

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ  
ДЕРЖАВНИЙ ЗАКЛАД  
„ЛУГАНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ  
ІМЕНІ ТАРАСА ШЕВЧЕНКА”

Навчально-науковий інститут математики та інформаційних технологій

Кафедра математики та інформатики

**Друпп Олександр Вадимович**

**ПЛАТФОРМА ОБМІНУ ЦИФРОВИМИ СЕРТИФІКАТАМИ НА  
ОСНОВІ БЛОКЧЕЙНУ**

**кваліфікаційна робота здобувача вищої освіти першого (бакалаврського)  
рівня освітньої програми « Комп’ютерні науки та інформаційні  
технології » за спеціальністю 122 Комп’ютерні науки**

Особистий підпис \_\_\_\_\_ Олександр ДРУПП

Науковий керівник \_\_\_\_\_ Микола СЕМЕНОВ,  
кандидат педагогічних наук, доцент  
кафедри інформаційних технологій  
та систем

Завідувач кафедри \_\_\_\_\_ Микола СЕМЕНОВ,  
кандидат педагогічних наук, доцент  
кафедри інформаційних технологій  
та систем

Полтава – 2025

Міністерство освіти і науки України  
Державний заклад „Луганський національний університет  
імені Тараса Шевченка”

Факультет (інститут)

Навчально-науковий інститут фізики,  
математики та інформаційних технологій

Кафедра

Інформаційних технологій та систем

Рівень освіти

перший (бакалаврський)

Спеціальність

Ф3 Комп'ютерні науки

(код, назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри ІТС

Микола СЕМЕНОВ

(підпис)

(ім'я, прізвище)

“ ”

2024 р.

**ЗАВДАННЯ  
НА КВАЛІФІКАЦІЙНУ РОБОТУ**

**Друппа Олександра Вадимовича**

(прізвище, ім'я, по батькові )

**1. Тема проекту (роботи)** Платформа обміну цифровими сертифікатами на основі блокчейну

Керівник кваліфікаційної роботи

Семенов М.А.

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджена наказом по університету

Від“ ”

2024 року №

**2. Строк подання студентом проекту (роботи)**

**3. Вихідні дані до роботи (проекту)** у результаті виконання роботи

повинно бути розроблена система захищеного обміну повідомленнями з використанням блокчейну

(визначаються кількісні або (та) якісні показники, яким повинен відповідати об'єкт розробки)

**4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)** АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

ПРОЕКТУВАННЯ СИСТЕМИ. РЕАЛІЗАЦІЯ ПРОЕКТУ.

(визначаються назви розділів або (та) перелік питань, які повинні увійти до тексту ПЗ)

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

**6. Консультанти розділів проекту (роботи)**

Розділ	Прізвище, ініціали та посада Консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання „\_\_\_\_\_” \_\_\_\_\_ 2024 р.

**КАЛЕНДАРНИЙ ПЛАН**

№ з / п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
	Вибір теми роботи, вивчення наукової літератури, затвердження теми та керівника.	До 15 жовтня	
	Аналіз літературних джерел за темою роботи. Розробка та апробація методики дослідно-експериментальної роботи. Подання структури теоретичної частини роботи та плану експериментальних досліджень.	Другий тиждень листопада (10 листопада)	
	Робота над теоретичною частиною. Подання теоретичної частини роботи для першого читання науковим керівником.	До 15 грудня	
	Усунення зауважень, урахування рекомендацій наукового керівника. Подання теоретичної частини роботи на друге читання.	До 28 січня	
	Проведення експериментальної роботи. Поетапний аналіз та обговорення її результатів. Перевірка стану виконання роботи.	Перший тиждень березня	
	Урахування рекомендацій наукового керівника, усунення недоліків, підготовка варіанта роботи до передзахисту. Розробка презентації.	До 31 березня	
	Попередній захист роботи на кафедрі	квітень	
	Доопрацювання роботи з урахуванням рекомендацій після передзахисту. Подання роботи науковому керівникові та рецензентові на підготовку відгуку та рецензії	За 10 днів до державної атестації	
	Подання на кафедру остаточного варіанта роботи, переплетеного та підписаного автором, науковим керівником і рецензентом.	За 5 днів до державної атестації	

