МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В.Н.Каразіна

Факультет математики і інформатики

Кафедра теоретичної та прикладної інформатики

Кваліфікаційна робота

Магістр

на тему: Застосування блокчейн-технологій у системах управління ланцюгами постачання

Виконав: студент 2 курсу, групи МФ-61
спеціальність 122 «Комп'ютерні науки»
освітньо-наукова програма «Інформатика»

Олефіренко І. І.
(прізвище та ініціали)

Керівник Фролов В. В.
(прізвище та ініціали)

Консультант Бережна Н. І.
(прізвище та ініціали)

Рецензент ______________________
(прізвище та ініціали)

Харків – 2025 року

ЗМІСТ

1. ВСТУП…………………………………………………………...3
1.1 Мета та задачі роботи……………………………………………..4
1.2 Актуальність роботи………………………………………………5
2. РОЗРОБКА АРХІТЕКТУРИ………………………………………7
2.1 Опис функціональності системи…………………………………7
2.2 Огляд використаних технологій…………………………………...12
2.3 Моделювання бази даних…………………………………………..17
2.4 Концептуальна модель предметної галузі………………………...25
3. РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ………………………………33
3.1 Основні архітектурні патерни та шаблони………………………..33
3.2 Опис моделі…………………………………………………………44
3.3 Реалізація логіки обробки запитів та бізнес логіки…………………48
3.4 Взаємодія між обробкою запитів та моделями даних………………56
4. ІНТЕГРАЦІЯ БЛОКЧЕЙН ТЕХНОЛОГІЙ………………………...61
4.1 Реалізація блокчейн технологій у серверній частині………………61
4.2 Використання блокчейн технологій у клієнтській частині………...71
4.3 Майбутні перспективи розвитку проекту…………………………...74
ВИСНОВКИ………………………………………………………………76
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ……………………………77

ВСТУП

Останнім часом світова економіка все більше відчуває на собі вплив глобалізації, одночасно стискаючись з політичними потрясіннями. Насамперед значного удару зазнала світова логістика. Через пандемію, несприятливі політичні умови та економічні негаразди ланцюги постачання стають все менш ефективними та несуть у собі куди більше витрат.

Одним із перспективних технологічних напрямків сьогодні є блокчейн технології. Використання блокчейну, як розподіленого реєстру дозволяє створювати прозору та захищену історію транзакцій. Також використання блокчейну може значно зменшити витрати та збільшити ефективність завдяки децентралізації та виключення посередників.

Зараз, більше и більше великих компаній намагаються відмовлятися від класичного підходу у ланцюгах постачання. Сучасні компанії роблять значні інвестиції та технологічні вкладення у інтеграцію блокчейну в ланцюги постачання. Такими діями вони рухаються у майбутнє та задають новітні світові тренди. Тепер усі бажаючі можуть приєднатися до цієї ідеї та допомогти розвинути блокчейн як альтернативу, все менш надійним, класичним ланцюгам постачання.

1.1 Мета та задачі роботи

Метою даної кваліфікаційної роботи є розробка децентралізованого сервісу на основі блокчейн-технологій, який забезпечує прозору та безпечну взаємодію між учасниками ланцюга постачання. У межах проекту реалізовано веб-сервіс, що дозволяє:

- Реєстрацію учасників, якими можуть виступати виробники, дистриб'ютори, роздрібні продавці, покупці- з використанням криптогаманців для ідентифікації;
- Зберігання даних у блокчейні як у розподіленому реєстрі, що фіксує інформацію про угоди, товари, їх переміщення, доставку та оплату;
- Інтеграцію блокчейн-платежів для автоматизації розрахунків між учасниками;
- Візуалізацію ланцюга постачання через веб-інтерфейс, що надає доступ до повної історії угод та ключової інформації;

Для досягнення описаної мети треба виділити наступні задачі:

- Аналіз функціональності;
- Розробка моделі системи;
- Проектування та розробка бази даних;
- Розробка серверної частини сервісу;
- Інтеграцій блокчейну у сервіс;

Розроблений сервіс сприятиме зменшенню бюрократії, запобіганню шахрайства та підвищенню довіри між учасниками за рахунок використання публічного реєстру транзакцій. У роботі досліджено принципи роботи блокчейну, проаналізовано існуючі рішення та впроваджено власну систему, яка може бути масштабована для різних галузей економіки.

1.2 Актуальність роботи

У сучасні часи ланцюги постачання у світовій економіці стають дедалі складнішими, що зумовлює потребу у впровадженні інноваційних технологій для підвищення їх прозорості, ефективності та безпеки. Традиційні системи управління, засновані на концепції централізації, часто стикаються з такими проблемами, як недостатній рівень довіри між учасниками, можливість фальсифікації даних, неефективність обміну інформацією та високі витрати на посередників.

Блокчейн-технологія, завдяки своїй децентралізованій природі, незмінності записів і можливості автоматизації, пропонує принципово новий підхід до організації ланцюгів постачання. Вона дозволяє створити загальну, захищену від несанкціонованих змін систему, доступ до якої мають усі зацікавлені учасники. Це значно підвищує прозорість усіх операцій, спрощує відстеження товарів, автоматизує платежі та зменшує ризики шахрайства.

Останні роки блокчейн-технології активно інтегруються в системи управління ланцюгами постачання (SCM), пропонуючи рішення для підвищення прозорості, відстежуваності та автоматизації бізнес-процесів. Можливо навести низку прикладів існуючих децентралізованих платформ, які вже змінюють індустрію:

- VeChainThor. Сфера застосування: логістика, роздрібна торгівля, боротьба з контрафактом;
- IBM Food Trust. Сфера застосування: харчова промисловість;
- OriginTrail. Сфера застосування: міжгалузеві ланцюги постачання;
- Ambrosus. Сфера застосування: фармацевтика, продукти харчування;
- Morpheus Network Сфера застосування: міжнародна логістика;

Ці платформи демонструють, що блокчейн уже зараз трансформує SCM, усуваючи традиційні проблеми довіри та неефективності. Впровадження подібних рішень дозволяє компаніям зменшити витрати та підвищити лояльність клієнтів за рахунок повної прозорості. Розробка децентралізованого сервісу може стати частиною цієї глобальної тенденції, запропонувавши новий рівень автоматизації та безпеки для учасників ринку.

Розробка децентралізованого сервісу на основі блокчейну є особливо актуальною, оскільки вона дозволяє об'єднати виробників, дистриб'юторів, продавців і споживачів у єдиному цифровому середовищі, де всі транзакції фіксуються миттєво, а оплата виконуються автоматично за умови виконання угод. Це не лише оптимізує бізнес-процеси, але й створює основу для розвитку нових моделей співпраці, зокрема в умовах зростання популярності електронної комерції та глобалізації ринків.

Таким чином, дана робота є актуальною, оскільки вона спрямована на вирішення ключових проблем сучасних ланцюгів постачання за допомогою інноваційних блокчейн-рішень, що відповідає світовим тенденціям, цифровізації економіки та підвищення ефективності бізнес-процесів.

2. РОЗРОБКА АРХІТЕКТУРИ

2.1 Опис функціональності системи

Розроблений децентралізований сервіс на основі блокчейн-технологій призначений для оптимізації управління ланцюгами постачання, забезпечуючи прозорість, безпеку та автоматизацію ключових процесів. Система надає наступні функціональні можливості:

1. Реєстрація учасників ланцюга постачання:
   - Можливість реєстрації для продавців та покупців;
   - Криптогаманці (наприклад Binance wallet) використовуються для ідентифікації користувачів, що підвищує рівень довіри та захищає від фальсифікації даних;
2. Децентралізоване зберігання даних у блокчейні:
   - Розподілений реєстр фіксує всю інформацію про товари, включаючи: унікальні ідентифікатори, дані про наявні угоди, транспортування, оплату та доставку, історію оплат та транзакцій;
   - Незмінність записів гарантує, що дані не можуть бути змінені або видалені непомітно;
3. Блокчейн-платежі та автоматизація розрахунків:
   - Використання блокчейну для автоматичного проведення платежів між учасниками (наприклад, оплата за товар або підтвердження отримання чи відправлення товару);
   - Фіксація транзакцій у блокчейні з деталізацією: учасник, характеристики товару (артикул, кількість, вартість), пункти відправлення та призначення;
   - Інтеграція з криптогаманцями (через API або Web3) для безпечного проведення платежів;
4. Відстеження ланцюга постачання через веб-інтерфейс:

- Прозора історія угод з можливістю перегляду: час та підтвердження кожного етапу (від замовлення до оплати), статус доставки;
- Можливість для учасників оновлювати статус угод;

5. Монетизація:
   - Сервіс може мати модель монетизації через оплату символічних сум для виконання транзакцій, щоб оновити статус угод;
   - Монетизаційні стратегії повинні бути розроблені з урахуванням інтересів користувачів та створити додаткову цінність для них;

У результаті аналізу предметної галузі було виділено 2 типи акторів покупець та продавець як учасники ланцюга постачання.

Функціональні можливості цих акторів у рамках сервісу формалізовані у вигляді відповідних use-case діаграм.

Нижче наведені можливості покупця у вигляді use case діаграми (Рис. 1):

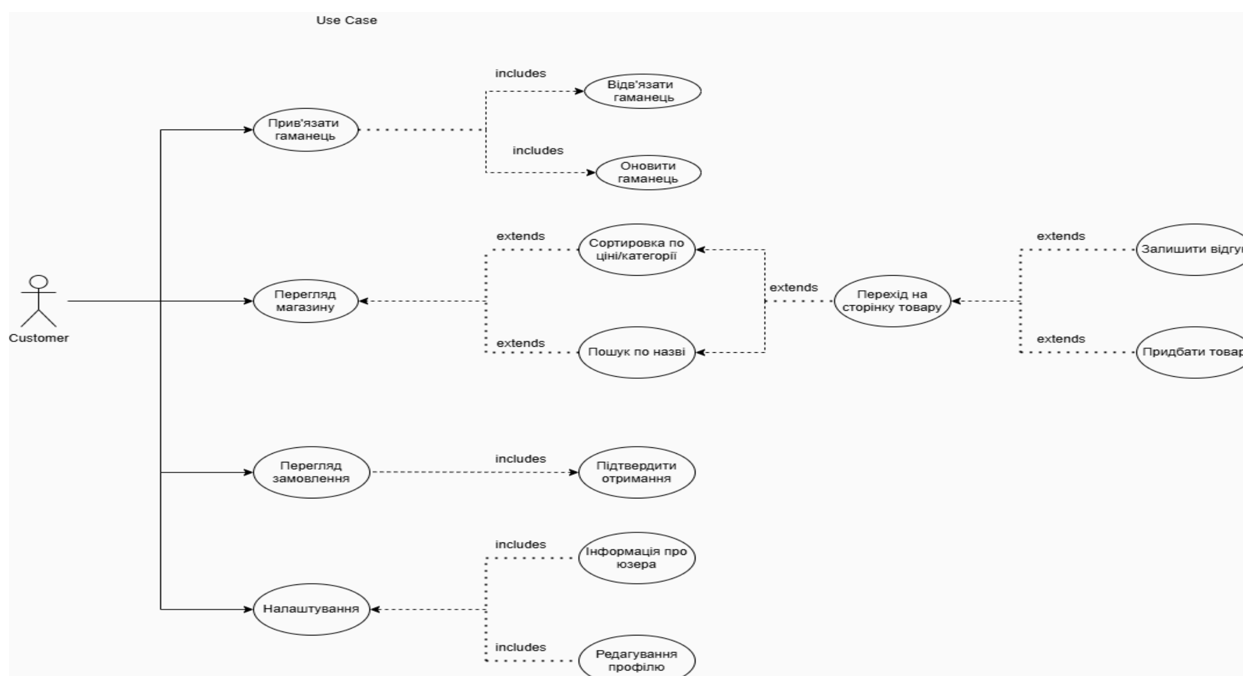

Рис. 1. Use-case buyer

Реєстрація та авторизація користувачів: Система надає можливість користувачам реєструватися за допомогою свого облікового запису. Також

вона забезпечує механізм авторизації, що дозволяє користувачам увійти в систему та мати доступ до своїх особистих даних. Чутливі дані такі як пароль користувача захищенні за допомогою хешування. Також після авторизації чи реєстрація створиться захищена сесія, яка має обмеження по часу, для захисту даних користувача та протидії підробки запитів.

Підключення гаманця: Система надає можливість зацікавленому користувачу підключити власний гаманець. Також підключення гаманця надає у подальшому можливість його відключити чи оновити за бажанням користувача.

Магазин: Система надає можливість користувачу переглянути магазин, провести сортування контенту магазина або скористатися пошуком по назві вже існуючих предметів. Користувач має можливість відкрити сторінку товару та замовити його або залишити відгук.

Угоди: Система дозволяє переглянути вже існуючі, актуальні або минулі угоди та підтвердити отримання товару у вже існуючих, актуальних угодах.

Налаштування: Система дає можливість переглянути користувачу надану ним інформацію щодо його профілю, та редагувати профіль у разі потреби.

Вихід з акаунту: Система дає можливість зробити вихід з профілю за потреби.

