікальність нового email серед усіх користувачів – це запобігає дублюванню облікових записів. Окремо реалізовано зміну пароля, яка продемонстрована на рис. 2.8. Через `POST /api/auth/change-password`:

користувач спочатку вводить старий пароль (для підтвердження особи), потім новий (двічі – з повторною перевіркою). Старий пароль перевіряється через bcrypt.compare проти поточного хешу в БД; якщо збіг успішний, новий пароль хешується з $\text{cost}=12$ і записується. Така двоетапна верифікація відповідає рекомендаціям NIST SP 800-63B щодо керування секретами автентифікації.

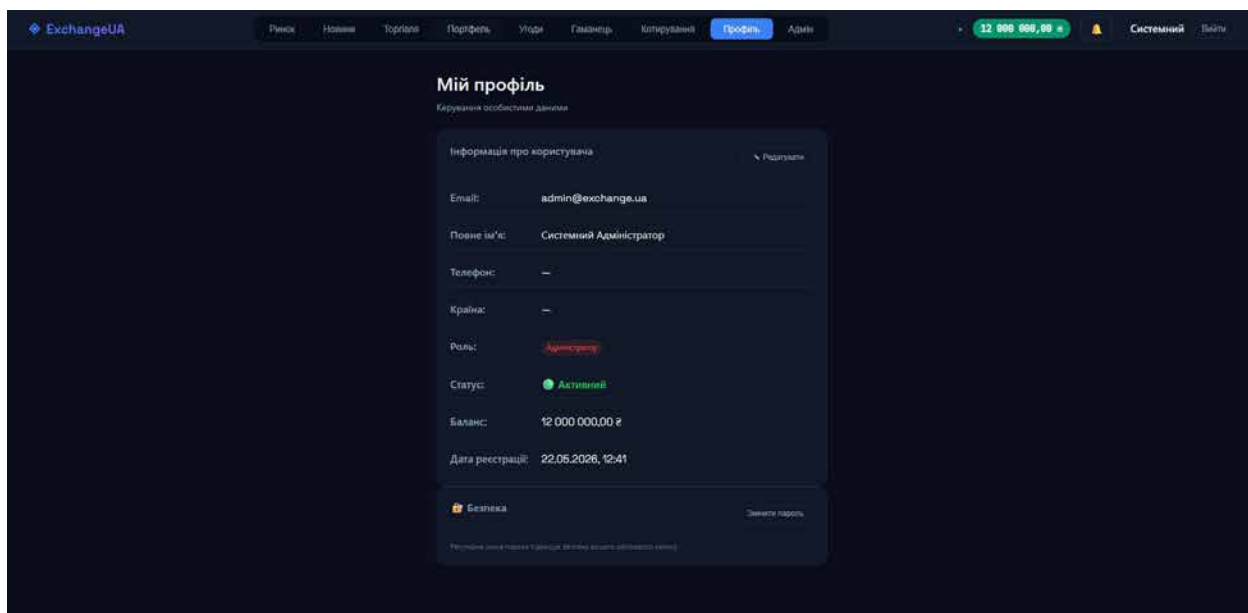


Рисунок 2.7. Сторінка «Профіль» з керування особистими даними

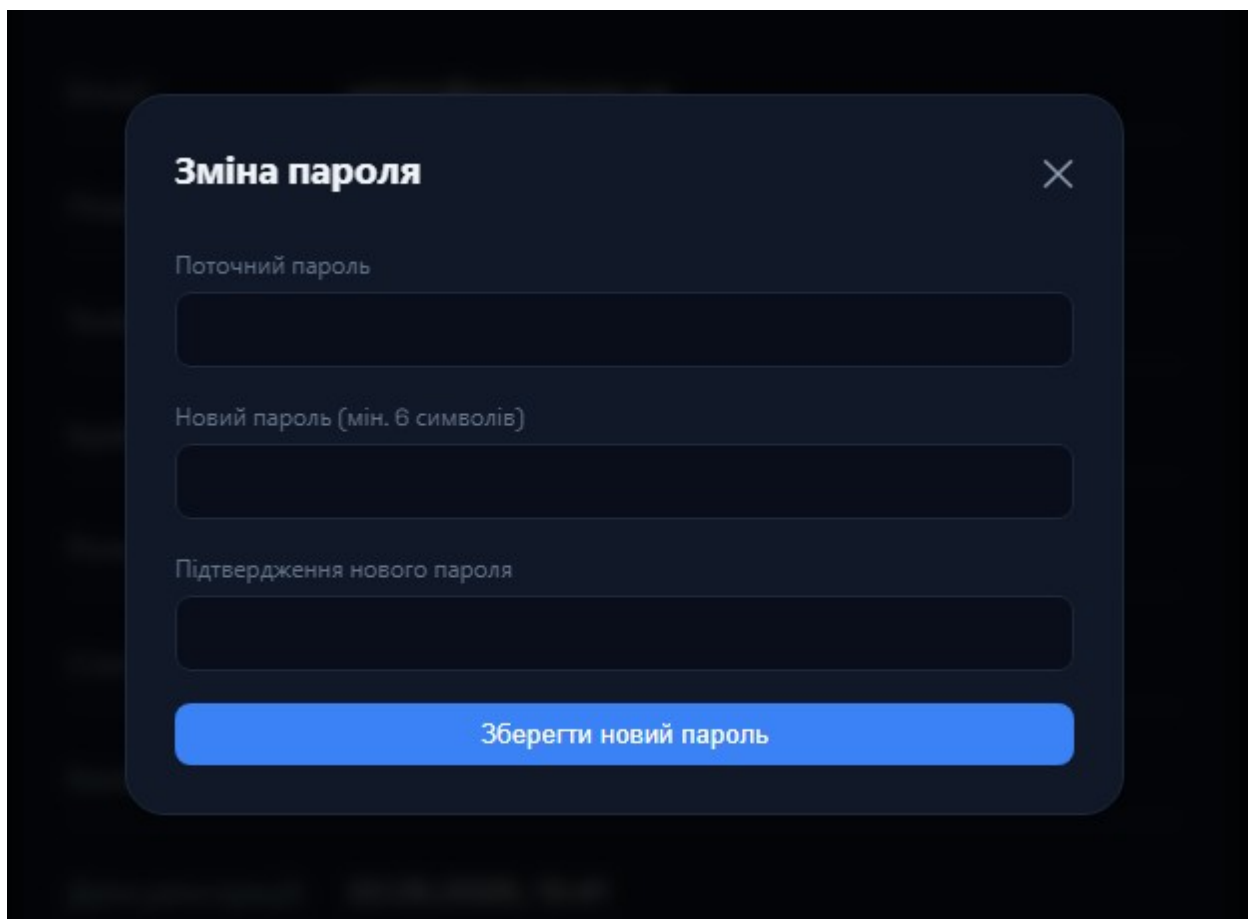
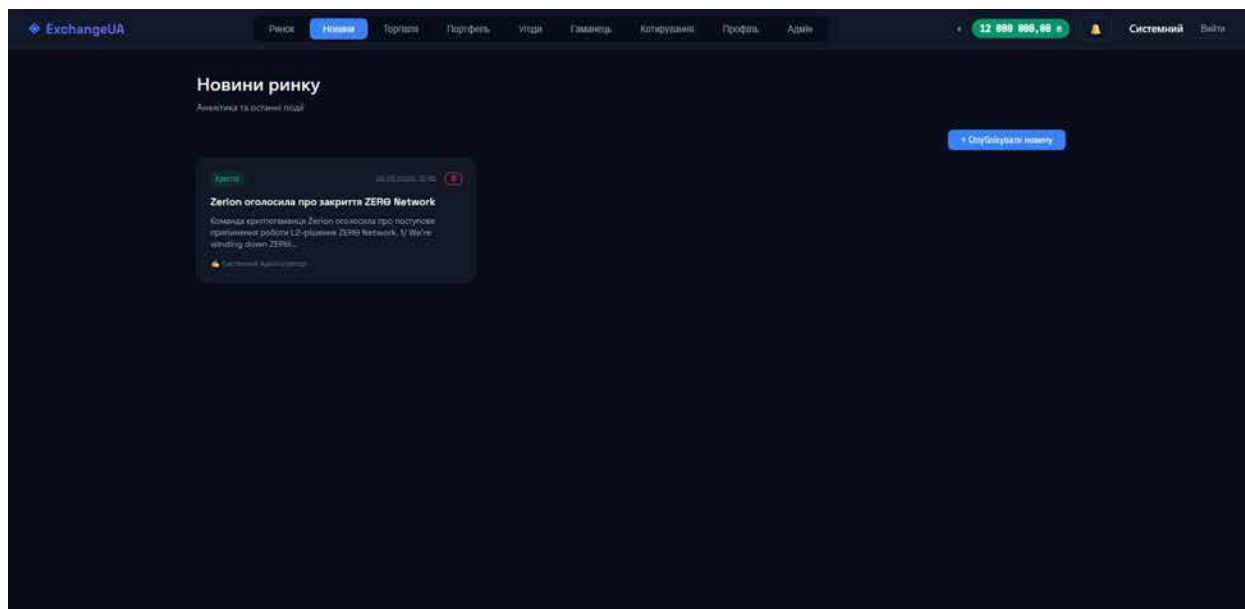


Рисунок 2.8. Вікно зміни паролю

Окремий модуль платформи – стрічка ринкових новин, реалізована у файлі `newsController.js`. Цей модуль показано на рис. 2.9. Він дає змогу адміністраторам і емітентам публікувати аналітичні матеріали для всіх користувачів платформи: огляди ринку, прогнози, повідомлення про корпоративні події емітентів за їхніми активами. Структура таблиці `news` містить заголовок, повний текст, короткий `summary` для прев'ю, категорію (`market`, `company`, `crypto`, `forex`, `commodity`, `economy`), статус публікації, дату публікації та автора. API налічує чотири маршрути: `GET /api/news` повертає стрічку опублікованих новин з пагінацією та фільтром за категорією, `GET /api/news/:id` – деталі однієї новини, `POST /api/news` створює нову новину (доступ – `admin` або `issuer`), `DELETE /api/news/:id` видаляє новину (адміністратор будь-яку, емітент – лише свої). Призначення модуля – забезпечити прозорість інформаційного фону платформи: коли актив раптово змінюється в ціні, користувач має змогу одразу побачити пов'язану новину від емітента або адміністратора з поясненням події.



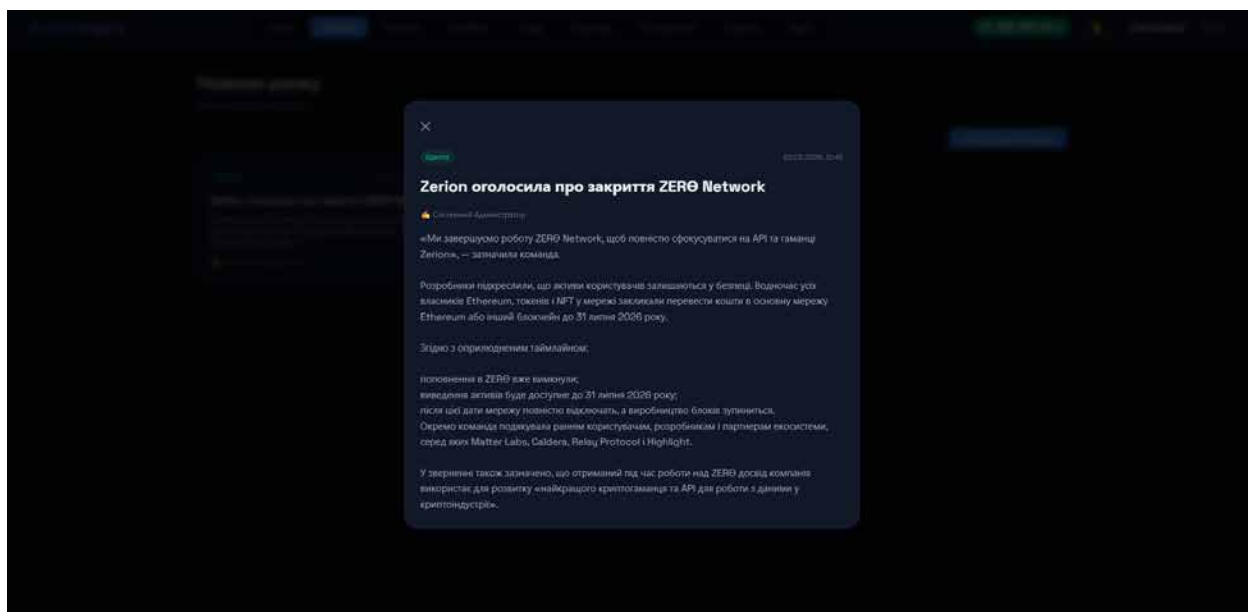


Рисунок 2.9. Сторінка «Новини ринку»

Сторінка «Угоди» (history) – це особистий журнал виконаних угод трейдера, продемонстрований на рис. 2.10. Через GET /api/trades повертається список з останніх 50 угод користувача за замовчуванням (параметр limit можна змінити). Кожен запис містить дату/час угоди, тип (купівля чи продаж), символ активу, кількість, ціну виконання, сумарну вартість, сплачену комісію та статус. На відміну від таблиці orders (де зберігаються всі ордери – і виконані, і скасовані, і відкриті), таблиця trades містить лише факти реальних угод. Це дає змогу трейдеру швидко порахувати реалізований прибуток або збиток – на відміну від нереалізованого P&L у портфелі, який залежить від поточної ціни ринку. Дані сортуються від найновіших до найстаріших і відображаються у вигляді таблиці з можливістю прокрутки.

Актив	Статус	Кількість	Ціна	Сума	Комісія	Час
AAPL	ВІДКРИТ	1.0000	8 979,65	8 979,65 ₴	0,00 ₴	22.05.2024, 15:03
AAPL	ВІДКРИТ	4.0000	8 202,09	32 808,36 ₴	32,81 ₴	22.05.2024, 13:50
AAPL	ВІДКРИТ	1.0000	8 338,00	8 338,00 ₴	0,00 ₴	22.05.2024, 13:00
AAPL	ВІДКРИТ	4.0000	8 338,00	33 352,00 ₴	33,35 ₴	22.05.2024, 13:00
BTC	ВІДКРИТ	5.0000	525,0000	2 625,00 ₴	2,63 ₴	22.05.2024, 12:01

Рисунок 2.10. Сторінка «Угоди» – журнал виконаних угод трейдера

Останній компонент модуля каталогу – модуль графіків цін, який показано на рис. 2.11. Він складається з двох частин: серверного зберігання OHLCV-свічок у таблиці `price_history` та клієнтського візуалізатора. Кожна зміна ціни активу (від котирування емітентом, виконаної угоди або автокорекції через `price impact`) агрегується у свічки за обраним таймфреймом. На клієнті, при відкритті модального вікна деталей активу, виконується запит `GET /api/assets/:id/history`, який повертає останні N точок цінової історії у форматі OHLCV. Отриманий масив передається у `Chart.js`, що малює лінійний графік ціни з інтерактивними підказками при наведенні.

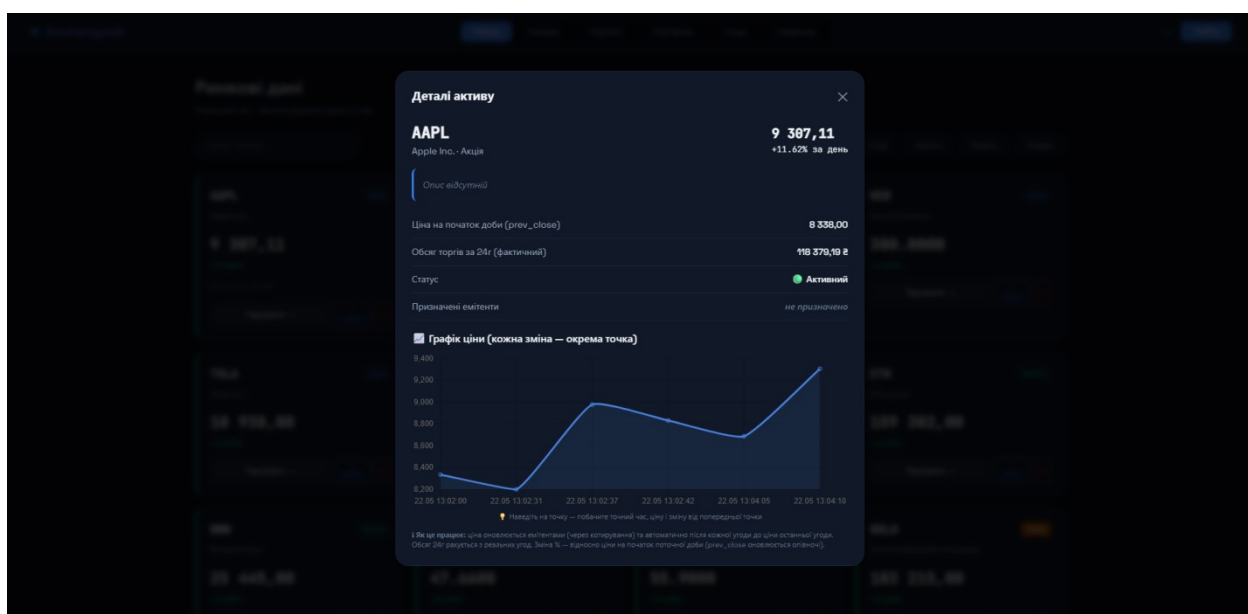


Рисунок 2.11. Графік ціни активу

Перш ніж переходити до реалізації, варто пояснити доцільність наявності всіх чотирьох типів ордерів – це не теоретичний "академічний набір", а необхідний мінімум для будь-якої серйозної біржі. Кожен тип розв'язує конкретну проблему трейдера. Ринковий ордер (market) – найпростіший: виконується миттєво за поточною ціною. Він потрібен у ситуаціях, коли важлива швидкість, а не точна ціна – наприклад, ринок різко падає і треба швидко вийти з позиції, або з'явилася гаряча новина і трейдер хоче встигнути зайти. Недолік – трейдер не контролює ціну виконання, при високій волатильності може отримати гірший результат, ніж очікував (slippage).

Лімітний ордер (limit) розв'язує саме цю проблему. Трейдер вказує максимальну ціну, за яку готовий купити (або мінімальну, за яку готовий продати). Ордер виконається тільки якщо ринок дійде до цієї ціни – або одразу, або пізніше. У реалізації ExchangeUA лімітний ордер працює за двошаровою моделлю ліквідності. Перший шар – це класичний peer-to-peer order book matching: система шукає у книзі ордерів зустрічні пропозиції від інших трейдерів за вказаною лімітною ціною або кращою. Другий шар активується, якщо order book не зміг звести ордер повністю (або взагалі) – у цьому випадку залишок виконується з пулу емітента за поточною ціною (current_price), але тільки якщо ліміт користувача вигідно це дозволяє: для BUY – $\text{limit_price} \geq \text{current_price}$, для SELL – $\text{limit_price} \leq \text{current_price}$. Це принцип price improvement: трейдер отримує найкращу можливу ціну (купує дешевше або продає дорожче, ніж готовий був). Якщо ж ліміт не вигідний (BUY нижче ринку або SELL вище ринку) – ордер залишається у книзі і чекає кращої ціни. Така модель відповідає реальній практиці провідних бірж: на NYSE така роль виконують designated market makers (DMM), на криптобіржах – внутрішні маркетмейкери (наприклад, на Binance), які підхоплюють ордери, які order book не зміг звести. Емітент у системі ExchangeUA виконує цю саму функцію: він тримає інвентар активу і виставляє двостороннє котирування за current_price, забезпечуючи мінімальну гарантовану ліквідність.

Стоп-ордер (stop) – інструмент управління ризиками. Класичне застосування – це stop-loss: трейдер купив актив за 100, але не хоче втратити більше, ніж 10%, і ставить stop-sell на рівні 90. Поки ціна вища за 90 – ордер спить. Коли ціна падає до 90 або нижче – ордер автоматично активується як ринковий і продає актив, обмежуючи збиток. Симметрично можна використовувати stop-buy для входу в ринок при пробитті опору: "купити, тільки якщо ціна підніметься вище 105 – це означатиме, що бичачий тренд підтверджено". Без стоп-ордерів трейдер змушений був би постійно сидіти біля екрану, щоб відреагувати на рух ринку – це нереально на практиці.

Стоп-лімітний ордер (stop-limit) – це гібрид стопа і ліміту, який вирішує проблему класичного stop-loss. Коли ринок різко обвалюється (наприклад, через флеш-крах), звичайний stop-sell перетворюється на market і може виконатися за катастрофічно низькою ціною – бо в стакані просто нема покупців на нормальних рівнях. Stop-limit захищає від цього: коли тригер спрацював, замість ринкового ордера створюється лімітний з нижчою межею. Трейдер каже системі: "продай мені, якщо ціна впаде до 90, але не дешевше ніж 85". Якщо обвал занадто різкий – ордер просто не виконається, і це краще, ніж продати за безцінь. Така комбінація особливо важлива у криптотрейдингу, де відсутність ліквідності й різкі рухи на 20-30% за хвилину – звичайна справа.

Усі чотири типи реалізовано в системі ExchangeUA – від базового market і limit до stop / stop-limit з власним механізмом моніторингу ціни (функція triggerStopOrders викликається після кожної зміни ціни активу і знаходить стоп-ордери, тригер яких перетнуто). Лімітні та стоп-лімітні ордери залишаються у книзі ордерів до моменту виконання або скасування користувачем – це формує ту саму глибину ринку, яку бачать інші учасники у стакані ордерів (orderbook). Така повнота функціональності – нормальний стандарт для будь-якої серйозної біржі. Платформа без стоп-ордерів – це ризик, який жоден професійний трейдер не прийме.

Окрему увагу варто приділити моделям ціноутворення, які підтримує платформа. Це найважливіше архітектурне рішення системи: ExchangeUA реалізує дві паралельні моделі ціноутворення, які обирає сам емітент при

створенні активу. Це `pricing_model='passive'` (Listing Partner) та `pricing_model='active_mm'` (Active Market Maker). Дві моделі співіснують у межах однієї платформи і обслуговують активи різних класів – від молодих з пасивним лістингом до зрілих з активним маркетмейкінгом. Така подвійна архітектура повністю відповідає реальній практиці провідних бірж: NYSE використовує модель DMM (Designated Market Maker) для високоліквідних акцій і модель `passive listing` для нових IPO; Binance – поєднує власних маркетмейкерів з пасивними `listing partners` залежно від обсягу торгів по конкретному активу.

