Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента	*Михайловського Іллі Сергійовича*
	(ПІБ)
академічної групи	*122м-21-1*
	(шифр)
спеціальності	*122 Комп'ютерні науки*
	(код і назва спеціальності)
освітньої програми	*Комп'ютерні науки*
на тему:	*Розробка інформаційної системи обліку енергоресурсів з використанням технології блокчейн*

_____ *І.С. Михайловський*

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**Інституційною**	
розділів кваліфікаційної роботи				
спеціальний	*Проф. Слєсарєв В.В.*			
Рецензент	*Доц. Кожевніков А.В.*			
Нормоконтролер	*Проф. Лактіонов І.С.*			

Дніпро
2022

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп'ютерних систем
(повна назва)

М.О. Алексєєв
(підпис) (прізвище, ініціали)

« » 20 22 Року

ЗАВДАННЯ
на виконання кваліфікаційної роботи магістра

Спеціальності *122 Комп'ютерні науки*
(код і назва спеціальності)

Студенту *122М-21-2* *Михайловському Іллі Сергійовичу*
(група) (прізвище та ініціали)

Тема кваліфікаційної роботи *Розробка інформаційної системи обліку енергоресурсів з використанням технології блокчейн*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 31.10.2022 р. № 1200-с

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт дослідження – процес обліку енергоресурсів на базі блокчейн технологій.

Мета даної роботи полягає в проектуванні і створенні повноцінної автоматизованої системи з обліку та продажу енергоресурсів, яка змогла би запропонувати альтернативу обліку і продажу енергоресурсів на ринку.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Основна задача системи – проведення фінансових операцій над енергоресурсами за новою концепцією, де кожен із користувачів в мережі має безпосередній зв'язок один із одним і здатний комунікувати без посередників, що потенціально може змінити ринок енергоресурсів та в загалом зробить ресурси дешевшими.

Даний підхід вимагає спеціальних алгоритмів, таких як хешування та побудова даних у вигляді ланцюгів, де забезпечується висока стійкість до підробки.

Проектування системи було виконано таким чином, щоб було можливо забезпечити масштабування та простоту у внесенні змін або додаванні нового

функціоналу.

Система являє собою децентралізовану однорангову структуру, де кожен користувач являється вузлом мережі. Сам вузол розроблений за принципом клієнт-серверного підходу. Доступ до бази даних здійснюється засобами Node.js.

4 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – кінець**
Аналіз існуючих рішень та постановка задачі роботи.	12.09.2022-30.09.2022
Проектування і створення повноцінної автоматизованої системи з обліку та продажу енергоресурсів.	01.10.2022-31.10.2022
Розробка програмного забезпечення та дослідження ефективності запропонованих рішень.	01.11.2022-10.12.2022

Завдання видав _____ *Слєсарєв В.В.*
(підпис) (прізвище, ініціали)

Завдання прийняв до виконання _____ *Михайловський І.С.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *10.09.2022 р.*

Термін подання до ЕК *20.12.2022 р.*

РЕФЕРАТ

Пояснювальна записка: 52 стор., 14 рис., 2 таблиці, 2 додатка, 23 джерел.

Об’єкт дослідження – процес обліку енергоресурсів на базі блокчейн технологій.

Мета даної роботи полягає в проектуванні і створенні повноцінної автоматизованої системи з обліку та продажу енергоресурсів, яка змогла би запропонувати альтернативу обліку і продажу енергоресурсів на ринку.

Основна задача системи – проведення фінансових операцій над енергоресурсами за новою концепцією, де кожен із користувачів в мережі має безпосередній зв’язок один із одним і здатний комунікувати без посередників, що потенціально може змінити ринок енергоресурсів та в загалом зробить ресурси дешевшими.

Даний підхід вимагає спеціальних алгоритмів, таких як хешування та побудова даних у вигляді ланцюгів, де забезпечується висока стійкість до підробки.

Проектування системи було виконано таким чином, щоб було можливо забезпечити масштабування та простоту у внесенні змін або додаванні нового функціоналу.

Система являє собою децентралізовану однорангову структуру, де кожен користувач являється вузлом мережі. Сам вузол розроблений за принципом клієнт-серверного підходу. Доступ до бази даних здійснюється засобами Node.js.

СПИСОК КЛЮЧОВИХ СЛІВ: ОДНОРАНГОВА СТРУКТУРА, БЛОКЧЕЙН ТЕХНОЛОГІЇ, ХЕШУВАННЯ ФУНКЦІЙ, АЛГОРИТМ, КЛІЄНТ-СЕРВЕРНИЙ ПІДХІД.

ABSTRACT

Explanatory note: 52 pages, 14 figures, 2 tables, 2 appendices, 23 sources.

The object of research is the process of accounting for energy resources based on blockchain technologies.

The purpose of this work is to design and create a full-fledged automated system for the accounting and sale of energy resources, which would be able to offer an alternative to the accounting and sale of energy resources on the market.

The main task of the system is to conduct financial transactions over energy resources according to a new concept, where each of the users in the network has a direct connection with each other and is able to communicate without intermediaries, which can potentially change the energy market and make resources cheaper in general.

This approach requires special algorithms, such as hashing and building data in the form of chains, which ensures high resistance to forgery.

The design of the system was done in such a way that it was possible to provide scalability and ease of making changes or adding new functionality.

The system is a decentralized peer-to-peer structure where each user is a network node. The node itself is designed according to the principle of a client-server approach. Access to the database is carried out using Node.js.

LIST OF KEYWORDS: PEER STRUCTURE, BLOCKCHAIN TECHNOLOGY, HASHING FUNCTIONS, ALGORITHM, CLIENT-SERVER APPROACH.

ЗМІСТ

ВСТУП

Розвиток IT-технологій щодня пропонує світові нові інструменти для оптимізації бізнес-процесів. Одним з останніх таких інструментів є технологія ― блокчейн (blockchain technology).

Блокчейн ― технологія здатна докорінно змінити енергетичну систему, спочатку шляхом трансформації окремих секторів і, нарешті, шляхом трансформації всього ринку електроенергії.

Основна перевага даної технології ― однорангова взаємодія між користувачами. Іншими словами, користувачі мають можливість здійснювати передачу цінної інформації або грошові перекази один між одним без посередників, що потенціально може зменшити ціну на товари, якими торгують за допомогою блокчейну. Даний підхід вимагає спеціальних алгоритмів та особливої архітектури.

Мета даної роботи полягає в проектуванні і створенні повноцінної автоматизованої системи з обліку та продажу енергоресурсів, яка змогла би запропонувати альтернативу обліку і продажу енергоресурсів на ринку.

При проектуванні системи застосовано технології React.js, Node.js, MongoDB. У першому розділі описано поточний стан на ринку енергоресурсів та можливі способи удосконалення торгівлі енергоресурсами. У другому розділі стисло описано принцип роботи блокчейну. В третьому представлено технології, які були використані при проектуванні та створенні автоматизованої системи. Розділ демонструє програмне рішення з детальним описом технічних деталей та показує взаємодію користувача із створеною системою.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ РОБОТИ

1.1. Аналіз існуючих рішень.

Думки експертів щодо ідеї впровадження криптовалют розділилися: одні вважають це справді новим етапом, але не зовсім зрозуміло, чи буде дане рішення революційним. Другі — це інновації, які потребують значної адаптації. Тому питання подолання багатозначності і практичного використання є актуальними та потребують більшого дослідження.

В сучасних умовах прийнята технологія обліку і контролю енергоресурсів застаріла через організаційну та технічну недосконаліть структур. Ці проблеми стають причиною постійних збитків, що показує про необхідність створення нової автоматизованої системи.

Метою дослідження є аналіз перспектив та варіантів використання блокчейн технологій в енергетиці та розгляд системи обліку споживання енергоресурсів.

Міжнародні енергетичні компанії розробляють проекти, які надалі з'єднають усіх споживачів в одну мережу — децентралізовану систему. Існуюча багаторівнева система складається з виробників електроенергії, операторів розподільної мережі, операторів-постачальників, постачальників платіжних послуг банківських послуг, споживачів та трейдерів. Усі транзакції щодо отримання та оплати за енергію здійснюватимуться в мережі, об'єднуючи учасників — виробників та споживачів енергії. Це зробить енергію дешевшою.

Всі транзакції будуть відкритими. Люди не зможуть прострочити платіж за споживання енергії — буде контролюватися виконання всіх операцій. Система сама заплатить за себе, тобто спише стільки криптовалюти, скільки Вам знадобиться для транзакції по передачі енергії.

Завдяки блокчейну всі дії будуть захищені від сторонніх користувачів. Це дозволить сертифікувати електроенергію, перевірити квоти на допустимі викиди. Ця технологія функціонує як база даних транзакцій, тому за допомогою блокчейну можна створити архів для збереження всіх даних за виставленими рахунками за електроенергію. Споживачі отримають можливість контролю за своїми договорами на постачання електроенергії, а також дані про споживання електроенергії. Усі записи зберігатимуться у відкритому

доступі в блокчейні, який буде коригувати всі питання права власності та поточний стан активів.

Технологія блокчейну, крім того, що використовується для проведення операцій з постачання енергії, може бути основою для процесів вимірювання кількості споживаної електроенергії, формування рахунків за спожиту енергію та подальшу їх оплату. Інші можливі додатки включають право власності на активи, управління активами, систему сертифікатів квоти на викиди вуглекислого газу та сертифікати, що підтверджують виробництво електроенергії на основі використання відновлюваних джерел енергії (ВДЕ). Можливості використання технологій в енергетиці представлені в таблиці 1.1.