Актор з роллю продавець має ті самі можливості які надані покупцю, а також і деякі додаткові. Повні можливості продавця наведені у відповідній use-case діаграмі (рис. 2):

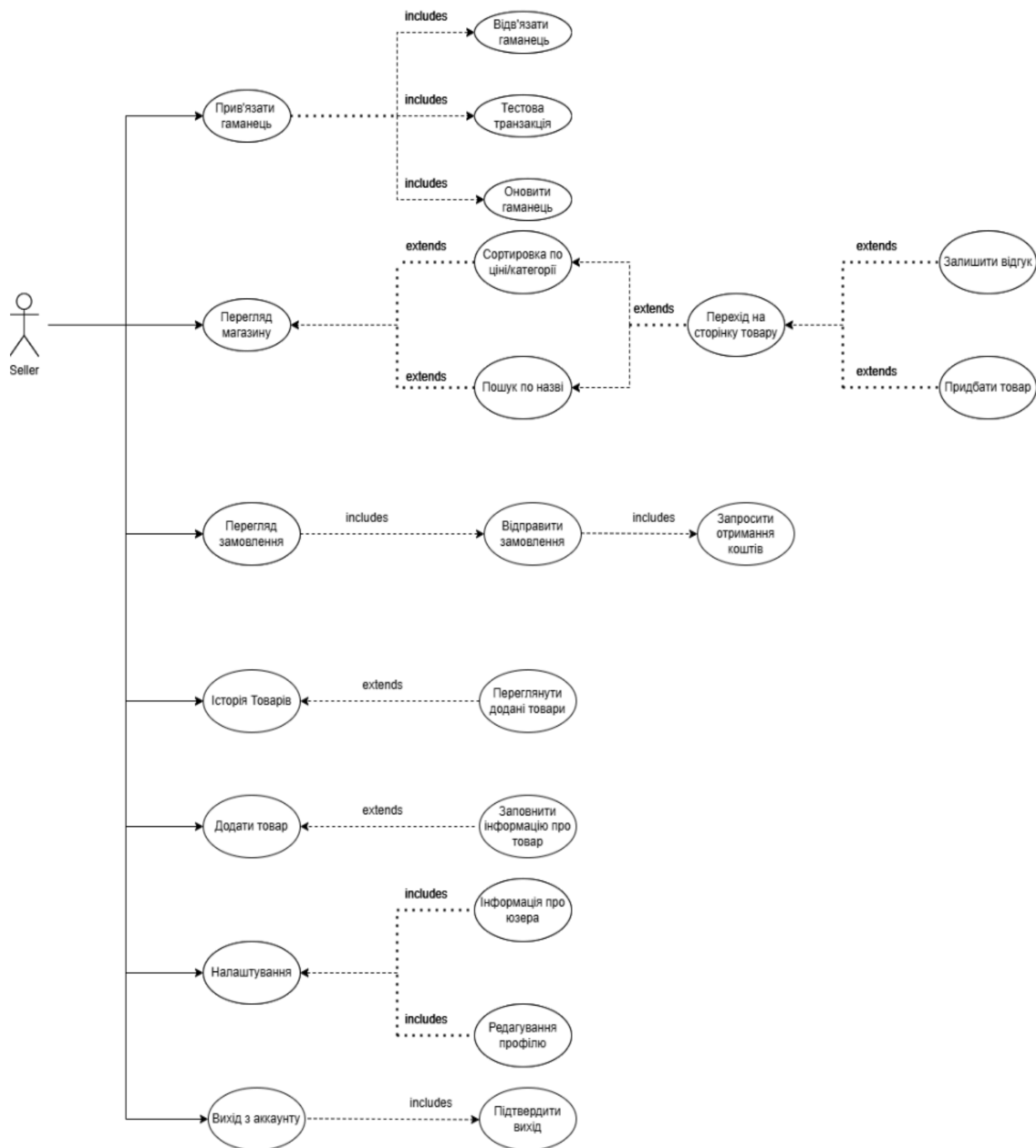

Рис. 2. Use-case seller

Угоди: Система дозволяє переглянути вже існуючі, актуальні або минулі угоди та підтвердити відправку товару у вже існуючих, актуальних угодах. Також я разі підтвердження отримання товару покупцем, продавець має

змогу запросити отримання коштів. Система програмно обробить запит та у разі виконання умов кошти надійдуть на гаманець продавця автоматично.

Історія товарів: Система надає можливість переглянути історію товарів, там буде відображена необхідна інформація, зокрема відстеження актуальної кількості товарів у наявності для інформування продавця та інше.

Додати товар: Система дає можливість продавцю додати новий товар та надати необхідну інформацію щодо товару для його подальшої демонстрації користувачам.

2.2 Огляд використаних технологій

JavaScript та Node.js - це потужні інструменти, які надають можливості розробки програмного забезпечення. Одна з найбільш значних переваг JavaScript - його універсальність: він використовується як для фронтенду, так для бекенду, завдяки Node.js, що дозволяє розробникам працювати в єдиному середовищі та зменшувати накладні витрати. Node.js, побудований на рушії V8, забезпечує високу продуктивність завдяки асинхронній, архітектурі, що ідеально підходить для масштабованих додатків з великою кількістю одночасних з'єднань. Технологія дозволяє використовувати JavaScript на сервері, що спрощує та пришвидшує розробку full-stack додатків, а велика екосистема npm надає доступ до тисяч бібліотек та фреймворків, прискорюючи процес розробки. Разом JavaScript і Node.js утворюють потужний дует, який забезпечує швидкість, гнучкість та ефективність у сучасній веб-розробці [1].

Vite + React + TypeScript - це стек для швидкої та ефективної розробки веб-сервісів, який поєднує сучасні технології для максимальної продуктивності. Одна з головних переваг Vite - неймовірно швидкий запуск сервера та гаряче оновлення модулів завдяки використанню ES-модулей, що значно прискорює процес розробки. React, як популярна бібліотека для створення інтерфейсів, забезпечує компонентний підхід, що спрощує розробку складних UI та підтримує реалізацію динамічних додатків. TypeScript додає статичну типізацію, що підвищує надійність коду, допомагає уникнути помилок на ранніх етапах та покращує стабільність роботи додатків. Vite прискорює збірку та розгортання, React надає гнучкість у створенні інтерфейсів, а TypeScript забезпечує безпеку типів. Ці технології ідеально підходить для створення високопродуктивних додатків [2], [3].

Express.js - це потужний веб-фреймворк для Node.js, який дозволяє швидко та якісно створювати серверні додатки. Його головна перевага - простота та гнучкість: завдяки модульній архітектурі та мінімалістичному підходу розробники можуть легко налаштовувати маршрути, middleware та обробку запитів, не перевантажуючи код. Express підтримує RESTful API, WebSockets та різноманітні шаблони, що робить його універсальним інструментом для створення як простих серверів, так і складних бекенд-систем. Завдяки великій екосистемі додаткових бібліотек, наприклад, для аутентифікації, валідації чи роботи з базами даних, можна легко розширювати функціонал. Фреймворк має відмінну продуктивність, оскільки працює на базі Node.js і використовує його асинхронні можливості, що дозволяє обробляти тисячі запитів без серйозного навантаження. Крім того, Express має велику документацію, що спрощує навчання та вирішення проблем. Разом із Node.js він утворює гарне рішення для розробки швидких, легких та масштабованих серверних додатків, що робить його одним із найпопулярніших фреймворків у бекенд-розробці.

PostgreSQL - це потужна, реляційна система управління базами даних (СУБД) з відкритим кодом, яка поєднує надійність, гнучкість та високу продуктивність. Однією з ключових переваг Postgres є підтримка стандарту SQL, що дозволяє ефективно працювати із даними, виконувати складні запити, агрегації та аналітику. Він підтримує ACID-транзакції, що гарантує цілісність даних навіть у разі збоїв, а також має розвинені механізми індексування та оптимізації запитів. На відміну від багатьох інших СУБД, Postgres пропонує розширені можливості, такі як робота з JSON та повнотекстовий пошук. SQL як мова запитів дозволяє легко будувати логіку роботи з даними. Використання PostgreSQL як бази даних для системи також має свої переваги і обґрунтування. PostgreSQL є однією

з найбільш надійних та стабільних реляційних баз даних. Вона відома своєю високою стійкістю до відмов та забезпечує цілісність даних [4].

Низка сучасних технологій для серверної частини, що включає jose, dotenv, @orbs-network/ton-access, cors, cookie-parser та Sequelize, надає розробникам потужний інструментарій для створення безпечних та ефективних бекенд-рішень.

Jose - це бібліотека для роботи з JWT, яка дозволяє легко генерувати, підписувати та верифікувати токени, забезпечуючи безпеку та зручність автентифікації та авторизації. Вона підтримує сучасні алгоритми шифрування та спрощує роботу з токенами без зайвих накладних витрат.

Dotenv - простий, але надзвичайно корисний модуль для роботи з змінними оточення. Він дозволяє зберігати конфіденційні дані (API-ключі, паролі, посилання тощо) окремо від коду, що підвищує безпеку та спрощує налаштування системи для різних середовищ.

@orbs-network/ton-access - бібліотека для взаємодії з блокчейном, яка надає зручний інтерфейс для роботи зі смарт-контрактами, транзакціями та іншими функціями мережі. Це дозволяє інтегрувати криптовалютні можливості в додатки без необхідності глибокого занурення в низькорівневі деталі блокчейну.

CORS - middleware для Express.js, який керує політикою Cross-Origin Resource Sharing, дозволяючи безпечні міждоменні запити. Він налаштовує заголовки, що захищає додаток від небажаних запитів із сторонніх джерел, але при цьому дає контроль над тим, які саме домени можуть взаємодіяти з API.

Cookie-parser це ще один middleware для Express, який спрощує роботу з HTTP-кукі. Він автоматично розпарсює кукі з запитів, дозволяючи зручно зчитувати та встановлювати їх, що особливо корисно для сесійної автентифікації або зберігання тимчасових даних.

Sequelize - Object-Relational Mapping для Node.js, який значно спрощує роботу з реляційними базами даних. Завдяки ньому можливо працювати з даними як зі звичайними JavaScript-об'єктами, автоматизує створення міграцій, дозволяє будувати складні запити та зв'язки між таблицями, економлячи час на написанні SQL-коду.

Разом ці технології дають можливість розробити безпечні API, крипто інтеграцій, зручну роботу з куками та сессіями та ефективну взаємодію з базою даних.

Сучасні фронтенд технології, що включають Tailwind CSS, React, Redux з reduxjs/toolkit, RTK Query, tonconnect/ui-react та Eruda, надає розробникам потужний інструментарій для створення сучасних, продуктивних та складних веб-сервісів.

Tailwind CSS - це утилітарний CSS-фреймворк, який дозволяє швидко будувати інтерфейси без необхідності перемикатися між файлами стилів. Завдяки готовим класам розробка стає набагато швидшою, а дизайн - привабливим, при цьому зберігається повний контроль над стилями без зайвих дій [6].

React - провідна бібліотека для створення інтерактивних UI, яка забезпечує компонентний підхід та ефективне оновлення інтерфейсу. Разом із Redux та reduxjs/toolkit вона надає надійне управління станом клієнта. Redux Toolkit спрощує роботу зі станами, автоматизуючи створення редюсерів, екшенів та налаштування стора, а RTK Query додає

можливості кешування API-запитів, що значно скорочує код та підвищує продуктивність.

Tonconnect/ui-react - це бібліотека для інтеграції гаманців у React-додатки. Вона дозволяє легко підключити криптовалютні транзакції, автентифікацію через блокчейн та взаємодію зі смарт-контрактами без необхідності глибокого знання специфіки Web3 [7].

Eruda - зручна консоль браузерів, яка дозволяє відлагоджувати додатки прямо на місці. Вона надає інструменти для аналізу логів, мережевих запитів, помилок JavaScript тощо, що робить розробку та тестування на пристроях набагато комфортнішою.

2.3 Моделювання бази даних

Цей пункт описує структуру бази даних для децентралізованого сервісу, який використовує блокчейн-транзакції у ланцюгах постачання. Для зберігання інформації було прийняте рішення використовувати реляційну базу даних.

Першим етапом проектування бази даних є виділення сутностей предметної галузі. Проаналізувавши як безпосередньо предметну галузь, так і функціонал системи, спираючись на розроблені use-case діаграми, можна виділити наступні сутності предметної галузі:

1. User;
2. Product;
3. Review;
4. Order;
5. Payment;

Кожна таблиця містить поля, які зберігають інформацію про сутності системи, а також зв’язки між ними для забезпечення цілісності даних.

На другому етапі проектування бази даних були визначені зв'язки між сутностями.

1 Таблиця User:

Ця таблиця зберігає дані про всіх користувачів системи, включаючи покупців, продавців, дистриб’юторів та виробників;

id – унікальний ідентифікатор користувача (первинний ключ, автоінкремент);

name – ім’я користувача;

surname – прізвище користувача;

email – електронна пошта, потрібна, як унікальне поле для входу в систему;

password – захешований пароль для автентифікації;
walletAddress – адреса криптогаманця користувача яка є унікальною та використовується для транзакцій;
role – роль користувача (CUSTOMER, SELLER, DISTRIBUTOR, MANUFACTURER);
country – країна користувача (з обмеженим списком);
createdAt – дата створення запису;
updatedAt – дата останнього оновлення запису;

Зв’язки: моделювання бази даних для зв'язку користувача (юзера) з товарами було здійснено за допомогою використання реляційної моделі даних. У такій моделі будуть існувати таблиці: "User" (Користувач) та "Product" (Товар). Доречно буде використати тип зв'язку один до багатьох (OneToMany) Зв'язок між ними був встановлений за допомогою стовпця "id" в таблиці "User" який є первинним ключем, та "userId" в таблиці "Product" який є зовнішнім ключем та буде посилатися на User.id. Цей підхід дозволяє кожному користувачу мати багато різних товарів і навпаки, кожен товар пов'язаний з одним відповідним йому користувачем. Таким чином, кожен запис в таблиці " Product " буде мати посилання на відповідний запис в таблиці " User ". Таке моделювання бази даних дозволить вам легко здійснювати операції з отримання та збереження даних про користувачів та товари. Наприклад, можливо зручно отримати інформацію про товар, пов'язаний з конкретним користувачем, або знайти користувача, який додавав певні товари.

2. Таблиця Product:

Ця таблиця містить інформацію про товари, які додані певними користувачами, включаючи їх характеристики та поточне місцезнаходження;

id – унікальний ідентифікатор товару (первинний ключ, автоінкремент);
name – назва товару;
description – детальний опис товару;
price – ціна товару;
quantity – кількість товару у наявності;
category – категорія товару (CLOTHS, DEVICES, OFFICE EQUIPMENT, DECOR, OTHER);
originCountry – країна походження товару;
currentCountry – поточна країна знаходження товару (для відстеження ланцюжка постачання);
photoUrl – посилання на фото товару для того щоб зручніше його відобразити на клієнтській частині сервісу;
userId – ідентифікатор користувача, який додав товар (продавець/виробник та інше);
createdAt, updatedAt – дати створення та оновлення запису;

Зв’язки: далі розглянемо зв'язки товарів та інших сутностей. Було створено таблицю "Product" для зберігання даних про товари. Було додано первинний ключ "id" до таблиці "Product", який буде використовуватися для посилатися на відповідну строку в таблиці "Product". Він потрібен для створення зв’язку між товарами та іншими сутностями. "Product" буде мати зв’язки один до багатьох (OneToMany) з сутностями "Review" та "Order" через зовнішні ключи цих таблиць, які будуть посилатися на "id" у таблиці "Product" відповідно: Review.productId => Product.id та Order.productId => Product.id.

3. Таблиця Review:

Ця таблиця зберігає оцінки та коментарі покупців про товари;

id – унікальний ідентифікатор відгуку (первинний ключ, автоінкремент);

rating – оцінка товару (від 1 до 5);

text – текстовий відгук;

userId – ідентифікатор користувача, який залишив відгук;

productId – ідентифікатор товару, до якого належить відгук;

createdAt, updatedAt – дати створення та оновлення;

Зв’язки: тепер опишемо зв’язок відгуку і користувача, було реалізоване з’єднання зв’язком багато до одного (ManyToOne). Для моделювання зв'язку "багато до одного" між відгуком та юзером, необхідно створити зовнішній ключ у таблиці відгук який буде посилатися на первинний ключ відповідного користувача: Review.userId => User.id. Така модель дозволяє зв'язувати кожного користувача з багатьма відгуками, але кожен відгук може бути пов'язаний з тільки з одним користувачем. Ця модель дозволяє ефективно виконувати операції, такі як отримання всіх відгуків, пов'язаних з певним користувачем або всіх користувачів, які залишили відгук з певною оцінкою товару.