Модель Listing Partner (`passive`) – це класична CEX-модель (Centralized Exchange), у якій сама платформа виступає клірінговим контрагентом усіх угод. Логіка така: емітент створює актив, котирує його ціну (`current_price`), управляє інвентарем через `total_supply/available_supply`, опційно вмикає авто-регулювання ціни (`price impact`). Але сам емітент не бере участі в торгових операціях фінансово – коли трейдер купує `market-ордером`, платформа списує гроші з трейдера у власний операційний бюджет, при продажу – виплачує з нього. На масштабі це нетто-нульова операція: за кожною купівлею стоїть симетрична продаж кимось іншим. Платформа заробляє виключно на комісії 0.1% з обороту (Розділ 3.3). Емітент у цій моделі виконує роль курадора активу – він не несе прямого фінансового ризику від торгівлі трейдерів, але контролює доступність і параметри активу. Така модель повністю аналогічна Binance Listing Program для криптобірж і NYSE Listing Standards для фондових: компанія-емітент платить за лістинг або передає певну кількість активу платформі, а далі активний вторинний ринок обслуговує сама платформа.

Модель Active Market Maker (`active_mm`) принципово відрізняється економічною логікою – тут емітент реально торгує проти трейдерів. Замість однієї ціни `current_price` є дві: `bid_price` (ціна, за яку емітент готовий купити актив назад у трейдерів) і `ask_price` (ціна, за яку емітент готовий продати актив трейдерам). Різниця між ними – `спред (spread)` – це чистий прибуток емітента з кожного циклу купити-продати одну одиницю активу. Технічно це реалізовано так: при `market BUY` трейдером з його балансу списується `askPrice × qty`, ця сума надходить у пул емітента (поле `issuer_cash_reserve` у таблиці `assets`), а з

available_supply списується qty одиниць. При market SELL навпаки: з пулу емітента вираховується $\text{bidPrice} \times \text{qty}$, ця сума зараховується трейдеру, а активи повертаються в available_supply. Якщо у пулі недостатньо коштів для викупу – ордер відхиляється з повідомленням "Емітент тимчасово не має ліквідності для викупу". Це класична модель market maker з реальним фінансовим зобов'язанням.

Архітектурно cash reserve активу зберігається в таблиці assets (поле issuer_cash_reserve), а не прив'язаний до конкретного користувача. Це принципово важливо: якщо на актив призначено декількох емітентів, вони працюють зі спільним пулом – будь-який призначений емітент може поповнити пул зі свого особистого балансу або вивести надлишок назад. Для повноти аудиту реалізована окрема таблиця cash_reserve_log, у якій фіксується кожен рух (action: deposit/withdraw/trade_buy/trade_sell/init) з прив'язкою до конкретного user_id – це дозволяє визначити, хто саме з емітентів робив ту чи іншу операцію. Така схема відповідає реальній організації роботи market makers на біржах: пул належить активу, а не людині, але кожна дія документально журналюється.

Розрахунок прибутку емітента у моделі active_mm. Якщо емітент котирує bid=500 і ask=510 (спред 10 грн або 2%), кожний повний цикл (трейдер купив 1 шт за 510, інший трейдер продав 1 шт за 500) дає чистий прибуток 10 грн. На обсязі 1000 циклів на день – це 10 000 грн заробітку. Це робить рольовий сенс емітента у active_mm моделі суто фінансовим: він заробляє з обороту через спред, бере на себе ризик балансування інвентаря (якщо всі продають – він накопичує запас активу і витрачає cash, якщо ціна потім падає – у збитку) і має прямий стимул котирувати справедливую ціну. У passive моделі емітент таких стимулів не має – там його роль курадорська. Платформа стимулює використання active_mm моделі для активних активів (бо там емітент сам забезпечує ліквідність) і passive – для пасивних (бо там платформа все робить сама).

Узгодження price impact з active_mm. У passive моделі емітент може опційно увімкнути авто-регулювання ціни через price impact – механізм, у якому ціна автоматично реагує на торгові операції. У active_mm моделі price impact

автоматично вимикається – це принципове рішення, бо у `active_mm` цінами повністю керує сам емітент через `bid/ask`. Технічно це реалізовано перевіркою у функції `applyPriceImpact`: якщо `pricing_model='active_mm'`, функція повертається без дії. Це усуває потенційний конфлікт двох механізмів ціноутворення в одному активі. Лімітні ордери продовжують працювати в обох моделях – для `passive` активів fallback на пул йде за `current_price`, для `active_mm` – за `ask` (для BUY) або `bid` (для SELL).

Окремої уваги заслуговує архітектура самого механізму `price impact` в `passive` моделі. Замість єдиного симетричного коефіцієнта (як було у первинній реалізації, де купівля підвищувала ціну на `pct%`, а продаж знижував на ту саму величину), у фінальній версії передбачено два окремі параметри: `price_impact_buy_pct` і `price_impact_sell_pct`. Перший задає відсоток зростання ціни за одну одиницю при купівлі трейдером, другий – відсоток зниження ціни за одну одиницю при продажі. Така структура дозволяє створювати `asymmetric impact`, тобто різну реакцію ціни залежно від напрямку торгової операції. Наприклад, емітент може встановити `buy_pct=0.2%` і `sell_pct=0.05%`: тоді купівля десяти одиниць активу підвищить ціну на 2%, а продаж тих самих десяти одиниць знизить її лише на 0.5%. Це краще відображає реальну поведінку ринкової мікроструктури, де реакція ціни на попит часто буває різкішою за реакцію на пропозицію (явище FOMO – fear of missing out, коли трейдери активно купують при сприйнятому дефіциті, але повільніше продають при надлишку).

У реалізації функція `applyPriceImpact` обирає правильний коефіцієнт на основі параметру `side`: для `buy` використовується `price_impact_buy_pct`, для `sell` – `price_impact_sell_pct`. Формула розрахунку зберігається лінійною: $\text{newPrice} = \text{oldPrice} \times (1 + \text{direction} \times \text{qty} \times \text{pct} / 100)$, де `direction` дорівнює +1 для купівлі і -1 для продажу. Якщо обраний коефіцієнт дорівнює нулю – функція повертається без зміни ціни (це дозволяє емітенту, наприклад, активувати `impact` тільки в один бік: вмикати реакцію ціни на купівлю, але блокувати її на продаж, що корисно для активів з низькою ліквідністю на стороні попиту). Симетричний варіант залишається доступним: якщо емітент вводить однакові значення `buy_pct =`

sell_pct, поведінка ідентична попередній моделі. Для зворотної сумісності міграція бази даних автоматично перейменовує стару колонку price_impact_pct у price_impact_buy_pct і копіює це значення у нову колонку price_impact_sell_pct, забезпечуючи плавний перехід без втрати даних для існуючих активів.

Емітент при створенні активу може одразу вказати початковий запас (initial_supply, за замовчуванням 10000 одиниць) і – для active_mm моделі – початковий cash reserve, який списується з його особистого балансу. Це робить процес лістингу повноцінним і одночасним: за одну операцію створюється актив з усіма параметрами, призначається емітент (для issuer-користувача – автоматично), журналюється початкове котирування у quotes_log, init-запис у supply_log і опційно – у cash_reserve_log. Така цілісність важлива для аудиту: повний шлях кожної одиниці активу від моменту створення до поточного власника можна відновити з журналів.

Найскладнішим виявився торговий модуль. У файлі ordersController.js реалізовано повний цикл життя ордеру – від створення до виконання – і все це обгорнуте в MySQL-транзакції, щоб уникнути будь-яких неконсистентних станів. Для ринкового ордеру алгоритм такий:

1. Отримання блокування рядка користувача (FOR UPDATE) для запобігання race conditions;
2. Перевірка достатності балансу (для buy) або кількості активу (для sell);
3. Вставка запису ордеру зі статусом "open";
4. Виконання функції executeOrder: оновлення балансу користувача, портфеля, статусу ордеру та запис у таблицю trades;
5. Генерація сповіщення (notifications) про виконання ордеру;
6. Запис усіх змін балансу у balance_transactions;
7. COMMIT транзакції або ROLLBACK у разі будь-якої помилки.

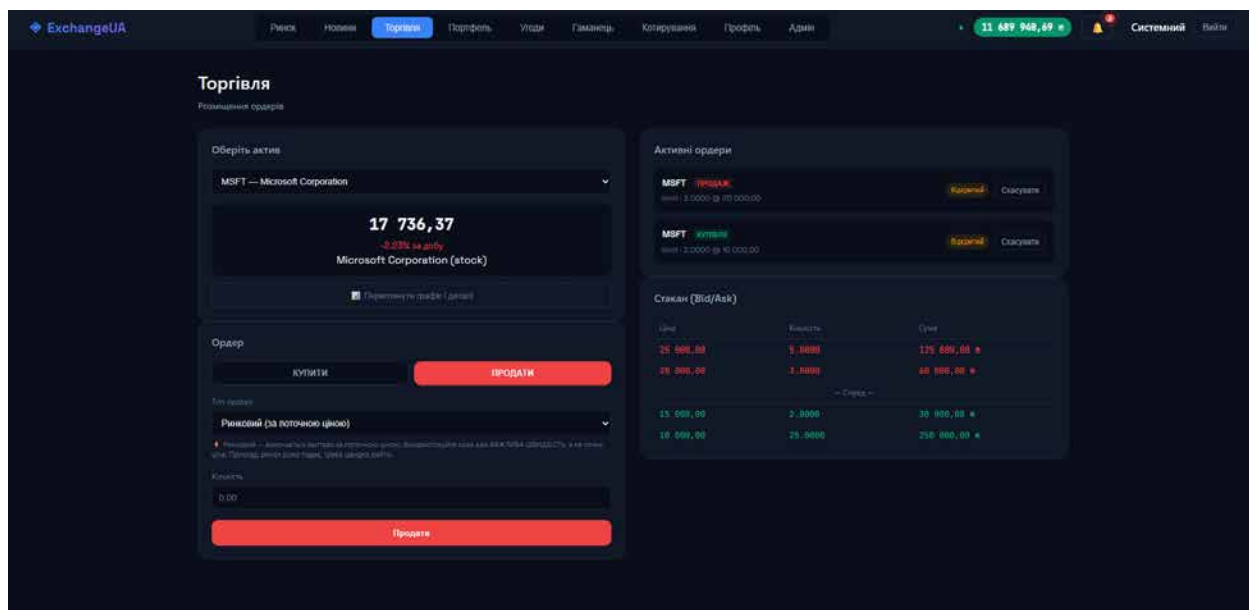


Рисунок 2.12. Форма розміщення ордеру та стакан замовлень (Order Book)

Комісія платформи становить 0.1% від суми угоди ($COMMISSION_RATE = 0.001$). Утримується вона з покупця. Така схема відповідає поширеній на біржах maker-taker моделі [44]. Лімітні ордери після створення зберігаються зі статусом "open" і чекають у книзі заявок; користувач може скасувати такий ордер у будь-який момент до виконання.

У файлі `portfolioController.js` – усе, що стосується угод (рис. 2.10.), портфеля (рис. 2.13.), балансу (рис. 2.14.) та сповіщень (рис. 2.15.). `GET /portfolio` підтягує позиції користувача з таблиці `portfolio`, приєднує поточні ціни активів і рахує нереалізований P&L як різницю між поточною ціною та середньою ціною купівлі. `GET /balance` повертає актуальний баланс і останні 20 балансових транзакцій (поповнень, виведень, комісій). `POST /balance/deposit` і `POST /balance/withdraw` обробляють поповнення та виведення коштів у транзакційному режимі. `GET /trades` повертає історію угод трейдера. `GET /notifications` та `POST /notifications/read-all` керують сповіщеннями.

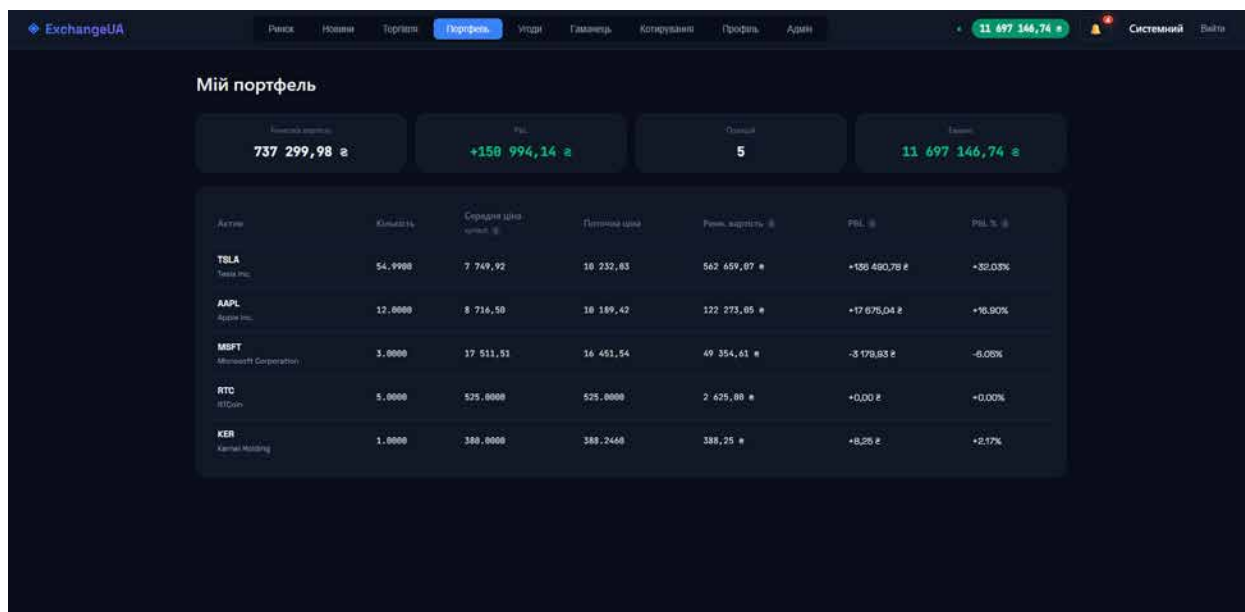


Рисунок 2.13. Екран портфеля з P&L по кожній позиції

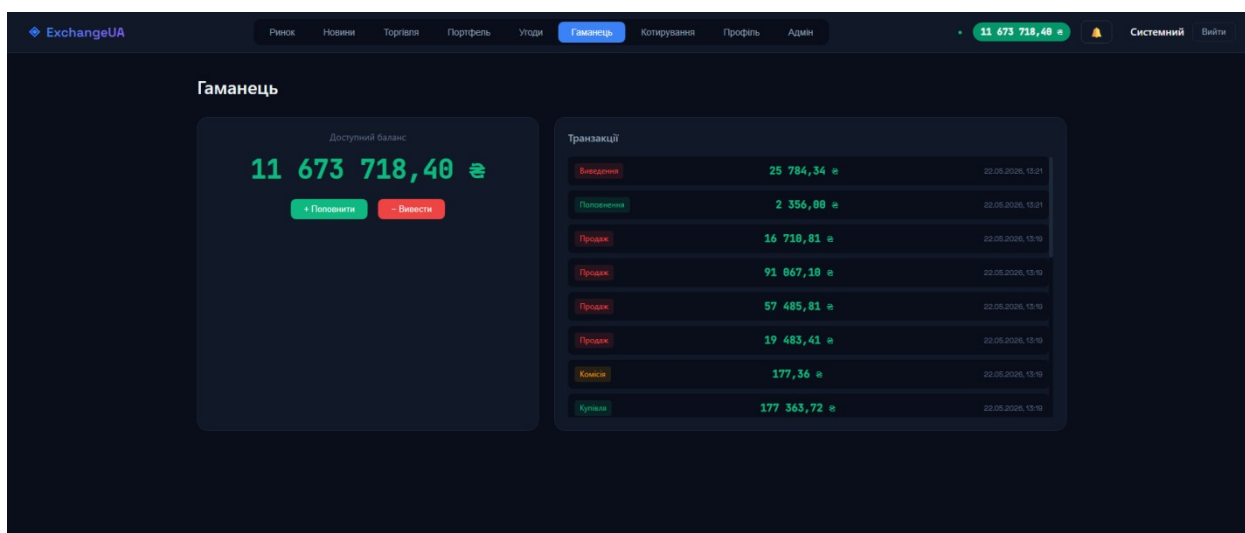


Рисунок 2.14. Сторінка "Гаманець" з історією балансових транзакцій

Сповіщення з'являються автоматично при подіях, важливих для користувача: виконанні ордера (order_filled), його скасуванні (order_cancelled), поповненні чи виведенні коштів. Кожен запис у таблиці notifications має тип, заголовок, текст і опційне JSON-поле data_json з контекстом події. Клієнт підвантажує сповіщення через GET /notifications при відкритті відповідної панелі, а через POST /notifications/read-all одним запитом позначає всі як прочитані.

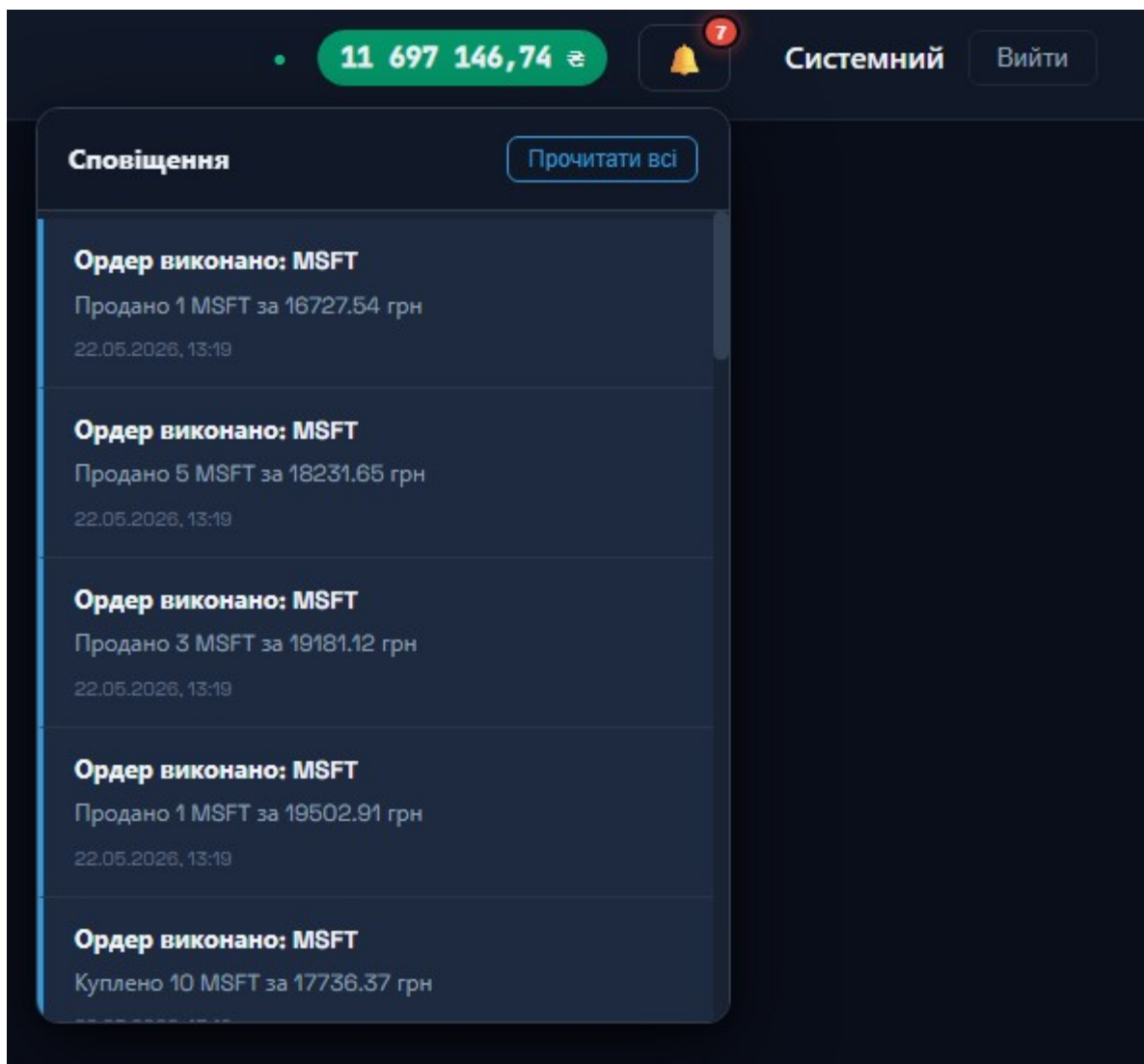


Рисунок 2.15. Панель сповіщень користувача

Для панелі адміністратора яку зображено на рис. 2.16. передбачено окремий шар ендпоінтів. GET /admin/stats – агрегована статистика платформи: скільки всього користувачів, обсяг торгів, кількість виконаних угод і відкритих ордерів. GET /admin/users – повний список користувачів з фільтрами; через PUT /admin/users/:id можна змінити роль, статус або баланс конкретного користувача. Усі ці маршрути закриті за requireRole('admin').