Таблиця 1.1. Варіанти використання блокчейну в енергетиці

<table>
<tr><th>Транзакції і «розумні контракти»</th><th>Права власності на активи і управління ними</th><th>Децентралізовані інформаційні системи</th></tr>
<tr><td>Децентралізована торгівля</td><td>Реєстрація власності та ведення реєстру активів</td><td>Облік електроспоживання та виставлення рахунків за електроенергію</td></tr>
<tr><td>Можливості для просьюмерів</td><td>«Зелені» сертифікати</td><td>Облік споживання тепла і виставлення рахунків за нього</td></tr>
<tr><td>Впровадження криптовалют</td><td rowspan="3">Квоти на викиди вуглекислого газу і сертифікація виробництва електроенергії на основі відновлюваних джерел енергії</td><td rowspan="3">Оплата зарядки електромобілів</td></tr>
<tr><td>Зарядка електромобілів</td></tr>
<tr><td>IOT</td></tr>
</table>

Існуючі блокчейн додатки можна розділити на три великі категорії залежно від рівня розробки: додатки версій 1.0, 2.0 та 3.0. Blockchain 3.0 — це етап розвитку технологій, на якому здійснюється подальший розвиток концепції "смарт контракту" з метою створення децентралізованих, автономних організаційних підсистем, які керуються власними законами та працюють майже незалежно. Децентралізована система енергетичних транзакцій та постачання енергії представлена на рисунку 1.1.

Рис. 1.1. Децентралізована система енергетичних транзакцій і енергопостачання

Звернемо увагу на проблему обліку споживання та сбір платежів за енергоресурси. Технологія обліку і контролю енергоресурсів, яка сьогодні застосовується, завдає шкоди суб'єктам господарювання. Головна причина це організаційна і технічна деградація структур. Ці проблеми стають причиною постійних збитків, що свідчить про необхідність створення нової автоматизованої системи.

Нижче наводяться вимоги, які необхідно виконати для створення повноцінної автоматизованої системи:

a. Система має проводити облік та торгівлю енергоресурсів за принципами блокчейну.
b. Система повинна бути децентралізована;
c. Користувач повинен керувати лише власними даними;
d. Користувач має можливість переглядати будь-які дані блокчейну;

е. Користувач може вносити цінну інформацію і проводити оплату за енергоресурси за допомогою криптовалюти.

Крім того, система повинна мати зручний та сучасний графічний інтерфейс користувача з можливістю працювати у веб-браузері та бути імплементована відповідно до шаблонів задля гнучкого внесення змін та додавання нового функціоналу.

1.2. Принципи побудови блокчейн технологій.

Головний принцип блокчейн технології – перманентне обчислення зашифрованої хеш-функції. Хешування – перетворення n-бітового рядку (n – задається) довільного входу за допомогою хеш-функції , наприклад, у 256-бітний хеш-рядок. При зміні вхідного повідомлення (хоч на знак!), результат, що отримується застосуванням хеш-функції – повністю змінюється (забезпечується стійкість криптокоду).

Приклад хеш-функції – діленням входу на поліном по модулю 2:

$$h(n,m) = nmod(m) \qquad (1.1.)$$

де n – вхід («ключ»), m – кількість входів («хешів»), mod – цілочисельне ділення.

Можна хеш-функцію представити набором коефіцієнтів полінома. Для цього ділимо дані (вхід) по модулю 2, m – ступінь 2, бінарні ключі

$$K = K_{n-1}K_{n-2} \dots K_0 \qquad (1.2.)$$

представляють поліномами, а хеш-код – значеннями коефіцієнтів:

$$a_{m-1}, a_{m-2}, \dots, a_0$$

Полінома, отриманого як залишок ділення полінома K(x) на інший, що задається поліном ступеня m:

$$K(x)modP(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \dots + a_1x^1 + a_0 \qquad (1.3.)$$

$$h(x) = a_{m-1}, a_{m-2}, \dots, a_1a_0 \qquad (1.4.)$$

Майнинг блоків базується на криптоалгоритмі SHA-256 (обчислення криптокоду по хеш-функції зі значеннями 256-бітових рядків).

Як висновок, можна зазначити, що дані можливо вносити без участі посередників. Ввесь ланцюг розподілений між комп'ютерами по всьому світу. Центральний сервер, який було б можливо зламати не існує, щоб нанести шкоду системі. Нова технологія дозволяє продавцю і покупцю електроенергії підключившись до мережі, напряму взаємодіяти один із одним через Інтернет, проводячи грошові розрахунки. Традиційні посередники, такі як банки, платіжні системи в даній моделі не потрібні, оскільки всі абоненти мережі виступають свідками транзакцій і можуть підтвердити її деталі.

1.3 Постановка задачі роботи.

Мета даної роботи полягає в проектуванні і створенні повноцінної автоматизованої системи з обліку та продажу енергоресурсів, яка змогла би запропонувати альтернативу обліку і продажу енергоресурсів на ринку.

Для вирішення даного завдання у роботі пропонується вирішити ряд наступних задач:

1) провести аналіз предметної області,
2) обрати технології для розробки,
3) спроектувати моделі даних, що будуть використовуватися в системі,
4) запропонувати архітектурне рішення системи,
5) спроектувати серверну реалізацію,
6) спроектувати графічний інтерфейс користувача та міжсерверну комунікацію.

РОЗДІЛ 2
ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ ОБЛІКУ ТА ПРОДАЖУ ЕНЕРГОРЕСУРСІВ

2.1 Обґрунтування вибору технологій та їх опис

Прототип системи для дослідження можливостей концепції з мінімальними часовими затратами було виконано мовою Java. Реалізація системи була здійснена за допомогою мови програмування Javascript, де серверна частина написана за допомогою фреймворку Node.js, графічний інтерфейс користувача – React.js. Вище зазначені фреймворки дозволяють писати шаблонізований код, який простіше супроводжувати і змінювати у порівнянні із ситуацією без їх використання.

Вказані вище технології базуються на мові програмування Javascript, в них одна екосистема, широкий вибір додаткових бібліотек з простотою їх підключення. Мова Javascript швидка у вивченні і зручна для використання.

Автоматизована система представлена у вигляді веб додатку, взаємодія з яким забезпечена за допомогою веб-браузеру. Припускається, що система буде працювати в мережі Інтернет. Вся взаємодія між користувачем та системою в мережі Інтернет відбувається за допомогою протоколу HTTPS. Передача даних здійснюється на основі архітектурного шаблону REST, де всі операції маніпулювання з даними в рамках Інтернету виконуються за допомогою 4 типів HTTP запитів:

- GET – на отримання інформації;
- POST – на збереження інформації;
- PUT – на оновлення інформації;
- DELETE - на видалення інформації.

Корисна інформація в рамках HTTP запитів передається у форматі JSON.

Реалізована система використовує перші два типи запитів.

Сховищем даних було вирішено обрати NoSQL (нереляційну) базу даних MongoDB. Як відомо, нереляційні бази не зберігають логічної цілісності даних, тому для роботи з ними необхідно забезпечувати додаткові застережні методи валідації та контролю збереженої інформації. Цей факт також унеможливлює наявність транзакцій в рамках терміну SQL, але нереляційні БД не мають механізмів забезпечення логічної цілісності і тому їх швидкодія більше, вони більш гнучкі в

проектах з важко формалізованими вимогами, прості у масштабуванні (як у горизонтальному, так і у вертикальному), що важливо для децентралізованої блокчейн-системи.

2.2 Графічний інтерфейс користувача

Наразі найбільшого використання отримали односторінкові веб-застосунки (SPA). Веб-застосунок - тип сайту, який вміщується на одній сторінці з метою забезпечити користувачу досвід близький до користування настільною програмою. Для Спрощення їх побудови використовуються бібліотеки та фреймворки.

Одним з найпопулярніших інструментів, що використовується для побудови веб-застосунків даного типу є React.js.

React.js - це бібліотека для створення користувацьких інтерфейсів. Її головне завдання - спростити і автоматизувати написання веб UI інтерфейсів.

React значно полегшує створення інтерфейсів завдяки ефективному підходу розбиття кожної сторінки на невеликі фрагменти. Фрагменти називаються компонентами. Кожен виділений

Рис. 2.1. Приклад розбивки сторінки на компоненти

Компонент React – це ділянка коду, який представляє частину веб-сторінки. Кожен компонент – це JavaScript функція, яка повертає частину коду, що представляє фрагмент сторінки [7].

Для формування сторінки ці функції викликаються в певному порядку. Після цього зібраний воєдино з компонентів результат відображається користувачеві.

React розроблений навколо концепції багаторазових компонентів.

Будуються невеликі за розміром та логікою невеликі компоненти, які відповідають та описують одну функцію. Далі вони поєднуються, щоб сформувати більші компоненти. Всі компоненти, маленькі чи великі, можуть використовуватися повторно, навіть у різних проектах.

Компоненти бувають 2 типів:

1. Презентаційні (dummy) – відповідають за відображення контенту.
2. Контейнери (smart) – відповідають за надання логіки і даних для презентації компонентів.

```
const App = ({ location }) => (
    <div className="ui-container">
        <Route path="/" exact component={LoginPage} />
        <Route location={location} exact path="/login" component={LoginPage} />
        <Route location={location} exact path="/dashboard" component={Dashboard} />
    </div>
)
```

Рис. 2.2. Приклад компонента

React використовує мову розмітки, так званий JSX, який схожий на мову розмітки HTML, але працює всередині мови JavaScript, що відрізняє його від HTML.

Проте в браузері використовується скомпільовану версію. JSX - це технологія, яка дозволяє нам розробляти компоненти React використовуючи HTML-подібний синтаксис (рисунок 2.3).

```
render() {
    return (
        <div className="login-page">
            <Grid centered>
                <Grid.Row>
                    <Grid.Column mobile={14} tablet={10} widescreen={6} largeScreen={6}>
                        <h1 className="login-h1">Login Page</h1>
                        <LoginForm submit={this.submit} />
                    </Grid.Column>
                </Grid.Row>
            </Grid>
        </div>
    );
}
```

Рис. 2.3. Приклад розмітки

Компоненти React можна поміщати в інші компоненти. Саме так сторінки збирають з фрагментів, написаних на React - вкладаючи компоненти один в одного. В компонент LoginPage поміщений компонент LoginForm (рисунок 2.4).

Рис. 2.4. LoginPage компонент

Стан – це інструмент, що дозволяє оновлювати призначений для користувача інтерфейс, ґрунтуючись на події [7]. У кожного компоненту є власний стан. У стані зберігається внутрішні дані компонента, при зміні яких відбувається перемалювання компонента (і вкладених в нього компонентів, якщо вони є) (рисунок 2.5).

```
state = {
    table: [],
    user: {
        id: localStorage.getItem("userToken")
    },
    loading: false,
    isLeftMenuActive: false,
    activeTable: ""
}
```

Рис. 2.5. Приклад стану компонента

Коли користувач клацає на кнопку login, відбувається оновлення стану компонента і з цієї причини запит на сервер буде відправлений.