4. Таблиця Order:

Ця таблиця відстежує замовлення, включаючи інформацію про доставку та блокчейн-хеші для підтвердження транзакцій та відстеження статусу замовлення;

id – унікальний ідентифікатор замовлення (первинний ключ, автоінкремент);

quantity – кількість товару в замовленні;

deliveryAddress – адреса доставки;

paymentHash – хеш транзакції оплати;

sendHash – хеш підтвердження відправки;

shipmentHash – хеш підтвердження доставки;

receptionHash – хеш підтвердження отримання;

transferHash – хеш переказу коштів продавцю;

isCompleted – чи завершено замовлення;

userId – ідентифікатор покупця;

productId – ідентифікатор товару;

sellerId – ідентифікатор продавця;

createdAt, updatedAt – дати створення та оновлення;

Зв'язки: тепер опишемо зв’язки для таблиці "Order", очевидно, що один користувач може мати багато замовлень, в яких користувач може бути у якості продавця чи замовника, тож зв’язок буде один до багатьох. Для моделювання зв'язку "один до багатьох" між юзером і замовленням, було додано стовпець "userId" та "sellerId" до таблиці "Order", який буде посилатися на первинний ключ "id" в таблиці "User". У цьому моделюванні, таблиця "User" містить дані про користувачів, а таблиця "Order" містить дані про замовлення. У стовпці "userId" таблиці "Order" зберігається ідентифікатор користувача, який є замовником, а у стовпці "sellerId" зберігається посилання на користувача який є продавцем. Ця модель дозволяє кожному користувачу мати багато замовлень, але кожне замовлення належить лише одному користувачеві. Ця модель дозволяє легко

виконувати операції, такі як отримання всіх замовлень, пов'язаних з певним користувачем та інше.

5. Таблиця Payment:

Ця таблиця фіксує всі транзакції, пов’язані з замовленнями, включаючи оплату, відправку, отримання товару або транзакції де продавець робить запит на отримання коштів та транзакцію отримання коштів продавцем;

id – унікальний ідентифікатор платежу (первинний ключ, автоінкремент);

senderAddress – адреса гаманця відправника;

amount – сума платежу;

sellerWalletAddress – адреса гаманця продавця;

hash – хеш транзакції в блокчейні;

type – тип транзакції (payment, sending, shipment, reception, transfer);

isConfirmed – чи підтверджено транзакцію;

userId – ідентифікатор користувача, який ініціював платіж;

orderId – ідентифікатор замовлення;

createdAt, updatedAt – дати створення та оновлення;

Зв'язки: тепер розглянемо транзакцію та її зв1язки з іншими сутностями. Для цього використаємо зв’язок один до багатьох. Для моделювання зв'язку "один до багатьох" між транзакцією і користувачем було створено зовнішній ключ, який використовується для зв'язку цих сутностей. У цьому моделюванні, таблиця "Payment" містить дані про користувачів, через посилання на первинний ключ

відповідних користувачів: Payment.userId => User.id. Також кожен зовнішній ключ "orderId" посилається на відповідний первинний ключ у таблиці "Order": Payment.OrderId => Order.id. Ця модель дозволяє кожному користувачу та кожному замовленню мати багато транзакцій, а також кожна транзакція може бути пов'язаний з певним користувачем та відповідним замовленням. Ця модель дозволяє легко виконувати операції, такі як отримання списку транзакцій, пов'язаних з певним користувачем або певним замовленням та отримати конкретне замовлення, що відповідає певній транзакції.

Загалом дана модель даних забезпечить ефективне управління користувачами, зручний каталог товарів, систему відгуків для оцінки якості товарів, відстеження замовлень з використанням блокчейн хешів для прозорості транзакцій та Історію платежів з підтвердженням кожного етапу угоди.

Така структура дозволяє створити масштабований сервіс з підтримкою криптоплатежів та відстеження, а також дозволяє ефективно керувати даними, робити складні SQL-запити та забезпечує прозорість транзакцій у системі.

Нижче наведено ER-діаграму з встановленими зв’язками та потрібними стовпцями (Рис. 3):

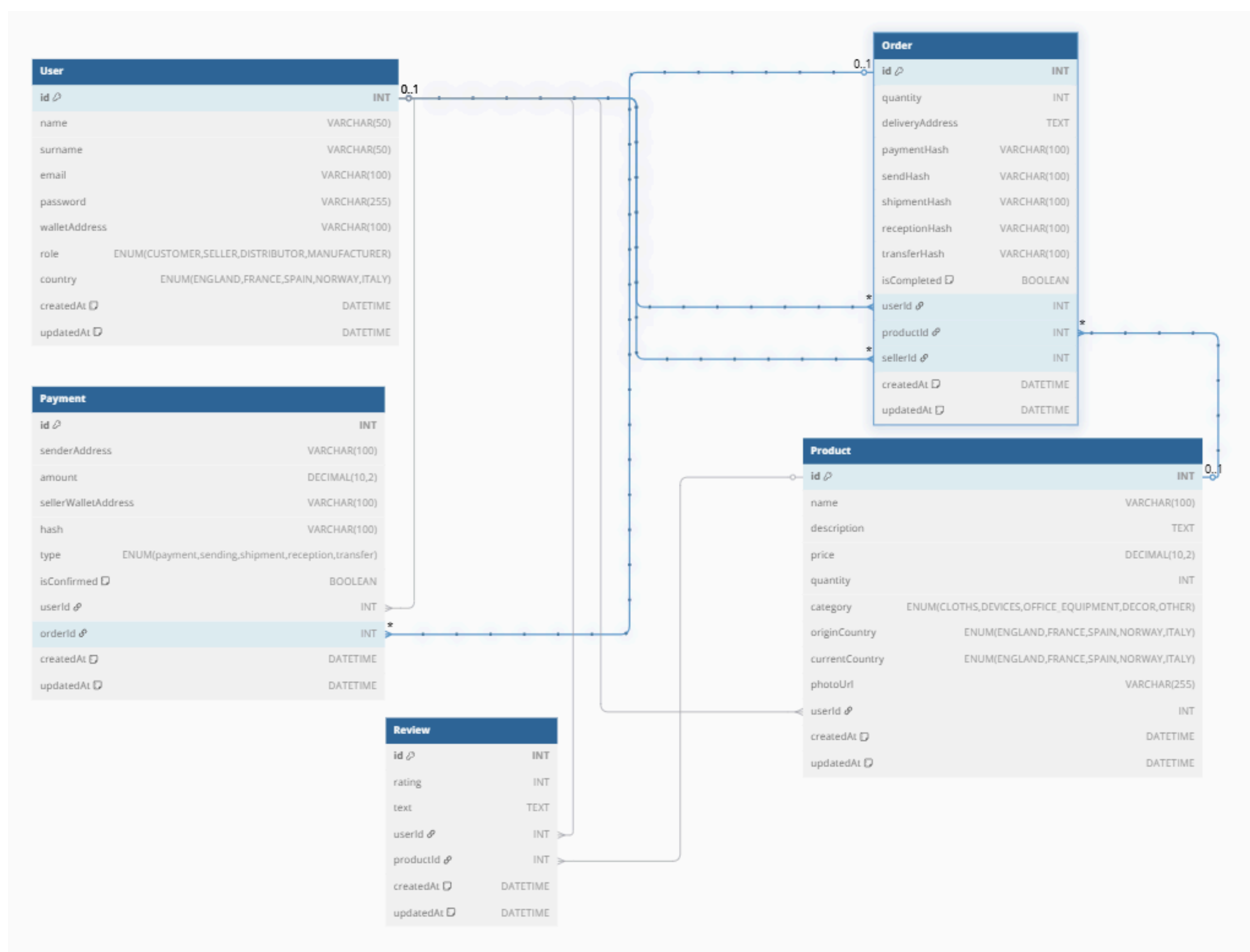

Рис. 3. ER-діаграма

2.4 Концептуальна модель предметної галузі

Розглянемо створений клас користувача. Він носить User та містить наступні атрибути:

- Особисті дані (ім'я, прізвище, email, пароль);
- Гаманець (walletAddress);
- Роль (CUSTOMER, SELLER, DISTRIBUTOR, MANUFACTURER);
- Країна (ENGLAND, FRANCE, SPAIN, NORWAY, ITALY);
- Дати створення/оновлення;

Він має наступні відносини: один-до-багатьох (1:N) з Product, тобто користувач може мати багато товарів (як продавець). Один-до-багатьох з Review, користувач залишає відгуки до товарів, та один користувач може надіслати відгуки до багатьох товарів. Має двонаправлений зв’язок один-до-багатьох з Order, як покупець або як продавець. Замовлення належить покупцю та продавцю. Також користувач має зв’язок один-до-багатьох з Payment, тобто одному користувачу належить багато транзакцій.

Наведено клас User з діаграми класів (Рис 4.)

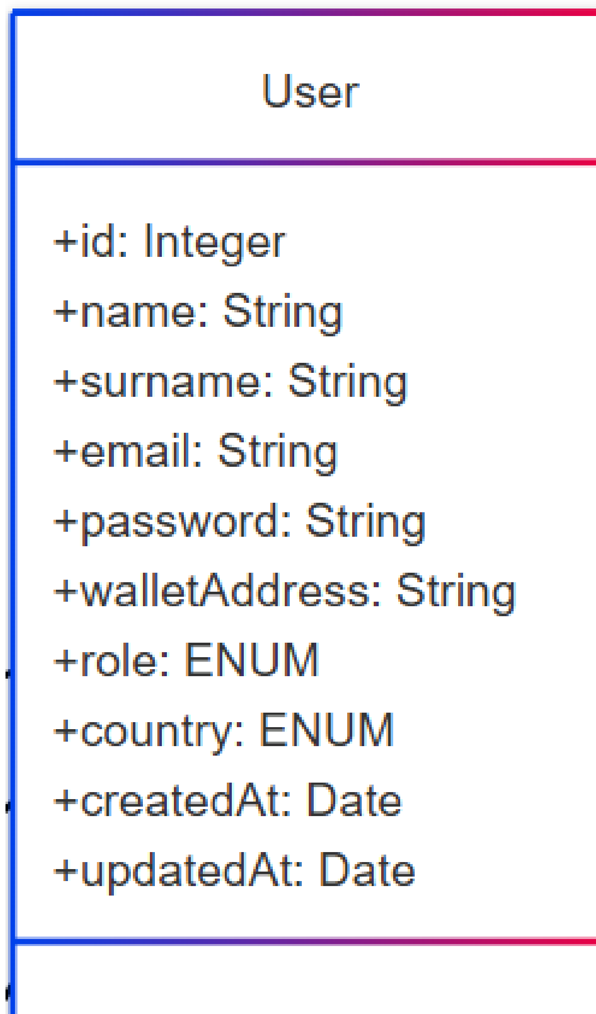

Рис. 4. Class-Diagram User

Для зразка було наведемо з діаграми класів зв'язок користувача та товару із повністю заповненими атрибутами (Рис. 5):

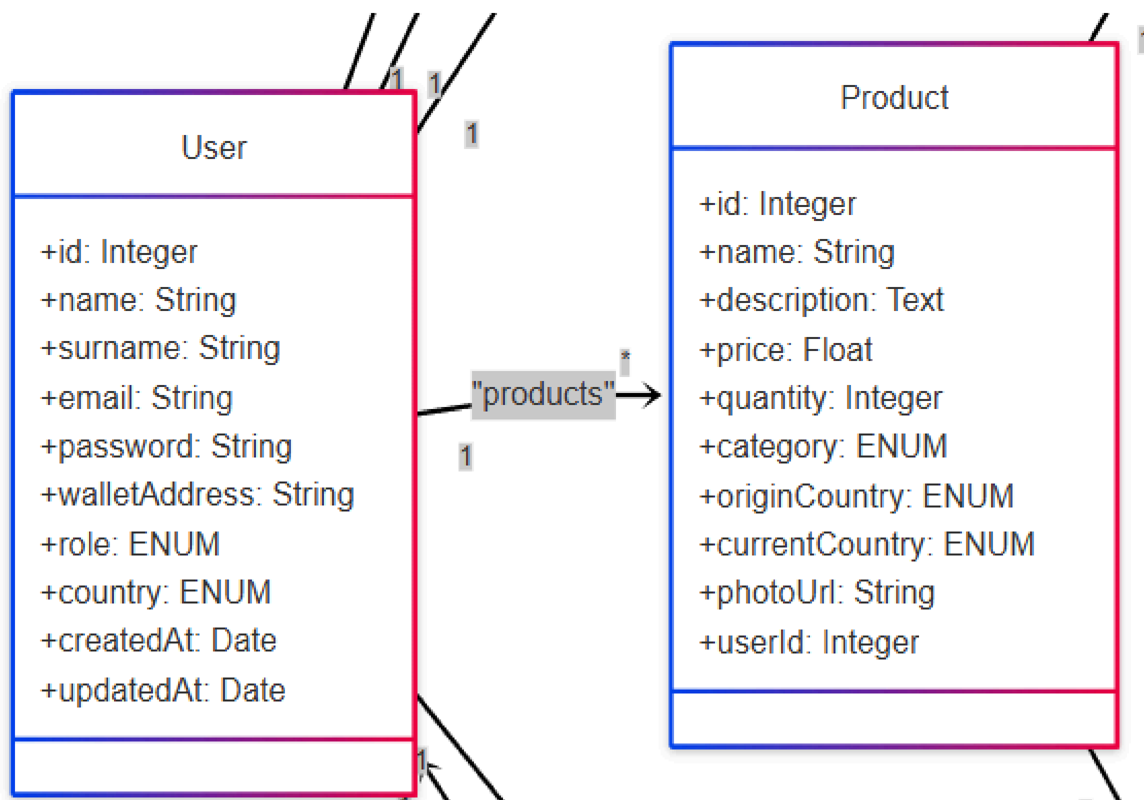

Рис. 5. User-Product class diagram

Можна побачити реалізоване відношення "один до багатьох" користувача та товару.

Клас Product – товар, наведений на попередній діаграмі (Рис. 5) та він має асоціацію "багато до одного" з класом User, та "один до багатьох" з класами Review та Order. Клас Product має наступні атрибути:

- Характеристики (назва, опис, ціна, кількість);
- Категорія (CLOTHS, DEVICES, OFFICE EQUIPMENT, DECOR, OTHER);
- Країна походження та поточна країна;
- Посилання на фото;
- Власник;

Клас Order (Рис. 6) – угода, містить двонаправлену асоціацію "багато до одного" з класом User, асоціацію "один до багатьох" з класом Payment, та асоціацію "багато до одного" з класом Product.