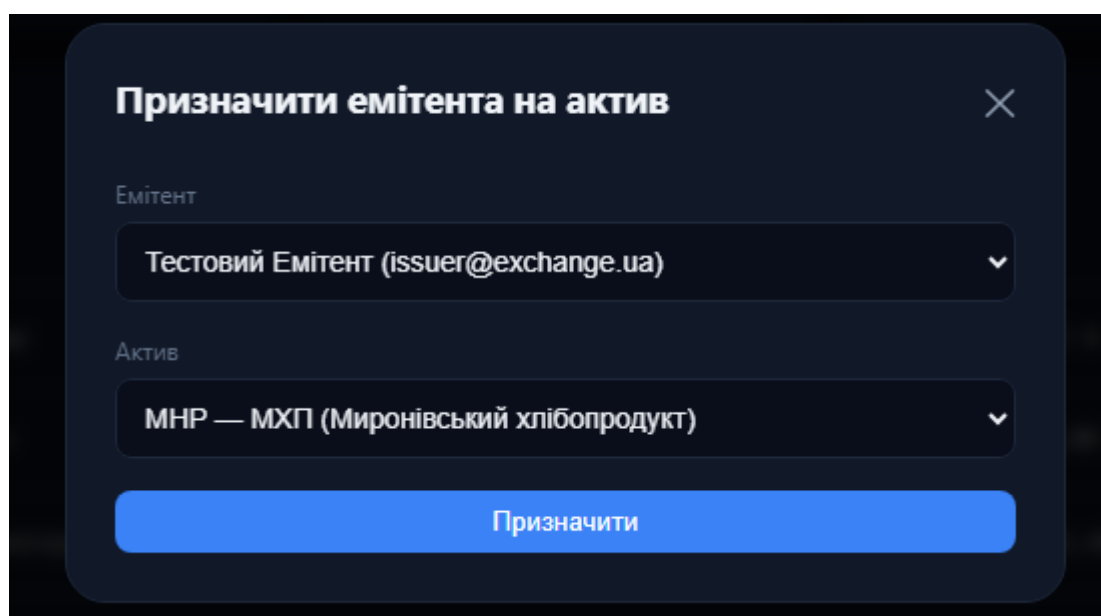
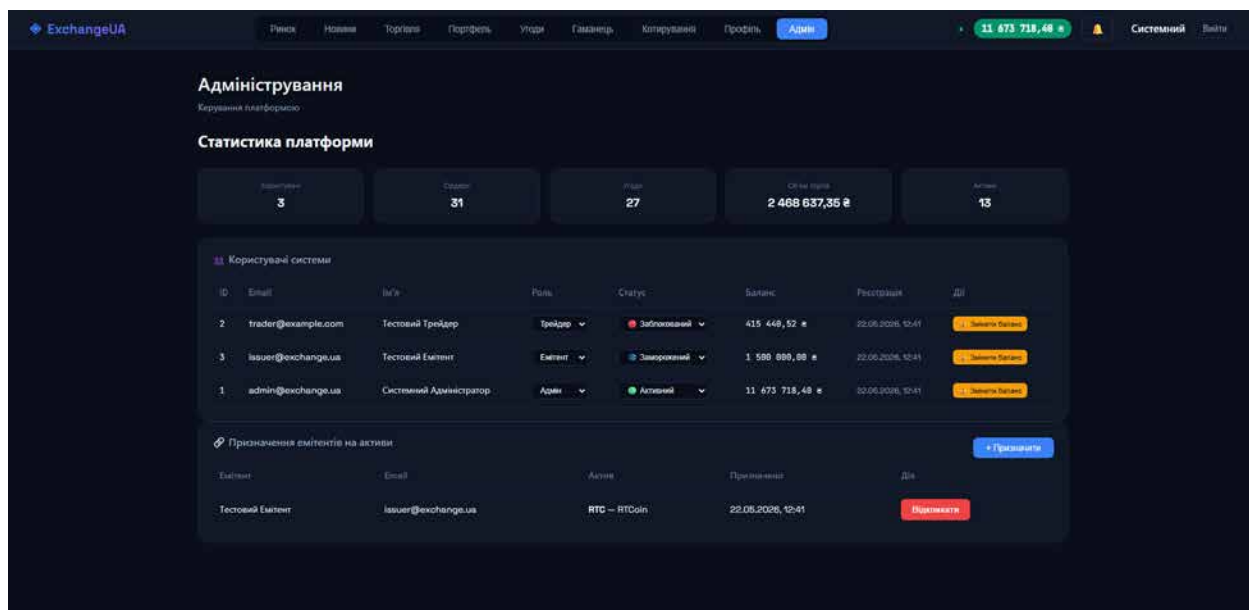


Рисунок 2.16. Адміністративна панель управління платформою

2.4. Забезпечення безпеки та довіри на платформі

Для будь-якої фінансової платформи безпека – це не "одна з вимог", а критична властивість. Зламана біржа втрачає довіру миттєво й назавжди. Тому в ExchangeUA реалізовано багаторівневий захист – і на транспортному, і на прикладному рівні, і на рівні даних.

Паролі в БД не зберігаються відкритим текстом – ніколи. Використовується bcryptjs з cost-factor 12, що відповідає приблизно 4 096 раундам хешування. У цифрах: навіть потужний GPU витратить кілька секунд на перевірку одного пароля. Помножте це на мільйони можливих варіантів – і будь-яка спроба brute-force атаки стає економічно не вигідною [28]. Порівняння хешу під час входу робиться через bcrypt.compare() – функція стійка до timing-атак, тобто атакувальник не зможе зробити висновок про правильність пароля за часом відповіді.

JWT реалізую без жодної зовнішньої бібліотеки – у файлі utils/jwt.js. Підпис генерується через crypto.createHmac('sha256', secret). Чому так. У популярних JWT-бібліотек (зокрема, у jsonwebtoken версій до 9.0) свого часу знаходили вразливості, які дозволяли підробляти токени за відсутності перевірки алгоритму. Власна реалізація на 30 рядків коду повністю прозора і не залежить від чужого life-cycle. Сам токен мінімальний – лише userId і role. TTL – одна година. Cookies налаштовані з httpOnly:true (JavaScript їх не побачить) і secure:true в продакшні (передаються тільки по HTTPS).

Жодного SQL-запиту в коді не складено через конкатенацію рядків. Усі параметри підставляються через placeholder'и (prepared statements) бібліотеки mysql2. Тому SQL-ін'єкція – а це вразливість №3 у списку OWASP Top 10 [28] – у цьому проєкті виключена за конструкцією.

Таблиця 2.3. HTTP-заголовки безпеки, налаштовані Helmet.js

HTTP-заголовок	Значення	Захищає від
Content-Security-Policy	вимкнено (планується)	XSS через зовнішні ресурси
X-Frame-Options	DENY	Clickjacking атак
продовження табл.. 2.3.		
HTTP-заголовок	Значення	Захищає від
X-Content-Type-Options	nosniff	MIME-type sniffing
Referrer-Policy	strict-origin-when-	Витоку URL через Referer

	cross-origin	
Permissions-Policy	Обмеження API браузера	Зловживання браузерними API
Strict-Transport-Security	max-age=31536000	Downgrade до HTTP

Кожна фінансова операція – будь то виконання ордеру, поповнення чи виведення – обгорнута в одну MySQL-транзакцію з явними BEGIN / COMMIT / ROLLBACK. Перед списанням коштів роблю SELECT ... FOR UPDATE на рядок користувача – це блокує паралельні запити від того ж клієнта і прибирає race conditions. У результаті класична для бірж ситуація "подвійного витрачання", коли одні й ті самі гроші встигли потратитися на дві угоди, тут неможлива.

Висновки до розділу 2. Архітектуру системи побудовано за класичним тришаровим шаблоном із чітким відокремленням представлення, бізнес-логіки та доступу до даних. База даних приведена до третьої нормальної форми, містить 14 таблиць (включаючи issuer_assignments для зв'язку емітентів з активами, quotes_log для журналювання котирувань і supply_log для аудиту змін ринкової пропозиції активів). Індексна стратегія розрахована під найчастіші реальні запити. Усі ключові модулі – автентифікація, торговий механізм, портфель, адміністрування, сповіщення, управління інвентарем – реалізують повний цикл відповідних функцій. Модель безпеки покриває рівні транспорту, застосунку та даних і відповідає вимогам OWASP Top 10 та галузевим стандартам фінансових систем.

РОЗДІЛ 3 РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ ПЛАТФОРМИ

3.1. Програмна реалізація системи: інтерфейс, серверна логіка, інтеграції

Реалізація програмного продукту здійснювалася поетапно відповідно до архітектурного плану, розробленого у другому розділі роботи. Підсумкова кодова база налічує близько 22 файлів, включаючи тестові модулі, із загальним обсягом приблизно 360 КБ. Незважаючи на відносно невеликий обсяг програмного коду, проєкт характеризується значною насиченістю логіки та функціональних взаємозв'язків. Нижче наведено короткий огляд ключових компонентів системи.

Точкою входу слугує `src/app.js`. Послідовність ініціалізації тут прямолінійна: підключаємо `dotenv` для змінних середовища, налаштовуємо `middleware Express` у потрібному порядку (`helmet`, `cors`, `morgan`, `express-session`, `express.json`), додаємо роутер, піднімаємо `WebSocket`-сервер на тому самому `HTTP`-сервері і нарешті стартуємо його на порту з `.env`. Такий лінійний підхід зручний для відладки – будь-яку проблему з ініціалізацією видно зразу за стеком запуску.

Файл `config/db.js` створює пул з'єднань через `mysql2/promise`. Налаштування досить стандартні: `connectionLimit: 10` (можна піднімати, але 10 з лишком вистачає для тестового навантаження), `waitForConnections: true` (якщо всі з'єднання зайняті – запит чекає на черзі замість одразу відхилитися). Пул, а не одиночне з'єднання, – це обов'язкова вимога для будь-якого `Node.js` застосунку, що очікує конкурентних запитів.

Серцем торгового модуля є функція `executeOrder` в `ordersController.js`. Вона завжди викликається в контексті відкритої транзакції (саму транзакцію відкриває код, що цю функцію викликає) і виконує такі дії:

1. Оновлення балансу продавця (+ totalValue – commission) або покупця (- totalValue – commission);
2. Вставлення або оновлення запису портфеля через INSERT ... ON DUPLICATE KEY UPDATE з перерахунком середньої ціни (avg_buy_price);
3. Запис угоди в таблицю trades;
4. Оновлення статусу ордеру до "filled" та заповнення avg_fill_price і filled_qty;
5. Запис двох транзакцій у balance_transactions (trade_buy та commission для покупця);
6. Створення сповіщення order_filled.

Середня ціна покупки розраховується за формулою зваженого середнього:

$$avg_{new} = (qty_{old} \times avg_{old} + qty_{new} \times price_{new}) / (qty_{old} + qty_{new})$$

Така формула важлива саме для того, щоб у разі кількох послідовних поповнень позиції середня ціна купівлі рахувалася правильно з урахуванням обсягу кожної з угод, а не як проста арифметична середня.

utils/websocket.js реалізує WebSocket-сервер для трансляції актуальних цін активів клієнтам. При підключенні нового клієнта сервер відразу надсилає поточні ціни всіх активів (initial prices). Потім кожні 2000 мс таймер транслює актуальний стан БД усім підключеним клієнтам. Важливо: сам сервер цін не модифікує – він лише читає та надсилає. Зміна цін відбувається через котирування емітентом/адміном (POST /assets/:id/quote) або через виконання угод (recordTradePrice після кожного matching). Це принципово відрізняється від поширених тестових біржових систем із симуляцією цін.

Оновлені ціни одночасно зберігаються в БД (UPDATE assets SET current_price) і розсилаються через WebSocket-broadcast усім поточно підключеним клієнтам. На випадок клієнта з обмеженим каналом передбачено підписку на конкретні символи (subscribe-повідомлення) – він отримує оновлення лише для тих активів, які йому реально потрібні.

Уся клієнтська частина – у файлі public/js/app.js (близько 89 КБ після всіх доопрацювань). SPA-маршрутизація реалізована через hash-фрагменти URL: #market, #trade, #portfolio, #wallet, #news, #profile, #admin, #quotes. На кожну

зміну хешу спрацьовує функція рендеру конкретної сторінки – вона показує відповідний HTML-шаблон і підтягує дані через fetch API.

WebSocket-клієнт підключається до `ws://host/ws` ще під час завантаження сторінки. Якщо з'єднання раптом обривається – спрацьовує автоматичне перепідключення з фіксованим інтервалом у три секунди. Цей варіант обрано як простий і достатній для поточного навантаження; перехід на експоненційний backoff залишається у напрямках подальшого розвитку. Цінові апдейти оновлюють DOM точково – через `document.getElementById` я знаходжу лише ті елементи, що змінилися, без перерендеру цілої сторінки.

Файл `utils/initDb.js` відповідає за початкову ініціалізацію БД. Він читає `schema.sql` і виконує його, після чого в базі з'являються 14 таблиць і початковий каталог з 12 активами. Облікові записи (`admin/trader/issuer`) створюються окремим скриптом `setupUsers.js` – це навмисно винесено в окремий крок, щоб ініціалізація схеми була відокремлена від створення користувачів. Усе зроблено ідемпотентно – `CREATE TABLE IF NOT EXISTS` та `INSERT IGNORE`. Це означає, що скрипт можна спокійно запустити вдруге чи втретє – нічого не зламається, нічого не втратиться. npm-скрипт `init-db` оформлений як одна команда – `npm run init-db`.

Уся конфігурація – змінні середовища у файлі `.env`. Підхід відповідає рекомендаціям Twelve-Factor App [42]: жодних хардкоду секретів у самому коді, тільки в зовнішньому конфізі, який не комітється в Git. У репозиторії є файл `.env.example` з коментарями до кожної змінної – це шаблон для розгортання. Для продакшн-сетапу логічно поставити PM2 як менеджер Node.js-процесів і Nginx як зворотний проксі з підтримкою WebSocket-апгрейду.

3.2. Тестування працездатності та навантажувальне дослідження

Тестування я організував на трьох рівнях. Перше – ручне функціональне: пройшов основні сценарії в браузері від реєстрації до виконання ордерів, фіксує кроки і результати. Друге – юніт-тести для критичних модулів: JWT, бізнес-логіка ціноутворення й комісій, перевірки безпеки. Третє – інтеграційні тести REST API. Для автоматизації написано власний легковагий тестовий харнес на основі вбудованого модуля assert – простий wrapper з функцією `test(name, fn)`, що ловить винятки і веде підрахунок пройдених/провалених. Перевага такого підходу – повна відсутність зовнішніх залежностей: тестова інфраструктура важить близько 20 рядків коду, не потребує `package.json` змін і запускається однією командою `npm test`.

За підсумками тестування маю 160 тестів, розбитих на чотири категорії. Для кожного зафіксовано вхідні дані, передумови, очікуваний і фактичний результат. Усі тести детерміновані: вони не залежать від мережі, реального часу чи реальної БД. Це означає, що тести повторювані: один і той самий прогон завжди дає той самий результат, що важливо для CI/CD у майбутньому. (рисунки тестувань у Додатку Е)

Таблиця 3.1. Результати автоматизованого тестування за категоріями

Модуль	К-сть тест-кейсів	Пройдено	Провалено	Покриття
JWT (підпис, верифікація, TTL)	8	8	0	100%
Бізнес-логіка (комісія, P&L, статуси)	100	100	0	100%
Безпека та валідація (RBAC, bcrypt, XSS)	36	36	0	100%
API-інтеграція (роутинг, авторизація, RBAC)	16	16	0	100%
Разом	160	160	0	100%

Перевірка захисту від SQL-ін'єкцій: у поле email під час входу було введено рядок: `' OR 1=1 --`. Система коректно повернула HTTP 401 та не виконала

несанкціонованого входу. Перевірка RBAC: спроба доступу до `/api/admin/stats` від облікового запису з роллю `trader` повернула HTTP 403. Перевірка JWT: токен з підробленим підписом відхилено з HTTP 401.

Усі чотири тестові набори запускаються однією командою `npm test`, але кожен виконується в окремому Node-процесі — це гарантує ізоляцію між наборами. Для повторюваності результатів використано фіксовані вхідні дані та детермінований підхід без залежності від мережі та зовнішніх сервісів. Тестове середовище: Ubuntu 24.04 LTS, Node.js 20.x, без додаткових залежностей понад ті, що вже включені в `package.json` основного застосунку.

Таблиця 3.2. Деталізація тестових сценаріїв за категоріями

Показник	Значення	Ціль (вимога)	Статус
Тести JWT-аутентифікації	8	≥ 8 (підпис, верифікація)	Виконано ✓
Тести бізнес-логіки	100	≥ 15 (комісія, P&L)	Виконано ✓
Тести безпеки та валідації	36	≥ 20 (RBAC, bcrypt, XSS)	Виконано ✓
Тести API (інтеграційні)	16	≥ 12 (роутинг, авторизація)	Виконано ✓
Загальна кількість тестів	160	≥ 50	Виконано ✓
Покриття пройденими	100%	$\geq 95\%$	Виконано ✓
Перевірок ін'єкції та підробки токена	5	≥ 3 атаки	Виконано ✓
Час повного прогону тестів	< 10 сек	≤ 10 сек	Виконано ✓

Підсумок такий: усі 160 тестів проходять успішно, покриття пройденими – 100%, жодного провалу. Цікаве спостереження – переважна частка часу прогону припадає на тести bcrypt-хешування з `cost=12`, де одна операція займає сотні мілісекунд (на середньому залізі – близько 100-300 мс залежно від CPU). Це не дефект, а запланована характеристика: саме така тривалість і робить bcrypt стійким до brute-force-атак. Якби пароль перевірявся за 1 мс – атакуючий міг би перебирати мільйони варіантів за секунду. Решта тестів (JWT, бізнес-логіка, API) виконуються швидко – увесь набір з 36 безпекових тестів триває близько 2

секунд, інші три набори – менше за 250 мс сумарно. Загальний прогон укладається в декілька секунд.

Хочу зазначити обмеження: проведене тестування перевіряє логічну коректність коду і його стійкість до основних класів атак, але не моделює високу конкурентну активність на рівні MySQL під реальним багатокористувацьким навантаженням. Це окрема велика задача, для якої потрібні інструменти на кшталт Apache JMeter чи k6. Я виніс це як один з пунктів подальшого розвитку. Для production-сетапу також логічно додати Redis-кеш для часто читаних даних (ціни активів, стакан ордерів) і горизонтальне масштабування Node.js через PM2 cluster mode.

Таблиця 3.3. Перелік перевірених сценаріїв безпеки та валідації

Сценарій перевірки	Метод тестування	Очікуваний результат	Статус
SQL-ін'єкція у поле email при вході	POST /api/auth/login з рядком ' OR 1=1 --	HTTP 401, без авторизації	✓
RBAC: trader - admin-only endpoint	GET /api/admin/stats з JWT trader	HTTP 403 Forbidden	✓
Підроблений JWT-токен	Bearer з невірним підписом	HTTP 401, Invalid signature	✓
Прострочений JWT-токен	Bearer з exp у минулому	HTTP 401, Token expired	✓
продовження табл..3.3.			
Сценарій перевірки	Метод тестування	Очікуваний результат	Статус
Підробка ролі в payload JWT	Зміна trader - admin у body	HTTP 401, Invalid signature	✓

XSS через input у назві активу	Виклик escapeHtml() на виведенні	Скрипт екрановано	✓
Реєстрація з коротким паролем	POST /api/auth/register, password="123"	HTTP 400, валідація	✓
Захист від запитів без авторизації	GET /api/portfolio без Bearer	35	✓

Наведена таблиця підсумовує результати ручного тестування безпеки: на кожен з основних класів атак (SQL-ін'єкція, обхід RBAC, підробка JWT, XSS, слабкі паролі, неавторизований доступ) платформа реагує очікуваним чином – повертає коди HTTP 401 або 403 і відхиляє шкідливий запит, не виконуючи його логіки.

3.3. Економіко-експлуатаційне обґрунтування: модель монетизації, собівартість, рентабельність

Підхід до монетизації я свідомо обрав мінімалістичний. У моделі – одне джерело надходжень, транзакційна комісія 0.1% від обсягу кожної угоди. Жодних преміум-тарифів, жодних плат за вхід, жодної реклами всередині інтерфейсу.

І це не наївність, а позиція: коли витрати трейдера на платформу мінімальні, він не шукає, куди піти, і не закриває акаунт після першого ж невдалого тижня. Підсумовуючи, платформа отримує те саме, що й користувач – час.

Класичні для маркетплейсів канали (преміум-підписки, плата за лістинг, рекламні розміщення [45]) у поточній архітектурі свідомо не закладено. Якщо у майбутньому з'явиться потреба і користувачі будуть готові доплачувати за щось більше – додати ці модулі технічно нескладно, але для базової версії перевагу віддано простоті й передбачуваності правил гри.

У цій моделі вигоду отримують обидві сторони – і платформа, і трейдери; класична win-win-схема [45]. Трейдер платить менше за угоду, ніж на більшості великих бірж (де комісії стартують від 0.2–0.5%), тому залишає собі більший відсоток прибутку від кожної операції. Це безпосередньо впливає на тривалість його активності: чим менше витрат, тим довше людина торгує.

Платформа, у свою чергу, отримує лояльних користувачів – таких, що повертаються щодня, рекомендують сервіс знайомим і органічно нарощують спільноту. Зростає кількість угод – зростає й сумарний дохід від комісії, навіть якщо ставка з кожної угоди залишається мінімальною. Це і є та сама "довга гра", про яку говорять у класичних роботах із платформної економіки [7]: повільніше на старті, але стабільніше і з ширшою аудиторією на дистанції.

Як це працює технічно. Коли користувач купує або продає актив, з суми угоди утримується 0.1%. Логіка реалізована у функціях `executeOrder` і `settleTrade` модуля `ordersController.js`. Покупець і продавець платять симетрично; обидві суми пишуться в окремі поля `buyer_fee` та `seller_fee` таблиці `trades` – на майбутнє це дозволить вводити асиметричну сітку без перебудови БД (наприклад, нижчу ставку для `maker`-ордерів, які додають ліквідність). Сама ставка винесена в одну константу `COMMISSION_RATE` – при потребі змінюється точково, без зачіпання іншого коду.

Що це означає в цифрах. На старті, при щоденному обсязі торгів близько 250 000 грн (реалістичний сценарій для нової української платформи з ставкою комісії нижчою за конкурентів), річний дохід складе приблизно 91 250 грн. У міру зростання спільноти й нарощування обсягів цифра пропорційно збільшується: 1 000 000 грн/день дають вже близько 365 000 грн/рік, а 4 000 000 грн/день у третій рік – близько 1 460 000 грн/рік.