Компоненти можуть «спілкуватися» один з одним. Можлива передача даних від одного компонента до іншого через властивості (props).

Властивості – це інформація, колективно використовувана батьківським компонентом і компонентами-нащадками (рисунок 2.6).

Рис. 2.6. Path, exact, component - приклад властивостей компонентів.

Таким чином можна реалізувати інкапсуляцію даних. Вкладений компонент отримує тільки необхідні йому дані для роботи через властивості. Це дозволяє створювати слабозв’язні компоненти і перевикористовувати їх кілька разів у проекті за необхідністю.

2.3 Серверна частина

Node.js призначений для створення масштабованих мережевих додатків як асинхронна машина виконання JavaScript. Після кожного з'єднання зворотний виклик буде знято, але якщо роботи не буде виконано то Node.js перейде у режим сну.

Це контрастує на відміну від більш поширеної моделі сучасності, в якій використовуються потоки ОС. Мережа на основі потоків порівняно неефективна і дуже складна у використанні. Крім того, користувачі Node.js звільняються від турботи про “dead locking” процесу, оскільки блокувань взагалі немає. Майже жодна функція в Node.js безпосередньо не виконує введення-виведення, тому процес ніколи не блокується. Оскільки нічого не блокує, масштабовані системи дуже резонно розробляти з використанням Node.js.

Node.js схожий за дизайном та під впливом таких систем, як Ruby's Event Machine і Python's Twisted. Node.js розширює поняття моделі подій. Він представляє цикл подій як конструкцію виконання, а не як бібліотеку. В інших системах завжди є виклик блокування для запуску циклу подій. Зазвичай поведінка визначається через зворотні виклики на початку сценарію, а в кінці сервер запускається через блокуючий виклик. У Node.js немає такого виклику циклу запуску події. Node.js просто входить у цикл подій після виконання сценарію введення і виходить з циклу подій, коли більше немає зворотних викликів для виконання. Така поведінка схожа на JavaScript браузера — цикл

подій прихований від користувача.

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Рис. 2.7. Приклад створення серверу

HTTP – це модуль у Node.js, розроблений з урахуванням потокової та низької затримки. Це робить Node.js добре підходящим для створення веб-бібліотеки чи фреймворку. Node.js, розроблений без потоків, це означає, що не можливо скористатися кількома ядрами в оточенні. Дочірні процеси можна породжувати за допомогою API child_process.fork і вони розроблені так, щоб було легко комунікувати. На цьому ж інтерфейсі побудований модуль кластера, який дозволяє обмінюватися сокетами між процесами, щоб забезпечити балансування навантаження над вашими ядрами.

Блокування – це коли виконання додаткового JavaScript коду у процесі Node.js має зачекати, поки не завершиться операція, що не стосується JavaScript. Це трапляється тому, що цикл подій не в змозі продовжувати працювати під час операції блокування.

У Node.js операції, які демонструють низьку продуктивність через інтенсивність процесора, а не очікування операції, наприклад, вводу/виводу, зазвичай не відноситься до блокуючих. Синхронні методи в стандартній бібліотеці Node.js, які використовують libuv, є найбільш часто використовуваними операціями блокування.

Усі методи вводу-виводу в стандартній бібліотеці Node.js забезпечують асинхронні версії, які не блокують, і приймають функції зворотного виклику (рисунки 2.8 – 2.9).

```
const fs = require('fs');
const data = fs.readFileSync('/file.md'); // blocks here until file is read
```

Рис. 2.8. Приклад читання файлу

```
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
});
```

Рис. 2.9. Приклад читання файлу із зворотним викликом

Будь-який веб додаток рано чи пізно повинен створити об’єкт веб-сервера. Це

Можливо зробити, використовуючи метод http.createServer(). Функція, яка передається в createServer викликається для кожного HTTP запиту який надходить до серверу, тому її називають обробником запиту. Насправді, об’єкт Server, що повертає createServer є емітером події.

Аналогічно NodeJS має зручні засоби для обробки заголовків запитів, робота з помилками. Код для серверу, який приймає HTTP запит, оброблює тіло запиту та обробляє помилки, наведено нижче:

Отже внаслідок того, що технологія Node.js як відносно простий інструмент для розробки системи до якої вимагається наявність потенціалу з масштабування, готовність до внесення архітектурних змін із найменшими втратами часу найкраще підходить до поставленої задачі.

```
const http = require('http');

http.createServer((request, response) => {
  const { headers, method, url } = request;
  let body = [];
  request.on('error', (err) => {
    console.error(err);
  }).on('data', (chunk) => {
    body.push(chunk);
  }).on('end', () => {
    body = Buffer.concat(body).toString();
    // BEGINNING OF NEW STUFF

    response.on('error', (err) => {
      console.error(err);
    });

    response.statusCode = 200;
    response.setHeader('Content-Type', 'application/json');
    // Note: the 2 lines above could be replaced with this next one:
    // response.writeHead(200, {'Content-Type': 'application/json'})

    const responseBody = { headers, method, url, body };

    response.write(JSON.stringify(responseBody));
    response.end();
    // Note: the 2 lines above could be replaced with this next one:
    // response.end(JSON.stringify(responseBody))

    // END OF NEW STUFF
  });
}).listen(8080);
```

Рис. 2.10. Приклад створення серверу для оброки HTTP запиту

2.4 Документоорієнтована система управління базами даних з відкритим вихідним кодом MongoDB.

MongoDB – це база даних загального призначення, заснована на документах, створена для сучасного застосування.

MongoDB реалізує новий підхід до побудови базових даних, де немає таблиць, схем, запитів SQL, наявних ключів і багатьох інших речей, які наявні в об'єктах даних, що мають абсолютно реальне значення.

Відмінність від реляційних даних бази MongoDB пропонує документоорієнтовану модель даних, завдяки чому MongoDB в багатьох випадках працює швидше, маючи кращу спроможність до масштабування.

Але навіть зважаючи на всі недоліки реляційних баз даних і переваги MongoDB,

важливо врахувати, що задачі бувають різні, як і методи їх рішення. У якомусь випадку ситуація MongoDB дійсно спрощує розробку, наприклад, якщо потрібно зберігати складні дані за структурою. У іншій ситуації найкраще використовувати традиційні реляційні бази даних. Крім того, можна використовувати смішний підхід: зберігати один тип даних у MongoDB, а інший тип даних - у традиційних БД.

Вся система MongoDB може представляти не тільки одну базу даних, що знаходиться на одному фізичному сервері. Функціональність MongoDB дозволяє розташувати кілька баз даних на декількох фізичних серверах, і ці бази даних зможуть легко обмінюватися даними і зберігати цілісність.

Одним з популярних стандартів обміну даними та їх зберігання є JSON (JavaScript Object Notation). JSON ефективно описує складні за структурою дані. Спосіб зберігання даних в MongoDB в цьому плані схожий на JSON, хоча формально JSON не використовується. Для зберігання в MongoDB застосовується формат, який називається BSON (Бісон) або скорочення від binary JSON.

MongoDB написана на C++, тому її легко перенести на найрізноманітніші платформи. MongoDB може бути розгорнута на платформах Windows, Linux, MacOS, Solaris. Можна також завантажити вихідний код і самому скомпілювати MongoDB.

Однак при всіх відмінностях є одна особливість, яка зближує MongoDB і реляційні бази даних. У реляційних СУБД зустрічається таке поняття як первинний ключ. Це поняття описує якийсь стовпець, який має унікальні значення. У MongoDB для кожного документа є унікальний ідентифікатор, який називається _id. І якщо явно не вказати його значення, то MongoDB автоматично згенерує для нього значення.

Якщо в традиційному світі SQL є таблиці, то в світі MongoDB є колекції. І якщо в реляційних БД таблиці зберігають однотипні жорстко структуровані об'єкти, то в колекції можуть містити найрізноманітніші об'єкти, що мають різну структуру і різний набір властивостей.

Система зберігання даних в MongoDB представляє набір реплік. У цьому наборі є основний вузол, а також може бути набір вторинних вузлів. Всі вторинні вузли зберігають цілісність і автоматично оновлюються разом з оновленням головного вузла. І якщо основний вузол з якихось причин виходить з ладу, то один з вторинних вузлів стає головним.

На відміну від реляційних СУБД MongoDB дозволяє зберігати різні документи з різним набором даних, однак при цьому розмір документа обмежується 16 мб. Але MongoDB пропонує рішення – спеціальну технологію GridFS, яка дозволяє зберігати дані за розміром більше, ніж 16 мб.

Документ можна уявити як об'єкт, який зберігає деяку інформацію. У певному сенсі він подібний до рядкам в реляційних СУБД, де рядки зберігають інформацію про окремий елемент. Наприклад, типовий документ:

Документ являє набір пар ключ-значення. Ключі представляють рядки. Значення ж можуть відрізнятися за типом даних. В даному випадку у нас майже всі значення також представляють строковий тип, і лише один ключ (company) посилається на окремий об'єкт. Всього є наступні типи значень:

1) string: строковий тип даних, як в наведеному вище прикладі (для рядків використовується кодування UTF-8);
2) array (масив): тип даних для зберігання масивів елементів;
3) binary data (двійкові дані): тип для зберігання даних в бінарному форматі;
4) boolean: булевий тип даних, який зберігає логічні значення TRUE або FALSE;
5) date: зберігає дату в форматі часу Unix;
6) double: числовий тип даних для зберігання чисел з плаваючою точкою;
7) integer: використовується для зберігання цілочисельних значень;
8) javaScript: тип даних для зберігання коду javascript;
9) min key / max key: використовуються для порівняння значень з найменшим / найбільшим елементів BSON;
10) null: тип даних для зберігання значення Null;
11) object: строковий тип даних, як в наведеному вище прикладі;
12) objectID: тип даних для зберігання id документа;
13) regular expression: застосовується для зберігання регулярних виразів;
14) symbol: тип даних, ідентичний строковому. Використовується переважно для тих мов, в яких є спеціальні символи;
15) timestamp: застосовується для зберігання часу.