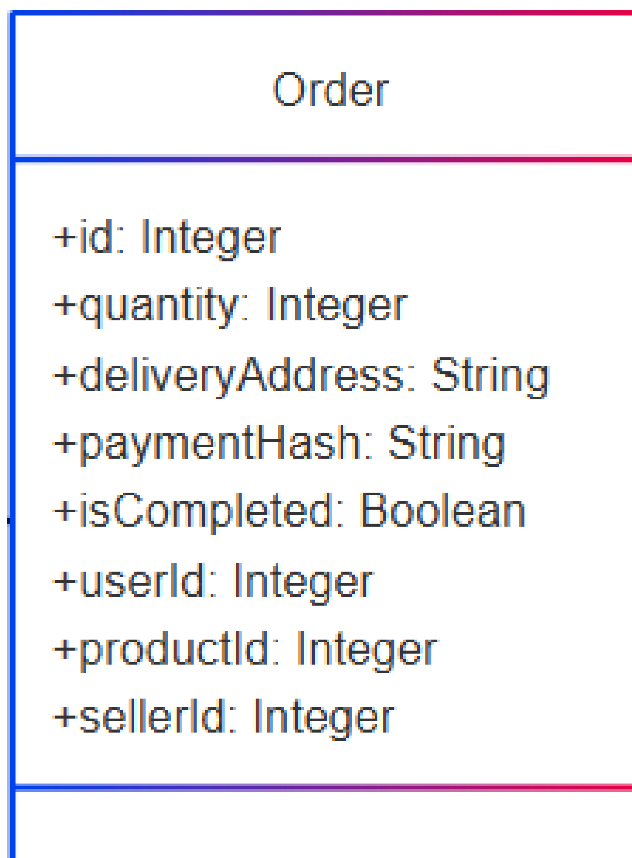

Рис. 6. Order class diagram

Клас Order має наступні атрибути:

- Інформацію щодо замовлення (кількість, адреса доставки);
- Хеш-коди для різних етапів (оплата, відправка, отримання, запит на отримання коштів, отримання коштів);
- Статус завершення;
- Посилання: Покупець, продавець, товар;

Нижче наведено приклад асоціацій класу Order з класами Payment та Product з діаграми класів (Рис. 7):

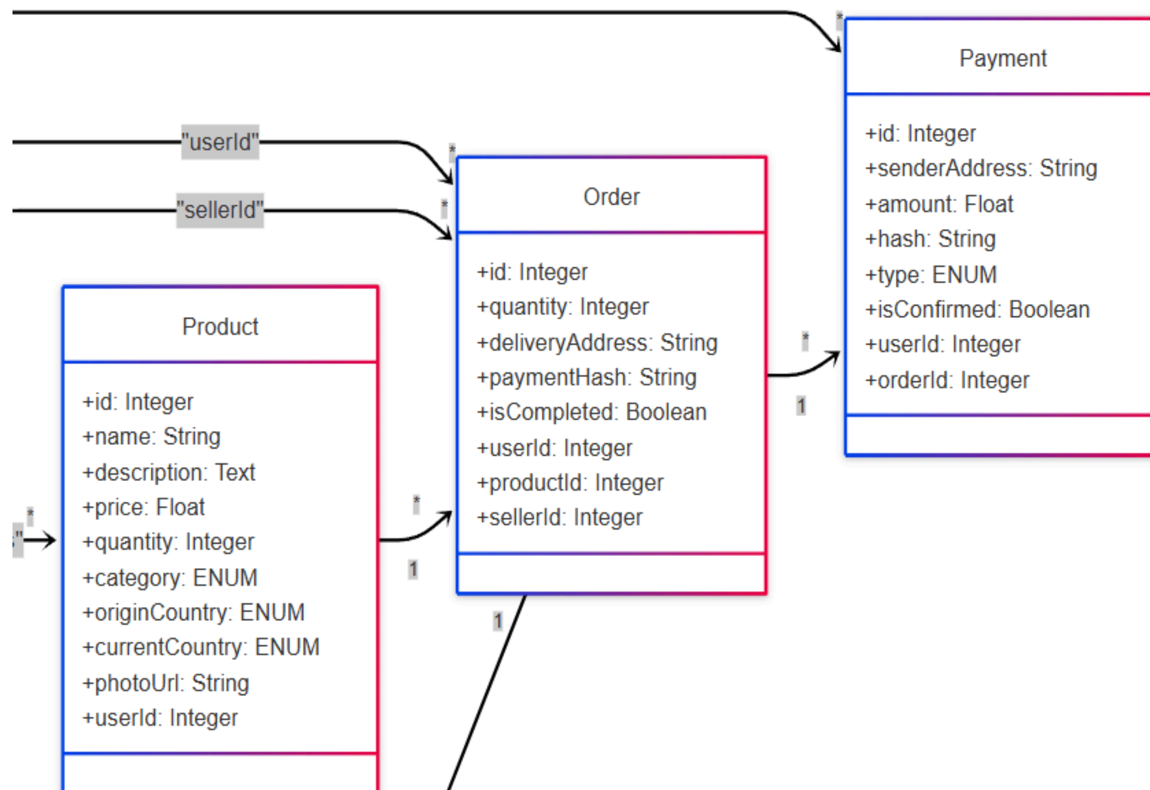

Рис. 7. Product- Order- Payment class diagram

Клас Review – тест, містить наступні атрибути (Рис. 8):

- Рейтинг (1-5), текст відгуку;
- Посилання: автор та товар;

Клас Review має асоціації типу "багато до одного" з класом User та з класом Product. Відповідні асоціації продемонстровані з діаграми класів (Рис. 8):

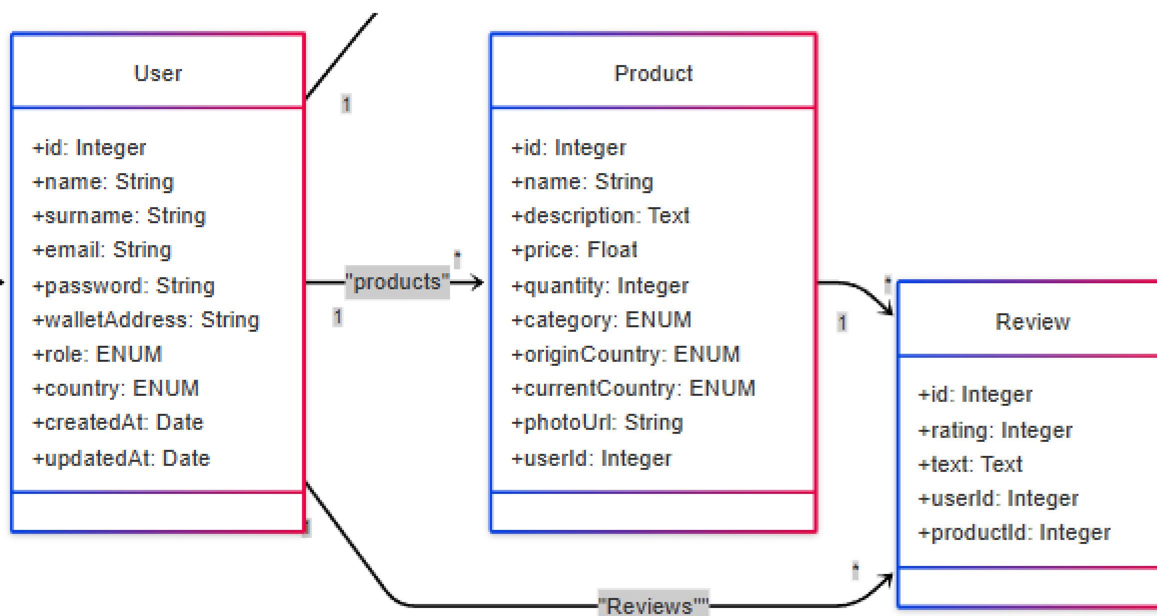

Рис 8. Review-Product-User class diagram

Клас Payment (Рис. 9) – транзакції, містить наступні атрибути та асоціацію "багато до одного" з класом Order, та асоціацію "багато до одного" з User.

Атрибути:

- Адреси гаманців;
- Сума, тип операції (payment, sending, shipment, receiving, transfer);
- Підтвердження (isConfirmed);
- Хеш транзакції;

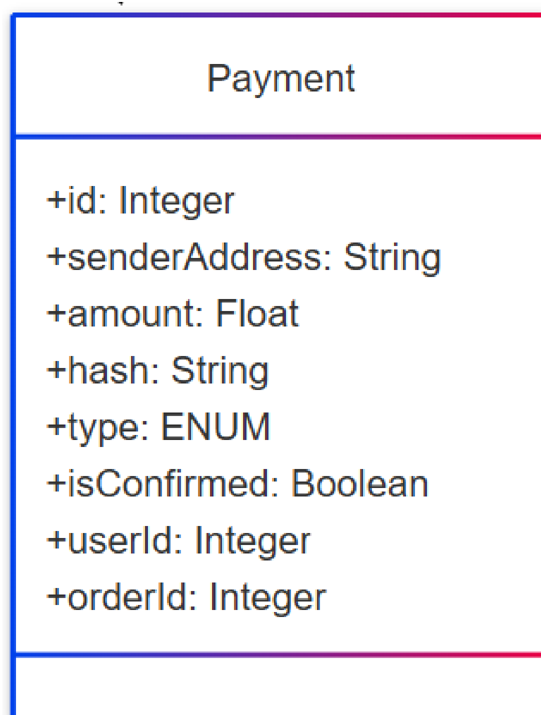

Рис. 9. Payment class diagram

Нижче наведено приклад асоціацій класу Payment з класами Order та User з діаграми класів (Рис. 10):

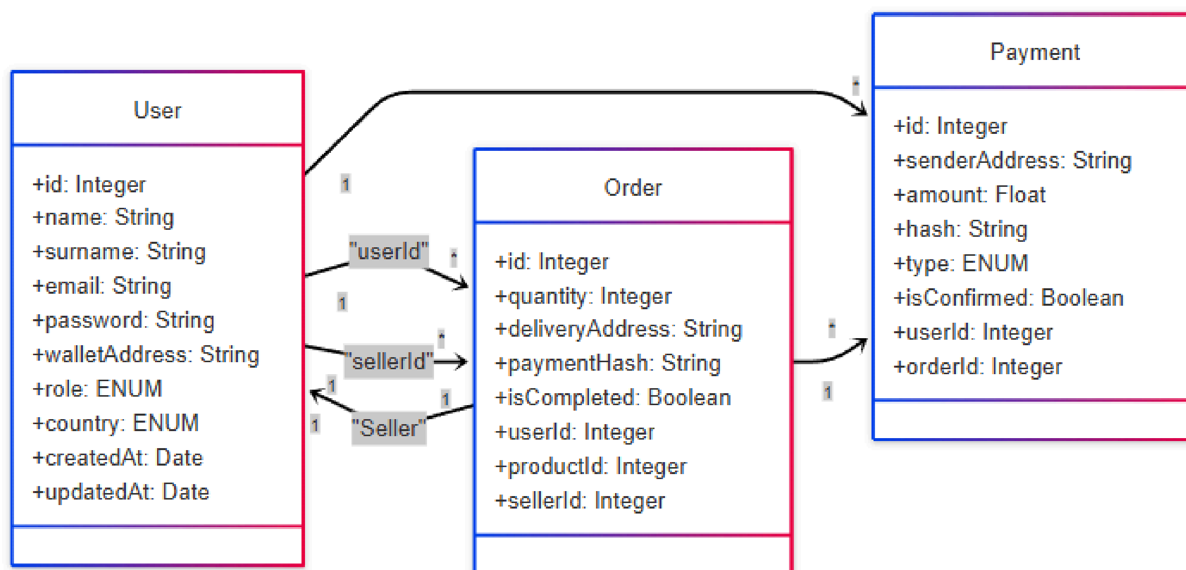

Рис. 10. User-Order-Payment class diagram

Бізнес-логіка:

1. Ролі користувачів:

- CUSTOMER: робить замовлення, залишає відгуки;
- SELLER DISTRIBUTOR MANUFACTURER: робить замовлення, залишає відгуки, додає товари, отримує платежі;

2. Життєвий цикл замовлення:

- Створення та оплата => Відправка => Отримання => запит на отримання переказу => переказ за отриманий товар => замовлення виконане;

3. Перевірки даних:

- Валідація даних;
- Обмеження рейтингу від 1 до 5;
- Негативні значення ціни та кількості заблоковані;

4. Верифікація транзакцій через блокчейн, отримання та перевірка інформації;

Концепт моделі реалізує повноцінний сервіс з: користувачами різних ролей, товарами з категоріями та характеристиками, відгуками для оцінки якості, замовленнями з підтвердженням етапів через блокчейн за допомогою хешів, платежів з підтримкою різних типів операцій.

3. РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ

3.1 ОСНОВНІ АРХІТЕКТУРНІ ПАТЕРНИ ТА ШАБЛОНИ

1. Патерн MVC (Model – View – Controller):

Паттерн MVC (Model-View-Controller) є одним з найпоширеніших архітектурних паттернів для розробки програмного забезпечення. Він використовується для розділення логіки програми на три різні компоненти: Модель (Model), Представлення (View) та Контролер (Controller). Кожен з цих компонентів виконує свою відповідальність і взаємодіє з іншими компонентами для забезпечення роботи програми.

(Model-View-Controller) — це архітектурний патерн, який розділяє додаток на три основні частини:

- Модель (Model): Модель представляє дані та стан програми. Вона відповідає за обробку даних та логіки, без прив'язки до способу відображення або взаємодії з користувачем. Модель може містити методи для отримання та зміни даних, а також виконання розрахунків або операцій над даними. У розробленій серверній частині модель знаходиться у папці models та описує структуру даних та взаємодію з БД, через Sequelize;
- Представлення (View): Представлення відповідає за візуалізацію даних, відображення графічного інтерфейсу користувача та взаємодію з користувачем. Воно отримує дані з моделі і відображає їх у зрозумілій формі. Представлення може також реагувати на події користувача і сповіщати контролер про ці події;
- Контролер (Controller): Контролер служить посередником між моделлю і представленням. Він обробляє події користувача, взаємодіє з представленням та моделлю для забезпечення обробки запитів та оновлення стану програми. Контролер також відповідає за перетворення даних, отриманих від користувача або представлення,

перед тим, як вони будуть передані до моделі для обробки. У розробленому сервісу контролери, вбудовані у роути, що знаходяться у папці routes. У класичному MVC контролери мають бути окремо, але у Express.js їх часто об'єднують із роутами для зручності;

Загалом проект частково використовує MVC, але більше у стилі Express-сервісу з розділенням на моделі та роути. Також у зв'язку з тим що мова йде саме про серверну частину View буде представлений на клієнтській частині сервісу.