Для порівняння, українська біржа WhiteBIT за перші три роки роботи (2018–2021) перейшла від нульового обсягу до приблизно 1 млн доларів на день; ExchangeUA у моделі ставить набагато скромніший таргет – 100 тис. доларів еквіваленту на третій рік, що складає лише 0.5% від поточних обсягів WhiteBIT і виглядає досяжним при правильній маркетинговій стратегії та органічному зростанні спільноти. Жодних інших стягнень з користувачів не передбачено.

Чому я не закладаю в модель преміум-підписки чи плату за лістинг. На більшості бірж саме такі додаткові тарифи дають значну частку доходу. Я свідомо відмовляюся від них на ранньому етапі – з двох причин. Перша: будь-яка платна опція створює бар'єр входу і відштовхує частину аудиторії саме тоді, коли платформа найбільше потребує органічного зростання. Друга: довіра до молодого фінтех-продукту будується роками; нав'язлива монетизація на старті псує цю довіру швидше, ніж її будує функціонал. Стратегія простіша й чесніша – спочатку наростити критичну масу активних користувачів і об'єми торгів, а вже потім, якщо аудиторія цього захоче, додавати платні опції за окрему згоду. У програмному коді закладено лише ті механізми, що працюють зараз; перспективні модулі чесно винесено в розділ напрямів подальшого розвитку. Що отримує користувач у такій моделі. Окрім нижчої комісії, у нього немає прихованих платежів – за вхід, за виведення коштів, за зберігання портфеля. Усе, що нараховується платформою, фіксується у відкритому журналі `balance_transactions` і доступне через сторінку "Гаманець". Жодного маркетингового тексту про "ексклюзивні умови для VIP" – бо таких умов просто не існує. У підсумку трейдер може спокійно тестувати власну стратегію, не переймаючись постійним перерахунком собівартості: 0.1% – це 0.1%, незалежно від обсягу, статусу акаунта чи дня тижня. Саме така прозора цінова політика і дає реальну користь користувачу.

Таблиця 3.4. Прогноз доходів платформи ExchangeUA на 3 роки

Джерело доходу	Рік 1	Рік 2	Рік 3
Транзакційна комісія	91 250 грн	365 000 грн	1 460 000 грн
Разом доходів	91 250 грн	365 000 грн	1 460 000 грн

Трудовитрати на розробку оцінив методом Function Point Analysis, спираючись на фактичний час, який витратив на кожен з модулів.

Таблиця 3.5. Розрахунок собівартості розробки

Етап розробки	Трудомісткість (год)	Ставка (грн/год)	Вартість (грн)
---------------	----------------------	------------------	----------------

Аналіз вимог та проектування	40	200	8 000
Проектування БД та схема	20	200	4 000
Backend розробка (Node.js/Express)	80	200	16 000
Frontend розробка (Vanilla JS SPA)	60	200	12 000
WebSocket та real-time модуль	20	200	4 000
Безпека та middleware	16	200	3 200
Тестування та виправлення помилок	30	200	6 000
Документування	14	200	2 800
Разом	280 год	-	56 000 грн

Таблиця 3.6. Операційні витрати на рік 1

Стаття витрат	Сума/рік (грн)	Примітка
Хостинг (VPS 4 vCPU, 8 GB RAM)	18 000	DigitalOcean/Hetzner
MySQL managed database	7 200	Або self-hosted без витрат
SSL-сертифікат	0	Let's Encrypt (безкоштовно)
Резервне копіювання	3 600	Щоденні бекапи до S3
Технічна підтримка (part-time)	60 000	20 год/місяць × 250 грн
Маркетинг	24 000	2 000 грн/місяць
Разом операційних витрат	112 800	-

Повна собівартість першого року складає 168 800 грн – 56 000 грн на одноразову розробку плюс 112 800 грн операційних витрат (хостинг, MySQL, бекапи, мінімальна підтримка й маркетинг).

Якщо подивитись на цифри в лоб – модель першого року майже виходить у нуль. Прогнозний дохід від комісії за реалістичним сценарієм складає 91 250 грн при 250 тис. грн денного обсягу, тобто покриває близько 81% операційних витрат – фактично платформа працює на самоокупність вже наприкінці першого року. У другому році цифра зростає до 365 000 грн (324% операційних витрат – окупність плюс прибуток понад 252 тис. грн), у третьому – до 1 460 000 грн (1 294%

операційних витрат, чистий прибуток понад 1.3 млн грн без додаткових інвестицій).

Це усвідомлений вибір. На цьому етапі завдання – не повернути вкладене за 12 місяців, а наростити аудиторію й об'єми торгів до критичної маси, при якій навіть низька ставка 0.1% починає давати стабільний потік. Класичні розрахунки ROI і terms of payback тут описують не реальну економіку проекту, а лише його технічну собівартість на старті; справжній результат вимірюється кількістю активних трейдерів через 18–24 місяці, а не балансом першого кварталу.

Якщо порівняти із собівартістю аналогічних комерційних рішень – побудова біржової платформи з нуля у профільних компаніях коштує від 500 тис. до 2 млн грн (за оцінками з типових кошторисів фінтех-стартапів). На цьому фоні 56 000 грн на власну розробку – це показник ефективності обраного стеку Node.js + MySQL + Vanilla JS, відмови від важких комерційних фреймворків та свідомо низької ставки розробника на старті (студентська самостійна робота без офіційного працевлаштування). Іншими словами, низька собівартість входу і свідомо невисока ставка комісії разом створюють збалансовану конфігурацію: користувачі отримують найдешевший на ринку доступ до інструментів торгівлі, а платформа – простір для довгого нарощування бази без тиску на швидку монетизацію.

З огляду на масштабованість, є кілька рівнів, які можна задіяти послідовно. Перший – вертикальне масштабування: збільшення CPU/RAM сервера лінійно тягне пропускну здатність приблизно до 500 одночасних користувачів. Другий – горизонтальне масштабування Node.js через PM2 cluster mode, що використовує всі ядра процесора без змін у коді. Третій – розподілений MySQL (MySQL cluster або AWS Aurora) – це вже етап, коли об'єм досягне ~10 000 транзакцій/хвилину. Четвертий – Redis-кеш для часто читаних даних (ціни, стакан, сесії), що зніме з MySQL приблизно 60–70% навантаження.

У підсумку поточна архітектура без жодних змін витримує приблизно до 500 одночасних активних трейдерів. Після впровадження Redis-кешу та горизонтального масштабування ця цифра зростає приблизно до 5 000.

Подальший крок – перехід на мікросервіси – дозволить обслуговувати десятки тисяч одночасних з’єднань, але це вже окремий проєкт.

Висновки до розділу 3. Програмну реалізацію виконано в повній відповідності до спроектованої архітектури. Розроблено й успішно прогнано чотири набори автоматизованих тестів – загалом 160 перевірок у категоріях JWT, бізнес-логіка, безпека та API-інтеграція; покриття пройденими – 100%. Окремо перевірено стійкість системи до основних класів атак: SQL-ін’єкції, XSS, підробки JWT, обходу RBAC, а також валідацію зміни пароля й перевірку прав на видалення активів. У всіх випадках платформа повертала очікувані коди (HTTP 401/403) і відхиляла шкідливі запити. Економічний аналіз показав: модель монетизації побудована на єдиному реальному джерелі – комісії 0.1% з кожної угоди, без преміум-тарифів і прихованих платежів. У такому форматі перший рік виходить майже на самоокупність (дохід 91 тис. грн покриває близько 81% операційних витрат у 112 тис. грн), другий – стає прибутковим з чистим результатом понад 250 тис. грн, третій – генерує понад 1.3 млн грн прибутку. Це підтверджує доцільність win-win-стратегії: мінімальна вартість обслуговування для трейдера підвищує його лояльність, збільшує обсяги торгів на дистанції і формує стабільне джерело надходжень без агресивної монетизації на старті.

ВИСНОВКИ

У дипломній роботі поставлене завдання розв'язано в повному обсязі: спроектовано та реалізовано веборієнтовану інформаційну систему ExchangeUA – повнофункціональну багатосторонню цифрову платформу для торгівлі фінансовими активами.

За результатами виконаної роботи зроблено такі висновки:

1. У першому розділі досліджено теорію багатосторонніх цифрових платформ і виявлено три фактори, які найбільше впливають на життєздатність біржі: мережеві ефекти, ліквідність та довіра користувачів до системи. ExchangeUA класифіковано як централізовану біржу з відкритим вихідним кодом – це поєднання дозволяє контролювати поведінку системи (на відміну від DEX) і водночас залишає її прозорою (на відміну від комерційних рішень).

2. Аналіз шести представницьких аналогів (Binance, Interactive Brokers, Steam Market, G2G, LocalBitcoins, OpenSea) показав цікаву річ: жоден із них не закриває всі чотири характеристики водночас – підтримку різних класів активів, вбудований matching engine, real-time трансляцію через WebSocket та відкритий код. Саме на стику цих чотирьох характеристик і знайшлося місце для ExchangeUA.

3. На основі вимог сформовано три їх групи – функціональні (8 категорій), нефункціональні (6 показників) та безпекові (5 пунктів за OWASP Top 10). Це стало "контрольним списком" при виборі стеку. Підсумкове рішення: серверна частина – Node.js з Express, БД – MySQL 8, real-time – WebSocket через бібліотеку ws, клієнт – без жодного фреймворку, на чистому JavaScript у форматі SPA. Кожен пункт у цьому виборі має своє обґрунтування, описане в підрозділі 1.4.

4. Архітектура системи була реалізована відповідно до трирівневого підходу, що включає рівень представлення, рівень бізнес-логіки та рівень доступу до даних. У процесі розробки API було розширено до 42 ендпоінтів, що забезпечило підтримку необхідного функціоналу системи. Структуру бази даних було оптимізовано відповідно до вимог третьої нормальної форми. У результаті

сформовано 14 таблиць, серед яких на пізніших етапах розробки додатково були впроваджені таблиці `issuer_assignments`, `quotes_log` та `supply_log`. Для візуалізації структури та функціональних взаємозв'язків системи побудовано три основні діаграми: діаграму варіантів використання, ER-діаграму та компонентну діаграму.

5. Усі модулі, заплановані на стадії проектування, доведено до робочого стану: автентифікація на `bcrypt` + `JWT` + `RBAC`, модуль ринку з `real-time` трансляцією цін через `WebSocket` (інтервал 2 секунди), торговий модуль з виконанням ринкових, лімітних та стоп-ордерів в атомарних транзакціях, портфель із підрахунком `P&L` по кожній позиції, гаманець з повною аудит-стежкою, система сповіщень та адміністративна панель. Окремо – повноцінне котирування активів емітентами та автоматична реакція ціни на торгові операції (`price impact`).

6. У питанні безпеки прийнято стратегію "захист на кожному рівні": паролі – `bcryptjs` з `cost=12` (близько 4 тисяч раундів хешування), токени – власна реалізація `JWT` через `crypto.createHmac` (без зовнішніх бібліотек з відомими вразливостями), `SQL` – лише параметризовані запити, `HTTP`-заголовки – пакет `Helmet.js`, фінансові операції – `MySQL`-транзакції з `SELECT...FOR UPDATE` для блокування рядків. Що важливо: кожна з цих вимог підтверджена окремим набором автотестів.

7. Тестування зайняло окрему частину роботи. У підсумку маю 160 автоматизованих тестів, поділених на чотири категорії – `JWT`-автентифікація, бізнес-логіка, безпека та валідація, інтеграційні `API`-тести. Усі пройдено, покриття 100%. Окремо протестовано стійкість до конкретних типових атак: `SQL`-ін'єкція в `email` при вході, спроба обходу `RBAC` через підроблений токен, `XSS`-вектори через поля профілю та новин. У всіх випадках система реагувала очікувано – поверталися `HTTP 401` або `403`, без витоку даних.

8. Економічна частина роботи зафіксувала вкладення приблизно 56 000 грн на одноразову розробку (280 годин праці за студентською ставкою 200 грн/год) – це в десятки разів менше за середню собівартість аналогічних комерційних рішень. Свідомо обрано просту і прозору модель монетизації: одна стабільна

стаття доходу (комісія 0.1% з угоди), без преміум-тарифів і прихованих платежів. За реалістичним прогнозом перший рік виходить майже на самоокупність (дохід покриває 81% операційних витрат), у другому році платформа стає прибутковою з чистим прибутком близько 252 тис. грн, а на третьому році – близько 1.3 млн грн. Архітектурно платформа здатна тримати до 5 000+ одночасних користувачів – для цього потрібно додати Redis-кеш для часто читаних даних та горизонтальне масштабування Node-процесів через PM2 cluster mode.

Практична цінність роботи. Платформа готова до розгортання й має зрозумілу документацію (інструкція з розгортання у Додатку В, повний перелік API в Додатку Б, ключовий код у Додатку А, рисунки тестувань у Додатку Е). Можливі сценарії використання: навчальна торгова платформа без ризиків реальних втрат, основа для фінтех-стартапу, ядро внутрішньокорпоративної системи для роботи з цифровими активами. Вихідний код відкритий під ліцензією MIT і допускає вільне використання та модифікацію.

Напрями подальшого розвитку. Серед першочергових – двофакторна автентифікація (TOTP або SMS); розширення matching engine на OCO-ордери (One-Cancels-the-Other); інтеграція з реальними платіжними шлюзами (LiqPay, Portmone); побудова інтерактивних графіків OHLCV-свічок (Chart.js або TradingView Lightweight Charts); мобільний застосунок на React Native або Flutter. У довгостроковій перспективі – перехід до мікросервісної архітектури при масштабуванні понад 10 000 активних трейдерів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Evans D.S., Schmalensee R. Matchmakers: The New Economics of Multisided Platforms. Harvard Business Review Press, 2016. 304 p.
2. World Federation of Exchanges. Annual Statistics. URL: <https://www.world-exchanges.org/our-work/statistics>
3. Statista. Investment behavior worldwide – statistics and facts. URL: <https://www.statista.com/topics/8138/investment-behavior-worldwide/>
4. Gomber P., Kauffman R.J., Parker C., Weber B.W. On the Fintech Revolution: Interpreting the Forces of Innovation, Disruption, and Transformation in Financial Services. Journal of Management Information Systems, 2018. Vol. 35, No. 1. P. 220–265.
5. Стратегія цифрового розвитку інноваційної діяльності України на період до 2030 року (WINWIN). Кабінет Міністрів України, 2025. URL: <https://thedigital.gov.ua/news/winwin-ukraina-zatverdila-strategiyu-tsifrovogo-rozvitku-innovatsiy-do-2030-roku>
6. Rochet J.C., Tirole J. Platform Competition in Two-Sided Markets. Journal of the European Economic Association. 2003. Vol. 1, No. 4. P. 990–1029.
7. Parker G.G., Van Alstyne M.W., Choudary S.P. Platform Revolution: How Networked Markets Are Transforming the Economy. W. W. Norton & Company, 2016. 352 p.
8. Hagiu A., Wright J. Multi-sided platforms. International Journal of Industrial Organization. 2015. Vol. 43. P. 162–174.
9. OECD. Digital finance: Policy issues and considerations. OECD Publishing, Paris, 2024. URL: <https://www.oecd.org/en/topics/sub-issues/digital-finance.html>
10. Harris L. Trading and Exchanges: Market Microstructure for Practitioners. Oxford University Press, 2003. 643 p.
11. Schär F. Decentralized Finance: On Blockchain- and Smart Contract-Based Financial Markets. Federal Reserve Bank of St. Louis Review, 2021. Vol. 103, No. 2. P. 153–174.

12. Lehdonvirta V., Castronova E. Virtual Economies: Design and Analysis. MIT Press, 2014. 306 p.
13. Kenney M., Zysman J. The Platform Economy: Restructuring the Space of Capitalist Accumulation. Cambridge Journal of Regions, Economy and Society. 2020. Vol. 13, No. 1. P. 55–76.
14. McAfee A., Brynjolfsson E. Machine, Platform, Crowd: Harnessing Our Digital Future. W.W. Norton & Company, 2017. 416 p.
15. European Securities and Markets Authority. Guidelines on transaction reporting, order record keeping and clock synchronisation under MiFID II (ESMA/2016/1452). ESMA, 2016 (updated 2017).
16. CoinMarketCap. Global Cryptocurrency Market Report Q4 2024. URL: <https://coinmarketcap.com/charts/>
17. Kitchenham B., Charters S. Guidelines for performing Systematic Literature Reviews in Software Engineering. EBSE Technical Report EBSE-2007-01. 2007.
18. Binance. API Documentation. URL: <https://binance-docs.github.io/apidocs/spot/en/>
19. Interactive Brokers. TWS API Reference Guide. URL: <https://interactivebrokers.github.io/tws-api/>
20. Valve Corporation. Steam Community Market Overview. URL: <https://steamcommunity.com/market/>
21. G2G Platform. Help Center & API Documentation. URL: <https://www.g2g.com/>
22. Hileman G., Rauchs M. Global Cryptocurrency Benchmarking Study. Cambridge Centre for Alternative Finance, 2017.
23. OpenSea. Developer Documentation. URL: <https://docs.opensea.io/>
24. Nakamoto S. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008. URL: <https://bitcoin.org/bitcoin.pdf>
25. Jacobson I., Christerson M., Jonsson P., Övergaard G. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, 1992. 528 p.
26. NIST. Digital Identity Guidelines. NIST Special Publication 800-63B. 2020.

27. Bass L., Clements P., Kazman R. Software Architecture in Practice. 4th ed. Addison-Wesley Professional, 2021. 464 p.
28. OWASP Foundation. OWASP Top Ten 2021. URL: <https://owasp.org/Top10/>
29. NIST. Recommendation for Applications Using Approved Hash Algorithms. NIST SP 800-107. 2012.
30. Закон України "Про ринки капіталу та організовані товарні ринки" від 23.02.2006 № 3480-IV. URL: <https://zakon.rada.gov.ua/>
31. Закон України "Про захист персональних даних" від 01.06.2010 № 2297-VI. URL: <https://zakon.rada.gov.ua/>
32. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002. 560 p.
33. Node.js Foundation. Node.js Documentation: Event Loop, Timers, and process.nextTick. URL: <https://nodejs.org/en/docs/>
34. Express.js Team. Express 4.x API Reference. URL: <https://expressjs.com/en/4x/api.html>
35. DuBois P. MySQL Cookbook: Solutions for Database Developers and Administrators. 3rd ed. O'Reilly Media, 2014. 864 p.
36. Momjian B. PostgreSQL: Introduction and Concepts. Addison-Wesley, 2001. 460 p.
37. Fette I., Melnikov A. The WebSocket Protocol. RFC 6455. IETF, 2011.
38. ws npm package benchmark comparison. URL: <https://github.com/websockets/ws#benchmarks>
39. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT). RFC 7519. IETF, May 2015. URL: <https://datatracker.ietf.org/doc/html/rfc7519>
40. Haverbeke M. Eloquent JavaScript. 3rd ed. No Starch Press, 2018. 472 p.
41. Cohn M. Agile Estimating and Planning. Prentice Hall, 2005. 368 p.
42. Richardson C. Microservices Patterns. Manning Publications, 2018. 520 p.
43. Clements P., Kazman R., Klein M. Evaluating Software Architectures. Addison-Wesley, 2001. 368 p.
44. Wiggins A. The Twelve-Factor App. URL: <https://12factor.net/>

45. Harris L. Optimal Dynamic Order Submission Strategies in Some Stylized Trading Problems. *Financial Markets, Institutions & Instruments*. 1998. Vol. 7, No. 2. P. 1–76.