Для кожного документа в MongoDB визначено унікальний ідентифікатор, який називається _id. При додаванні документа в колекцію, даний ідентифікатор

створюється автоматично. Однак, розробник може сам явно задати ідентифікатор, а не покладатися на автоматично генеруються, вказавши відповідний ключ і його значення в документі.

2.5. Опис архітектурного рішення

Система представлена у вигляді децентралізованої мережі серверів для кожного користувача в залежності від відведеної ролі та одного завантажуючого сервера (bootstrapper server) який виконує синхронізацію користувачів між собою. Завантажуючий сервер містить колекцію користувачів.

Для того, щоб почати роботу із системою, користувачеві необхідно запустити у себе на локальній машині сервер, який приймає запити і виконує обробку; та сервер, який надає графічний інтерфейс користувача.

У мережі наявно 2 ролі користувачів: користувач, що виконує майнинг блоків; користувач, що виконує створення транзакцій для оплати рахунків.

Для отримання даних про інших користувачів мережі, між користувачем та завантажуючим сервером здійснюється взаємодія. Аналогічна робота між майнером та завантажуючим сервером.

Робота між майнером та користувачем полягає у тому, що майнер отримує транзакції від користувача і після здійснення майнингу блоку здійснює поширення всім вузлам нового блоку; якщо блок пройшов валідацію і не був раніше доданий до блокчейну – він додається, майнер отримує грошову винагороду за майнинг.

Робота між користувачем та майнером полягає у тому, що користувач створює транзакцію і поширює її всім майнерам.

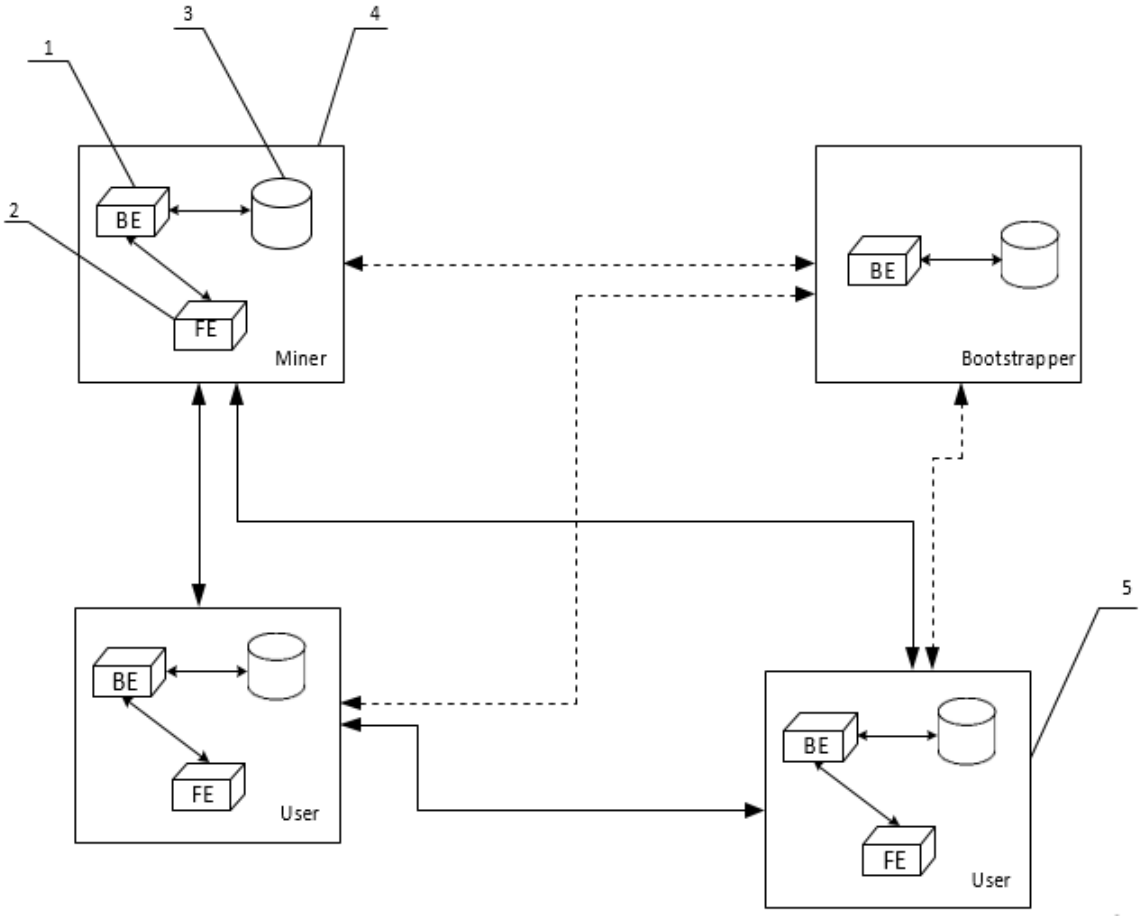

Рис. 2.11. Приклад мережі з двома користувачами та одним майнером

Рис. 2.12. Діаграма прецедентів системи, що проектується

Рисунок 2.13 показує структуру проекту серверу користувача, а рисунки 2.14 – 2.16 показують колекції в БД MongoDB:

Рис. 2.13. Структура серверу користувача

Так як була обрана база даних MongoDB, система має 3 основних колекції: db.blockchain – колекція, що зберігає блокчейн; db.nodes – колекція, що зберігає інформацію про сервери, які працюють в мережі; db.users – колекція, що зберігає особисту інформацію про користувачів.

Структура db.blockchain:

- _id: унікальне-хеш значення яке ідентифікує запис;

- hash: текстове хеш-значення блоку;
- date: дата створення блоку;
- nonce: значення лічильника при якому майнинг блоку був завершений;
- transactions: список транзакцій у вигляді масиву;
- minerReward: числове значення, вказує яку нагороду отримає майнер після успішного завершення майнингу блоку.

Структура transaction:

- _id: унікальне-хеш значення яке ідентифікує запис;
- from: публічний ключ особи, що відправляє транзакцію;
- to: публічний ключ особи, що отримує транзакцію;
- amount: числове значення суми, що передається (у криптовалюті);
- description: текстовий опис транзакції, заповнюється користувачем.

Рис. 2.14. Колекція db.blockchain

Структура db.nodes:

- _id: унікальне-хеш значення яке ідентифікує запис;
- email: текстове значення, адреса електронної пошти користувача;
- firstName: текстове значення, ім’я користувача;

- lastName: текстове значення, прізвище користувача;
- phoneNumber: текстове значення, номер телефону користувача;
- registrationDate: текстове значення, дата реєстрації користувача;
- passwordHash: текстове значення, пароль користувача у вигляді хеш-значення;
- publicKey: текстове значення, публічний ключ користувача, іншими словами – адреса гаманця користувача
- privateKey: текстове значення, приватний ключ користувача за допомогою якого здійснюється підпис транзакцій.

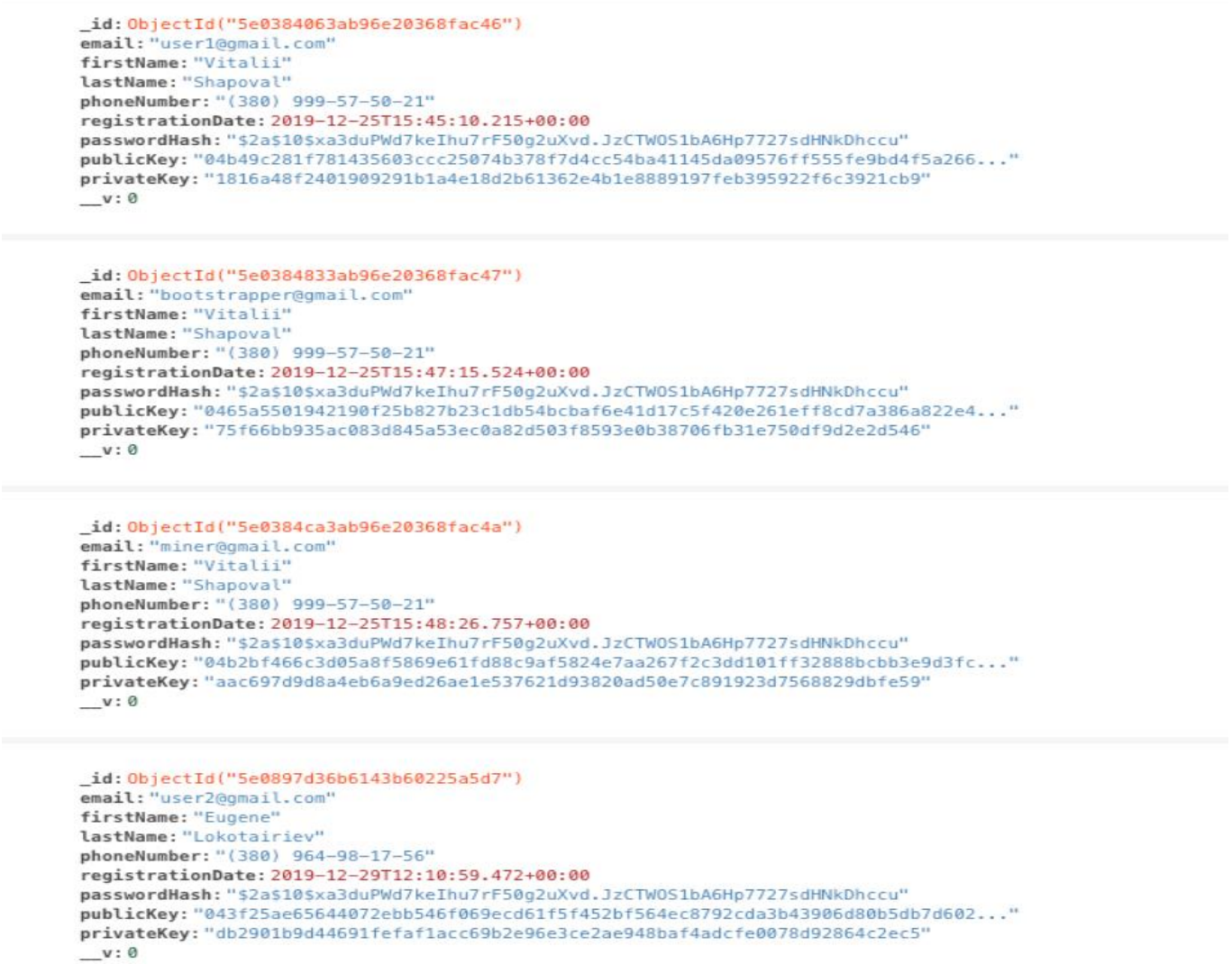

Рис. 2.15. Колекція db.nodes

Структура db.nodes:

- _id: унікальне-хеш значення яке ідентифікує запис;
- walletAdress: текстове значення, адреса електронної пошти користувача;

- __v: числове значення, поточний грошовий баланс користувача;
- ip: текстове значення, IP-адреса, за якою працює сервер в мережі;
- port: текстове значення, номер порту, на якому працює сервер;
- role: текстове значення, описує які задачі доступні для користувача;

Рис. 2.16. Колекція db.users

РОЗДІЛ 3
РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ЙОГО ЗАСТОСУВАННЯ

3.1. Механізм авторизації та аутентифікації користувачів за допомогою JSON Web tokens

Генерація токена:

Відправлення POST запитів /signup, POST /login. При подальших запитах якщо сервер не знайде токену, то видасть у відповіді помилку 401 з описом проблеми.