**Студент**

\_\_\_\_\_

підпис

**Олександр ДРУГІЙ****Керівник проекту (роботи)**

\_\_\_\_\_

підпис

**Микола СЕМЕНОВ**

## АНОТАЦІЯ

**Друпп О.В.**

**Тема:** Платформа обміну цифровими сертифікатами на основі блокчейну.

**Спеціальність:** 122 „Комп’ютерні науки”

**Установа:** ДЗ ЛНУ імені Т. Шевченка, 202\_р.

**Кваліфікаційна робота містить:**

**Об’єкт дослідження** – процес обміну цифровими сертифікатами між користувачами в інформаційних системах.

**Предмет дослідження** – програмна платформа для створення, зберігання та перевірки цифрових сертифікатів із використанням технології блокчейн.

**Мета роботи** – розробка вебплатформи обміну цифровими сертифікатами з реалізацією функціональності блокчейну, автентифікації користувачів та захисту даних.

**Результати роботи** - розроблено повнофункціональну вебплатформу з можливістю реєстрації, видачі сертифікатів, їх збереження у блокчейні та перевірки достовірності. Реалізовано REST API, frontend-інтерфейс на React.js та захист даних за допомогою токенів.

**Висновки** - підтверджено ефективність використання блокчейну для забезпечення цілісності та автентичності цифрових сертифікатів. Обґрунтовано вибір стеку технологій. Система може масштабуватись і використовуватись у навчальних або корпоративних середовищах.

**Ключові слова:** блокчейн, цифровий сертифікат, MongoDB, Node.js, React, REST API, авторизація, хешування, вебплатформа.

## ABSTRACT

**Drupp**

**O.V.**

**Topic:** A Blockchain-Based Digital Certificate Exchange Platform

**Specialty:** 122 “Computer Science”

**Institution:** State Institution Luhansk Taras Shevchenko National University, 202\_

**The qualification thesis includes:**

**Object of research** – the process of exchanging digital certificates between users in information systems.

**Subject of research** – a software platform for the creation, storage, and verification of digital certificates using blockchain technology.

**Purpose of the study** – to develop a web platform for digital certificate exchange with implemented blockchain functionality, user authentication, and data protection.

**Research results** – a fully functional web platform was developed, enabling user registration, certificate issuance, storage in the blockchain, and verification of authenticity. The system implements a REST API, a React.js frontend interface, and data protection through token-based authentication.

**Conclusions** – the effectiveness of using blockchain to ensure the integrity and authenticity of digital certificates has been confirmed. The choice of the technology stack has been justified. The system is scalable and can be used in educational or corporate environments.

**Keywords:** blockchain, digital certificate, MongoDB, Node.js, React, REST API, authorization, hashing, web platform.

## ЗМІСТ

ВСТУП .....	8
РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ .....	10
1.1. Загальна характеристика предметної області.....	10
1.2. Технологічні аспекти проблеми.....	10
1.3. Можливості блокчейн-технології у сертифікації.....	12
1.4. Актуальність проблеми .....	14
1.5. Існуючі рішення та їх порівняльна характеристика .....	16
1.6. Висновки до розділу .....	17
РОЗДІЛ 2. ПРОЕКТУВАННЯ СИСТЕМИ .....	18
2.1 Модель проекту .....	18
2.2 Архітектура системи .....	21
2.3 Вибір технологій .....	22
2.4 Структура бази даних .....	24
2.5 Взаємодія компонентів системи .....	26
2.6 Безпека та авторизація .....	27
2.7 Висновки до розділу .....	27
РОЗДІЛ 3. РЕАЛІЗАЦІЯ ПРОЕКТУ .....	28
3.1 Реалізація серверної частини проекту .....	28
3.1.1 Ініціалізація проекту.....	28
3.1.2 Ініціалізація серверу та підключення маршрутів .....	28
3.1.3 Реалізація підключення до бази даних MongoDB .....	30
3.1.4 Реалізація функціоналу серверної частини .....	31
3.1.5 Налаштування маршрутів серверної частини .....	37
3.1.6 Реалізація допоміжних функцій безпеки та автентифікації .....	40
3.2 Реалізація клієнтської частини .....	44
3.2.1 Ініціалізація проекту.....	44
3.2.2 Реалізація маршрутизації та перевірки автентифікації користувача у клієнтській частині. ....	45