2. Патерн Singleton:

Патерн Singleton — це шаблон проектування, який гарантує, що клас має тільки один екземпляр, і надає глобальну точку доступу до нього. У сервісі з використанням Node.js-сервісу цей патерн використовується для керування підключенням до бази даних та моделями Sequelize, що є класичним прикладом його застосування.

У файлі models/index.js створюється єдиний екземпляр Sequelize, який використовується усіма моделями (Рис. 11):

```
let sequelize;
if (config.use_env_variable) {
  sequelize = new Sequelize(process.env[config.use_env_variable], config);
} else {
  sequelize = new Sequelize(config.database, config.username, config.password, {
    host: config.host || 'localhost',
    dialect: 'postgres',
  });
}
```

Рис. 11. Index.js sequelize initialization

Як можна побачити, об'єкт sequelize ініціалізується лише один раз при запуску додатка. Усі подальші звернення до БД використовують це

саме з'єднання, також експорт db.sequelize дозволяє отримувати цей екземпляр з будь-якого місця програми.

Ще один приклад застосування цього патерну у проекті: Кожна модель (наприклад, User) експортується як функція, яка повертає клас. Node.js кешує результат require(), тому при повторному імпорті моделі повертається той самий екземпляр. Наприклад, при виконанні const { User } = require('../models'); у різних файлах, отримується доступ до однієї й тієї ж моделі. Тож можна зазначити, що При старті додатка: ініціалізується єдине з'єднання з БД (sequelize), а моделі створюються один раз і реєструються в db (об'єкт з models/index.js) (рис. 12).

```
fs
  .readdirSync(__dirname)
  .filter(file => {
    return (
      file.indexOf('.') !== 0 &&
      file !== basename &&
      file.slice(-3) === '.js' &&
      file.indexOf('.test.js') === -1
    );
  })
  .forEach(file => {
    const model = require(path.join(__dirname, file))(sequelize, Sequelize.DataTypes);
    db[model.name] = model;
  });

Object.keys(db).forEach(modelName => {
  if (db[modelName].associate) {
    db[modelName].associate(db);
  }
});

db.sequelize = sequelize;
db.Sequelize = Sequelize;

module.exports = db;
```

Рис. 12. Index.js models init

3. Патерн Facade:

Facade — це структурний патерн проектування, який надає простий інтерфейс для роботи зі складною підсистемою, приховуючи її внутрішню

логіку. Він діє як "обгортка", що об'єднує взаємопов'язані класи або функції в єдиний зручний для використання інтерфейс [8].

У серверному сервісі патерн Facade реалізований через модулі в папці utils, які приховують складну логіку за простими інтерфейсами.

Спочатку, розглянемо фасад для роботи з блокчейном(рис. 13), цей фасад реалізує:

- Генерація ключів гаманця з мнемонічної фрази;
- Взаємодія з API (отримання ендпоінту, відправка транзакцій);
- Відстеження статусу транзакції через seqno;
- Обробка помилок та повторні спроби;

```
const endpoint = await getHttpEndpoint({
    network: process.env.TON_NETWORK || "testnet"
});
const client = new TonClient({ endpoint });

if (!await client.isContractDeployed(wallet.address)) {
    throw new Error("Sender wallet is not deployed");
}

const walletContract = client.open(wallet);
const seqno = await walletContract.getSeqno();
const walletAddress = process.env.WALLET_ADDRESS;

console.log(`[1/5] Sending transfer from ${walletAddress} with seqno: ${seqno}`);

await walletContract.sendTransfer({
    secretKey: key.secretKey,
    seqno: seqno,
    messages: [
        internal({
            to: recipientAddress,
            value: amount,
            body: message,
            bounce: bounce,
        }),
    ],
});
```

Рис. 13. Send-transfer.js sendTransfer

Тобто, загалом, можливо імпортувати метод const { sendTransfer } = require('../utils/send-transfer'); який зовні надає простий інтерфейс для використання у routes, хоча всередині відбуваються складні дії, такі як: ініціалізація контракту, відправка транзакції, очікування підтвердження та перевірка статусу [5].

Ще один приклад фасаду можна навести з файлу utils/session.js (Рис. 14). У session.js відбувається:

- Генерація та перевірка JWT-токенів за допомогою бібліотеки jose;
- Робота з заголовками HTTP (Authorization);
- Обробка помилок, протерміновані токени, невірні підписи;

```
async function decrypt(token) {
    try {
        const { payload } = await jwtVerify(token, key, {
            algorithms: ['HS256'],
        });
        return payload;
    } catch (error) {
        console.error('JWT verification failed:', error);
        return null;
    }
}

async function getSession(req) {
    const sessionToken = req.headers.authorization?.split(' ')[1];
    if (!sessionToken) return null;
    return await decrypt(sessionToken);
}

const authMiddleware = async (req, res, next) => {
    try {
        const session = await getSession(req);
        if (!session) {
            return res.status(401).json({ message: 'Unauthorized: Please log in.' });
        }
        req.user = session.user;

        next();
    } catch (error) {
        console.error('Error in authMiddleware:', error);
        return res.status(500).json({ message: 'Internal server error' });
    }
};
```

Рис. 14. session.js

У session.js загалом надано 2 методи та константа, що приховують складну логіку, які вже готові для подальшого використання.

4. Патерн Repository

Repository — це патерн проектування, який виступає проміжним шаром між бізнес-логікою програми та джерелом даних (наприклад, базою даних). Він інкапсулює логіку доступу до даних, надаючи зручний

інтерфейс для операцій CRUD (Create, Read, Update, Delete) без розкриття деталей їх реалізації.

(Рис. 15) демонструє часткову реалізацію репозиторію з імпортом моделі User, де саме ця модель виконує роль репозиторію у Sequelize. Але ця реалізацію часткова в даному прикладі, оскільки є інкапсуляція запитів до БД (User.findOne(), user.update()), проте відсутній явний інтерфейс репозиторію, окремий сервіс, що абстрагує роботу з даними.

```
const existingUser = await User.findOne({
  where: { walletAddress: newWalletAddress }
});

if (existingUser && existingUser.id !== user.id) {
  return res.status(400).json({
    success: false,
    message: 'This wallet address is already in use by another user'
  });
}

await user.update({ walletAddress: newWalletAddress });

res.status(200).json({
  success: true,
  message: 'Wallet address was successfully updated',
  newWalletAddress: newWalletAddress
});
```

Рис. 15. routes/user.js

5. Патерн Dependency Injection

Dependency Injection (DI) — це патерн проектування, при якому залежності (сервіси, об'єкти, налаштування) не створюються всередині класу, а передаються ззовні. Це робить код більш гнучким і модульним.

Першим прикладом цього патерну може бути впровадження sequelize та DataTypes у моделі (Рис. 16).

```
'use strict';
const { Model } = require('sequelize');

module.exports = (sequelize, DataTypes) => {
  class Product extends Model {
    static associate(models) {
      Product.belongsTo(models.User, { foreignKey: 'userId', as: 'user'  });
      Product.hasMany(models.Review, { foreignKey: 'productId' });
      Product.hasMany(models.Order, { foreignKey: 'productId' });
    }
  }

  Product.init(
    {
      id: {
        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true,
      },
```

Рис. 16. models/product.js

Аналогічно приведеному прикладу кожна модель отримує екземпляр sequelize та DataTypes (типи полів: STRING, INTEGER, інші). Sequelize автоматично передає ці залежності при ініціалізації моделей.

Другим прикладом цього патерну може служити провадження залежностей у асоціації (Рис. 12).

А саме частина:

```
Object.keys(db).forEach(modelName => {
  if (db[modelName].associate) {
    db[modelName].associate(db);
  }
});
```

При виклику associate(db) моделі отримують доступ до всіх інших моделей через об'єкт db. Це дозволяє описувати зв'язки (hasMany, belongsTo) без явного імпорту моделей.

6. Патерн Middleware

Middleware — це патерн, який дозволяє обробляти HTTP-запити пошарово, передаючи їх через ланцюжок обробників перед тим, як вони потраплять до фінального маршруту.

- Middleware у Express:

Основна ціль використання Middleware перехоплення запитів перед обробкою в маршруті (/signup, /signin), модифікування req, res або перервати ланцюжок (наприклад, якщо користувач не авторизований), передача контролю наступному middleware або маршруту (Рис. 14).

На серверній частині він також використовується для захисту маршрутів (Рис. 17), Обробки помилок, Cross-Origin Resource Sharing та аутентифікації. У прикладі (Рис. 17) Middleware виконує ключову роль у захисті даних. Метод authMiddleware (Рис. 14) перевіряє JWT-токен з заголовка запиту (Authorization). Якщо токен валідний, він додає дані користувача до об'єкта req (наприклад, req.user.id) та викликає next() для передачі контролю до обробника маршруту .Якщо токен невалідний або відсутній: Повертає 401 Unauthorized. Користувач спочатку робить GET-запит після цього Express спочатку викликає authMiddleware, який перевіряє токен. Якщо все гаразд — маршрут отримує доступ до req.user.id. У тілі маршруту: використовується req.user.id для пошуку замовлень, де користувач є покупцем або продавцем. Завдяки цьому дані повертаються лише для поточного авторизованого користувача. Без authMiddleware будь-хто міг би отримати всі замовлення. Завдяки його

використанню маршрут фільтрує дані на основі req.user.id та логіка авторизації винесена окремо і не дублюється в кожному маршруті [9];

```
router.get('/orders', authMiddleware, async (req, res) => {
    try {
        const userId = req.user.id;

        const orders = await Order.findAll({
            where: {
                [Op.or]: [
                    { userId: userId },
                    { sellerId: userId }
                ]
            },
```

Рис. 17. routes/order.js

- Middleware у Redux:

У Redux middleware використовується для обробки actions перед тим, як вони потраплять до редюсера. Він може перехоплювати, модифікувати або відкладати дії, Використовується для логування, асинхронних запитів RTK Query.

У прикладі (Рис. 18) middleware обробляє HTTP-запити через createApi (RTK Query). кешує дані, обробляє помилки. Дозволяє використовувати useFetchOrdersQuery() та інше;

```
middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(
        userApiSlice.middleware,
        productApiSlice.middleware,
        reviewApiSlice.middleware,
        orderApiSlice.middleware,
```

Рис. 18. store.ts

7. Патерн Database Migrations

Database Migrations як патерн проектування — це спосіб керування змінами у базі даних за допомогою версіонованих скриптів, які: описують зміни схеми БД (таблиці, стовпці, індекси), мають механізм застосування та відкату, зберігають історію змін для синхронізації між середовищами.

Міграції — це механізм контролю версій схеми БД. Проект містить приклад у директорії migrations. Кожен файл міграції описує певні в БД та розробник може цим скористуватися аби застосувати або відкатити певні зміни.

Приклад частини міграції init (Рис. 19).

```
down: async (queryInterface, Sequelize) => {
  await queryInterface.dropTable('Payments');
  await queryInterface.dropTable('Reviews');
  await queryInterface.dropTable('Orders');
  await queryInterface.dropTable('Products');
  await queryInterface.dropTable('Users');
},
```

Рис. 19. 20250313154947-init.js

Кожна міграція має унікальну назву та timestamp, який присвоюється їй після створення. Команда npx sequelize-cli migration:generate --name init у Sequelize створює новий файл міграції з базовим шаблоном для змін у базі даних. Команда npx sequelize-cli db:migrate застосовує наявні міграції.

Історія міграцій зберігається в самій БД, спеціальна таблиця SequelizeMeta, яка містить тільки назву виконаних міграцій. Міграції можуть бути структурні та скриптові, структурні змінюють схему БД, тоді як скриптові виконують SQL-запити. Якщо дуже потрібно на етапі розробки можна видалити міграцію із SequelizeMeta, редагувати її та виконати повторно. Але не потрібно змінювати міграції великих проектів чи проектів на етапі продакшену.

8. Патерн Environment Configuration

Патерни для конфігурації — це підходи до організації та управління налаштуваннями програми, які забезпечують гнучкість, безпеку та зручність роботи з конфігураційними даними. Патерн, при якому налаштування програми зберігаються у зовнішніх файлах, наприклад, .env та config.json, а не вписані в код (Рис. 20).

Компоненти:

- .env-файл: Містить змінні оточення (наприклад, DATABASE_URL=postgres://user:pass@localhost:5432/db), зазвичай додається до .gitignore, щоб секрети не потрапляли до репозиторію або використовується приватний репозиторій, завантажується під час запуску програми (наприклад, за допомогою бібліотеки dotenv у Node.js [10];
- config.json (або інші JSON/YAML/TOML-файли):
  Зберігає структуровані налаштування (порти, URL-адреси, параметри логування тощо) та може бути залежним від середовища (наприклад, config.dev.json, config.prod.json);

```
"development": {
  "username": "myuser",
  "password": "mypassword",
  "database": "mydb",
  "host": "localhost",
  "dialect": "postgres"
},
"test": {
  "username": "myuser",
  "password": "mypassword",
  "database": "mydb_test",
  "host": "localhost",
  "dialect": "postgres"
},
"production": {
```

Рис. 20. config.json

3.2 Опис моделі

У Express.js з Sequelize (ORM для Node.js), модель даних — це JavaScript-об'єкт, який представляє структуру таблиці в базі даних. Модель створюється за допомогою sequelize.define() або через декоратори. Ключові елементи моделі:

- Атрибути (поля) – визначають структуру даних (типи: STRING, INTEGER, BOOLEAN, DATE тощо);
- Валідації – перевірка даних перед записом у БД (наприклад, isEmail);
- Зв'язки – belongsTo, hasMany, many-to-many (через belongsToMany);

Модель у Sequelize – це абстракція над таблицею БД, яка дозволяє працювати з даними через JavaScript-методи, не пишучи запити. Sequelize автоматизує: створення таблиць, додавання та вибірку даних, валідацію та обробку помилок.

Для роботи з Sequelize в Express потрібно правильно підключити модель до системи (через sequelize.sync() або міграції).

ORM (Object-Relational Mapping) – це технологія, яка дозволяє працювати з базою даних через об'єкти у коді, замість SQL-запитів. Sequelize – популярна ORM для Node.js, що підтримує PostgreSQL, MySQL, SQLite та інші СУБД. Моделі допомагають структурувати дані, уникаючи незручних запитів, і значно прискорюють розробку.

Розглянемо реалізацію моделі Review для прикладу (Рис. 21):

'use strict' – вмикає строгий режим JavaScript (запобігає потенційним помилкам).

const { Model } – імпортує базовий клас Model з Sequelize, від якого наслідуватимется модель Review.

Експорт функції, що повертає модель. Функція приймає sequelize – підключення до БД та DataTypes – об'єкт з типами даних (STRING, INTEGER тощо).

Оголошення класу Review: Review extends Model – клас успадковує всі методи Sequelize (CRUD, хуки тощо), tatic associate(models) – місце для визначення зв'язків з іншими моделями.