Процес перевірка токена:

Здійснюється перевірка заголовку Autorization на наявність тексту. Якщо він відсутній – відправляється відповідне повідомлення. В іншому випадку проводиться валідація токена і пошук користувача; якщо процес успішний – HTTP запит виконується, інакше забороняється.

3.2. Алгоритм валідації нового блоку

Для визнання блоку валідним, необхідно виконання наступних умов:

- Попереднє хеш-значення в новому блоці повинно бути таким же самим як і хеш-значення попереднього блоку;
- Хеш-значення що вказане в новому блоці повинно бути так же саме, якщо його перерахувати хеш-значенням враховуючи поля:

```
- previous hash (хеш-значення попереднього блоку)
- transactions (массив транзакцій блоку)
- nonce (число, яке було отримано під час майнингу блоку)
- date (дата створення блоку)
```

При умові виконання цих двох умов новий блок вважається валідним, його ніхто не підробив і його можна зберігати в блокчейн.

3.3. Опис завантажуючого (bootstrapper) сервера

Спочатку сервер встановлює зв’язок з БД:

```
const connection = await mongoose.connect(process.env.MONGODB_URL,
{
  useNewUrlParser: true,
  useFindAndModify: false,
  useUnifiedTopology: true,
});
```

Якщо підключення не буде успішним – подальша робота зупиняється.

Далі йде процес налаштування express server. Здійснюється процес налаштування роботи з HTTP, CORS.

Після цього здійснюється запис в пул користувачів інформації про себе:

```
- ip,
- port
- role (user | boostrapper | miner)
- email
- wallet address (публічний ключ)
```

На останньому етапі налаштовується завантажувач блокчейну (blockchain-loader). Здійснюється перевірка первинного блоку (genesis block) в блокчейні. Якщо його немає (це означає, що блокчейн пустий) – створюється первинний блок.

Після завантаження, сервер може приймати наступні HTTP запити:

GET /users – отримання всіх вузлів. Не приймає вхідних даних. Результатом запиту буде список всіх користувачів з полями:

```
- ip,
- port
- role
- email
- wallet address (публічний ключ)
```

POST /user – додавання нового вузла. Даний запит очікує на вході наступні поля:

```
- ip,
- port,
- wallet address,
- role,
- email
```

Унікальність користувачів визначається за адресою електронної пошти. Якщо користувач з такою поштою існує, система видає відповіддю відповідне повідомлення.

GET /blockchain. Не приймає вхідних даних. Результатом запиту буде отримання всього блокчейну. Дана кінцева точка (endpoint) необхідний у випадку, коли новий користувач долучається до мережі і необхідно провести перше

завантаження блокчейну.

3.4.Опис сервера майнера

Сервер встановлює зв’язок з БД.

Якщо підключення не буде успішним – подальша робота зупиняється. Налаштування express server. Здійснюється процес налаштування роботи з HTTP, CORS.

Здійснюється запис в пул користувачів інформації про себе:

Відправлення запиту на отримання всіх вузлів мережі до завантажуючого сервера.

Після отримання списку вузлів, сервер зберігає в себе локальну копію в колекцію nodes. При подальшому оновленні інформації про користувачів, сервер при новому запуску здійснить новий аналогічний запит.

Оновлення блокчейну. Здійснюється запит до завантажуючого серверу на отримання блокчейну, валідації і подальшого збереження в локальній базі даних у випадку якщо валідація успішна.

POST /transaction На вході приймає інформацію про нову транзакцію. Додає транзакцію в чергу транзакцій, які очікують майнингу. Після завершення майнингу черга очищається.

POST /new-block. Приймає на вході інформацію відносно нового блоку. Якщо блок валідний, то сервер додає його в колекцію blockchain.

Окремо слід зазначити, що на даному сервері наявний механізм здійснення майнингу у вигляді утилітного методу:

```
minePendingTransactions(
  DIFFICULTY - складність майнингу
  transactions - черга транзакцій
  lastBlockHash - хеш попереднього блоку
  date - дата майнингу блоку
)
```

В основі майнингу закладена концепція доведення виконаної роботи (proof of work).

3.5.Опис сервера користувача

Сервер встановлює зв’язок з БД

Якщо підключення не буде успішним – подальша робота зупиняється.

Налаштування express server. Здійснюється процес налаштування роботи з HTTP, CORS.

Здійснюється запис в пул користувачів інформації про себе.

Відправлення запиту на отримання всіх вузлів мережі до завантажуючого сервера

Після отримання списку вузлів, сервер зберігає в себе локальну копію в колекцію nodes. При подальшому оновленні інформації про користувачів, сервер при новому запуску здійснить новий аналогічний запит.

Оновлення блокчейну. Здійснюється запит до завантажуючого серверу на отримання блокчейну, валідації і подальшого збереження в локальній базі даних у випадку якщо валідація успішна.

Після завантаження, сервер може приймати наступні запити:

POST /signup. На вході приймаються поля як email, password, first name, last name, phone number для реєстрації нового користувача. У випадку успішної реєстрації користувача генерується JSON Web Token.

POST /login. На вході приймаються як email, password здійснення процесу аутентифікації та авторизації. У випадку успішної реєстрації створюється персональний токен який перенаправляє користувача до власного кабінету та надає можливість виконувати HTTP запити.

GET /transactions. На вході – інформація, за якою можна ідентифікувати користувача.

З початку виконується запит до колекції blockchain, знаходяться всі транзакції, у яких поля from, to співпадають з публічним ключем користувача;

По знайденим транзакціям виконується запит для отримання інформації по користувачам за електронною адресою та публічним ключем;

Знайдені транзакції відправляються користувачу у наступній формі:

```
{
  data: {
    from - публічний ключ відправника
    to - публічний ключ отримувача
    amount - кількість монет
    description - описання транзакції (опціонально)
  }
  from: {
    email: email відправника
  },
  to: {
    email: email отримувача
  }
}
```

POST /new-block. На вході – інформація про новий блок. Після валідації блока, він вноситься в локальну БД користувача.

POST /transaction/create – сервер отримує дані від користувача на створення транзакції:

```
{
  to - кому
  amount - кількість відправленої криптовалюти
  description - опис транзакції
}
```

Додатково із локальної БД здійснюється процес отримання публічного та приватного ключа.

Далі розраховується хеш транзакції. На основі хешу транзакції та приватного ключа здійснюється цифровий підпис транзакції і транзакція поширюється всім майнерам у вигляді:

```
{
  data: {
    from,
    to,
    amount,
    description,
  },
  signature,
  hash,
}
```

Для роботи необхідно апаратне забезпечення, що задовольняє вимогам: процессор Intel Core i3, 8 Гб оперативної пам'яті, 256 ГБ вільного місця на жорсткому диску.

Додатково необхідно встановити програмне забезпечення: MongoDB, NPM, Node.js збережння блокчейну на локальній машині та для запуску серверів.

3.6. **Результати тестування системи**

Після того як користувач має все необхідне програмне забезпечення, йому необхідно перейти на голову сторінку и одну з дій: вхід в систему, або реєстрація нового користувача. Сторінка входу систему показано на рисунку 3.1, де необхідно ввести електронну пошту и пароль.

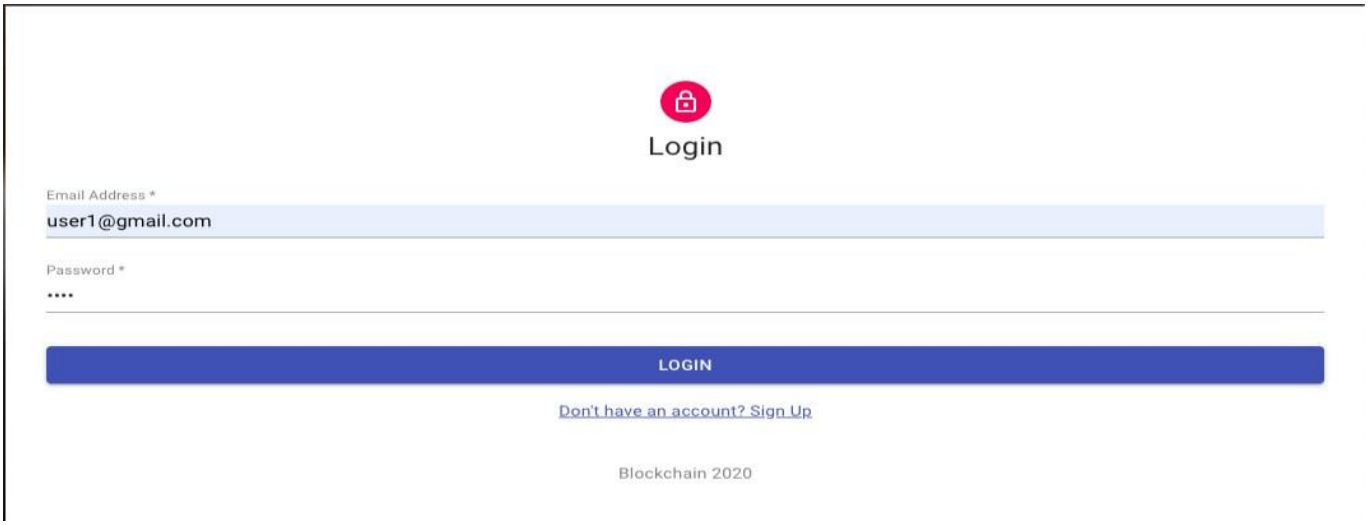

Рис. 3.1. Сторінка аутентифікації та авторизації

Для реєстрації нового користувача необхідно перейти на відповідну сторінку, ввести електронну адресу, прізвища, ім’я, номер телефону та пароль (рисунок 3.2). Після чого натиснути кнопку «Sign in».