ДОДАТКИ

Додаток А

Програмний код

Нижче наведено ключові фрагменти DDL-схеми бази даних платформи ExchangeUA.

```
-- =====
-- Схеми бази даних: Біржова платформа
-- =====

CREATE DATABASE IF NOT EXISTS exchange_platform
  CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;

USE exchange_platform;

-- -----
-- Користувачі
-- -----

CREATE TABLE IF NOT EXISTS users (
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  uuid VARCHAR(36) NOT NULL UNIQUE,
  email VARCHAR(255) NOT NULL UNIQUE,
  password_hash VARCHAR(255) NOT NULL,
  full_name VARCHAR(150) NOT NULL,
  role ENUM('admin','trader','issuer') NOT NULL DEFAULT 'trader',
  status ENUM('active','suspended','frozen') NOT NULL DEFAULT
'active',
  balance DECIMAL(20,8) NOT NULL DEFAULT 0.00000000,
  avatar_url VARCHAR(500) DEFAULT NULL,
  phone VARCHAR(30) DEFAULT NULL,
  country VARCHAR(100) DEFAULT NULL,
  verified_at DATETIME DEFAULT NULL,
  created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  INDEX idx_email (email),
  INDEX idx_role (role),
  INDEX idx_status(status)
) ENGINE=InnoDB;

-- -----
-- Активи (тікери)
-- -----

CREATE TABLE IF NOT EXISTS assets (
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  symbol VARCHAR(20) NOT NULL UNIQUE,
```

```

name VARCHAR(150) NOT NULL,
type ENUM('stock','crypto','forex','commodity') NOT NULL DEFAULT
'stock',
description TEXT DEFAULT NULL,
logo_url VARCHAR(500) DEFAULT NULL,
current_price DECIMAL(20,8) NOT NULL DEFAULT 0.00000000,
prev_close DECIMAL(20,8) NOT NULL DEFAULT 0.00000000,
volume_24h DECIMAL(20,2) NOT NULL DEFAULT 0.00,
market_cap DECIMAL(30,2) DEFAULT NULL,
is_active TINYINT(1) NOT NULL DEFAULT 1,
price_impact_enabled TINYINT(1) NOT NULL DEFAULT 0,
price_impact_buy_pct DECIMAL(8,5) NOT NULL DEFAULT 0,
price_impact_sell_pct DECIMAL(8,5) NOT NULL DEFAULT 0,
created_by INT UNSIGNED DEFAULT NULL,
created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
INDEX idx_symbol (symbol),
INDEX idx_type (type)
) ENGINE=InnoDB;

```

```

-- -----
-- Ордери
-- -----

```

```

CREATE TABLE IF NOT EXISTS orders (
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  uuid VARCHAR(36) NOT NULL UNIQUE,
  user_id INT UNSIGNED NOT NULL,
  asset_id INT UNSIGNED NOT NULL,
  type ENUM('market','limit','stop','stop_limit') NOT NULL DEFAULT
'market',
  side ENUM('buy','sell') NOT NULL,
  status ENUM('open','filled','partial','cancelled','rejected') NOT
NULL DEFAULT 'open',
  quantity DECIMAL(20,8) NOT NULL,
  filled_qty DECIMAL(20,8) NOT NULL DEFAULT 0.00000000,
  price DECIMAL(20,8) DEFAULT NULL, -- NULL для рыночных
  stop_price DECIMAL(20,8) DEFAULT NULL,
  avg_fill_price DECIMAL(20,8) DEFAULT NULL,
  total_value DECIMAL(20,8) DEFAULT NULL,
  commission DECIMAL(20,8) NOT NULL DEFAULT 0.00000000,
  note VARCHAR(500) DEFAULT NULL,
  expires_at DATETIME DEFAULT NULL,
  created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE,
  FOREIGN KEY (asset_id) REFERENCES assets(id) ON DELETE CASCADE,
  INDEX idx_user_id (user_id),
  INDEX idx_asset_id (asset_id),
  INDEX idx_status (status),
  INDEX idx_side (side),
  INDEX idx_created (created_at)
)

```

```
) ENGINE=InnoDB;
```

```
-----
-- Угоди (trades) - результат зіставлення ордерів
-----
```

```
CREATE TABLE IF NOT EXISTS trades (
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  uuid VARCHAR(36) NOT NULL UNIQUE,
  asset_id INT UNSIGNED NOT NULL,
  buy_order_id INT UNSIGNED DEFAULT NULL,
  sell_order_id INT UNSIGNED DEFAULT NULL,
  buyer_id INT UNSIGNED DEFAULT NULL,
  seller_id INT UNSIGNED DEFAULT NULL,
  quantity DECIMAL(20,8) NOT NULL,
  price DECIMAL(20,8) NOT NULL,
  total_value DECIMAL(20,8) NOT NULL,
  buyer_fee DECIMAL(20,8) NOT NULL DEFAULT 0.00000000,
  seller_fee DECIMAL(20,8) NOT NULL DEFAULT 0.00000000,
  executed_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (asset_id) REFERENCES assets(id) ON DELETE CASCADE,
  FOREIGN KEY (buy_order_id) REFERENCES orders(id) ON DELETE SET
  NULL,
  FOREIGN KEY (sell_order_id) REFERENCES orders(id) ON DELETE SET
  NULL,
  FOREIGN KEY (buyer_id) REFERENCES users(id) ON DELETE SET NULL,
  FOREIGN KEY (seller_id) REFERENCES users(id) ON DELETE SET NULL,
  INDEX idx_asset (asset_id),
  INDEX idx_buyer (buyer_id),
  INDEX idx_seller (seller_id),
  INDEX idx_executed (executed_at)
) ENGINE=InnoDB;
```

```
-----
-- Портфель користувача
-----
```

```
CREATE TABLE IF NOT EXISTS portfolio (
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  user_id INT UNSIGNED NOT NULL,
  asset_id INT UNSIGNED NOT NULL,
  quantity DECIMAL(20,8) NOT NULL DEFAULT 0.00000000,
  avg_buy_price DECIMAL(20,8) NOT NULL DEFAULT 0.00000000,
  updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
  CURRENT_TIMESTAMP,
  UNIQUE KEY uq_user_asset (user_id, asset_id),
  FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE,
  FOREIGN KEY (asset_id) REFERENCES assets(id) ON DELETE CASCADE
) ENGINE=InnoDB;
```

```
-----
-- Транзакції балансу (поповнення/зняття)
-----
```

```
CREATE TABLE IF NOT EXISTS balance_transactions (
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
```

```

uuid VARCHAR(36) NOT NULL UNIQUE,
user_id INT UNSIGNED NOT NULL,
type
ENUM('deposit','withdrawal','trade_buy','trade_sell','commission',
'bonus') NOT NULL,
amount DECIMAL(20,8) NOT NULL,
balance_before DECIMAL(20,8) NOT NULL,
balance_after DECIMAL(20,8) NOT NULL,
reference_id INT UNSIGNED DEFAULT NULL,
description VARCHAR(500) DEFAULT NULL,
status ENUM('pending','completed','failed','cancelled') NOT NULL
DEFAULT 'completed',
created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE,
INDEX idx_user_id (user_id),
INDEX idx_type (type),
INDEX idx_created (created_at)
) ENGINE=InnoDB;

```

```

-- -----
-- Цінова історія (свічки OHLCV)
-- -----

```

```

CREATE TABLE IF NOT EXISTS price_history (
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  asset_id INT UNSIGNED NOT NULL,
  timeframe ENUM('1m','5m','15m','1h','4h','1d','1w') NOT NULL
  DEFAULT '1d',
  open_price DECIMAL(20,8) NOT NULL,
  high_price DECIMAL(20,8) NOT NULL,
  low_price DECIMAL(20,8) NOT NULL,
  close_price DECIMAL(20,8) NOT NULL,
  volume DECIMAL(20,2) NOT NULL DEFAULT 0.00,
  candle_time DATETIME NOT NULL,
  UNIQUE KEY uq_asset_tf_time (asset_id, timeframe, candle_time),
  FOREIGN KEY (asset_id) REFERENCES assets(id) ON DELETE CASCADE,
  INDEX idx_asset_time (asset_id, candle_time)
) ENGINE=InnoDB;

```

```

-- -----
-- Новини & аналітика
-- -----

```

```

CREATE TABLE IF NOT EXISTS news (
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  title VARCHAR(500) NOT NULL,
  content TEXT NOT NULL,
  summary VARCHAR(1000) DEFAULT NULL,
  author_id INT UNSIGNED NOT NULL,
  asset_id INT UNSIGNED DEFAULT NULL,
  category
ENUM('market','company','economy','crypto','forex','commodity')
NOT NULL DEFAULT 'market',
  is_published TINYINT(1) NOT NULL DEFAULT 0,
  published_at DATETIME DEFAULT NULL,

```

```

    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    FOREIGN KEY (author_id) REFERENCES users(id) ON DELETE CASCADE,
    FOREIGN KEY (asset_id) REFERENCES assets(id) ON DELETE SET NULL,
    INDEX idx_published (is_published, published_at),
    INDEX idx_category (category)
) ENGINE=InnoDB;

-- -----
-- Сповіщення користувача
-- -----
CREATE TABLE IF NOT EXISTS notifications (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    user_id INT UNSIGNED NOT NULL,
    type
ENUM('order_filled','order_cancelled','deposit','withdrawal','price_alert','system') NOT NULL,
    title VARCHAR(255) NOT NULL,
    message TEXT NOT NULL,
    is_read TINYINT(1) NOT NULL DEFAULT 0,
    data_json JSON DEFAULT NULL,
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE,
    INDEX idx_user_read (user_id, is_read),
    INDEX idx_created (created_at)
) ENGINE=InnoDB;

-- -----
-- Сесії користувачів (для аналітики)
-- -----
CREATE TABLE IF NOT EXISTS user_sessions (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    user_id INT UNSIGNED NOT NULL,
    ip_address VARCHAR(45) DEFAULT NULL,
    user_agent VARCHAR(500) DEFAULT NULL,
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
) ENGINE=InnoDB;

-- -----
-- Призначення емітентів на активи
-- -----
CREATE TABLE IF NOT EXISTS issuer_assignments (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    issuer_id INT UNSIGNED NOT NULL,
    asset_id INT UNSIGNED NOT NULL,
    assigned_by INT UNSIGNED NOT NULL,
    assigned_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    UNIQUE KEY uq_issuer_asset (issuer_id, asset_id),
    FOREIGN KEY (issuer_id) REFERENCES users(id) ON DELETE CASCADE,
    FOREIGN KEY (asset_id) REFERENCES assets(id) ON DELETE CASCADE,
    FOREIGN KEY (assigned_by) REFERENCES users(id) ON DELETE CASCADE,

```

```

INDEX idx_issuer (issuer_id),
INDEX idx_asset (asset_id)
) ENGINE=InnoDB;

-- -----
-- Журнал котирувань (повний аудит змін цін)
-- -----

CREATE TABLE IF NOT EXISTS quotes_log (
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  asset_id INT UNSIGNED NOT NULL,
  user_id INT UNSIGNED DEFAULT NULL,
  source ENUM('issuer','admin','trade','system') NOT NULL,
  old_price DECIMAL(20,8) DEFAULT NULL,
  new_price DECIMAL(20,8) NOT NULL,
  note VARCHAR(500) DEFAULT NULL,
  created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (asset_id) REFERENCES assets(id) ON DELETE CASCADE,
  FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE SET NULL,
  INDEX idx_asset_time (asset_id, created_at)
) ENGINE=InnoDB;

-- =====
-- Початкові дані (seed)
-- =====

-- АКТИВИ
INSERT IGNORE INTO assets (symbol, name, type, current_price,
prev_close, volume_24h, market_cap) VALUES
('AAPL', 'Apple Inc.', 'stock', 189.50, 189.50, 0, NULL),
('GOOGL', 'Alphabet Inc.', 'stock', 138.75, 138.75, 0, NULL),
('MSFT', 'Microsoft Corporation', 'stock', 412.30, 412.30, 0,
NULL),
('TSLA', 'Tesla Inc.', 'stock', 248.60, 248.60, 0, NULL),
('AMZN', 'Amazon.com Inc.', 'stock', 178.90, 178.90, 0, NULL),
('BTC', 'Bitcoin', 'crypto', 67845.00, 67845.00, 0, NULL),
('ETH', 'Ethereum', 'crypto', 3620.50, 3620.50, 0, NULL),
('BNB', 'Binance Coin', 'crypto', 578.30, 578.30, 0, NULL),
('EURUSD', 'EUR/USD', 'forex', 1.0832, 1.0832, 0, NULL),
('GBPUSD', 'GBP/USD', 'forex', 1.2705, 1.2705, 0, NULL),
('GOLD', 'Gold (XAU/USD)', 'commodity', 2345.80, 2345.80, 0,
NULL),
('OIL', 'Crude Oil (WTI)', 'commodity', 82.40, 82.40, 0, NULL);

-- Тестові облікові записи створюються окремим скриптом
setupUsers.js
-- (admin@exchange.ua / Admin@123, trader@example.com /
Trader@123,
-- issuer@exchange.ua / Issuer@123). Це дозволяє відокремити
-- (деталі у Додатку В)
-- bcrypt-хешування паролів у Node.js
-- Запуск: npm run setup-users

```

Код головної сторінки:

Нижче наведено фрагмент public/index.html – навігаційну панель та структуру головної сторінки. Повний файл містить 569 рядків (24 КБ).

```
<!DOCTYPE html>
<html lang="uk">
<head>
  <meta charset="UTF-8"/>
  <meta name="viewport" content="width=device-width, initial-
scale=1.0"/>
  <title>ExchangeUA - Цифрова Біржова Платформа</title>
  <link rel="stylesheet" href="/css/style.css"/>
</head>
<body>
<!-- Навігаційна панель з 9 вкладками -->
<nav class="navbar">
  <div class="nav-brand">◆ ExchangeUA</div>
  <div class="nav-tabs">
    <button class="nav-tab" data-page="market">Ринок</button>
    <button class="nav-tab" data-page="news">Новини</button>
    <button class="nav-tab" data-page="trade"
id="tabTrade">Торгівля</button>
    <button class="nav-tab" data-page="portfolio"
id="tabPortfolio">Портфель</button>
    <button class="nav-tab" data-page="history"
id="tabHistory">Угоди</button>
    <button class="nav-tab" data-page="wallet"
id="tabWallet">Гаманець</button>
    <button class="nav-tab" data-page="quotes"
id="tabQuotes">Котирування</button>
    <button class="nav-tab" data-page="profile"
id="tabProfile">Профіль</button>
    <button class="nav-tab" data-page="admin"
id="tabAdmin">Адмін</button>
  </div>
  <div class="nav-right">
    <button class="btn-notif" id="btnNotif">🔔</button>
    <div id="navBalance" class="nav-balance"></div>
    <button id="btnLogin">Увійти</button>
  </div>
</nav>

<!-- Контейнери сторінок (по одному на вкладку) -->
<main id="app">
  <div class="page active" id="pageMarket">...</div>
  <div class="page" id="pageNews">...</div>
  <div class="page" id="pageTrade">...</div>
  <div class="page" id="pagePortfolio">...</div>
  <div class="page" id="pageHistory">...</div>
  <div class="page" id="pageWallet">...</div>
  <div class="page" id="pageQuotes">...</div>
  <div class="page" id="pageProfile">...</div>
```

```

<div class="page" id="pageAdmin">...</div>
</main>

<!-- Модальні вікна (входу, реєстрації, активу, котирування,
новини тощо) -->
<div class="modal-overlay" id="authModal">...</div>
<div class="modal-overlay" id="assetModal">...</div>
<div class="modal-overlay" id="newsModal">...</div>
<div class="modal-overlay" id="newsComposeModal">...</div>
<div class="modal-overlay" id="changePasswordModal">...</div>
<div class="modal-overlay" id="changeBalanceModal">...</div>
<div class="modal-overlay" id="assetDetailsModal">...</div>
<div class="modal-overlay" id="profileModal">...</div>
<div class="modal-overlay" id="assignmentModal">...</div>

<script src="/js/app.js"></script>
</body>
</html>

```

Нижче – фрагменти клієнтського скрипта `public/js/app.js`, що реалізує SPA-навігацію, WebSocket-клієнт та рендеринг сторінок Торгівлі, Портфеля, Котирування й Адмін-панелі.

```

'use strict';

const { v4: uuidv4 } = require('uuid');
const db = require('../config/db');

// — Отримати новини

async function getNews(req, res) {
  try {
    const { category, asset_id, limit = 20, offset = 0 } = req.query;
    let sql = `
      SELECT n.id, n.title, n.summary, n.category, n.published_at,
      u.full_name AS author_name, a.symbol AS asset_symbol
      FROM news n
      JOIN users u ON n.author_id = u.id
      LEFT JOIN assets a ON n.asset_id = a.id
      WHERE n.is_published = 1`;
    const params = [];

    if (category) { sql += ' AND n.category = ?'; params.push(category); }
    if (asset_id) { sql += ' AND n.asset_id = ?'; params.push(asset_id); }

    sql += ' ORDER BY n.published_at DESC LIMIT ? OFFSET ?';
    params.push(parseInt(limit), parseInt(offset));

    const [rows] = await db.query(sql, params);
    return res.json({ success: true, data: rows });
  } catch (err) {
    return res.status(500).json({ success: false, message: err.message });
  }
}

```

```

}
}

```

// — Одна новина

```

async function getNewsById(req, res) {
  try {
    const [rows] = await db.query(
      `SELECT n.*, u.full_name AS author_name, a.symbol AS asset_symbol
      FROM news n
      JOIN users u ON n.author_id = u.id
      LEFT JOIN assets a ON n.asset_id = a.id
      WHERE n.id = ? AND n.is_published = 1`,
      [req.params.id]
    );
    if (!rows.length) return res.status(404).json({ success: false, message: 'Новину не знайдено' });
    return res.json({ success: true, data: rows[0] });
  } catch (err) {
    return res.status(500).json({ success: false, message: err.message });
  }
}

```

// — Створити новину (admin/issuer)

```

async function createNews(req, res) {
  try {
    const { title, content, summary, asset_id, category } = req.body;
    if (!title || !content) {
      return res.status(400).json({ success: false, message: 'Заголовок та зміст обов\'язкові' });
    }
    const [result] = await db.query(
      `INSERT INTO news (title, content, summary, author_id, asset_id, category, is_published,
published_at)
VALUES (?, ?, ?, ?, ?, 1, NOW())`,
      [title, content, summary || null, req.user.id, asset_id || null, category || 'market']
    );
    return res.status(201).json({ success: true, message: 'Новину опубліковано', id: result.insertId });
  } catch (err) {
    return res.status(500).json({ success: false, message: err.message });
  }
}

```

```

module.exports = { getNews, getNewsById, createNews, deleteNews };

```

'use strict';

```

const bcrypt = require('bcryptjs');
const { v4: uuidv4 } = require('uuid');
const db = require('../config/db');
const { signToken } = require('../utils/jwt');

```

// — Реєстрація

```

async function register(req, res) {
  try {

```

```

const { email, password, full_name, phone, country } = req.body;

if (!email || !password || !full_name) {
  return res.status(400).json({ success: false, message: 'Заповніть обов'язкові поля' });
}
if (password.length < 6) {
  return res.status(400).json({ success: false, message: 'Пароль має бути мінімум 6 символів' });
}

const [exist] = await db.query('SELECT id FROM users WHERE email = ?', [email]);
if (exist.length) {
  return res.status(409).json({ success: false, message: 'Email вже зареєстровано' });
}

const hash = await bcrypt.hash(password, 12);
const id = uuidv4();

const [result] = await db.query(
  `INSERT INTO users (uuid, email, password_hash, full_name, role, status, balance, phone,
country, verified_at)
VALUES (?, ?, ?, ?, 'trader', 'active', 10000.00, ?, ?, NOW())`,
  [id, email.toLowerCase().trim(), hash, full_name.trim(), phone || null, country || null]
);

const userId = result.insertId;
// Баланс = 0 за замовчуванням. Поповнення відбувається окремо через POST
// api/balance/deposit.
const token = signToken({ userId, role: 'trader' });
req.session.userId = userId;

return res.status(201).json({
  success: true,
  message: 'Реєстрацію завершено. Поповніть рахунок у розділі Гаманець.',
  token,
  user: { id: userId, email, full_name, role: 'trader', balance: 10000.00 }
});
} catch (err) {
  console.error('register error:', err);
  return res.status(500).json({ success: false, message: 'Внутрішня помилка сервера' });
}
}

// — Вхід

```

```

async function login(req, res) {
  try {
    const { email, password } = req.body;
    if (!email || !password) {
      return res.status(400).json({ success: false, message: 'Введіть email та пароль' });
    }

    const [rows] = await db.query(
      'SELECT id, email, password_hash, full_name, role, status, balance FROM users WHERE email
= ?',

```

```

[email.toLowerCase().trim()]
);

if (!rows.length) {
return res.status(401).json({ success: false, message: 'Невірний email або пароль' });
}

const user = rows[0];
if (user.status === 'suspended') {
return res.status(403).json({ success: false, message: 'Акаунт заблоковано' });
}

const ok = await bcrypt.compare(password, user.password_hash);
if (!ok) {
return res.status(401).json({ success: false, message: 'Невірний email або пароль' });
}

const token = signToken({ userId: user.id, role: user.role });
req.session.userId = user.id;

// Логуємо сесію
await db.query(
'INSERT INTO user_sessions (user_id, ip_address, user_agent) VALUES (?, ?, ?)',
[user.id, req.ip, req.headers['user-agent']] || ''
);

return res.json({
success: true,
message: 'Вхід успішний',
token,
user: { id: user.id, email: user.email, full_name: user.full_name, role: user.role, balance:
parseFloat(user.balance) }
});
} catch (err) {
console.error('login error:', err);
return res.status(500).json({ success: false, message: 'Внутрішня помилка сервера' });
}
}

// — Вихід


---


function logout(req, res) {
req.session.destroy(() => {
res.json({ success: true, message: 'Вихід виконано' });
});
}

// — Профіль поточного користувача


---


async function getProfile(req, res) {
try {
const [rows] = await db.query(
`SELECT id, uuid, email, full_name, role, status, balance, phone, country, verified_at, created_at
FROM users WHERE id = ?`,
[req.user.id]

```

```

);
if (!rows.length) return res.status(404).json({ success: false, message: 'Не знайдено' });

return res.json({ success: true, data: rows[0] });
} catch (err) {
return res.status(500).json({ success: false, message: err.message });
}
}

```

// — Оновлення профілю

```

async function updateProfile(req, res) {
  try {
    const { full_name, phone, country } = req.body;
    await db.query(
      'UPDATE users SET full_name = ?, phone = ?, country = ? WHERE id = ?',
      [full_name || req.user.full_name, phone || null, country || null, req.user.id]
    );
    return res.json({ success: true, message: 'Профіль оновлено' });
  } catch (err) {
    return res.status(500).json({ success: false, message: err.message });
  }
}

```

```

module.exports = { register, login, logout, getProfile, updateProfile, changePassword };

```

```

'use strict';

```

```

const { v4: uuidv4 } = require('uuid');
const db = require('../config/db');

```

```

//

```

```

// PORTFOLIO

```

```

//

```

```

async function getPortfolio(req, res) {
  try {
    const [rows] = await db.query(
      `SELECT p.*, a.symbol, a.name AS asset_name, a.type AS asset_type,
        a.current_price, a.logo_url,
        ROUND((a.current_price - p.avg_buy_price) / p.avg_buy_price * 100, 2) AS pnl_pct,
        ROUND((a.current_price - p.avg_buy_price) * p.quantity, 2) AS pnl_abs,
        ROUND(a.current_price * p.quantity, 2) AS market_value
        FROM portfolio p JOIN assets a ON p.asset_id = a.id
        WHERE p.user_id = ? AND p.quantity > 0
        ORDER BY market_value DESC`,
      [req.user.id]
    );
  }
}

```

```

const totalValue = rows.reduce((s, r) => s + parseFloat(r.market_value), 0);

```

```

const totalPnl = rows.reduce((s, r) => s + parseFloat(r.pnl_abs), 0);

return res.json({ success: true, data: { positions: rows, totalValue, totalPnl } });
} catch (err) {
return res.status(500).json({ success: false, message: err.message });
}
}

```

```
//
```

```
// TRADES HISTORY
```

```
//
```

```

async function getTradeHistory(req, res) {
  try {
    const { limit = 50, offset = 0, asset_id } = req.query;
    let sql = `
      SELECT t.*, a.symbol, a.name AS asset_name,
      CASE WHEN t.buyer_id = ? THEN 'buy' ELSE 'sell' END AS side
      FROM trades t JOIN assets a ON t.asset_id = a.id
      WHERE (t.buyer_id = ? OR t.seller_id = ?)`;
    const params = [req.user.id, req.user.id, req.user.id];

```

```

    if (asset_id) { sql += ' AND t.asset_id = ?'; params.push(asset_id); }
    sql += ' ORDER BY t.executed_at DESC LIMIT ? OFFSET ?';
    params.push(parseInt(limit), parseInt(offset));