Визначення зв'язків (асоціацій): Review.belongsTo(models.User, { foreignKey: 'userId', as: 'User' }), тобто користувач може мати багато відгуків, foreignKey: 'userId' – зовнішній ключ у таблиці Review, as: 'User' – псевдонім для доступу до зв'язаних даних (через user.getReviews()).

Аналогічно для товарів (Product), тощо.

Ініціалізація моделі (Review.init):

- Review.init() – метод, що ініціалізує модель;
- Перший аргумент – опис атрибутів таблиці;
- Другий аргумент – налаштування;
- sequelize – передане підключення до БД;
- modelName – ім'я моделі для використання у запитах;
- tableName – реальна назва таблиці в БД;
- timestamps – якщо true, додає поля createdAt та updatedAt. Для відстеження змін у рядках;

Опис атрибутів моделі:

- DataTypes – тип даних колонки;
- primaryKey – первинний ключ;
- autoIncrement – автоматичне збільшення значення;
- validate – валідація формату;
- allowNull – параметр обов'язкове поле чи ні;

- unique – унікальне значення;
- defaultValue – значення за замовчуванням;
- ENUM – обмежений набір значень;

Повернення моделі: return Review;

Функція повертає готову модель для використання в інших частинах програми.

Загалом можна підсумувати, що для створення моделі потрібно: створити модель через наслідування від Model, поля описуються в .init() з типами, обмеженнями та валідацією, зв’язки з іншими моделями (Product, тощо) визначаються в associate(), налаштування (tableName, timestamps) передаються другим аргументом до init(). Така модель дозволить працювати з даними через зручні методи.

```
module.exports = (sequelize, DataTypes) => {
  class Review extends Model {
    static associate(models) {
      Review.belongsTo(models.User, { foreignKey: 'userId', as: 'User' });
      Review.belongsTo(models.Product, { foreignKey: 'productId', as: 'Product' 
    }
  }

  Review.init(
    {
      id: {
        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true,
      },
      rating: {
        type: DataTypes.INTEGER,
        allowNull: false,
        validate: {
          min: 1,
          max: 5,
        },
      },
      text: {
        type: DataTypes.TEXT,
        allowNull: true,
      },
      userId: {
        type: DataTypes.INTEGER,
        allowNull: false,
        references: {
          model: 'Users',
          key: 'id',
```

Рис. 21. models/review.js

Інші моделі даних були створені схожим чином.

3.3 Реалізація логіки обробки запитів та бізнес логіки

Реалізація логіки обробки запитів та бізнес-логіки у сучасних веб-застосунках є одним із ключових аспектів створення безпечних, масштабованих і підтримуваних систем. За допомогою Node.js із використанням Express.js, Sequelize ORM та бібліотекою bcrypt для хешування паролів, можна розглянути, як виконана ця реалізація на практиці.

Першим етапом обробки запитів є валідація вхідних даних. У випадку маршруту /signup, перевіряється наявність усіх обов'язкових полів (Рис. 22): name, surname, email, password, country. Ця перевірка реалізована у вигляді умовного блоку if, і в разі виявлення помилки повертається відповідь зі статусом 400 (Bad Request). Це дозволяє уникнути створення об'єктів з неповними або невалідними та помилковими даними, що є важливою частиною бізнес-логіки.

```
const express = require('express');
const bcrypt = require('bcrypt');
const { User } = require('../models');
const { encrypt, SESSION_DURATION } = require('../utils/session');

const router = express.Router();

router.post('/signup', async (req, res) => {
    try {
        const { name, surname, email, password, role, country, walletAddress } = req.body;

        if (!name || !surname || !email || !password || !country) {
            return res.status(400).json({ message: 'Missing required fields' });
        }
```

Рис. 22. routes/auth.js

Наступним кроком є перевірка унікальності певних значень, таких як email або walletAddress. Використовується ORM Sequelize для запитів до бази даних методом findOne (Рис. 23). У разі виявлення збігів повертається

відповідне повідомлення про помилку. Це частина логіки бізнесу, що захищає від дублювання записів і забезпечує цілісність даних.

Окремо слід відзначити процес хешування паролів за допомогою бібліотеки bcrypt. Це критично важливий аспект безпеки, який унеможливлює зберігання паролів у відкритому вигляді. У реалізованому алгоритму застосовується 5 раундів хешування (saltRounds = 5), що є компромісом між швидкістю обробки та захищеністю.

```
const existingUser = await User.findOne({ where: { email } });
if (existingUser) {
    return res.status(400).json({ message: 'User with this email already exists' });
}

if (walletAddress) {
    const existingWallet = await User.findOne({ where: { walletAddress } });
    if (existingWallet) {
        return res.status(400).json({ message: 'This wallet address is already in use' });
    }
}

const saltRounds = 5;
const hashedPassword = await bcrypt.hash(password, saltRounds);

const user = await User.create({
    name,
    surname,
    email,
    password: hashedPassword,
    role: role || 'CUSTOMER',
    country,
    walletAddress: walletAddress || null
});
```

Рис. 23. routes/auth.js

Після успішного створення користувача відбувається генерація сесійного токену за допомогою функції encryptSession. Вона створює об'єкт сесії із зазначенням строку дії (SESSION_DURATION) та виконує його шифрування. Це дозволяє реалізувати механізм аутентифікації через токени без збереження сесій на стороні сервера — сучасний підхід, що дозволяє легко масштабувати систему.

Реалізація логіки входу в систему /signin також містить низку перевірок. По-перше, перевіряється, чи існує користувач з вказаним email. Потім відбувається порівняння введеного пароля з хешованим паролем, збереженим у базі даних, знову ж таки з використанням методу bcrypt.compare. У разі успіху користувач отримує новий токен доступу та свої дані без пароля. Такий підхід відповідає загальноприйнятим практикам інформаційної безпеки (Рис. 24).

Загалом, при виконанні умов успішної авторизації для користувача будуть доступні більшість запитів. Це обов'язкова умова задля дотримання інформаційної безпеки. У подальших маршрутах майже всюди буде використовуватись authMiddleware, та за його допомогою більшість чутливих даних користувача будемо отримувати через req.user, що дозволить захистити приватні дані користувача та дотриматися загальним правилам інформаційної безпеки.

SESSION_SECRET розміщений у дотенв файлі, що надійно його захищає від несанкціонованого доступу третім особам та дозволить переконатися у безпеці приватних даних користувачів.

```
router.post('/signin', async (req, res) => {
    try {
        const { email, password } = req.body;

        if (!email || !password) {
            return res.status(400).json({
                message: 'Email and password are required'
            });
        }

        const user = await User.findOne({
            where: { email },
            attributes: { include: ['password'] }
        });

        if (!user) {
            return res.status(401).json({
                message: 'Invalid email or password'
            });
        }

        const isPasswordValid = await bcrypt.compare(password, user.password);
        if (!isPasswordValid) {
            return res.status(401).json({
                message: 'Invalid email or password'
            });
        }

        const token = await encryptSession(user);

        const userResponse = {
            id: user.id,
            name: user.name,
            surname: user.surname,
            email: user.email,
            role: user.role,
            country: user.country,
            walletAddress: user.walletAddress
        };

        res.status(200).json({
            token,
            user: userResponse
        });
```

Рис. 24. routes/auth.js

Обробка товарів, реалізована у маршрутах /products та /add-product, демонструє логіку роботи з базою даних у контексті бізнес-процесів

платформи. У маршруті GET /products спочатку здійснюється запит до бази для отримання усіх товарів, включаючи дані продавця. Використання include з Sequelize дозволяє робити складні запити із приєднанням пов’язаних моделей, зменшуючи кількість запитів до бази. Після цього дані форматуються у зручний для клієнта формат, що забезпечує ізоляцію внутрішньої структури бази від фронтенду (Рис. 25).

```
router.get('/products', async (req, res) => {
  try {
    const products = await Product.findAll({
      include: [{
        model: User,
        as: 'user',
        attributes: ['id', 'name', 'role', 'walletAddress']
      }],
      order: [['createdAt', 'DESC']]
    });

    const formattedProducts = products.map(product => ({
      id: product.id,
      name: product.name,
      description: product.description,
      price: product.price,
      quantity: product.quantity,
      category: product.category,
      originCountry: product.originCountry,
      currentCountry: product.currentCountry,
      photoUrl: product.photoUrl,
      seller: {
        id: product.user.id,
        name: product.user.name,
        role: product.user.role,
        walletAddress: product.user.walletAddress
      },
      createdAt: product.createdAt
    }));

    res.status(200).json({
      success: true,
      count: products.length,
      products: formattedProducts
    });
```

Рис. 25. routes/product.js

Маршрут POST /add-product включає автентифікацію користувача через middleware authMiddleware, валідацію ролі користувача (дозволено лише SELLER, MANUFACTURER, DISTRIBUTOR), перевірку наявності

гаманця, та валідацію полів товару. Це все приклади реалізації комплексної бізнес-логіки, яка забезпечує правила користування сервісом: хто може додавати товари, з якими параметрами, і за яких умов. Важливо, що валідація типів і значень реалізована вручну, що хоча і потребує більше коду, зате дає повний контроль над поведінкою застосунку (Рис. 26).

```
const allowedRoles = ['SELLER', 'MANUFACTURER', 'DISTRIBUTOR'];
if (!allowedRoles.includes(user.role)) {
  return res.status(403).json({
    success: false,
    message: 'Only sellers, manufacturers and distributors can add products',
    yourRole: user.role
  });
}

const requiredFields = {
  name: 'string',
  description: 'string',
  price: 'number',
  quantity: 'number',
  category: 'string',
  originCountry: 'string',
  currentCountry: 'string'
};

const errors = [];

Object.entries(requiredFields).forEach(([field, type]) => {
  if (!req.body[field]) {
    errors.push(`${field} is required`);
  } else if (typeof req.body[field] !== type) {
    errors.push(`${field} must be a ${type}`);
  }
});

if (errors.length > 0) {
  return res.status(400).json({
    success: false,
    message: 'Validation failed',
    errors
  });
}
```

Рис. 26. routes/product.js

Маршрути для додавання та перегляду відгуків /products/:productId/reviews також мають чітко визначену логіку. Наприклад, користувач може залишити відгук тільки один раз, а сам відгук

повинен містити оцінку від 1 до 5. Це запобігає зловживанням системою оцінювання (Рис. 27).

```
router.post('/products/:productId/reviews', authMiddleware, async (req, res) => {
  try {
    const { productId } = req.params;
    const { rating, text } = req.body;
    const userId = req.user.id;

    if (typeof rating !== 'number' || rating < 1 || rating > 5) {
      return res.status(400).json({
        success: false,
        message: 'Rating is required, must be a number between 1 and 5'
      });
    }

    const product = await Product.findByPk(productId);
    if (!product) {
      return res.status(404).json({
        success: false,
        message: 'Product not found'
      });
    }

    const existingReview = await Review.findOne({
      where: { userId, productId }
    });

    if (existingReview) {
      return res.status(400).json({
        success: false,
        message: 'You have already reviewed this product'
      });
    }

    const review = await Review.create({
      rating,
      text: text || null,
      userId,
      productId
    });
```

Рис. 27. routes/reviews.js

Окремо варто згадати оновлення гаманця користувача у маршруті /update-wallet. Логіка передбачає перевірку, чи не співпадає новий гаманець із уже зареєстрованим у базі, що гарантує унікальність гаманців.

Це ще один приклад реалізації обов’язкових правил, які контролюють унікальність та цілісність важливої фінансової інформації.

На завершення, можна сказати, що було досягнено класичну реалізацію багаторівневої архітектури з чітким розділенням відповідальності: Express відповідає за маршрутизацію, Sequelize — за доступ до бази, middleware — за автентифікацію, а бізнес-логіка розташована безпосередньо в обробниках маршрутів. Хоча в більших проектах часто рекомендується виносити логіку у окремі сервіси або контролери для кращої модульності, даний приклад чудово ілюструє принципи роботи сучасного backend-застосунку.

3.4 Взаємодія між обробкою запитів та моделями даних

Веб-розробка з використанням Node.js, Express та Sequelize передбачає тісну взаємодію між обробкою HTTP-запитів і моделями даних. Ця взаємодія є основою для створення надійних, масштабованих і безпечних сервісів. У цьому контексті моделі даних визначають структуру та взаємозв'язки між сутностями, тоді як обробники запитів реалізують бізнес-логіку, використовуючи ці моделі.

Sequelize — це ORM (Object-Relational Mapping) для Node.js, який дозволяє розробникам працювати з базами даних, використовуючи об'єктно-орієнтований підхід. Моделі в Sequelize представляють таблиці в базі даних і визначають їхню структуру, включаючи поля, типи даних, обмеження та взаємозв'язки з іншими моделями.

Наприклад, модель User має наступні поля: id, name, email, password, role, country, walletAddress. Ця модель має деякі зв’язки з іншими моделями, такими як Product, Review та інші розглянуті вище, що дозволяє легко отримувати пов'язані дані. Додавання у БД даних та отримання їх за допомогою цієї моделі продемонстровано за допомогою (Рис. 23) та (Рис. 24) відповідно.

Також можна побачити, як використовується метод encryptSession(), для отримання захищеної сесії користувачем (Рис. 28). Також там є параметр, за допомогою якого можна відстежити чи сплинула сесія за часом. У випадку якщо сесія сплинула, токен стає невалідним і відповідний метод дасть нам про це дізнатися.

```
async function encryptSession(user) {
    const sessionExpires = new Date(Date.now() + SESSION_DURATION);
    return await encrypt({
        user: {
            id: user.id,
            name: user.name,
            email: user.email,
            role: user.role,
            country: user.country,
        },
        expires: sessionExpires,
    });
}
```

Рис. 28. encryptSession.js

Express — це мінімалістичний веб-фреймворк для Node.js, який дозволяє створювати маршрути для обробки HTTP-запитів. Кожен маршрут відповідає певному URL і типу запиту (GET, POST, PUT, DELETE) та виконує відповідну функцію-обробник.

У функції-обробнику можна отримати дані з запиту (req), взаємодіяти з моделями Sequelize для читання або запису даних у базу, а також формувати відповідь (res) для клієнта, для подальшої відправки на клієнтську частину. Але непотрібно забувати, що деякі дані, можуть бути чутливими, або небажаними для відправки.