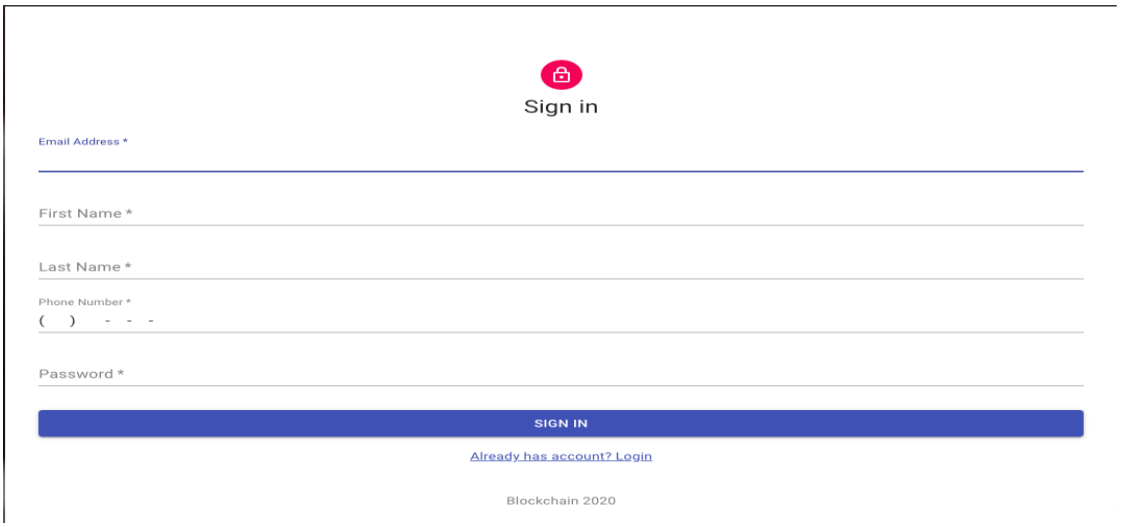

Рис. 3.2. Сторінка реєстрації

Після того, як користувач увійшов в систему, він може створити транзакцію на переведення криптовалюти на спеціальній для цього сторінці (рисунок 3.3), де користувачу необхідно ввести адресу електронного гаманця отримувача, кількість валюти, яку необхідно передати та опис транзакції, в якому можна вказати для чого передаються гроші.

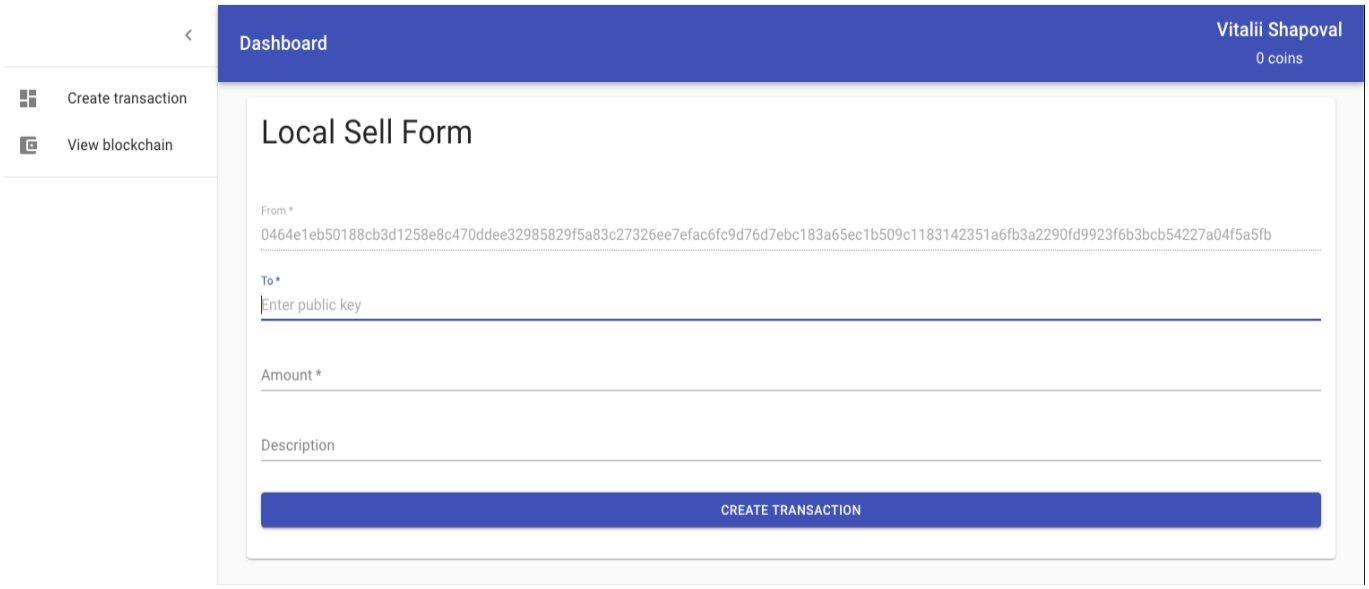

Рис. 3.3. Сторінка створення транзакції

Після того, як користувач створив транзакцію, вона була внесена в блок і був успішно проведений майнинг, криптовалюта буде переведена, а користувач зможе побачити виконану транзакцію (рисунок 3.4).

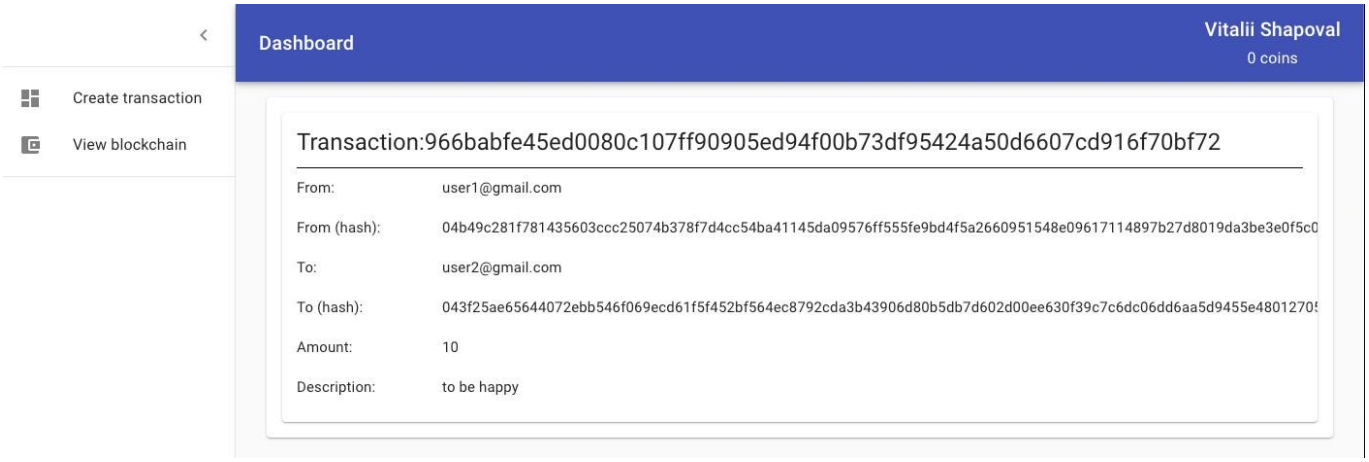

Рис. 3.4. Сторінка перегляду транзакцій

Система являє собою децентралізовану однорангову структуру, де кожен користувач являється вузлом мережі. Сам вузол розроблений за принципом клієнт-серверного підходу. Доступ до бази даних здійснюється засобами Node.js.

Після запуску серверів необхідно в браузері перейти на адресу локального хосту із вказанням порту та зареєструватися як новий користувач, або увійти як вже існуючий. Після успішної авторизації, користувач має можливість здійснювати облік та продаж енергоресурсів.

Компоненти необхідні для установки системи: NPM, Node.js, React.js.

Використана мова програмування системи – Javascript.

Розроблена автоматизована система працює в операційних системах Windows7, Windows8, Windows10 та потребує встановлення компонентів: NPM, Node.js, MongoDB.

Для запуску необхідно викликати shell команди і дочекатися запуску серверів. Після цього необхідно в браузері перейти за адресою локального хосту із вказанням порту на головну сторінку з подальшою реєстрацією нового або авторизацією вже існуючого користувача. Після успішної авторизації користувач може працювати із автоматизованою системою.

Вхідні дані можуть бути текстового або цифрового значення в залежності від призначення форми. Вхідні дані вводяться та зчитуються засобами React.js, логіка обробки даних здійснюється за допомогою мови Javascript.

Аналогічно, вихідні дані представлені у вигляді текстового або цифрового значення у форматі JSON та збережені в базах даних користувачів.

Вихідні дані отримуються з бази даних і представляються користувачам у текстовому або цифровому вигляді.

Висновки

У ході виконання роботи був проведений аналіз предметної області, аналіз вимог та проектування архітектури системи і як наслідок було створено програмний продукт, який дозволяє організувати однорангову взаємодію між користувачами за допомогою технології блокчейн.

Основна задача системи – проведення фінансових операцій над енергоресурсами за новою концепцією, де кожен із користувачів в мережі має безпосередній зв'язок один із одним і здатний комунікувати без посередників, що потенціально може змінити ринок енергоресурсів та в загалом зробить ресурси дешевшими.

Даний підхід вимагає спеціальних алгоритмів, таких як хешування та побудова даних у вигляді ланцюгів, де забезпечується висока стійкість до підробки.

Проектування системи було виконано таким чином, щоб було можливо забезпечити масштабування та простоту у внесенні змін або додаванні нового функціоналу.