3.2.3 Реалізація клієнтської логіки авторизації та роботи з сертифікатами .....	48
3.2.4 Розробка користувацького інтерфейсу для роботи з цифровими сертифікатами.....	55
3. Відображення .....	59
3.2.5 Контекст автентифікації та валідація реєстрації користувача. ....	71
3.3 Тестування .....	74
<b>ВИСНОВКИ .....</b>	<b>76</b>
<b>СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....</b>	<b>78</b>

## ВСТУП

У цифрову епоху важливість забезпечення достовірності та збереження електронних документів зростає з кожним роком. Особливо це стосується сертифікатів, які підтверджують кваліфікацію, завершення курсів чи участь у заходах. Блокчейн як децентралізована технологія пропонує надійне рішення для перевірки автентичності таких документів, усуваючи потребу в довірених третіх сторонах. Реалізація платформи на основі блокчейну забезпечує прозорість, захист від фальсифікацій та зручність у використанні, що є надзвичайно актуальним для освітніх та професійних середовищ.

Мета роботи — розробити вебплатформу для створення, зберігання та перевірки цифрових сертифікатів з використанням технології блокчейн.

### **Завдання роботи:**

1. Провести аналіз сучасних технологій для створення безпечної системи обміну сертифікатами.
2. Розробити архітектуру системи, що включає базу даних, логіку блокчейну та вебінтерфейс.
3. Реалізувати функціонал генерації цифрових сертифікатів.
4. Забезпечити інтеграцію кожного сертифіката у блокчейн.
5. Реалізувати перевірку автентичності сертифікатів через блокчейн.
6. Розробити клієнтську частину зручну для взаємодії користувачів із системою.

У роботі використано методи об'єктно-орієнтованого програмування, розробки REST API, клієнт-серверної архітектури та принципи побудови блокчейну. Серверна частина реалізована на Node.js із використанням фреймворку Express. Для збереження даних застосовано MongoDB у поєднанні з бібліотекою Mongoose. Клієнтська частина розроблена на основі бібліотеки React.js. Для забезпечення безпеки реалізовані механізми хешування та перевірки цілісності даних у блокчейні.

Розроблена система може бути застосована в освітніх, сертифікаційних, корпоративних та інших структурах, де важливо забезпечити надійність та



достовірність цифрових сертифікатів. Платформа може легко масштабуватись і доповнюватись додатковими функціональними модулями (наприклад, QR-перевіркою, імпортом користувачів або статистикою). Вона також демонструє практичне використання блокчейн-технологій у реальних застосунках.

## **РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ**

### **1.1. Загальна характеристика предметної області**

У сучасному інформаційному суспільстві стрімко зростає роль цифрових технологій. Практично кожна сфера людської діяльності зазнає цифрової трансформації, що передбачає впровадження інноваційних технологій для підвищення ефективності, прозорості та доступності процесів. Сфера документообігу не є винятком. Значна частина документів, які раніше існували виключно у паперовій формі, нині трансформуються у цифровий формат. Серед таких документів — сертифікати, дипломи, довідки, ліцензії, атестати та інші документи, що мають юридичну силу.

Цифрові сертифікати є не лише сучасною альтернативою паперовим документам, а й інструментом, що дозволяє здійснювати автоматизовану перевірку автентичності, пришвидшує обмін інформацією, знижує ризики втрат або підробки документів. У навчальних закладах, компаніях, органах державного управління все частіше застосовуються системи електронного сертифікування для підтвердження результатів навчання, професійної кваліфікації, проходження курсів чи участі в подіях.

Це зумовлює зростання попиту на системи, які можуть забезпечити зручне створення, зберігання, передачу та перевірку цифрових сертифікатів. Проте перехід до цифрових документів супроводжується рядом проблем, пов'язаних із забезпеченням безпеки, довіри та надійності таких документів.