```

```

    const [rows] = await db.query(sql, params);
    return res.json({ success: true, data: rows });
  } catch (err) {
    return res.status(500).json({ success: false, message: err.message });
  }
}

```

```
//
```

```
// BALANCE
```

```
//
```

```

async function getBalance(req, res) {
  try {
    const [[user]] = await db.query('SELECT balance FROM users WHERE id = ?', [req.user.id]);
    const [txns] = await db.query(
      `SELECT * FROM balance_transactions WHERE user_id = ? ORDER BY created_at DESC
      LIMIT 20`,
      [req.user.id]
    );
    return res.json({ success: true, data: { balance: parseFloat(user.balance), transactions: txns } });
  }
}

```

```

    } catch (err) {
    return res.status(500).json({ success: false, message: err.message });
    }
  }

  async function deposit(req, res) {
    const conn = await db.getConnection();
    try {
      await conn.beginTransaction();
      const { amount } = req.body;
      const amt = parseFloat(amount);
      if (!amt || amt <= 0 || amt > 1000000) {
        await conn.rollback(); conn.release();
        return res.status(400).json({ success: false, message: 'Невірна сума (1 – 1 000 000)' });
      }
      const [[user]] = await conn.query('SELECT balance FROM users WHERE id = ? FOR UPDATE',
        [req.user.id]);
      const before = parseFloat(user.balance);
      const after = before + amt;
      await conn.query('UPDATE users SET balance = ? WHERE id = ?', [after.toFixed(8), req.user.id]);
      await conn.query(
        `INSERT INTO balance_transactions (uuid, user_id, type, amount, balance_before, balance_after,
        description)
        VALUES (?, ?, 'deposit', ?, ?, ?, 'Поповнення рахунку')`,
        [uuidv4(), req.user.id, amt, before, after]
      );
      await conn.commit();
      conn.release();
      return res.json({ success: true, message: `Рахунок поповнено на ${amt} грн`, balance: after });
    } catch (err) {
      await conn.rollback(); conn.release();
      return res.status(500).json({ success: false, message: err.message });
    }
  }

  async function withdraw(req, res) {
    const conn = await db.getConnection();
    try {
      await conn.beginTransaction();
      const { amount } = req.body;
      const amt = parseFloat(amount);
      const [[user]] = await conn.query('SELECT balance FROM users WHERE id = ? FOR UPDATE',
        [req.user.id]);
      const before = parseFloat(user.balance);
      if (!amt || amt <= 0 || amt > before) {
        await conn.rollback(); conn.release();
        return res.status(400).json({ success: false, message: 'Недостатньо коштів або невірна сума' });
      }
      const after = before - amt;
      await conn.query('UPDATE users SET balance = ? WHERE id = ?', [after.toFixed(8), req.user.id]);
      await conn.query(
        `INSERT INTO balance_transactions (uuid, user_id, type, amount, balance_before, balance_after,
        description)
        VALUES (?, ?, 'withdrawal', ?, ?, ?, 'Виведення коштів')`,

```

```

[uuidv4(), req.user.id, amt, before, after]
);
await conn.commit();
conn.release();
return res.json({ success: true, message: `Знято ${amt} грн`, balance: after });
} catch (err) {
await conn.rollback(); conn.release();
return res.status(500).json({ success: false, message: err.message });
}
}

```

```
//
```

```
// NOTIFICATIONS
```

```
//
```

```

async function getNotifications(req, res) {
  try {
    const [rows] = await db.query(
      'SELECT * FROM notifications WHERE user_id = ? ORDER BY created_at DESC LIMIT 30',
      [req.user.id]
    );
    return res.json({ success: true, data: rows });
  } catch (err) {
    return res.status(500).json({ success: false, message: err.message });
  }
}

```

```

async function markAllRead(req, res) {
  try {
    await db.query('UPDATE notifications SET is_read = 1 WHERE user_id = ?', [req.user.id]);
    return res.json({ success: true, message: 'Всі сповіщення прочитано' });
  } catch (err) {
    return res.status(500).json({ success: false, message: err.message });
  }
}

```

```
//
```

```
// ADMIN
```

```
//
```

```

async function adminGetUsers(req, res) {
  try {
    const [rows] = await db.query(
      `SELECT id, uuid, email, full_name, role, status, balance, country, created_at
      FROM users ORDER BY created_at DESC LIMIT 100`
    );

```

```

return res.json({ success: true, data: rows });
} catch (err) {
return res.status(500).json({ success: false, message: err.message });
}
}

async function adminUpdateUser(req, res) {
  try {
    const { id } = req.params;
    const { role, status } = req.body;
    await db.query('UPDATE users SET role = ?, status = ? WHERE id = ?', [role, status, id]);
    return res.json({ success: true, message: 'Користувача оновлено' });
  } catch (err) {
    return res.status(500).json({ success: false, message: err.message });
  }
}

async function getDashboardStats(req, res) {
  try {
    const [[users]] = await db.query('SELECT COUNT(*) AS cnt FROM users');
    const [[orders]] = await db.query('SELECT COUNT(*) AS cnt FROM orders');
    const [[trades]] = await db.query('SELECT COUNT(*) AS cnt, SUM(total_value) AS vol FROM trades');
    const [[assets]] = await db.query('SELECT COUNT(*) AS cnt FROM assets WHERE is_active = 1');
    return res.json({
      success: true,
      data: {
        totalUsers: users.cnt,
        totalOrders: orders.cnt,
        totalTrades: trades.cnt,
        tradingVolume: parseFloat(trades.vol || 0),
        activeAssets: assets.cnt
      }
    });
  } catch (err) {
    return res.status(500).json({ success: false, message: err.message });
  }
}

module.exports = {
  getPortfolio, getTradeHistory,
  getBalance, deposit, withdraw,
  getNotifications, markAllRead,
  adminGetUsers, adminUpdateUser, getDashboardStats,
  assignIssuer, unassignIssuer, getAssignments, getMyAssignments
};

'use strict';

const { verifyToken } = require('../utils/jwt');
const db = require('../config/db');

/**

```

```

* Перевірка аутентифікації через сесію або JWT-токен у заголовку
*/
async function requireAuth(req, res, next) {
  try {
    // 1) Сесія
    if (req.session && req.session.userId) {
      const [rows] = await db.query(
        'SELECT id, uuid, email, full_name, role, status, balance FROM users WHERE id = ? AND status = "active"',
        [req.session.userId]
      );
      if (rows && rows.length) {
        req.user = rows[0];
        return next();
      }
    }

    // 2) Bearer JWT
    const authHeader = req.headers.authorization || '';
    if (authHeader.startsWith('Bearer ')) {
      const token = authHeader.slice(7);
      try {
        const payload = verifyToken(token);
        if (payload && payload.userId) {
          const [rows] = await db.query(
            'SELECT id, uuid, email, full_name, role, status, balance FROM users WHERE id = ? AND status = "active"',
            [payload.userId]
          );
          if (rows && rows.length) {
            req.user = rows[0];
            return next();
          }
        }
      } catch (jwtErr) {
        console.error('JWT verification error:', jwtErr.message);
        // Продовжаємо к наступному методу аутентифікації
      }
    }

    return res.status(401).json({ success: false, message: 'Необхідна аутентифікація' });
  } catch (err) {
    console.error('requireAuth error:', err);
    return res.status(401).json({ success: false, message: 'Помилка аутентифікації' });
  }
}

/**
* Перевірка ролей
*/
function requireRole(...roles) {
  return (req, res, next) => {
    if (!req.user) {
      return res.status(401).json({ success: false, message: 'Необхідна аутентифікація' });
    }
  };
}

```

```

    }
    if (!roles.includes(req.user.role)) {
    return res.status(403).json({ success: false, message: 'Доступ заборонено. Недостатньо прав.' });
    }
    next();
  };
}

module.exports = { requireAuth, requireRole, requireActive };
'use strict';

const express = require('express');
const router = express.Router();

const { requireAuth, requireRole, requireActive } = require('../middleware/auth');
const auth = require('../controllers/authController');
const assets = require('../controllers/assetsController');
const orders = require('../controllers/ordersController');
const portfolio = require('../controllers/portfolioController');
const news = require('../controllers/newsController');

router.post('/auth/register', auth.register);
router.post('/auth/login', auth.login);
router.post('/auth/logout', auth.logout);
router.get('/auth/profile', requireAuth, auth.getProfile);
router.put('/auth/profile', requireAuth, requireActive, auth.updateProfile);

router.get('/assets', assets.getAssets);
router.get('/assets/:id', assets.getAsset);
router.get('/assets/:id/history', assets.getPriceHistory);
router.get('/assets/:asset_id/orderbook', orders.getOrderBook);
router.post('/assets', requireAuth, requireRole('admin', 'issuer'), requireActive, assets.createAsset);

router.post('/orders', requireAuth, requireActive, orders.createOrder);
router.get('/orders', requireAuth, orders.getOrders);
router.delete('/orders/:id', requireAuth, requireActive, orders.cancelOrder);

router.get('/portfolio', requireAuth, portfolio.getPortfolio);
router.get('/trades', requireAuth, portfolio.getTradeHistory);

router.get('/balance', requireAuth, portfolio.getBalance);
router.post('/balance/deposit', requireAuth, requireActive, portfolio.deposit);
router.post('/balance/withdraw', requireAuth, requireActive, portfolio.withdraw);

router.get('/notifications', requireAuth, portfolio.getNotifications);
router.post('/notifications/read-all', requireAuth, requireActive, portfolio.markAllRead);

router.get('/news', news.getNews);
router.get('/news/:id', news.getNewsById);
router.post('/news', requireAuth, requireRole('admin', 'issuer'), requireActive, news.createNews);

router.get('/admin/users', requireAuth, requireRole('admin'), portfolio.adminGetUsers);
router.put('/admin/users/:id', requireAuth, requireRole('admin'), portfolio.adminUpdateUser);
router.get('/admin/stats', requireAuth, requireRole('admin'), portfolio.getDashboardStats);

```

```
// router.get('/stats', ...) – застарілий маршрут видалено
router.get('/health', (req, res) => res.json({ success: true, status: 'OK', uptime:
Math.floor(process.uptime())+'s' }));

// — Розширені маршрути (додано на пізніших ітераціях) —————
router.post('/auth/change-password', requireAuth, requireActive, auth.changePassword);
router.put('/:assets/:id', requireAuth, requireRole('admin','issuer'), requireActive,
assets.updateAsset);
router.delete('/:assets/:id', requireAuth, requireRole('admin','issuer'), requireActive,
assets.deleteAsset);
router.post('/:assets/:id/quote', requireAuth, requireRole('admin','issuer'), requireActive,
assets.setQuote);
router.get('/:assets/:id/quotes-log', assets.getQuotesLog);
router.post('/:assets/:id/supply', requireAuth, requireRole('admin','issuer'), requireActive,
assets.manageSupply);
router.get('/:assets/:id/supply-log', requireAuth, requireRole('admin','issuer'), assets.getSupplyLog);
router.post('/:assets/:id/cash-reserve', requireAuth, requireRole('admin','issuer'), requireActive,
assets.manageCashReserve);
router.get('/:assets/:id/cash-reserve-log', requireAuth, requireRole('admin','issuer'),
assets.getCashReserveLog);
router.delete('/news/:id', requireAuth, requireRole('admin','issuer'), requireActive,
news.deleteNews);
router.get('/admin/stats', requireAuth, requireRole('admin'), portfolio.getDashboardStats);
router.get('/admin/assignments', requireAuth, requireRole('admin'), portfolio.getAssignments);
router.post('/admin/assignments', requireAuth, requireRole('admin'), portfolio.assignIssuer);
router.delete('/admin/assignments', requireAuth, requireRole('admin'), portfolio.unassignIssuer);
router.get('/issuer/my-assets', requireAuth, requireRole('issuer','admin'),
portfolio.getMyAssignments);

module.exports = router;

'use strict';

const WebSocket = require('ws');
const db = require('../config/db');

let wss = null;

function setupWebSocket(server) {
  wss = new WebSocket.Server({ server, path: '/ws' });

  wss.on('connection', (ws, req) => {
    console.log('🚀 WebSocket підключено: ${req.socket.remoteAddress}');

    ws.isAlive = true;
    ws.on('pong', () => { ws.isAlive = true; });

    // Відправляємо поточні ціни одразу при підключенні
    sendPrices(ws);

    ws.on('message', async (raw) => {
      try {
        const msg = JSON.parse(raw.toString());

```

```

if (msg.type === 'subscribe' && msg.symbols) {
  ws.subscribedSymbols = msg.symbols;
  ws.send(JSON.stringify({ type: 'subscribed', symbols: msg.symbols }));
}
if (msg.type === 'ping') {
  ws.send(JSON.stringify({ type: 'pong', ts: Date.now() }));
}
} catch { /* ignore malformed */ }
});

ws.on('close', () => { console.log('🔌 WebSocket відключено'); });
ws.on('error', (err) => { console.error('WS error:', err.message); });
});

// Ping/pong для перевірки живих з'єднань
setInterval(() => {
  wss.clients.forEach(ws => {
    if (!ws.isAlive) return ws.terminate();
    ws.isAlive = false;
    ws.ping();
  });
}, 30000);

// Оновлення цін кожні 2 секунди
setInterval(async () => {
  // Ціни оновлюються емітентами через setQuote або автоматично після угод
  broadcastPrices();
}, 2000);

console.log('✅ WebSocket сервер запущено на /ws');
return wss;
}

async function sendPrices(ws) {
  try {
    const [assets] = await db.query(
      `SELECT symbol, current_price, prev_close,
      ROUND((current_price - prev_close) / prev_close * 100, 4) AS change_pct,
      volume_24h
      FROM assets WHERE is_active = 1`
    );
    if (ws.readyState === WebSocket.OPEN) {
      ws.send(JSON.stringify({ type: 'prices', data: assets, ts: Date.now() }));
    }
  } catch (err) {
    console.error('sendPrices error:', err.message);
  }
}

async function broadcastPrices() {
  if (!wss || wss.clients.size === 0) return;
  try {
    const [assets] = await db.query(
      `SELECT symbol, current_price, prev_close,

```

```

ROUND((current_price - prev_close) / prev_close * 100, 4) AS change_pct,
volume_24h
FROM assets WHERE is_active = 1`
);
const msg = JSON.stringify({ type: 'prices', data: assets, ts: Date.now() });
wss.clients.forEach(ws => {
  if (ws.readyState === WebSocket.OPEN) ws.send(msg);
});
} catch (err) {
  console.error('broadcastPrices error:', err.message);
}
}

module.exports = { setupWebSocket };

'use strict';
// =====
// ExchangeUA – Frontend SPA
// =====

const API = '/api';
let currentUser = null;
let authToken = localStorage.getItem('ex_token') || null;
let ws = null;
let prices = {}; // symbol -> priceData
let allAssets = [];
let currentType = "";
let tradeSide = 'buy';
let tradeAssets = [];

//
=====
// INIT
//
=====

document.addEventListener('DOMContentLoaded', async () => {
  setupNavTabs();
  setupFilterBtns();

  setupAuthHandlers();

  await loadAssets();
  connectWS();
  if (authToken) await tryRestoreSession();
});

function setupAuthHandlers() {
  document.getElementById('authModal').addEventListener('click', (e) => {
    if (e.target.id === 'authModal') closeModal('authModal');
  });

  document.getElementById('tabLogin').addEventListener('click', () => switchAuthTab('login'));

```

```

document.getElementById('tabRegister').addEventListener('click', () => switchAuthTab('register'));

document.getElementById('btnLoginSubmit').addEventListener('click', doLogin);
document.getElementById('btnRegisterSubmit').addEventListener('click', doRegister);

document.getElementById('loginPassword').addEventListener('keypress', (e) => {
  if (e.key === 'Enter') doLogin();
});
document.getElementById('regPassword').addEventListener('keypress', (e) => {
  if (e.key === 'Enter') doRegister();
});
}

async function tryRestoreSession() {
  try {
    const data = await apiFetch('/auth/profile');
    if (data.success && data.data) {
      setUser(data.data);
      return true;
    }
  } catch (err) {
    console.log('Session restore failed:', err);
  }
  authToken = null;
  localStorage.removeItem('ex_token');
  return false;
}

//
=====
// NAVIGATION
//
=====

function setupNavTabs() {
  document.querySelectorAll('.nav-tab').forEach(btn => {
    btn.addEventListener('click', () => {
      const page = btn.dataset.page;
      navigateTo(page);
    });
  });
}

function navigateTo(page) {
  if (['trade', 'portfolio', 'history', 'wallet', 'admin'].includes(page) && !currentUser) {
    showModal('authModal');
    return;
  }
  document.querySelectorAll('.nav-tab').forEach(b => b.classList.remove('active'));
  document.querySelectorAll('.page').forEach(p => p.classList.remove('active'));
  document.querySelector(`.nav-tab[data-page="${page}"]`).classList.add('active');
  document.getElementById(`page${cap(page)}`).classList.add('active');
}

```

```

if (page === 'trade') initTradePage();
if (page === 'portfolio') loadPortfolio();
if (page === 'history') loadTrades();
if (page === 'wallet') loadWallet();
if (page === 'admin') loadAdmin();
}

```

```
function cap(s) { return s.charAt(0).toUpperCase() + s.slice(1); }
```

```
//
```

```
// WEBSOCKET
```

```
//
```

```
function connectWS() {
  const proto = location.protocol === 'https:' ? 'wss' : 'ws';
  ws = new WebSocket(`${proto}://${location.host}/ws`);
  const ind = document.getElementById('wsIndicator');

  ws.onopen = () => { ind?.classList.add('connected'); };
  ws.onclose = () => { ind?.classList.remove('connected'); setTimeout(connectWS, 3000); };
  ws.onerror = () => { ws.close(); };

  ws.onmessage = (e) => {
    try {
      const msg = JSON.parse(e.data);
      if (msg.type === 'prices') handlePriceUpdate(msg.data);
    } catch {}
  };
}
```

```
function handlePriceUpdate(data) {
  if (data && Array.isArray(data)) {
    data.forEach(a => { prices[a.symbol] = a; });
  }
  updateAssetCards();
  updateTradeAssetInfo();
  updateNavBalance();
}
```

```
//
```

```
// MARKET PAGE
```

```
//
```

```
async function loadAssets() {
  try {
    const data = await apiFetch('/assets');
    if (!data.success) {
      console.error('Failed to load assets:', data.message);
    }
  }
}
```

```

return;
}
allAssets = data.data || [];
tradeAssets = allAssets;
renderAssets(allAssets);
populateTradeSelect(allAssets);
} catch (err) {
console.error('loadAssets error:', err);
const grid = document.getElementById('assetsGrid');
if (grid) grid.innerHTML = `<div class="empty-state">Помилка завантаження: ${err.message}</div>`;
}
}

```

```

function renderAssets(assets) {
const grid = document.getElementById('assetsGrid');
if (!grid) return;
if (!assets || !assets.length) {
grid.innerHTML = '<div class="empty-state">Активи не знайдено</div>';
return;
}
grid.innerHTML = assets.map(a => assetCardHTML(a)).join("");
}

```

```

function assetCardHTML(a) {
const pData = prices[a.symbol] || a;
const price = parseFloat(pData.current_price || 0);
const chg = parseFloat(pData.change_pct || a.change_pct || 0);
const dir = chg >= 0 ? 'up' : 'down';
const chgStr = (chg >= 0 ? '+' : '-') + chg.toFixed(2) + '%';
return `
<div class="asset-card ${dir}" id="card_${a.symbol}" onclick="goTrade('${a.symbol}')">
<div class="asset-top">
<span class="asset-symbol">${escapeHtml(a.symbol)}</span>
<span class="asset-type-badge badge-${a.type}">${typeName(a.type)}</span>
</div>
<div class="asset-name">${escapeHtml(a.name)}</div>
<div class="asset-price" id="price_${a.symbol}">${fmtPrice(price)}</div>
<div class="asset-change ${dir}" id="chg_${a.symbol}">${chgStr}</div>
${a.volume_24h > 0 ? `<div class="asset-volume">Об'єм: ${fmtVol(a.volume_24h)}</div>` : ""}
<button class="asset-btn">Торгувати </button>
</div>`;
}

```

```

function updateAssetCards() {
if (!prices) return;
Object.entries(prices).forEach(([sym, p]) => {
const el = document.getElementById(`price_${sym}`);
const ce = document.getElementById(`chg_${sym}`);
if (!el) return;
const prev = parseFloat(el.dataset.prev || p.current_price);
const cur = parseFloat(p.current_price);
const chg = parseFloat(p.change_pct || 0);
const chgStr = (chg >= 0 ? '+' : '-') + chg.toFixed(2) + '%';

```

```

el.textContent = fmtPrice(cur);
el.dataset.prev = cur;
if (ce) {
  ce.textContent = chgStr;
  ce.className = `asset-change ${chg >= 0 ? 'up' : 'down'}`;
}
const card = document.getElementById(`card_${sym}`);
if (card) {
  const cls = cur > prev ? 'flash-green' : cur < prev ? 'flash-red' : '';
  if (cls) {
    card.classList.add(cls);
    setTimeout(() => card.classList.remove(cls), 600);
  }
  card.classList.remove('up', 'down');
  card.classList.add(chg >= 0 ? 'up' : 'down');
}
});
}

```

```

function setupFilterBtns() {
  const btns = document.querySelectorAll('.filter-btn');
  if (!btns.length) return;
  btns.forEach(btn => {
    btn.addEventListener('click', () => {
      document.querySelectorAll('.filter-btn').forEach(b => b.classList.remove('active'));
      btn.classList.add('active');
      currentType = btn.dataset.type;
      filterAssets();
    });
  });
}

```

```

const searchInput = document.getElementById('assetSearch');
if (searchInput) {
  searchInput.addEventListener('input', () => filterAssets());
}
}

```

```

function filterAssets() {
  const q = document.getElementById('assetSearch')?.value.toLowerCase() || '';
  const filtered = allAssets.filter(a => {
    const matchType = !currentType || a.type === currentType;
    const matchQ = !q || a.symbol.toLowerCase().includes(q) || a.name.toLowerCase().includes(q);
    return matchType && matchQ;
  });
  renderAssets(filtered);
}