Система являє собою децентралізовану однорангову структуру, де кожен користувач являється вузлом мережі. Сам вузол розроблений за принципом клієнт-серверного підходу. Доступ до бази даних здійснюється засобами Node.js. автоматизованою системою.

Вхідні дані можуть бути текстового або цифрового значення в залежності від призначення форми. Вхідні дані вводяться та зчитуються засобами React.js, логіка обробки даних здійснюється за допомогою мови Javascript.

Аналогічно, вихідні дані представлені у вигляді текстового або цифрового значення у форматі JSON та збережені в базах даних користувачів.

Вихідні дані отримуються з бази даних і представляються користувачам у текстовому або цифровому вигляді.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Сегеда І.В., Локотарев Є.О., Шаповал В.О. Реалізація використання блокчейн-технологій у енергетичному секторі. / Вчені записки Таврійського національного університету імені В.І. Вернадського Серія:Економіка і управління Том 30 (69). № 4, 2019, С. 160-165 (DOI: https://doi.org/10.32838/2523-4803/69-4-51) .
2. Segeda I. Blockchain as a digital economy promotion tool in energy industry. Modern Aspects of Software Development: Proceedings of VI International Scientific and Practical Virtual Conference of Software Development Specialists, June, 24 2019 p. – Kyiv: Igor Sikorsky KPI, 2019. – p. 139-146 .
3. Nakamoto S. A Peer-to-Peer Electronic Cash System [Електронний ресурс] // Bitcoin. – Режим доступ до ресурсу: https://bitcoin.org/bitcoin.pdf .
4. How Blockchain Technology Works. Guide for Beginners [Електронний ресурс] / – Режим доступу до ресурсу: https://cointelegraph.com/bitcoin-for-beginners/how-blockchain-technology-works-guide-for-beginners#distributed-database
5. Блокчейн: как он работает, и почему эта технология изменит мир [Електронний ресурс] /– Режим доступу до ресурсу: https://habr.com/ru/company/iticapital/blog/340992/
6. Технічна документація React.js [Електронний ресурс] / – Режим доступу до ресурсу: https://devdocs.io/react/
7. Технічна документація npm [Електронний ресурс] / – Режим доступу до ресурсу: https://docs.npmjs.com/
8. Технічна документація Node.js [Електронний ресурс] / – Режим доступу до ресурсу: https://nodejs.org/dist/latest-v12.x/docs/api/
9. https://science.house.gov/imo/media/doc/Schmidt%20Testimony%20Att achment.pdf
10. https://keenlab.tencent.com/en/whitepapers/Experimental_Security_Rese arch_of_Tesla_Autopilot.pdf
11. https://www.science.org/doi/10.1126/science.aaw4399
12. https://arxiv.org/pdf/1707.08945.pdf
13. https://arxiv.org/abs/1812.00151

14. C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, “Intriguing properties of neural networks,” arXiv preprint arXiv:1312.6199, 2013.
15. I. J. Goodfellow, J. Shlens, and C. Szegedy, “Explaining and harnessing adversarial examples (2014),” arXiv preprint arXiv:1412.6572.
16. X. Yuan, P. He, Q. Zhu, R. R. Bhat, and X. Li, “Adversarial examples: Attacks and defenses for deep learning,” arXiv preprint arXiv:1712.07107, 2017.
17. A. Kurakin, I. Goodfellow, and S. Bengio, “Adversarial examples in the physical world,” arXiv preprint arXiv:1607.02533, 2016.
18. S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, “Deepfool: a simple and accurate method to fool deep neural networks,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2574– 2582, 2016.
19. U. Jang, X. Wu, and S. Jha, “Objective metrics and gradient descent algorithms for adversarial examples in machine learning,” in Proceedings of the 33rd Annual Computer Security Applications Conference, pp. 262– 277, 2017.
20. N. Carlini and D. Wagner, “Towards evaluating the robustness of neural networks,” in 2017 IEEE Symposium on Security and Privacy (SP), pp. 39–57, IEEE, 2017.
21. A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, “Towards deep learning models resistant to adversarial attacks,” arXiv preprint arXiv:1706.06083, 2017.
22. F. Croce and M. Hein, “Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks,” arXiv preprint arXiv:2003.01690, 2020.
23. S. Sabour, Y. Cao, F. Faghri, and D. J. Fleet, “Adversarial manipulation of deep representations,” arXiv preprint arXiv:1511.05122, 2015.

Додаток А

Лістинг програми

КОД ПРОГРАМИ

src/app.js

```
const express = require('express');
const dotenv = require('dotenv');

const loaders = require('./loaders');  const
mountRoutes = require('./routes');

const user1 = {
  publicKey:
'04b49c281f781435603ccc25074b378f7d4cc54ba41145da09576ff555fe9bd4f5a2660951548e09617114897b27d8019da3be3e0
f5c0d229bf5d9dbd71fd80882',
  email: 'user1@gmail.com', ip:
  '0.0.0.0',
  port: 5052, role:
                'user',
};

const user2 = {
  publicKey:
'043f25ae65644072ebb546f069ecd61f5f452bf564ec8792cda3b43906d80b5db7d602d00ee630f39c7c6dc06dd6aa5d9455e4801
2705f5e8d3013e8afdbc39100',
  email: 'user2@gmail.com', ip:
  '0.0.0.0',
  port: 5052, role:
                'user',
};

async function startServer() {
  const app = express();

  dotenv.config();
  await loaders({
    app, ...user2,
  }); mountRoutes(app);

  app.listen(5052, () => {
    console.log(`Running on ${5052}`);
  }); }

startServer();
```

src/constants.js

```
const modes = {
  USER: 'user',
  BOOTSTRAPPER: 'bootstrapper',
  MINER: 'miner',
};

module.exports = {
  modes,
};
```

src/package.json

```
{
  "name": "p2p",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "nodemon src/app.js",
    "lint": "eslint --debug src/**/*.{js,jsx}",
    "lint:write": "eslint --debug src/**/*.{js,jsx} --fix --quiet",
    "prettier": "prettier --write src/**/*.{js,jsx}",
    "test": "echo \"Error: no test specified\" && exit 1" },
  "keywords": [],
  "author": "", "license":
  "ISC", "dependencies":
  {
    "axios": "^0.19.0", "bcryptjs":
    "^2.4.3", "body-parser":
    "^1.19.0", "cors": "^2.8.5",
    "crypto-js": "^3.1.9-1",
    "dotenv": "^8.1.0",
    "elliptic": "^6.5.0",
    "express": "^4.17.1",
    "express-promise-router": "^3.0.3", "get-
    port": "^5.0.0",
    "husky": "^3.0.1", "ip":
    "^1.1.5",
    "jsonwebtoken": "^8.5.1", "lint-
    staged": "^9.2.1", "lodash":
    "^4.17.15", "mongodb":
    "^3.3.2", "mongoose": "^5.6.13",
    "request-promise": "^4.2.4"
  },  "devDependencies": {
    "eslint": "^5.16.0",
    "eslint-config-airbnb-base": "^13.2.0",
    "eslint-plugin-import": "^2.18.2",
    "nodemon": "^1.19.1",
    "prettier": "^1.18.2", "prettier-
    eslint": "^9.0.0"
  },  "husky": {
    "hooks": {
      "pre-commit": "lint-staged", "pre-
      push": "npm test"
    } },
        "lint-staged": {
          "*.{js,jsx}": [
      "eslint --fix --quiet", "git
      add"
    ] }
}


const fetchUsers = async () => { try {
    const response = await axios.get('http://0.0.0.0:5050/users'); //
    console.log('users', response.data);

    return response.data.users; }
  catch (error) {
    console.log('\n fetch-users \n');
    console.log(error.response.data.error);
  } };

const registerUser = async ({ ip,
  port, walletAddress,
  role,
```

```
email, }) =>
                {
  try {
    await axios.post(
      'http://0.0.0.0:5050/user', {
        walletAddress, ip,
        port,   role,
        email,
      }, );
    console.log('user has been added to bootstrapper'); }
  catch (error) {
    console.log('\n register-user \n');
    console.log(error.response.data.error);
  } };

const fetchBlockchain = async () => { try {
    const response = await axios.get('http://0.0.0.0:5050/blockchain'); //
    console.log('blockchain', response.data.blockchain);
    return response.data.blockchain; }
  catch (error) {
    console.log('\n fetch-blockchain \n');
    console.log(error.response.data.error);
  } };

module.exports = {
  fetchUsers,
  registerUser,
  fetchBlockchain,
};
```

src/loaders/bootstrap.js

```
const {
fetchUsers, fetchBlockchain,


const          {
  Node,
  Block,
} = require('../services');

module.exports = async ({
  publicKey,
  ip, port,
  role, email,
}) => {
  await registerUser({ ip,
    port,
    walletAddress: publicKey, role,
    email, });

          const nodes = await fetchUsers();
          nodes.forEach(async (node) => {
    if (node.walletAddress !== publicKey) {
      await Node.addOrUpdateNode(node);
      } });

  const blocks = await fetchBlockchain();
  await Block.updateChain(blocks);
};
```

src/loaders/express.js

```
const bodyParser = require('body-parser');
const cors = require('cors');
```

```
module.exports = async ({ app }) => {
  app.use(
    bodyParser.urlencoded({
      extended: true,
    }), );
  app.use(bodyParser.json());
  app.use(cors());

  return app; };
```

src/loaders/index.js

```
const expressLoader = require('./express');  const
mongooseLoader = require('./mongoose'); const
bootstrapLoader = require('./bootstrap');

module.exports = async ({ app,
  publicKey, ip,
port, role,
  await mongooseLoader();
  console.log('MongoDB Initialized');

  await expressLoader({ app });
  console.log('Express Initialized');

  await bootstrapLoader({
    publicKey,
    ip, port,
    role, email,
  });
  console.log('Bootstrap Initialized'); };
```

src/loaders/mongoose.js

```
const mongoose = require('mongoose');

module.exports = async () => {
  const connection = await mongoose.connect(process.env.MONGODB_URL, {
      useNewUrlParser: true,
      useFindAndModify: false,
      useUnifiedTopology: true,
    });
  return connection.connection.db; };
```