Взаємодія між обробниками запитів і моделями даних є ключовою в архітектурі веб-сервісів. Розглянемо кілька прикладів:

- Реєстрація користувача (/signup): Обробник отримує дані з тіла запиту, перевіряє їхню валідність, хешує пароль за допомогою bcrypt, створює нового користувача в базі даних через модель User і повертає токен сесії;

- Авторизація користувача (/signin): Обробник перевіряє наявність користувача з вказаною електронною поштою, порівнює хеш пароля, і у разі успіху повертає токен сесії (Рис. 24);
- Додавання продукту (/add-product): Обробник перевіряє роль користувача, валідність введених даних, і створює новий запис у таблиці Product, пов'язаний з користувачем (Рис. 29);
- Додавання відгуку (/products/:productId/reviews): Обробник перевіряє, чи існує продукт, чи користувач вже залишав відгук, і створює новий запис у таблиці Review (Рис. 27);

У всіх цих випадках обробники запитів використовують методи моделей Sequelize (findOne, create, update, destroy) для взаємодії з базою даних.

```
if (errors.length > 0) {
  return res.status(400).json({
    success: false,
    message: 'Validation failed',
    errors
  });
}

const product = await Product.create({
  name,
  description,
  price,
  quantity,
  category,
  originCountry,
  currentCountry,
  photoUrl: photoUrl || null,
  userId: user.id
});
```

Рис. 29. routes/product.js

Важливою частиною взаємодії між обробниками запитів і моделями є валідація введених даних та обробка помилок. Перед створенням або оновленням записів у базі даних обробники повинні перевіряти, чи всі обов'язкові поля заповнені, чи мають правильний формат, чи не порушують унікальні обмеження (Рис 29).

У разі помилок, таких як SequelizeValidationError, обробники повинні повертати відповідні повідомлення клієнту з описом проблеми.

Для забезпечення безпеки веб-сервісів необхідно реалізувати механізми аутентифікації (перевірка особи користувача) та авторизації (перевірка прав доступу).

Вище вказувалося, що використовується токен сесії, створений за допомогою encryptSession, який містить інформацію про користувача та термін дії сесії. Цей токен передається в заголовку Authorization при наступних запитах і перевіряється за допомогою middleware authMiddleware.

Таким чином, обробники запитів можуть отримати інформацію про поточного користувача з req.user і перевіряти його роль або інші атрибути для визначення прав доступу.

Sequelize дозволяє визначати взаємозв'язки між моделями, такі як hasMany, belongsTo, belongsToMany. Це дозволяє легко отримувати пов'язані дані, наприклад, всі продукти, створені певним користувачем, або всі відгуки для певного продукту.

У попередніх пунктах згадувалося, що наприклад модель Product має зв'язок з моделлю User через поле userId, що дозволяє отримувати інформацію про продавця разом із даними продукту.

Загалом треба усвідомити, що взаємодія між обробкою запитів і моделями даних у сервісах на базі Node.js, Express і Sequelize є фундаментальною для реалізації бізнес-логіки, забезпечення безпеки та ефективної роботи з базою даних. Розуміння цієї взаємодії дозволяє розробникам створювати масштабовані та надійні програми, які відповідають сучасним вимогам до функціональності та безпеки.

4. ІНТЕГРАЦІЯ БЛОКЧЕЙН ТЕХНОЛОГІЙ

4.1 Реалізація блокчейн технологій у серверній частині

Блокчейн — це технологія розподілених реєстрів, де дані зберігаються у вигляді ланцюжка блоків, захищених криптографією. Кожен блок містить транзакції (або іншу інформацію), хеш попереднього блоку та власний хеш, що робить ланцюг практично незмінним. Основні переваги: децентралізація (немає єдиного контролюючого центру), прозорість (всі операції можна перевірити) та безпека (через криптографічні алгоритми).

Найвідоміший приклад — Bitcoin, де блокчейн використовується для запису транзакцій. Однак технологія значно ширша: вона застосовується для смарт-контрактів (Ethereum), логістики, цифрових ідентифікаторів тощо.

Головні виклики — масштабованість (обмежена кількість транзакцій за раз) та значні енергетичні витрати (у системах на кшталт Bitcoin). Проте блокчейн відкриває нові можливості для фінансів, юриспруденції та інших сфер, усуваючи необхідність довіри до посередників.

Транзакція в блокчейні — це цифрова операція, що включає передачу даних (наприклад, криптовалют) між учасниками мережі. Вона складається з таких основних елементів:

- Вхідні дані (Inputs) — посилання на попередні транзакції, які підтверджують наявність коштів у відправника (наприклад, сума BTC з його гаманця);
- Вихідні дані (Outputs) — інформація про одержувача (його публічну адресу) та суму, яку йому передають;

- Підпис (Digital Signature) — криптографічне підтвердження, що транзакцію авторизував власник приватного ключа (без нього операція неможлива);
- Комісія (Fee) — добровільна винагорода для майнерів/стейкерів, які включать транзакцію в блок (зазвичай чим вища комісія, тим швидше підтвердження);
- Ідентифікатор (TxID) — унікальний хеш транзакції, що дозволяє відстежити її статус в мережі;

Після створення транзакція потрапляє в пул непідтверджених операцій, а потім — у новий блок після перевірки вузлами (нодами). Наприклад, у Bitcoin цей процес відбувається через консенсус PoW (Proof of Work), а в Ethereum 2.0 — через PoS (Proof of Stake).

Головні особливості: безпека (неможливість підробити підпис), прозорість (всі транзакції відкриті) та незворотність (після підтвердження скасувати операцію неможливо).

У серверній частині блокчейн буде відповідати за низку наступних операцій:

- Оплата угоди;
- Зміна статусу угоди;
- Запит на отримання коштів при виконанні умов (За замовчуванням оплата та отримання замовлення покупцем, відправка продавцем);
- Отримання коштів;
- Верифікація транзакцій;

Тобто якщо підсумувати то маємо реалізований сервіс, який дозволяє покупцю перевести кошти на контракт сервісу і не перейматися за доброчесність продавця, так як до виконання умов кошти не надходять до продавця. В свою чергу продавці не будуть сумніватися у швидкості та

надійності вчасного переказу коштів. При виконанні умов та запиту на отримання вони надійдуть моментально та автоматично.

Для демонстрації реалізації використана тестова мережа The Open Network, вона обрана завдяки певним перевагам, таким як, швидкі та дешеві транзакцій, масштабованість, сучасна екосистема для розробників, ефективні інструменти та безпека і децентралізація.

Спочатку, важливо, усвідомити що один і той же адрес має різні представлення зокрема це мережа, яка може бути mainnet або testnet, та для кожної з мереж маємо Bounceable та Non-bounceable адреси. Знаючи це, можемо отримати 4 адреси. Bounceable та Non-bounceable мають відмінність у тому, що Bounceable адрес, на який можна надсилати монети, і якщо виникне помилка або баг, вони повернуться відправнику. Non-bounceable адрес, на який транзакції не можна скасувати але він використовується для смарт-контрактів. Вищезгадані адреси закодовані у форматі base64url. Але вони мають одну й ту саму HEX адресу, яка і є унікальним ідентифікатором гаманця в мережі, та саме ця адреса відображається у транзакціях.

Щоб HEX представлення маючи одну з відповідних йому адрес, було реалізовано метод formatAdress(base64urlAddress), який продемонстровано нижче (Рис. 30).

```
async function formatAddress(base64urlAddress) {
    try {
        const url = `https://toncenter.com/api/v2/unpackAddress?address=${base64urlAddress}`;

        const response = await axios.get(url);

        if (response.data.ok) {
            console.log('Address unpacked successfully:', response.data.result);
            return response.data.result;
        } else {
            throw new Error('Failed to unpack address');
        }
    } catch (error) {
        console.error('Error parsing TON address:', error);
        throw error;
    }
}
```

Рис. 30. formatAddress.js

Наступним кроком визначимося, саме за якими даними буде верифікуватися транзакція. Нам надійде tx_hash (унікальний ідентифікатор транзакції), але підтвердження тільки наявність транзакції у блокчейну буде недостатньо. Першим кроком нам потрібно отримати дані транзакції, за запитом вони будуть мати наступний вигляд (Рис. 31). Серед цих даних нас будуть цікавити: transactionData.success, transactionData.out_msgs, outMessage.source.address, outMessage.destination.address, outMessage.decoded_body.text, outMessage.value.

Рис. 31. Transaction response

В свою чергу outMessage.decoded_body.text матиме наступний вигляд (Рис. 32). Ці дані теж потрібно в свою чергу отримати, відсортувати та разпарсити для подальшої верифікації.

```
'decoded_body": {
  "text": "Order payment:{\"userId\":11,\"productId\":5,\"type\":\"transfer\",\"quantity\":2,\"fromLocation\":\"ENGLAND\",\"toLocation\":\"Germany Stuttgart Stuttgart Ost Karpalaz 12\",\"orderId\":13,\"amount\":4}"
'
```

Рис. 32. transactionData.outMessage.decoded_body.text

Загалом аналізуючи вищезгадане було розроблено метод для верифікації транзакції що містить десять основних кроків. Приклади частин метода наведено нижче (Рис. 33) та (Рис. 34).

```
const transactionData = response.data;
console.log('[1/9] Данные транзакции получены:', {
    success: transactionData.success,
    hash: transactionData.hash
});

console.log(`[2/9] Проверка успешности транзакции...`);
if (!transactionData.success) {
    console.error('[2/9] Транзакция не была успешной:', transactionData);
    return {
        success: false,
        error: 'Transaction failed'
    };
}
console.log('[2/9] Транзакция успешна');

console.log(`[3/9] Поиск исходящего сообщения...`);
const outMessage = transactionData.out_msgs?.[0];
if (!outMessage) {
    console.error('[3/9] Исходящее сообщение не найдено');
    return {
        success: false,
        error: 'No output message found'
    };
}
console.log('[3/9] Исходящее сообщение найдено');

console.log(`[4/9] Форматирование адреса отправителя...`);
const formattedSenderAddress = await formatAddress(senderAddress);
console.log('[4/9] Адрес отправителя отформатирован:', {
    original: senderAddress,
    formatted: formattedSenderAddress
});
```

Рис. 33. verifyTransaction1

```
console.log(`[5/9] Проверка адреса отправителя...`);
if (outMessage.source?.address !== formattedSenderAddress) {
    console.error('[5/9] Несоответствие адреса отправителя:', {
        expected: formattedSenderAddress,
        actual: outMessage.source?.address
    });
    return {
        success: false,
        error: 'Sender address mismatch'
    };
}
console.log('[5/9] Адрес отправителя подтвержден');

console.log(`[6/9] Проверка адреса получателя...`);
if (outMessage.destination?.address !== process.env.WALLET_ADDRESS) {
    console.error('[6/9] Несоответствие адреса получателя:', {
        expected: process.env.WALLET_ADDRESS,
        actual: outMessage.destination?.address
    });
    return {
        success: false,
        error: 'Recipient address mismatch'
    };
}
console.log('[6/9] Адрес получателя подтвержден');

console.log(`[7/9] Парсинг текста транзакции...`);
const transactionText = outMessage?.decoded_body?.text;
if (!transactionText) {
    console.error('[7/9] Текст транзакции не найден');
    return {
        success: false,
        error: 'Transaction text not found'
    };
}
console.log('[7/9] Текст транзакции:', transactionText);
```

Рис. 34. verifyTransactio2

У алгоритмі дані порівнюються за допомогою оператора !==, це строгий оператор нерівності, який порівняє тип та значення не використовуючи автоматичне приведення типів. Якщо дані виявляться нерівні то виконається умова и метод поверне помилку з логуваням даних.

Схожим чином будемо перевіряти кожну транзакцію.

Для реалізації автоматичного переведення на адресу продавця було реалізовано наступний метод (Рис. 36). Для його реалізації був розроблений наступний алгоритм. Завдяки чутливим даним отриманим з дотенв, було створено спочатку ключ та гаманець. Далі виконується підключення до мережі та перевірка чи розгорнуто контракт, якщо контракт не розгорнуто, то повертається помилка. Якщо перевірка завершилась успіхом то ми отримуємо seqno (порядковий номер транзакції), для відстеження статусу та подальшого отримання хеша. Чекаємо на відправку, та після успіху передаємо у метод для отримання хеша транзакції (Рис. 35).

```
async function getTransactionBySeqno(walletAddress, seqno) {
    try {
        const response = await axios.get(
            `https://testnet.tonapi.io/v2/blockchain/accounts/${walletAddress}/transactions`,
            {
                params: {
                    limit: 100
                }
            }
        );

        return response.data.transactions.find(tx =>
            tx.in_msg?.decoded_body?.seqno === seqno
        );
    } catch (error) {
        console.error('Error fetching transactions:', error);
        return null;
    }
}
```

Рис. 36. getTransactionBySeqno

```
const key = await mnemonicToWalletKey(process.env.TON_MNEMONIC.split(" "));
const wallet = WalletContractV4.create({
    publicKey: key.publicKey,
    workchain: 0
});

const endpoint = await getHttpEndpoint({
    network: process.env.TON_NETWORK || "testnet"
});
const client = new TonClient({ endpoint });

if (!await client.isContractDeployed(wallet.address)) {
    throw new Error("Sender wallet is not deployed");
}

const walletContract = client.open(wallet);
const seqno = await walletContract.getSeqno();
const walletAddress = process.env.WALLET_ADDRESS;

console.log(`[1/5] Sending transfer from ${walletAddress} with seqno: ${seqno}`);

await walletContract.sendTransfer({
    secretKey: key.secretKey,
    seqno: seqno,
    messages: [
        internal({
            to: recipientAddress,
            value: amount,
            body: message,
            bounce: bounce,
        }),
    ],
});

console.log(`[2/5] Transfer sent, waiting for confirmation...`);

let currentSeqno = seqno;
let attempts = 0;
const maxAttempts = 10;

while (currentSeqno === seqno && attempts < maxAttempts) {
    await sleep(1500);
    currentSeqno = await walletContract.getSeqno();
    attempts++;
    console.log(`[3/5] Waiting for confirmation... Attempt ${attempts}/${maxAttempts}`);
}
```

Рис. 36. sendTransaction1

Отримання хешу за допомогою запиту зв’язано з відсутністю можливості отримати хеш напряму, як у клієнтській частині.