### **1.2. Технологічні аспекти проблеми**

На сьогодні більшість існуючих систем цифрової сертифікації побудовані на основі централізованої архітектури. Це означає, що всі дані про видані документи зберігаються в одній централізованій базі даних, яка перебуває під контролем однієї організації чи установи. Такий підхід має низку суттєвих обмежень і викликів, що впливають на ефективність і безпеку системи.

#### **Недоліки централізованої архітектури:**

- **Централізована вразливість:** Оскільки вся інформація зберігається в одному місці, база даних стає єдиною точкою відмови. У разі злому, технічного збою або кібератаки існує високий ризик втрати даних, їх пошкодження або несанкціонованих змін. Це ставить під загрозу достовірність і доступність сертифікатів.
- **Обмежена довіра:** У централізованих системах користувачі та треті сторони змушені покладатися на доброчесність та компетентність організації, яка управляє базою даних. Відсутність можливості самостійної незалежної перевірки інформації породжує сумніви у достовірності документів і знижує рівень довіри.
- **Складність масштабування:** Зі зростанням кількості користувачів, обсягів даних та запитів до системи централізована архітектура часто не справляється з навантаженням. Підвищення продуктивності вимагає значних ресурсів та інвестицій у потужне серверне обладнання, що не завжди економічно виправдано.
- **Ризик корупції та маніпуляцій:** Централізоване управління даними створює можливості для зловживань, зокрема, фальсифікації або приховування інформації, що особливо актуально для державних та великих корпоративних систем.
- **Відсутність прозорості:** Зазвичай користувачі не мають відкритого доступу до реєстру виданих сертифікатів і не можуть незалежно перевірити їх статус або історію змін, що знижує рівень довіри та ускладнює аудит.

Ці проблеми стимулюють пошук і впровадження нових технологічних рішень, які здатні забезпечити більш високий рівень безпеки, прозорості, надійності і масштабованості. У цьому контексті особливу увагу привертає технологія блокчейн, яка завдяки своїй розподіленій та криптографічно захищеній природі пропонує принципово інший підхід до зберігання і верифікації даних.

### **Переваги блокчейн-технології як альтернативи:**

- Відсутність єдиної точки відмови завдяки децентралізованому зберіганню даних.
- Можливість прозорості і публічної перевірки інформації будь-яким користувачем.
- Захист від несанкціонованих змін через криптографічні механізми і консенсусні алгоритми.
- Легке масштабування за рахунок розподіленої мережі вузлів.
- Підвищення рівня довіри без необхідності покладатися на одну організацію.

Таким чином, впровадження блокчейн-технології у системах цифрової сертифікації є важливим кроком для подолання існуючих технічних та організаційних обмежень централізованих рішень, що сприятиме створенню більш надійних, прозорих і доступних сервісів.

### **1.3. Можливості блокчейн-технології у сертифікації**

Блокчейн — це розподілений цифровий реєстр, який зберігає дані у вигляді послідовно з'єднаних блоків. Кожен блок містить унікальний хеш попереднього блоку, набір транзакцій (у контексті цифрової сертифікації — інформацію про сертифікати) та часову мітку. Така структура гарантує цілісність і послідовність даних, оскільки будь-яка спроба змінити інформацію в одному блоці призведе до невідповідності хешів, що легко виявляється всіма учасниками мережі. Криптографічні механізми забезпечують надійний захист інформації від несанкціонованих змін і фальсифікації, а децентралізований характер блокчейну усуває потребу в центральному контролюючому органі, що значно підвищує рівень довіри до системи.

Основні переваги використання блокчейну у цифровій сертифікації полягають у наступному:

- **Незмінність даних (імутабельність):** Після внесення сертифікату до блокчейну його неможливо змінити або видалити без погодження більшості учасників мережі, що робить фальсифікацію

практично неможливою. Це гарантує збереження достовірності сертифікатів на довгі роки.

- **Децентралізація:** Завдяки розподіленій природі блокчейну, дані зберігаються на багатьох вузлах одночасно, що усуває ризики маніпуляцій або втрати інформації через збої чи атаки на один центральний сервер. Це робить систему більш стійкою і надійною.