src/middleware/autenticate.js

```
const jwt = require('jsonwebtoken');
const { User } = require('../services/index');

const authenticate = (req, res, next) => {
  const authorization = req.header('Authorization');

  if (!authorization) {
    return res.status(401).json({ error: 'No token provided' }); }

  jwt.verify(authorization, process.env.JWT_SECRET, async (err, { email }) => { if
    (err) {
      return res.status(401).json({ error: 'Invalid token' }); }

    try {
      const user = await User.find({ email });

      if (user.error) {
        return res.status(401).json({ error: 'No user with provided credentials' }); }

      req.currentUser = user;
```

```
    next();
  } catch (error) {
    res.status(401).json({ error: 'Something when wrong' }); }
module.exports = {
 authenticate,
};
```

src/models/Block.js

```
const mongoose = require('mongoose');

const { Schema } = mongoose;
const { TransactionSchema } = require('./Transaction');

const BlockSchema = new Schema({ date:
 {
  type: Date,
  required: true,
 }, previousHash: {
  type: String,
  required: true,
 }, hash: {
  type: String,
  required: true,
 }, nonce: {
        type: Number,
        required: true,
 },
 transactions: [TransactionSchema],
 minerReward: {
  wallet: { type:
    String,
    required: true, },
  amount: {
         type: Number,
         required: true,
  }, },
});

module.exports = mongoose.model('Blockchain', BlockSchema);
```

src/models/Node.js

```
const mongoose = require('mongoose');

const { Schema } = mongoose;

const NodeSchema = new Schema({ ip: {
  type: String,
  required: true,
  lowercase: true, trim:
  true,
 }, port: {
        type: Number,
        required: true,
},


  required: true,
  lowercase: true,
  unique: true,
 }, role: {
  type: String,
  required: true,
 }, email: {
  type: String,
  required: true,
```

```
    }, });

module.exports = mongoose.model('Node', NodeSchema);
```

src/models/Transaction.js

```
const mongoose = require('mongoose');

const { Schema } = mongoose;

const TransactionSchema = new Schema({
  hash: {
    type: String,
    required: true,
  },  signature: {
    type: String,
    required: true,
  },  data: {
    from: {
      type: String,
      required: true,
    },  to: {
      type: String,
      required: true,
    },  amount: {
            type: Number,
            required: true,
    },  description: {
      type: String, },
    }, });

module.exports = {
  TransactionSchema,
};
```

src/models/User.js

```
const mongoose = require('mongoose');
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
const { Schema } = mongoose;

const UserSchema = new Schema({
  email: {
    type: String,
    required: true,
    lowercase: true, trim:
    true, unique: true,
  },  passwordHash: {
    type: String,
    require: true,
  },  phoneNumber: {
    type: String,
    required: true,
  },  firstName: {
    type: String,
    required: true,
  },  lastName: {
    type: String,
    required: true,
  },  registrationDate: {
    type: Date, required:
    true, default: Date.now,
  },  publicKey: {
    type: String,
    required: true,
```

```
  },
  // TODO: salt privateKey?
  privateKey: {
    type: String,
    required: true,
     }, });

UserSchema.methods.isValidPassword = function isValidPassword(password) {
  return bcrypt.compareSync(password, this.passwordHash);
};

UserSchema.methods.setPassword = function setPassword(password) {
  this.passwordHash = bcrypt.hashSync(password, process.env.JWT_SECRET);
};

UserSchema.methods.generateJWT = function generateJWT() {
  return jwt.sign(
    {
      registrationDate: this.registrationDate,
      email: this.email,
    }, process.env.JWT_SECRET, {
expiresIn: '30d',
UserSchema.methods.generateKeyPair = function genereteKeyPair() { const
  { publicKey, privateKey } = generateKeyPair();

  this.publicKey = publicKey;
  this.privateKey = privateKey;
};

module.exports = mongoose.model('User', UserSchema);
```

src/routes/dashboard.js

```
const Router = require('express-promise-router');
const axios = require('axios');
const _ = require('lodash/fp');

const {
  authenticate,
} = require('../middleware/authenticate');
const {
  signTransaction,
  calculateHash,
} = require('../utils/transaction');
const {
  Block,
  User,
} = require('../services');

const router = new Router();

router.post('/transaction/create', authenticate, async (req, res) => {
  const {
    publicKey,
    privateKey,
  } = req.currentUser;
  const {
    to, amount,
    description,   } =
              req.body;

  const hash = calculateHash(publicKey, to, amount, description);
  const signature = signTransaction(hash, privateKey);

  try {
    await axios.post(
```

```
    'http://0.0.0.0:5051/transaction/', {
      data: {
        from: publicKey, to,
        amount,
        description,
      }, signature,
      hash,
    }, );

  res.json({}); });

router.post('/new-block', async (req, res) => {
  await Block.updateChain([req.body]); res.json({
  });
});

router.get('/transactions', authenticate, async (req, res) => {
  const getUserAdresses = _.pipe(
    _.map(tx => [tx.data.from, tx.data.to]),
    _.flatten,
    _.uniq, );

  const txs = await Block.findUserTransactions(req.currentUser.publicKey); const
  userAdresses = getUserAdresses(txs);

  const users = await User.findByPublicKey(userAdresses);

  const mappedTxsWithUsers = _.pipe(
    _.map(tx => ({
      data: tx.data, hash:
      tx.hash,
      from: users[tx.data.from], to:
      users[tx.data.to],
    })), _.reverse,
  );

  res.json({ txs: mappedTxsWithUsers(txs) }); });

module.exports = router;
```

src/routes/index.js

```
const keypair = require('./keypair');
const signup = require('./signup'); const
login = require('./login');
const dashboard = require('./dashboard');

module.exports = (app) => {
  app.use('/keypair', keypair);
  app.use('/signup', signup);
  app.use('/login', login);
  app.use('/dashboard', dashboard);
};
```

src/routes/keypair.js

```
const Router = require('express-promise-router');

const { keyPairFormatter, streamToString } = require('../utils/io');

const router = new Router();

router.get('/', async (req, res) => {
  const keyPair = await streamToString('./keypair.txt', keyPairFormatter);
```

```
    return; }

  res.json({
    publicKey: keyPair.publicKey, });
});

module.exports = router;
```

src/routes/login.js

```
const Router = require('express-promise-router');

const { User } = require('../services');

const router = new Router();

router.post('/', async (req, res) => { //
  TODO: add validation here

  const   {   email,
    password,
  } = req.body;

  try {
    const user = await User.find({ email });

    if (user && !user.error && user.isValidPassword(password)) {
      res.json({
        token: user.generateJWT(), });
    } else {
      res.status(400).json({
        error: 'Invalid credentials provided', });
    }
  } catch (error) {
    console.log(error);
    res.status(400).json({ error: error.message }); }
});

module.exports = router;
```

src/routes/signup.js

```
const Router = require('express-promise-router');

const { User } = require('../services');

const router = new Router();

router.post('/', async (req, res) => { //
  TODO: add validation here

const {



    password,
  } = req.body;

  try {
    const user = await User.create({
      email,
      firstName,
      lastName,
      phoneNumber,
      password,
    });
```

```
    res.json({ reregistered:
      true,
      token: user.generateJWT(), });
  } catch (error) {
    console.log(error);
    res.status(400).json({ error: error.message }); }
});

module.exports = router;
```

src/utils/io.js

```
const fs = require('fs');

const keyPairFormatter = (data) => {
  const [publicKey, privateKey] = data.split(':');

  return {
    publicKey,
    privateKey,
  }; };

const streamToString = (filePath, formatter = data => data) => {
  const stream = fs.createReadStream(filePath);
  const chunks = [];

  return new Promise((resolve, reject) => {
    stream.on('data', (chunk) => {
      chunks.push(chunk); });
    stream.on('error', reject);
    stream.on('end', () => {
      const data = formatter(Buffer.concat(chunks).toString());
      resolve(data);
    }); });
};

const stringToStream = (filePath, data) => { const
  stream = fs.createWriteStream(filePath);

  return new Promise((resolve, reject) => {
    stream.write(data);
    stream.end();

stream.on('error', reject);

};

module.exports = {
  stringToStream,
  streamToString,
  keyPairFormatter,
};
```

src/utils/keygenerator.js

```
const EC = require('elliptic').ec;

const ec = new EC('secp256k1');

const generateKeyPair = () => {
  const key = ec.genKeyPair();
  const publicKey = key.getPublic('hex'); const
  privateKey = key.getPrivate('hex');

  return {   key,
    publicKey,
    privateKey, };
```

```
};

module.exports = {
  generateKeyPair,
};
```

src/utils/request.js

```
const request = require('request-promise');

// FIXME
// For each request need to change IP
const registerUser = (walletAddress, ip, port) => {
  const options = {
    method: 'POST',
    uri: 'http://0.0.0.0:5050/user/',
    body: {
      walletAddress, ip,
      port, },
    json: true, };

  return request.post(options); };

const fetchUsers = () => {
  const options = {
    method: 'GET',
    uri: 'http://0.0.0.0:5050/users/', };
```

src/utils/transaction.js

```
const SHA256 = require('crypto-js/sha256');
const EC = require('elliptic').ec;

const ec = new EC('secp256k1');

// TODO: make calculating suitable to all cases
const calculateHash = (...params) => {
  const concatenatedString = params.reduce((acc, current) => acc + current, '');

  return SHA256(concatenatedString).toString(); };

const signTransaction = (hash, privateKey) => { const
  myKey = ec.keyFromPrivate(privateKey);

  const sig = myKey.sign(hash, 'base64');
  const signature = sig.toDER('hex');

  return signature; };

const verifyTransaction = (hash, publicKey, signature) => {
  const pKey = ec.keyFromPublic(publicKey, 'hex');

  return pKey.verify(hash, signature); };
```

Додаток Б

ПЕРЕЛІК ДОКУМЕНТІВ НА ОПТИЧНОМУ НОСІЇ

Ім’я файла	**Опис**
Пояснювальні документи	
Кваліфікаційна робота.doc	Пояснювальна записка. Документ Word.
Кваліфікаційна робота.pdf	Пояснювальна в форматі PDF
Программа	
Program.rar	Архів. Містить коди програми і откомпільовану програму
Презентація	
Презентація.ppt	Презентація