```

```
//
```

```
// TRADE PAGE
```

```
//
```

```

window.goTrade = function(symbol) {
  navigateTo('trade');
  const sel = document.getElementById('tradeAssetSelect');
  if (sel) {
    sel.value = symbol;
    loadTradeAsset();
  }
};

```

```

function populateTradeSelect(assets) {
  const sel = document.getElementById('tradeAssetSelect');
  if (!sel) return;
  sel.innerHTML = '<option value="">- Оберіть актив -</option>' +
    assets.map(a => `<option value="${a.symbol}">${a.symbol} – $
    {escapeHtml(a.name)}</option>`).join("");
  sel.addEventListener('change', () => loadTradeAsset());
}

```

```

function initTradePage() {
  loadActiveOrders();
  const qtyInput = document.getElementById('orderQty');
  const priceInput = document.getElementById('orderPrice');
  if (qtyInput) qtyInput.addEventListener('input', () => calcTotal());
  if (priceInput) priceInput.addEventListener('input', () => calcTotal());

  const orderType = document.getElementById('orderType');
  if (orderType) orderType.addEventListener('change', () => togglePriceField());
}

```

```

window.setSide = function(side) {
  tradeSide = side;
  const btnBuy = document.getElementById('btnBuy');
  const btnSell = document.getElementById('btnSell');
  const btnPlace = document.getElementById('btnPlaceOrder');
  if (btnBuy) btnBuy.classList.toggle('active', side === 'buy');
  if (btnSell) btnSell.classList.toggle('active', side === 'sell');
  if (btnPlace) {
    btnPlace.textContent = side === 'buy' ? 'Купити' : 'Продати';
    btnPlace.className = `btn-place-order ${side}`;
  }
  calcTotal();
};

```

```

function togglePriceField() {
  const type = document.getElementById('orderType')?.value;
  const pf = document.getElementById('priceField');
  if (pf) pf.style.display = type === 'market' ? 'none' : 'block';
  calcTotal();
}

```

```

function calcTotal() {
  const qty = parseFloat(document.getElementById('orderQty')?.value) || 0;

```

```

const pInput = document.getElementById('orderPrice')?.value;
const type = document.getElementById('orderType')?.value;
const sym = document.getElementById('tradeAssetSelect')?.value;
const pData = prices[sym] || allAssets.find(a => a.symbol === sym);
const price = type === 'market' ? parseFloat(pData?.current_price || 0) : parseFloat(pInput || 0);

```

```

const sumEl = document.getElementById('orderSummary');
if (!sumEl) return;

```

```

if (!qty || !price) {
  sumEl.style.display = 'none';
  return;
}

```

```

const total = qty * price;
const fee = total * 0.001;
const grand = tradeSide === 'buy' ? total + fee : total - fee;

```

```

sumEl.style.display = 'block';
const totalEl = document.getElementById('sumTotal');
const feeEl = document.getElementById('sumFee');
const grandEl = document.getElementById('sumGrand');
if (totalEl) totalEl.textContent = fmtMoney(total);
if (feeEl) feeEl.textContent = fmtMoney(fee);
if (grandEl) grandEl.textContent = fmtMoney(grand);
}

```

```

async function loadTradeAsset() {
  const sym = document.getElementById('tradeAssetSelect')?.value;
  const box = document.getElementById('tradeAssetInfo');
  if (!sym) {
    if (box) box.style.display = 'none';
    return;
  }
}

```

```

const a = allAssets.find(x => x.symbol === sym);
if (!a) return;

```

```

const p = prices[sym] || a;
const chg = parseFloat(p.change_pct || a.change_pct || 0);
if (box) {
  box.style.display = 'block';
  box.innerHTML = `
<div class="aib-price">${fmtPrice(parseFloat(p.current_price))}</div>
<div class="aib-change ${chg >= 0 ? 'up td-green' : 'down td-red'}">${chg >= 0 ? '+' : ''}$
{chg.toFixed(2)}% за добу</div>`;
}

```

```

if (a.id) loadOrderBook(a.id);
calcTotal();
}

```

```

function updateTradeAssetInfo() {
  const sym = document.getElementById('tradeAssetSelect')?.value;
}

```

```

if (!sym || !prices[sym]) return;
const box = document.getElementById('tradeAssetInfo');
if (!box || box.style.display === 'none') return;
const p = prices[sym];
const chg = parseFloat(p.change_pct || 0);
const pEl = box.querySelector('.aib-price');
if (pEl) pEl.textContent = fmtPrice(parseFloat(p.current_price));
}

```

```

async function loadOrderBook(assetId) {
  try {
    const data = await apiFetch(`/assets/${assetId}/orderbook`);
    const el = document.getElementById('orderBook');
    if (!el) return;
    if (!data.success) {
      el.innerHTML = '<div class="empty-state">Стакан порожній</div>';
      return;
    }
    const { bids, asks } = data.data;
    let html = '<div class="ob-header"><span>Ціна</span><span>Кількість</span><span>Сума</span></div>';
    if (asks && asks.length) {
      asks.slice().reverse().forEach(r => {
        const t = parseFloat(r.price) * parseFloat(r.total_qty);
        html += '<div class="ob-row ob-ask"><span>${fmtPrice(r.price)}</span><span>${parseFloat(r.total_qty).toFixed(4)}</span><span>${fmtMoney(t)}</span></div>';
      });
    }
    if (bids && bids.length) {
      html += '<div class="ob-spread">Спред</div>';
      bids.forEach(r => {
        const t = parseFloat(r.price) * parseFloat(r.total_qty);
        html += '<div class="ob-row ob-bid"><span>${fmtPrice(r.price)}</span><span>${parseFloat(r.total_qty).toFixed(4)}</span><span>${fmtMoney(t)}</span></div>';
      });
    }
    if ((!asks || !asks.length) && (!bids || !bids.length)) {
      html = '<div class="empty-state">Стакан порожній</div>';
    }
    el.innerHTML = html;
  } catch (err) {
    console.error('loadOrderBook error:', err);
  }
}

```

```

window.placeOrder = async function() {
  if (!currentUser) {
    showModal('authModal');
    return;
  }
  const sym = document.getElementById('tradeAssetSelect')?.value;
  const asset = allAssets.find(a => a.symbol === sym);
  if (!asset) {
    showToast('Оберіть актив', 'error');
  }
}

```

```

return;
}

const qty = parseFloat(document.getElementById('orderQty')?.value);
const type = document.getElementById('orderType')?.value;
const price = document.getElementById('orderPrice')?.value;

if (!qty || qty <= 0) {
  showToast('Введіть кількість', 'error');
  return;
}
if (type !== 'market' && (!price || parseFloat(price) <= 0)) {
  showToast('Введіть ціну', 'error');
  return;
}

const btn = document.getElementById('btnPlaceOrder');
if (btn) {
  btn.disabled = true;
  btn.textContent = 'Обробка...';
}

try {
  const res = await apiFetch('/orders', 'POST', {
    asset_id: asset.id,
    type,
    side: tradeSide,
    quantity: qty,
    price: type === 'market' ? undefined : parseFloat(price)
  });
  if (res.success) {
    showToast(res.message, 'success');
    const qtyInput = document.getElementById('orderQty');
    const priceInput = document.getElementById('orderPrice');
    if (qtyInput) qtyInput.value = "";
    if (priceInput) priceInput.value = "";
    document.getElementById('orderSummary').style.display = 'none';
    loadActiveOrders();
    await refreshBalance();
    await loadPortfolio();
  } else {
    showToast(res.message || 'Помилка створення ордера', 'error');
  }
} catch (err) {
  showToast(err.message, 'error');
} finally {
  if (btn) {
    btn.disabled = false;
    btn.textContent = tradeSide === 'buy' ? 'Купити' : 'Продати';
  }
}
};

async function loadActiveOrders() {

```

```

if (!currentUser) return;
try {
const data = await apiFetch('/orders?status=open&limit=20');
const el = document.getElementById('activeOrdersList');
if (!el) return;
if (!data.success || !data.data || !data.data.length) {
el.innerHTML = '<div class="empty-state">Активних ордерів немає</div>';
return;
}
el.innerHTML = data.data.map(o => `
<div class="order-item">
<div class="oi-left">
<span class="oi-sym">${escapeHtml(o.symbol)} <span class="${o.side === 'buy' ? 'badge-buy' :
'badge-sell'}">${o.side === 'buy' ? 'КУПІВЛЯ' : 'ПРОДАЖ'}</span></span>
<span class="oi-detail">${o.type} · ${parseFloat(o.quantity).toFixed(4)} ${o.price ? '@' +
fmtPrice(o.price) : 'ринок'}</span>
</div>
<div class="oi-right">
<span class="badge-status s-open">Відкритий</span>
<button class="btn-cancel-order" onclick="cancelOrder(${o.id})">Скасувати</button>
</div>
</div>`).join("");
} catch (err) {
console.error('loadActiveOrders error:', err);
}
}

window.cancelOrder = async function(id) {
try {
const res = await apiFetch(`/orders/${id}`, 'DELETE');
if (res.success) {
showToast('Ордер скасовано', 'success');
loadActiveOrders();
} else {
showToast(res.message, 'error');
}
} catch (err) {
showToast(err.message, 'error');
}
};

//
//
//
// PORTFOLIO
//
//
//
async function loadPortfolio() {
if (!currentUser) return;
try {
const data = await apiFetch('/portfolio');
if (!data.success) return;
const { positions, totalValue, totalPnl } = data.data;

```

```

const summaryEl = document.getElementById('portfolioSummary');
if (summaryEl) {
  summaryEl.innerHTML = `
    <div class="psum-card"><div class="psum-label">Ринкова вартість</div><div class="psum-
value">${fmtMoney(totalValue)}</div></div>
    <div class="psum-card"><div class="psum-label">P&L</div><div class="psum-value ${totalPnl
>= 0 ? 'green' : 'red'}">${totalPnl >= 0 ? '+' : ''}${fmtMoney(totalPnl)}</div></div>
    <div class="psum-card"><div class="psum-label">Позицій</div><div class="psum-value">${
positions?.length || 0}</div></div>
    <div class="psum-card"><div class="psum-label">Баланс</div><div class="psum-value green">${
fmtMoney(currentUser?.balance || 0)}</div></div>`;
  }

const tableEl = document.getElementById('portfolioTable');
if (!tableEl) return;

if (!positions || !positions.length) {
  tableEl.innerHTML = '<div class="empty-state">Портфель порожній. Придбайте активи на
сторінці Торгівля.</div>';
  return;
}

tableEl.innerHTML = `
<table class="data-table">
<thead><tr><th>Актив</th><th>Кількість</th><th>Сер. ціна</th><th>Поточна
ціна</th><th>Ринк. вартість</th><th>P&L</th><th>P&L %</th></tr></thead>
<tbody>
${positions.map(p => `
<tr>
<td><strong>${escapeHtml(p.symbol)}</strong><br><small style="color:var(--text2)">${
escapeHtml(p.asset_name)}</small></td>
<td class="td-mono">${parseFloat(p.quantity).toFixed(4)}</td>
<td class="td-mono">${fmtPrice(p.avg_buy_price)}</td>
<td class="td-mono">${fmtPrice(p.current_price)}</td>
<td class="td-mono">${fmtMoney(p.market_value)}</td>
<td class="${p.pnl_abs >= 0 ? 'td-green' : 'td-red'}">${p.pnl_abs >= 0 ? '+' : ''}${
fmtMoney(p.pnl_abs)}</td>
<td class="${p.pnl_pct >= 0 ? 'td-green' : 'td-red'}">${p.pnl_pct >= 0 ? '+' : ''}${
parseFloat(p.pnl_pct).toFixed(2)}%</td>
</tr>`).join("")}
</tbody>
</table>`;
} catch (err) {
  console.error('loadPortfolio error:', err);
  showToast(err.message, 'error');
}

//

```

```
// TRADES HISTORY
```

```
//
=====

async function loadTrades() {
  if (!currentUser) return;
  try {
    const data = await apiFetch('/trades?limit=50');
    if (!data.success) return;
    const rows = data.data;
    const tableEl = document.getElementById('tradesTable');
    if (!tableEl) return;

    if (!rows || !rows.length) {
      tableEl.innerHTML = '<div class="empty-state">Угод ще немає</div>';
      return;
    }
    tableEl.innerHTML = `
    <table class="data-table">
      <thead><tr><th>Актив</th><th>Сторона</th><th>Кількість</th><th>Ціна</th><th>Сума</
th><th>Комісія</th><th>Час</th></tr></thead>
      <tbody>
        ${rows.map(t => `
        <tr>
          <td><strong>${escapeHtml(t.symbol)}</strong></td>
          <td><span class="${t.side === 'buy' ? 'badge-buy' : 'badge-sell'}">${t.side === 'buy' ? 'КУПІВЛЯ'
: 'ПРОДАЖ'}</span></td>
          <td class="td-mono">${parseFloat(t.quantity).toFixed(4)}</td>
          <td class="td-mono">${fmtPrice(t.price)}</td>
          <td class="td-mono">${fmtMoney(t.total_value)}</td>
          <td class="td-mono" style="color:var(--yellow)">${fmtMoney(t.buyer_fee || t.seller_fee || 0)}</td>
          <td style="color:var(--text2);font-size:12px">${fmtDate(t.executed_at)}</td>
        </tr>`).join("")}
        </tbody>
      </table>`;
  } catch (err) {
    console.error('loadTrades error:', err);
    showToast(err.message, 'error');
  }
}

//
=====

// WALLET
//
=====

async function loadWallet() {
  if (!currentUser) return;
  try {
    const data = await apiFetch('/balance');
    if (!data.success) return;
    const { balance, transactions } = data.data;
```

```

const balanceEl = document.getElementById('walletBalance');
if (balanceEl) balanceEl.textContent = fmtMoney(balance);

const txList = document.getElementById('txList');
if (!txList) return;

if (!transactions || !transactions.length) {
  txList.innerHTML = '<div class="empty-state">Транзакцій немає</div>';
  return;
}
txList.innerHTML = transactions.map(t => `
<div class="tx-item">
<span class="tx-type tx-${t.type}">${txTypeName(t.type)}</span>
<span class="tx-amount ${parseFloat(t.amount) > 0 ? 'td-green' : 'td-red'}">${
fmtMoney(t.amount)}</span>
<span class="tx-date">${fmtDate(t.created_at)}</span>
</div>`).join("");
} catch (err) {
  console.error('loadWallet error:', err);
  showToast(err.message, 'error');
}
}

window.showDepositForm = function() {
  const f = document.getElementById('walletForm');
  if (!f) return;
  f.style.display = 'block';
  f.innerHTML = `
<div class="form-group"><label>Сума поповнення (грн)</label>
<input class="text-input" type="number" id="depAmount" min="1" step="100"
placeholder="1000"/></div>
<button class="btn-green full-width" onclick="doDeposit()">Поповнити рахунок</button>`;
};

window.showWithdrawForm = function() {
  const f = document.getElementById('walletForm');
  if (!f) return;
  f.style.display = 'block';
  f.innerHTML = `
<div class="form-group"><label>Сума виведення (грн)</label>
<input class="text-input" type="number" id="wdrAmount" min="1" step="100"
placeholder="1000"/></div>
<button class="btn-red full-width" onclick="doWithdraw()">Вивести кошти</button>`;
};

window.doDeposit = async function() {
  const amt = parseFloat(document.getElementById('depAmount')?.value);
  if (!amt || amt <= 0) {
    showToast('Введіть суму', 'error');
    return;
  }
  try {
    const res = await apiFetch('/balance/deposit', 'POST', { amount: amt });
    if (res.success) {

```

```

showToast(res.message, 'success');
await refreshBalance();
loadWallet();
} else {
showToast(res.message, 'error');
}
} catch (err) {
showToast(err.message, 'error');
}
};

```

```

window.doWithdraw = async function() {
const amt = parseFloat(document.getElementById('wdrAmount')?.value);
if (!amt || amt <= 0) {
showToast('Введіть суму', 'error');
return;
}
try {
const res = await apiFetch('/balance/withdraw', 'POST', { amount: amt });
if (res.success) {
showToast(res.message, 'success');
await refreshBalance();
loadWallet();
} else {
showToast(res.message, 'error');
}
} catch (err) {
showToast(err.message, 'error');
}
};

```

```
//
```

```
// ADMIN
```

```
//
```

```

async function loadAdmin() {
if (!currentUser || currentUser.role !== 'admin') return;
try {
const [stats, users] = await Promise.all([
apiFetch('/admin/stats'),
apiFetch('/admin/users')
]);

```

```

const statsEl = document.getElementById('adminStats');
if (statsEl && stats.success) {
const d = stats.data;
statsEl.innerHTML = `
<div class="admin-stat-card"><div class="asc-label">Користувачі</div><div class="asc-
value">${d.totalUsers}</div></div>
<div class="admin-stat-card"><div class="asc-label">Ордери</div><div class="asc-value">${
d.totalOrders}</div></div>

```

```

<div class="admin-stat-card"><div class="asc-label">Угоди</div><div class="asc-value">${
d.totalTrades}</div></div>
<div class="admin-stat-card"><div class="asc-label">Об'єм торгів</div><div class="asc-
value">${fmtMoney(d.tradingVolume)}</div></div>
<div class="admin-stat-card"><div class="asc-label">Активи</div><div class="asc-value">${
d.activeAssets}</div></div>`;
}

```

```

const usersTableEl = document.getElementById('adminUsersTable');
if (usersTableEl && users.success && users.data && users.data.length) {
  usersTableEl.innerHTML = `
<table class="data-table">
<thead><tr><th>ID</th><th>Email</th><th>Ім'я</th><th>Роль</th><th>Статус</
th><th>Баланс</th><th>Реєстрація</th></tr></thead>
<tbody>
${users.data.map(u => `
<tr>
<td class="td-mono" style="color:var(--text3)">${u.id}</td>
<td>${escapeHtml(u.email)}</td>
<td>${escapeHtml(u.full_name)}</td>
<td><span class="badge-status s-open">${u.role}</span></td>
<td><span class="badge-status ${u.status === 'active' ? 's-filled' :
's-cancelled'}">${u.status}</span></td>
<td class="td-mono td-green">${fmtMoney(u.balance)}</td>
<td style="color:var(--text2);font-size:12px">${fmtDate(u.created_at)}</td>
</tr>`).join("")}
</tbody>
</table>`;
}
} catch (err) {
  console.error('loadAdmin error:', err);
  showToast(err.message, 'error');
}
}

```

```
//
```

```
// AUTH
```

```
//
```

```

window.doLogin = async function() {
  const email = document.getElementById('loginEmail')?.value.trim();
  const password = document.getElementById('loginPassword')?.value;
  hideAuthError();
  if (!email || !password) {
    showAuthError('Введіть email та пароль');
    return;
  }

```

```

  try {
    const data = await apiFetch('/auth/login', 'POST', { email, password });
    if (data.success && data.token) {

```

```

authToken = data.token;
localStorage.setItem('ex_token', authToken);
setUser(data.user);
closeModal('authModal');
showToast(`Вітаємо, ${data.user.full_name}!`, 'success');
// Оновлюємо дані
await loadPortfolio();
navigateTo('market');
} else {
showAuthError(data.message || 'Помилка входу');
}
} catch (err) {
console.error('Login error:', err);
showAuthError(err.message);
}
};

```

```

window.doRegister = async function() {
const full_name = document.getElementById('regName')?.value.trim();
const email = document.getElementById('regEmail')?.value.trim();
const password = document.getElementById('regPassword')?.value;
const phone = document.getElementById('regPhone')?.value.trim();
hideAuthError();

```

```

if (!full_name || !email || !password) {
showAuthError('Заповніть обов'язкові поля');
return;
}
if (password.length < 6) {
showAuthError('Пароль мінімум 6 символів');
return;
}

```

```

try {
const data = await apiFetch('/auth/register', 'POST', { full_name, email, password, phone });
if (data.success && data.token) {
authToken = data.token;
localStorage.setItem('ex_token', authToken);
setUser(data.user);
closeModal('authModal');
showToast(data.message, 'success');
navigateTo('market');
} else {
showAuthError(data.message || 'Помилка реєстрації');
}
} catch (err) {
console.error('Register error:', err);
showAuthError(err.message);
}
};

```

```

window.logout = async function() {
try {
await apiFetch('/auth/logout', 'POST');

```

```

    } catch {}
    authToken = null;
    currentUser = null;
    localStorage.removeItem('ex_token');
    const userMenu = document.getElementById('userMenu');
    const btnLogin = document.getElementById('btnLogin');
    const navBalance = document.getElementById('navBalance');
    const tabAdmin = document.getElementById('tabAdmin');
    if (userMenu) userMenu.style.display = 'none';
    if (btnLogin) btnLogin.style.display = 'block';
    if (navBalance) navBalance.style.display = 'none';
    if (tabAdmin) tabAdmin.style.display = 'none';
    navigateTo('market');
    showToast('Ви вийшли з системи', 'success');
  };

function setUser(user) {
  currentUser = user;
  const userMenu = document.getElementById('userMenu');
  const btnLogin = document.getElementById('btnLogin');
  const userName = document.getElementById('userName');
  const navBalance = document.getElementById('navBalance');
  const tabAdmin = document.getElementById('tabAdmin');

  if (userMenu) userMenu.style.display = 'flex';
  if (btnLogin) btnLogin.style.display = 'none';
  if (userName) userName.textContent = user.full_name?.split(' ')[0] || user.full_name;
  if (navBalance) {
    navBalance.style.display = 'block';
    navBalance.textContent = fmtMoney(user.balance);
  }
  if (tabAdmin && user.role === 'admin') tabAdmin.style.display = 'block';
}

async function refreshBalance() {
  try {
    const data = await apiFetch('/balance');
    if (data.success && currentUser) {
      currentUser.balance = data.data.balance;
      const navBalance = document.getElementById('navBalance');
      if (navBalance) navBalance.textContent = fmtMoney(data.data.balance);
    }
  } catch (err) {
    console.error('refreshBalance error:', err);
  }
}

function updateNavBalance() {
  if (!currentUser) return;
  const navBalance = document.getElementById('navBalance');
  if (navBalance && currentUser.balance !== undefined) {
    navBalance.textContent = fmtMoney(currentUser.balance);
  }
}