- **Прозорість:** Всі транзакції у публічних або напівпублічних блокчейнах відкриті для перевірки будь-яким користувачем. Це означає, що кожен може самостійно перевірити справжність сертифікату, не звертаючись до посередників або органів сертифікації, що значно підвищує довіру та зменшує корупційні ризики.

- **Доступність:** Завдяки публічному характеру записів і можливості миттєвого доступу через інтернет, інформація про сертифікати стає доступною для широкого кола користувачів у будь-який час і в будь-якому місці, що зручно як для власників сертифікатів, так і для роботодавців, навчальних закладів чи контролюючих органів.

- **Автоматизація за допомогою смарт-контрактів:** Смарт-контракти — це самовиконувані програми, що працюють у блокчейні і автоматично виконують задані умови без участі людини. У контексті сертифікації це дозволяє автоматично видавати, перевіряти, оновлювати або анулювати сертифікати, що значно спрощує і прискорює процеси, зменшуючи ризик людської помилки.

- **Підвищена безпека:** Використання сучасних криптографічних алгоритмів забезпечує надійний захист персональних даних власників сертифікатів, дозволяючи контролювати доступ до чутливої інформації і зберігати конфіденційність.

- **Інтероперабельність:** Блокчейн може служити єдиною платформою, що об'єднує різні організації та установи, які видають сертифікати, незалежно від їхнього географічного розташування або сфери діяльності. Це дозволяє створити єдину екосистему цифрової

сертифікації з універсальним форматом і спрощує взаємодію між учасниками.

Завдяки цим характеристикам блокчейн-технологія відкриває нові горизонти для впровадження цифрової сертифікації як у межах окремих компаній або навчальних закладів, так і на рівні державних інституцій. Вона сприяє підвищенню довіри до документів, зменшенню адміністративних витрат, оптимізації процесів перевірки та надає можливість створення прозорих і ефективних систем, що відповідають сучасним викликам цифрової епохи.

#### **1.4. Актуальність проблеми**

У сучасному світі спостерігається стрімке зростання популярності онлайн-освіти, що стає доступною для все ширшого кола користувачів. Платформи на кшталт Coursera, edX, Udemu, Skillshare, а також численні національні освітні ініціативи надають можливість здобувати знання у різних галузях, незалежно від географічного розташування та соціального статусу. Кожен пройдений курс або навчальна програма зазвичай завершується видачею цифрового або електронного сертифіката, що підтверджує компетенції і набуті навички.

Проте, на практиці такі сертифікати часто не отримують належного визнання через відсутність централізованих та надійних систем їх перевірки. Багато роботодавців, навчальних закладів та інших зацікавлених сторін сумніваються в достовірності таких документів через можливість їх підробки або фальсифікації. Це створює значні бар'єри для кандидатів, які прагнуть підтвердити свою кваліфікацію, а також ускладнює процес прийняття рішень з боку роботодавців чи організацій.

Відтак постає нагальна потреба у впровадженні:

- **Універсального формату сертифікатів**, який би був зрозумілим і сумісним між різними освітніми платформами, організаціями та інституціями.

- **Надійної системи перевірки справжності сертифікатів,** яка не вимагала б безпосереднього звернення до організації-видавця та була б швидкою, прозорою і доступною для всіх зацікавлених сторін.

- **Масштабованого та доступного рішення,** яке було б інтуїтивно зрозумілим для користувачів, не потребувало спеціальних технічних знань і одночасно забезпечувало високий рівень безпеки.

Актуальність цифрової сертифікації не обмежується виключно сферою освіти. Потреба у надійних і прозорих цифрових документах зростає також у багатьох інших галузях:

- **HR та рекрутинг:** Підтвердження дипломів, професійних сертифікатів, проходження курсів та набуття нових навичок є важливою частиною процесу найму. Цифрова сертифікація дозволяє уникнути шахрайства та забезпечує швидку перевірку кандидатів.

- **Державне управління:** Видача ліцензій, сертифікатів відповідності, дозволів у різних сферах діяльності (наприклад, будівництво, екологія) потребує прозорості і надійної системи для забезпечення контролю і довіри.

- **Медицина:** Сертифікати про вакцинацію, навчання та підвищення кваліфікації медичного персоналу, ліцензії на медичну діяльність — всі ці документи потребують безпечного зберігання та швидкої перевірки.