Після реалізації мною було проведено багато тестів, основні та найважливіші для демонстрації роботу сервісу наведено нижче [11]:

Роль продавця, демонстрація відправки замовлення (Рис. 37):

```
Проверка транзакции отправки d9cb4030d6453e18a4a5edd6291ce25fa4dfaacbeee70f39ce7a701194214178...
[verifyTransaction] Начало проверки транзакции d9cb4030d6453e18a4a5edd6291ce25fa4dfaacbeee70f39ce7a701194214178 для пользователя 15
[1/9] Запрос данных транзакции...
[1/9] Данные транзакции получены: {
  success: true,
  hash: '914625d5f567bdcc9fc2a00b7c37c7cc621e8692efbb64015090f6658a738d61'
}
[2/9] Проверка успешности транзакции...
[2/9] Транзакция успешна
[3/9] Поиск исходящего сообщения...
[3/9] Исходящее сообщение найдено
[4/9] Форматирование адреса отправителя...
Address unpacked successfully: 0:2179b8ce49838f0225db1276e18a133eca14028f5c4ff8eab3de3aa246bb6cd6
[4/9] Адрес отправителя отформатирован: {
  original: '0QAhebjOSYOPAiXbEnbhihM-yhQCj1xP-Oqz3jqiRrts1o7P',
  formatted: '0:2179b8ce49838f0225db1276e18a133eca14028f5c4ff8eab3de3aa246bb6cd6'
}
[5/9] Проверка адреса отправителя...
[5/9] Адрес отправителя подтвержден
[6/9] Проверка адреса получателя...
[6/9] Адрес получателя подтвержден
[7/9] Парсинг текста транзакции...
[7/9] Текст транзакции: 15/6/sending/3/ENGLAND/Germany Berlin Berlin Ost Karlhorst Blumenstrasse 32
[8/9] Разбор данных транзакции...
[8/9] Данные транзакции: {
  userId: 15,
  productId: 6,
  type: 'sending',
  quantity: 3,
  fromLocation: 'ENGLAND',
  toLocation: 'Germany Berlin Berlin Ost Karlhorst Blumenstrasse 32'
}
[9/9] Проверка соответствия данных...
[9/9] Данные транзакции подтверждены
[10/10] Проверка суммы транзакции...
[10/10] Сумма транзакции подтверждена: 0.01
[verifyTransaction] Транзакция успешно проверена
Executing (default): UPDATE "Orders" SET "sendHash"=$1,"updatedAt"=$2 WHERE "id" = $3
Executing (default): UPDATE "Payments" SET "isConfirmed"=$1,"amount"=$2,"updatedAt"=$3 WHERE "id" = $4
Транзакция отправки d9cb4030d6453e18a4a5edd6291ce25fa4dfaacbeee70f39ce7a701194214178 подтверждена
```

Рис. 37. Transaction1 logs

Роль продавця, демонстрація запиту оплати при виконанні умов угоди (Рис. 38):

```
Проверка транзакции подтверждения получения 9a9f4050b7935d2c199aa604c0e8dcb5ec8c3a5ef3c613bef3f6f5245164a62a...
[verifyTransaction] Начало проверки транзакции 9a9f4050b7935d2c199aa604c0e8dcb5ec8c3a5ef3c613bef3f6f5245164a62a для пользователя 15
[1/9] Запрос данных транзакции...
[1/9] Данные транзакции получены: {
  success: true,
  hash: 'b43f12c6754db10788279b76c87ba7b569a232270d99edba1c7ffaa68b27a145'
}
[2/9] Проверка успешности транзакции...
[2/9] Транзакция успешна
[3/9] Поиск исходящего сообщения...
[3/9] Исходящее сообщение найдено
[4/9] Форматирование адреса отправителя...
Address unpacked successfully: 0:2179b8ce49838f0225db1276e18a133eca14028f5c4ff8eab3de3aa246bb6cd6
[4/9] Адрес отправителя отформатирован: {
  original: '0QAhebjOSYOPAiXbEnbhihM-yhQCj1xP-Oqz3jqiRrts1o7P',
  formatted: '0:2179b8ce49838f0225db1276e18a133eca14028f5c4ff8eab3de3aa246bb6cd6'
}
[5/9] Проверка адреса отправителя...
[5/9] Адрес отправителя подтвержден
[6/9] Проверка адреса получателя...
[6/9] Адрес получателя подтвержден
[7/9] Парсинг текста транзакции...
[7/9] Текст транзакции: 15/6/reception/3/ENGLAND/Germany Berlin Berlin Ost Karlhorst Blumenstrasse 32
[8/9] Разбор данных транзакции...
[8/9] Данные транзакции: {
  userId: 15,
  productId: 6,
  type: 'reception',
  quantity: 3,
  fromLocation: 'ENGLAND',
  toLocation: 'Germany Berlin Berlin Ost Karlhorst Blumenstrasse 32'
}
[9/9] Проверка соответствия данных...
[9/9] Данные транзакции подтверждены
[10/10] Проверка суммы транзакции...
[10/10] Сумма транзакции подтверждена: 0.01
[verifyTransaction] Транзакция успешно проверена
[1/5] Sending transfer from 0:624cfe50858fada7aaaa6b8c8de438122a64ac2653c96048d17c17683b98b103 with seqno: 44
[2/5] Transfer sent, waiting for confirmation...
[3/5] Waiting for confirmation... Attempt 1/10
[3/5] Waiting for confirmation... Attempt 2/10
[3/5] Waiting for confirmation... Attempt 3/10
[3/5] Waiting for confirmation... Attempt 4/10
[4/5] Transaction confirmed (seqno changed to 45), fetching tx hash...
[5/5] TRANSACTION HASH: 65b033a59c84e5843ead14977afe9cdf86a09294fa2e4aa4c3e10ea72c15b718
Executing (default): INSERT INTO "Payments" ("id","senderAddress","amount","sellerWalletAddress","hash","type","isConfirmed","userId","orderId","createdAt","updatedAt") VALUES (DEFAULT,$1,$2,$3,$4,$5,$6,$7,$8,
$9,$10) RETURNING "id","senderAddress","amount","sellerWalletAddress","hash","type","isConfirmed","userId","orderId","createdAt","updatedAt";
Executing (default): UPDATE "Orders" SET "receptionHash"=$1,"transferHash"=$2,"isCompleted"=$3,"updatedAt"=$4 WHERE "id" = $5
Executing (default): UPDATE "Payments" SET "isConfirmed"=$1,"amount"=$2,"updatedAt"=$3 WHERE "id" = $4
Успешно: заказ #14 завершен, средства переведены 9
```

Рис. 38. Transaction2 logs

Демонстрація надходження та змісту (Рис. 39).

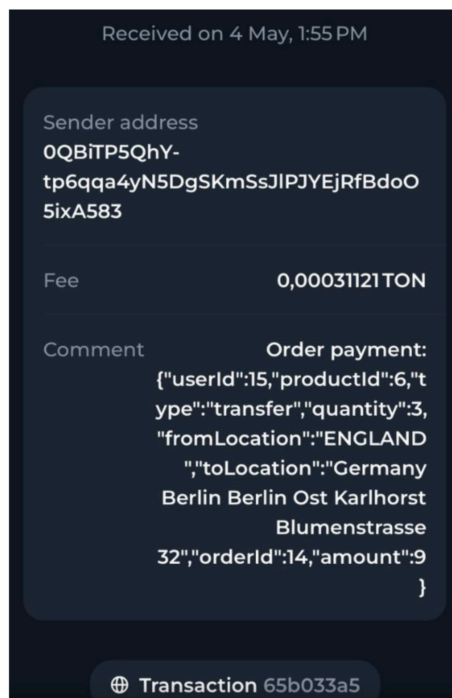

Рис. 39. Transaction proof

4.2 Використання блокчейн технологій у клієнтській частині

Для успішного використання блокчейн технологій у клієнтській частині спочатку треба обернути root компонент у відповідний провайдер(Рис 40).

```
export default function App() {
  return (
    <TonConnectUIProvider manifestUrl="https://aquamarine-odd-fox-362.mypinata.cloud/ipfs/bafkreih2jlfwj7vb6wjb4ebluo3iczquqlcrieuvisqpd7auaxuyqjnore">
      <PersistGate loading={<div>Loading...</div>} persistor={persistor}>
        <Router>
          <Routes>
```

Рис. 40. UI Provider

Провайдеру потрібно надати посилання manifestUrl на manifest.json. manifest.json, це файл з метаданими сервісу, який зберігається у IPFS(децентралізоване сховище).

Далі було створено хук для взаємодії з гаманцем (Рис. 42). Маючі хук, можна створити компонент для його використання, тоді як сам компонент в свою чергу буде використовуватися на сторінках. У разі підключеного гаманця отримаємо та підготуємо для зручного виводу його адресу(Рис. 41):

```
const formatAddress = (address: string) => {
  try {
    return Address.parse(address).toString({
      bounceable: false,
      urlSafe: true,
      testOnly: true
    });
  } catch {
    return address;
  }
};

const formatDisplayAddress = (address: string) => {
  const formatted = formatAddress(address);
  return `${formatted.slice(0, 4)}...${formatted.slice(-4)}`;
};
```

Рис. 41. UI Address

```
const useWallet = () => {
    const [tonWalletAddress, setTonWalletAddress] = useState<string | null>(null);
    const [isLoading, setIsLoading] = useState(true);
    const [tonConnectUI] = useTonConnectUI();

    const handleWalletConnection = useCallback((address: string) => {
        setTonWalletAddress(address);
        console.log("Wallet connected successfully!");
        setIsLoading(false);
    }, []);

    const handleWalletDisconnection = useCallback(() => {
        setTonWalletAddress(null);
        console.log("Wallet disconnected successfully!");
        setIsLoading(false);
    }, []);

    useEffect(() => {
        const checkWalletConnection = async () => {
            setIsLoading(true);
            if (tonConnectUI.account?.address) {
                handleWalletConnection(tonConnectUI.account.address);
            } else {
                handleWalletDisconnection();
            }
        };

        checkWalletConnection();

        const unsubscribe = tonConnectUI.onStatusChange((wallet) => {
            if (wallet) {
                handleWalletConnection(wallet.account.address);
            } else {
                handleWalletDisconnection();
            }
        });

        return () => {
            unsubscribe();
        };
    }, [tonConnectUI, handleWalletConnection, handleWalletDisconnection]);

    return { tonWalletAddress, isLoading, setIsLoading, handleWalletConnection, handleWalletDisconnection };
};
```

Рис. 42. useWallet.ts

Наступним кроком створили хук для ініціалізації транзакції. Алгоритм хуку складається з створення запиту на транзакцію. Вміст транзакції кодується у формат BOC, потім у base64. Далі надсилання транзакції, отримання хешу де ми приймаємо рядок у base64, перетворюємо його на структуру Cell, обчислюємо хеш та переводимо його у зручний формат. Завершемо хук поверненням результату (Рис. 43):

```
      const paymentRequest: SendTransactionRequest = {
        messages: [
          {
            address: params.destination,
            amount: toNano(params.amount).toString(),
            payload: body.toBoc().toString("base64"),
          },
        ],
        network: CHAIN.TESTNET,
        validUntil: Math.floor(Date.now() / 1000) + 360,
      };

      try {
        const transactionRes = await tonConnectUI.sendTransaction(paymentRequest);

        const cell = Cell.fromBase64(transactionRes.boc);
        const hashHex = cell.hash().toString("hex");
        console.log("Transaction hash:", hashHex);

        return {
          status: "success",
          hash: hashHex
        };
      } catch (error) {
        const err = error as Error;
        if (err.message.includes("cancelled")) {
          return { status: "cancelled" };
        }
        console.error("Transaction failed:", err);
        return { status: "failed" };
      }
    },
    [tonConnectUI]
  );
```

Рис. 43. initiateTransaction.ts

4.3 Майбутні перспективи розвитку проекту

У сучасному цифровому середовищі, де прозорість, довіра та швидкість прийняття рішень стають ключовими чинниками успіху бізнесу, проєкти, які базуються на інноваційних технологіях, отримують особливе значення. Особливо це стосується ланцюгів постачання — складної екосистеми взаємозв'язаних учасників, в якій кожен етап транспортування, обліку та оплати має бути не лише зафіксованим, а й перевіреним. У цьому контексті веб-сервіс, заснований на блокчейні, відкриває перед собою широкі горизонти розвитку.

Однією з основних перспектив є масштабування системи на міжнародному рівні. Це передбачає адаптацію сервісу до різних юрисдикцій, валют, мовних і технічних стандартів. Це не лише прискорить логістичні процеси, а й зменшить кількість шахрайських операцій.

Іншим важливим напрямком є використання штучного інтелекту для аналізу великих обсягів даних, які генеруються у блокчейні. У майбутньому, на основі цих даних, система зможе прогнозувати затримки, оптимізувати маршрути, оцінювати ризики для певних товарів чи регіонів. Наприклад, якщо виявляється, що постачання певної групи товарів регулярно затримується, система може автоматично інформувати зацікавлені сторони заздалегідь.

Не менш важливою перспективою є створення системи децентралізованих арбітражів. У разі спорів між учасниками (наприклад, щодо якості товару чи доставки), система зможе передбачити механізм голосування серед довірених арбітрів або використати DAO-моделі (децентралізовані автономні організації) для прийняття рішень. Це дозволить уникнути затяжних непорозумінь і вирішувати конфлікти швидко та прозоро.

Загалом можно підвести, що майбутнє такого проєкту — це не лише його технічне вдосконалення, а й побудова нової культури, в якій довіра та прозорість стають базовими принципами взаємодії. Це шанс перетворити традиційні ланцюги постачання на динамічні, автоматизовані, розумні системи нового покоління.

ВИСНОВКИ

Результатом моєї дипломної роботи стало розроблення сервісу з інтегруванням блокчейн технологій у системи управління ланцюгами постачання. Постановка задачі, пояснення актуальності теми у сьогоденні та аналітика майбутніх перспектив проекту. Проектування необхідної архітектури для створення сервісу. Реалізація серверної частини мовою програмування Javascript, за допомогою фреймворку Express, та СУБД PostgreSQL. Детальне пояснення використаних патернів та шаблонів. Покрокова реалізація з поясненням сутностей, моделей, та контролерів. Демонстрація роботи серверної частини системи та правильної обробки даних. В ході розробки було проведено роботу з патернами програмування. Також з фреймворком Express та створенням необхідних компонентів за його допомогою. Продемонстрована взаємодія СУБД PostgreSQL з додатком, системи з блокчейном, та правильність відображання та отримання інформації.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Node.js v23.11.0 documentation: веб-сайт. URL: https://nodejs.org/docs/latest/api/ (дата звернення 10.03.25)
2. React v19.1 documentation: веб-сайт. URL: https://react.dev/ (дата звернення 11.03.25)
3. TypeScript documentation: веб-сайт. URL: https://www.typescriptlang.org/docs/ (дата звернення 12.03.25)
4. PostgreSQL 17.4 documentation: веб-сайт. URL: https://www.postgresql.org/docs/current/index.html (дата звернення 13.03.25)
5. ORBS documentation: веб-сайт. URL: https://docs.orbs.network/v3 (дата звернення 20.03.25)
6. Tailwind CSS documentation: веб-сайт. URL: https://v2.tailwindcss.com/docs (дата звернення 22.03.25)
7. Tonconnect documentation: веб-сайт. URL: https://docs.ton.org/v3/guidelines/ton-connect/guidelines/developers (дата звернення 25.03.25)
8. Design patterns in node.js: веб-сайт. URL: https://medium.com/@techsuneel99/design-patterns-in-node-js-31211904903e (дата звернення 27.03.25)
9. Jonathan Lee Martin. (2019). Functional design patterns for express.js. (дата звернення 13.04.25)
10. The Twelve-Factor App: веб-сайт. URL: https://12factor.net/ (дата звернення 15.04.25)
11. Kevin Hoffman. (2016). Beyond the Twelve-Factor App (дата звернення 20.04.25)