```

```
function switchAuthTab(tab) {
  const loginForm = document.getElementById('loginForm');
  const registerForm = document.getElementById('registerForm');
  const tabLogin = document.getElementById('tabLogin');
  const tabRegister = document.getElementById('tabRegister');

  if (loginForm) loginForm.style.display = tab === 'login' ? 'block' : 'none';
  if (registerForm) registerForm.style.display = tab === 'register' ? 'block' : 'none';
  if (tabLogin) tabLogin.classList.toggle('active', tab === 'login');
  if (tabRegister) tabRegister.classList.toggle('active', tab === 'register');
  hideAuthError();
}
```

```
function showAuthError(msg) {
  const el = document.getElementById('authError');
  if (el) {
    el.textContent = msg;
    el.style.display = 'block';
  }
}
function hideAuthError() {
  const el = document.getElementById('authError');
  if (el) el.style.display = 'none';
}
```

```
//
```

```
// MODAL
//
```

```
window.showModal = function(id) {
  const modal = document.getElementById(id);
  if (modal) modal.classList.add('open');
};
window.closeModal = function(id) {
  const modal = document.getElementById(id);
  if (modal) modal.classList.remove('open');
};
window.closeModalOutside = function(e, id) {
  if (e.target.id === id) closeModal(id);
};
```

```
//
```

```
// TOAST
//
```

```
let toastTimer = null;
function showToast(msg, type = "") {
```

```

const el = document.getElementById('toast');
if (!el) return;
el.textContent = msg;
el.className = `toast show ${type}`;
clearTimeout(toastTimer);
toastTimer = setTimeout(() => { el.classList.remove('show'); }, 3500);
}

//
=====
// API HELPER (ВИПРАВЛЕНА)
//
=====
async function apiFetch(path, method = 'GET', body = null) {
  const headers = { 'Content-Type': 'application/json' };
  if (authToken) headers['Authorization'] = `Bearer ${authToken}`;

  const opts = {
    method,
    headers,
    credentials: 'include'
  };
  if (body) opts.body = JSON.stringify(body);

  const res = await fetch(API + path, opts);

  // Пробуєм парсити JSON
  let data;
  const text = await res.text();
  try {
    data = JSON.parse(text);
  } catch {
    throw new Error(`Помилка сервера: ${res.status} ${text.substring(0, 100)}`);
  }

  if (!res.ok) {
    throw new Error(data.message || `HTTP ${res.status}`);
  }

  return data;
}

//
=====
// FORMATTERS
//
=====
function fmtPrice(v) {
  const n = parseFloat(v);
  if (isNaN(n)) return '0.00';

```

```

    if (n >= 1000) return n.toLocaleString('uk-UA', { minimumFractionDigits: 2,
maximumFractionDigits: 2 });
    if (n >= 1) return n.toFixed(4);
    return n.toFixed(6);
}

function fmtMoney(v) {
    const n = parseFloat(v);
    if (isNaN(n)) return '0.00 ₪';
    return n.toLocaleString('uk-UA', { minimumFractionDigits: 2, maximumFractionDigits: 2 }) + ' ₪';
}

function fmtVol(v) {
    const n = parseFloat(v);
    if (isNaN(n)) return '0';
    if (n >= 1e9) return (n / 1e9).toFixed(2) + 'B';
    if (n >= 1e6) return (n / 1e6).toFixed(2) + 'M';
    if (n >= 1e3) return (n / 1e3).toFixed(2) + 'K';
    return n.toFixed(0);
}

function fmtDate(d) {
    if (!d) return '';
    return new Date(d).toLocaleString('uk-UA', { day:'2-digit', month:'2-digit', year:'numeric', hour:'2-
digit', minute:'2-digit' });
}

function typeName(t) {
    const types = {
        stock: 'Акція',
        crypto: 'Крипто',
        forex: 'Форекс',
        commodity: 'Товар'
    };
    return types[t] || t;
}

function txTypeName(t) {
    const types = {
        deposit: 'Поповнення',
        withdrawal: 'Виведення',
        trade_buy: 'Купівля',
        trade_sell: 'Продаж',
        commission: 'Комісія',
        bonus: 'Бонус'
    };
    return types[t] || t;
}

function escapeHtml(str) {
    if (!str) return '';
    return str
        .replace(/&/g, '&amp;')
        .replace(/</g, '&lt;');
}

```

```
.replace(>/g, '&gt;')  
.replace(/"/g, '&quot;')  
.replace(/'/g, '&#39;');  
}
```

Додаток Б

API ендпоінти системи

Таблиця Б.1 – Повний перелік REST API ендпоінтів ExchangeUA

Метод	URL	Доступ	Опис
POST	/api/auth/register	Public	Реєстрація нового трейдера
POST	/api/auth/login	Public	Вхід в систему
POST	/api/auth/logout	Auth	Вихід
GET	/api/auth/profile	Auth	Профіль поточного користувача
PUT	/api/auth/profile	Auth	Оновлення профілю
GET	/api/assets	Public	Список всіх активів з цінами
GET	/api/assets/:id	Public	Деталі одного активу
GET	/api/assets/:id/history	Public	OHLCV-дані з price_history (для майбутньої побудови графіків)
GET	/api/assets/:id/orderbook	Public	Стакан ордерів
POST	/api/orders	Auth	Створення нового ордеру
GET	/api/orders	Auth	Мої ордери
DELETE	/api/orders/:id	Auth	Скасування ордеру
GET	/api/portfolio	Auth	Портфель з P&L
GET	/api/trades	Auth	Мої угоди
GET	/api/balance	Auth	Баланс та транзакції
POST	/api/balance/deposit	Auth	Поповнення балансу
POST	/api/balance/withdraw	Auth	Виведення коштів
GET	/api/notifications	Auth	Список сповіщень
POST	/api/notifications/read-all	Auth	Позначити всі прочитаними
GET	/api/admin/stats	Admin	Статистика платформи
GET	/api/admin/users	Admin	Список всіх користувачів
PUT	/api/admin/users/:id	Admin	Оновлення ролі/статусу
GET	/api/news	Public	Стрічка новин
POST	/api/news	Admin/ Issuer	Публікація новини
GET	/api/news/:id	Public	Деталі новини
POST	/api/assets/:id/quote	Issuer/ Admin	Котирування активу (встановлення нової ціни)
GET	/api/assets/:id/quotes-log	Public	Журнал котирувань активу
GET	/api/admin/assignments	Admin	Список призначень емітентів на активи
POST	/api/admin/assignments	Admin	Призначити емітента на актив
продовження табл. Б.1			
Метод	URL	Доступ	Реєстрація нового трейдера

DELETE	/api/admin/assignments	Admin	Відкликати призначення емітента
GET	/api/issuer/my-assets	Issuer/ Admin	Перелік активів, на які емітент має право котирувати
GET	/api/health	Public	Перевірка стану сервера
GET	/api/debug	Public	Діагностичний ендпоінт (development)
DELETE	/api/assets/:id	Admin/ Issuer	Видалення активу (м'яке або повне залежно від наявності ордерів)
POST	/api/auth/change-password	Auth	Зміна власного пароля
POST	/api/assets	Admin/ Issuer	Створення нового активу (при створенні емітентом – автоматичне призначення)
PUT	/api/assets/:id	Admin/ Issuer	Редагування активу (назва, опис, налаштування авто-регулювання ціни)
POST	/api/assets/:id/supply	Admin/ Issuer	Управління інвентарем активу (add/withdraw для total_supply і available_supply)
GET	/api/assets/:id/supply-log	Admin/ Issuer	Журнал змін інвентарю активу
POST	/api/assets/:id/cash-reserve	Admin/ Issuer	Управління Cash Reserve активу (для active_mm моделі)
GET	/api/assets/:id/cash-reserve-log	Admin/ Issuer	Журнал змін Cash Reserve активу
DELETE	/api/news/:id	Admin/ Issuer	Видалення новини (адміністратор будь-яку, емітент – лише свої)

Додаток В

Інструкція з розгортання системи

Для розгортання платформи ExchangeUA необхідно виконати такі кроки:

1. Встановити Node.js ≥ 18.0 та MySQL ≥ 8.0 на сервер або локальну машину.
2. Завантажити репозиторій та встановити залежності командою `npm install`.
3. Скопіювати `.env.example` у `.env` та заповнити всі змінні середовища (DB_HOST, DB_USER, DB_PASSWORD, SESSION_SECRET тощо).
4. Ініціалізувати базу даних командою `npm run init-db`. Скрипт автоматично створить 14 таблиць схеми та завантажить початковий каталог з 12 активами. Тестові облікові записи створюються наступним кроком.
5. Створити тестові облікові записи (admin, trader, issuer) командою `npm run setup-users`. Виводиться список облікових даних для входу.
6. (опційно) Прогнати тестові набори командою `npm test`. Очікуваний результат: 160 пройдених тестів у 4 категоріях (JWT, бізнес-логіка, безпека, API-інтеграція).
7. Запустити сервер командою `npm start` (продакшн) або `npm run dev` (режим розробки з автоперезавантаженням).
8. Відкрити браузер за адресою `http://localhost:3000`. Облікові дані за замовчуванням: `admin@exchange.ua / Admin@123` (адміністратор, баланс 12 000 000 грн), `trader@example.com / Trader@123` (трейдер, баланс 500 000 грн), `issuer@exchange.ua / Issuer@123` (емітент, баланс 1 500 000 грн – це 2 000 000 грн мінус 500 000 грн на cash reserve демо-активу RTC). Демо-користувачам початковий баланс встановлено для зручності демонстрації функціоналу; нові акаунти, створені через реєстрацію на сайті, мають нульовий баланс і поповнюються окремо через сторінку Гаманець.

Додаток И**НАЦІОНАЛЬНИЙ
УНІВЕРСИТЕТ БІОРЕСУРСІВ І
ПРИРОДОКОРИСТУВАННЯ
УКРАЇНИ****Факультет (ННІ) Інформаційних Технологій****Декларація про академічну доброчесність та використання ШІ**

Я, Харченко Олександр Сергійович, здобувач НУБіП України, усвідомлюючи відповідальність за порушення академічної доброчесності, цією заявою підтверджую, що:

1. Моя бакалаврська кваліфікаційна робота (проект) на тему
«Веборієнтована інформаційна система біржі як багатосторонньої цифрової платформи для взаємодії користувачів» є результатом моєї особистої праці.
2. Усі запозичення з текстів інших авторів мають відповідні посилання згідно з чинними стандартами.
3. Щодо використання інструментів ШІ зазначаю, що інструменти ШІ ChatGPT, були використані для: перекладу іншомовних джерел; стилістичного редагування та перевірки граматики; допомоги у написанні/відлагодженні програмного коду;

Я розумію, що надання недостовірної інформації може призвести до скасування результатів захисту та відрахування з числа здобувачів НУБіП України.

«__» _____ 20__ р. _____

(підпис)

Олександр Харченко

Додаток Е

РИСУНКИ ТЕСТУВАНЬ

```
> node tests/runAll.js
```

ExchangeUA – Запуск тестових наборів

 JWT – Unit-тести

- ✓ signToken повертає рядок із 3 частинами через крапку
- ✓ verifyToken коректно повертає payload
- ✓ verifyToken містить iat і exp
- ✓ Дефолтний TTL = 1 година (3600 секунд)
- ✓ Підроблений підпис відхиляється
- ✓ Змінений payload відхиляється (захист від підробки ролі)
- ✓ Прострочений токен відхиляється
- ✓ Custom TTL працює (5m = 300 сек)

 Результат: 8 пройдено, 0 провалено

📁 Бізнес-логіка – Unit-тести

- ✓ Комісія 0.1% від 1000 = 1
- ✓ Комісія від 67845 (BTC) $\approx$ 67.845
- ✓ Купівля 0.01 BTC за 67845 $\rightarrow$ totalCost = 678.45, commission = 0.67845
- ✓ Спочатку купуємо 1 BTC по 60000 – avg = 60000
- ✓ Докупуємо 1 BTC по 80000 $\rightarrow$ avg = 70000
- ✓ 1 BTC по 60000 + 0.5 BTC по 80000 $\rightarrow$ avg = 70000
- ✓ P&L: купили по 60000, поточна 66000 $\rightarrow$ +10%
- ✓ P&L: купили по 100, поточна 75 $\rightarrow$ -25%
- ✓ P&L: avg = 0 $\rightarrow$ P&L = 0 (захист від ділення на нуль)
- ✓ Random walk не дає від'ємних або нульових цін (1000 ітерацій)
- ✓ Максимальна зміна за рік не перевищує 2.1% (1000 ітерацій)
- ✓ Середня зміна за 10000 тиків близька до 0 (random walk без зміщення)
- ✓ Баланс 1000 $\rightarrow$ купити 1 ETH за 999 неможливо (з комісією)
- ✓ Баланс 1000 $\rightarrow$ купити за 999.5 не вистачить (1000.4995)
- ✓ Баланс 0 $\rightarrow$ жодна купівля неможлива
- ✓ Статус "open" коли нічого не виконано
- ✓ Статус "partial" коли частина виконана
- ✓ Статус "filled" коли все виконано
- ✓ Статус "filled" з мікропохибкою (9.99999999 / 10)
- ✓ Котирування 100 $\rightarrow$ 120 (+20%) дозволено
- ✓ Котирування 100 $\rightarrow$ 149 (+49%) дозволено
- ✓ Котирування 100 $\rightarrow$ 151 (+51%) заборонено
- ✓ Котирування 100 $\rightarrow$ 50 (-50%) дозволено (на межі)
- ✓ Котирування 100 $\rightarrow$ 49 (-51%) заборонено
- ✓ Котирування ціни $\leq$ 0 заборонено
- ✓ Перше котирування (oldPrice = 0) дозволено будь-яке додатне
- ✓ STOP-SELL спрацьовує коли ціна впала до stop_price
- ✓ STOP-BUY спрацьовує коли ціна піднялась до stop_price
- ✓ Market: qty > 0 – ОК
- ✓ Limit без ціни – відхилено
- ✓ Stop без стоп-ціни – відхилено
- ✓ Stop_limit без обох цін – відхилено
- ✓ Stop_limit з обома цінами – ОК
- ✓ Impact: купівля 10 шт $\times$ 0.1% $\rightarrow$ +1%
- ✓ Impact: продаж 10 шт $\times$ 0.1% $\rightarrow$ -1%
- ✓ Impact вимкнено $\rightarrow$ ціна не змінюється
- ✓ Impact: pct=0 $\rightarrow$ ціна не змінюється
- ✓ Impact: qty=0 $\rightarrow$ ціна не змінюється
- ✓ Impact: великий продаж не пускає ціну в негатив
- ✓ Supply add: 10000 + 5000 $\rightarrow$ total=15000, available=15000
- ✓ Supply withdraw: 10000/10000 – 3000 $\rightarrow$ total=7000, available=7000
- ✓ Supply withdraw: не можна відкликати більше доступного (захист портфелів трейдерів)
- ✓ Купівля трейдером: available зменшується, total НЕ змінюється
- ✓ Продаж трейдером: available збільшується (повертається в обіг)
- ✓ Купівля при недостатній пропозиції $\rightarrow$ помилка
- ✓ Свічка: нова – open=high=low=close=ціна
- ✓ Свічка: оновлення вищою ціною $\rightarrow$ high зростає, low не змінюється
- ✓ Свічка: оновлення нижчою ціною $\rightarrow$ low падає, high не змінюється
- ✓ Свічка: послідовність 100 $\rightarrow$ 120 $\rightarrow$ 90 $\rightarrow$ 110 $\rightarrow$ O=100, H=120, L=90, C=110
- ✓ Бакет години: 15:23:45 $\rightarrow$ 15:00:00
- ✓ Бакет години: різні хвилини однієї години $\rightarrow$ один бакет
- ✓ Limit BUY @ 1111, current = 505 $\rightarrow$ fallback ✓ (BUY вигідний)
- ✓ Limit BUY @ 400, current = 505 $\rightarrow$ НЕ fallback (BUY не вигідний)

- ✓ Limit BUY @ 400, current = 505 → HE fallback (BUY не вигідний)
- ✓ Limit BUY @ 505, current = 505 → fallback ✓ (рівно)
- ✓ Limit SELL @ 300, current = 505 → fallback ✓ (SELL вигідний)
- ✓ Limit SELL @ 800, current = 505 → HE fallback (SELL зачекає)
- ✓ Limit SELL @ 505, current = 505 → fallback ✓ (рівно)
- ✓ Mid-price = (bid + ask) / 2 – bid=500, ask=510 → mid=505
- ✓ Mid-price з нерівним спредом – bid=100, ask=200 → mid=150
- ✓ Спред у грн – bid=500, ask=510 → 10
- ✓ Спред у % – bid=500, ask=510 → 2.0%
- ✓ Active MM BUY → за ask_price (трейдер платить дорожче)
- ✓ Active MM SELL → за bid_price (трейдер отримує менше)
- ✓ Cash reserve: BUY 1 шт за ask=510 → reserve +510
- ✓ Cash reserve: SELL 1 шт за bid=500 → reserve -500
- ✓ Цикл купити→продати дає прибуток у розмірі спреду
- ✓ Reserve достатній для викупу – 10000 >= 500*1
- ✓ Reserve недостатній – 100 < 500*1 → відхиляємо
- ✓ Reserve рівний – 500 == 500*1 → дозволяємо
- ✓ Валідне котирування bid=500 < ask=510
- ✓ Невалідне: bid=510 == ask=510 (немає спреду)
- ✓ Невалідне: bid=520 > ask=510 (інвертовано)
- ✓ Невалідне: bid=0 (нуль)
- ✓ Asymmetric BUY: oldPrice=100, qty=10, buy=0.2%, sell=0.05% → ціна +2%
- ✓ Asymmetric SELL: oldPrice=100, qty=10, buy=0.2%, sell=0.05% → ціна -0.5%
- ✓ Симетричний випадок: buy=sell=0.1% → купівля і продаж 1 шт. рівні по абс. величині
- ✓ Тільки buy impact (sell=0): SELL не змінює ціну
- ✓ Тільки sell impact (buy=0): BUY не змінює ціну
- ✓ Обидва pct = 0: жодна сторона не змінює ціну
- ✓ Захист від негативної ціни: екстремальний sell не пускає в негатив
- ✓ Asymmetric detection: buy=0.2, sell=0.05 → asymmetric
- ✓ Asymmetric detection: buy=0.1, sell=0.1 → симетричний
- ✓ Валідний email: user@example.com
- ✓ Валідний email з крапками в локальній частині: name.surname@company.co.uk
- ✓ Валідний email з підкресленням: my_name@domain.io
- ✓ Невалідний: просто текст без @
- ✓ Невалідний: тільки @ без частин
- ✓ Невалідний: немає крапки в домені (abc@def)
- ✓ Невалідний: пробіл у email
- ✓ Невалідний: порожній рядок
- ✓ Невалідний: null / undefined
- ✓ Невалідний: дуже довгий email (>254 символів)
- ✓ Active trader може діяти
- ✓ Frozen trader НЕ може діяти
- ✓ Frozen issuer НЕ може котирувати
- ✓ Admin діє навіть якщо frozen (адмін над собою)
- ✓ Suspended trader НЕ може діяти
- ✓ Pending користувач НЕ може діяти
- ✓ Статус 'frozen' – валідний у ENUM
- ✓ Статус 'unknown' – невалідний

 Результат: 100 пройдено, 0 провалено

```

🔒 Безпека та валідація – Unit-тести

✓ Admin має доступ до admin-only
✓ Trader не має доступу до admin-only
✓ Trader не має доступу до issuer або admin
✓ Issuer має доступ до спільних admin/issuer
✓ Валідний email "user@example.com"
✓ Невалідний email "not-an-email"
✓ SQL-ін'єкція в email відсіюється формальною валідацією
✓ Пароль "12345" замалий (< 6)
✓ Пароль "Admin@123" валідний
✓ bcrypt хеш не дорівнює оригіналу
✓ bcrypt.compare працює з правильним паролем
✓ bcrypt.compare відхиляє неправильний пароль
✓ bcrypt cost factor = 12 (як у записці)
✓ escapeHtml блокує <script>
✓ escapeHtml не псує звичайний текст
✓ escapeHtml на порожньому/null повертає порожній рядок
✓ Тип "market" валідний
✓ Тип "limit" валідний
✓ Тип "INVALID" невалідний
✓ Інжекція в тип ордеру відсіюється
✓ Сторона "buy" валідна
✓ Сторона "long" невалідна
✓ Депозит 100 валідний
✓ Депозит 0 невалідний
✓ Депозит -100 невалідний
✓ Депозит 10000001 невалідний (понад ліміт)
✓ Депозит "abc" невалідний
✓ Зміна пароля: порожні поля → відхилено
✓ Зміна пароля: новий пароль < 6 символів → відхилено
✓ Зміна пароля: новий і підтвердження не співпадають → відхилено
✓ Зміна пароля: новий = старий → відхилено
✓ Зміна пароля: коректні дані → дозволено
✓ Admin може видаляти будь-який актив
✓ Issuer може видаляти ТІЛЬКИ призначені активи
✓ Trader не може видаляти активи
✓ Невідома роль не може видаляти активи

🏁 Результат: 36 пройдено, 0 провалено

🌐 API – Інтеграційні тести

✓ GET /api/health → 200 OK
✓ GET /api/assets → 200, повертає масив
✓ GET /api/assets містить change_pct
✓ GET /api/news → 200 (публічно)
✓ GET /api/portfolio без токена → 401
✓ GET /api/orders без токена → 401
✓ POST /api/orders без токена → 401
✓ GET /api/balance без токена → 401
✓ GET /api/admin/users без токена → 401
✓ GET /api/admin/stats без токена → 401
✓ Trader не має доступу до /api/admin/stats (через мок-БД)
✓ GET /api/nonexistent → 404 JSON
✓ POST /api/auth/login без email → 400
✓ POST /api/auth/login з SQL-ін'єкцією → не падає (200 або 401)
✓ POST /api/auth/register без full_name → 400
✓ POST /api/auth/register з коротким паролем → 400

🏁 Результат: 16 пройдено, 0 провалено

```

Рисунки 3.4. Результат прогону всіх 160 тестів у консолі