- **Промисловість і виробництво:** Кваліфікаційні допуски, технічні дозволи, сертифікати якості продукції та безпеки — важливі документи, які забезпечують відповідність стандартам і правилам.

Розвиток блокчейн-систем цифрової сертифікації створює фундамент для вирішення багатьох актуальних проблем, пов'язаних з довірою, безпекою, фальсифікацією і втратою документів. Запровадження таких систем дозволить підвищити ефективність і прозорість процесів видачі і перевірки сертифікатів, сприятиме формуванню єдиної цифрової екосистеми, що буде корисною як для

окремих користувачів, так і для організацій на локальному, національному та міжнародному рівнях.

### 1.5. Існуючі рішення та їх порівняльна характеристика

Розглянемо найвідоміші приклади рішень у сфері блокчейн-сертифікації:

Система	Тип технології	Платформа	Переваги	Недоліки
Blockcerts	Відкритий блокчейн	Bitcoin/Ethereum	Прозорість, відкритий код	Складність впровадження
OpenCerts	Ethereum	Open-source	Державна підтримка в SG	Потребує навичок роботи з блокчейном
Credly	Централізована	SaaS	UI/UX, інтеграції	Не використовує блокчейн

Порівняльна таблиця існуючих рішень у сфері цифрової сертифікації дозволяє зробити кілька важливих висновків щодо стану розвитку та рівня придатності цих систем для впровадження у навчальних, корпоративних або державних закладах.

**Blockcerts** — це відкрита блокчейн-платформа, яка функціонує на базі публічних мереж Bitcoin або Ethereum. Вона має високу прозорість і використовує відкритий код, що дозволяє адаптувати систему під власні потреби. Проте її впровадження вимагає глибоких технічних знань і ресурсів, що може ускладнити використання для непідготовлених користувачів.

**OpenCerts** — державна ініціатива Сінгапуру, що ґрунтується на Ethereum. Вона також є відкритою та підтримує прозорість, але користувачеві все одно необхідно мати базові знання роботи з блокчейн-середовищем. Це обмежує її масове впровадження у менш технічно підготовлених закладах.

**Credly** — комерційне SaaS-рішення, що не базується на блокчейні. Воно пропонує зручний інтерфейс, широкі можливості інтеграції та популярне



серед HR-компаній, однак не забезпечує високого рівня довіри та незмінності даних, які властиві блокчейн-технології. Його централізований характер знижує довіру до процесу верифікації сертифікатів.

У результаті можна зробити висновок, що існуючі рішення або недостатньо зручні для широкого кола користувачів, або не забезпечують належного рівня захисту від фальсифікацій. Це відкриває простір для створення нової, адаптивної системи цифрової сертифікації на базі блокчейну, яка буде простішою у впровадженні, безпечнішою та більш гнучкою.

### **1.6. Висновки до розділу**

Аналіз предметної області цифрової сертифікації свідчить про зростаючу потребу у надійних, прозорих та безпечних системах для створення, зберігання та перевірки цифрових документів. Сучасні централізовані рішення мають низку суттєвих обмежень, серед яких – вразливість до технічних збоїв і кібератак, обмежена довіра користувачів, складнощі масштабування та ризики корупції. Ці фактори обумовлюють необхідність впровадження нових технологічних підходів.

Технологія блокчейн, завдяки своїй децентралізованій та криптографічно захищеній природі, пропонує перспективне рішення для подолання вказаних проблем. Вона забезпечує незмінність і прозорість даних, підвищує рівень довіри до цифрових сертифікатів, сприяє автоматизації процесів за допомогою смарт-контрактів і підтримує масштабованість систем.

Аналіз існуючих рішень на базі блокчейн-технологій показав, що хоча вони мають важливі переваги, їх впровадження часто ускладнене через технічні бар'єри або обмеження функціональності. Централізовані альтернативи, хоч і зручні, не забезпечують достатнього рівня безпеки і довіри.

Отже, виникає потреба у створенні нової, адаптивної системи цифрової сертифікації, що поєднуватиме простоту використання, високу безпеку та гнучкість, відкриваючи нові можливості для різних сфер діяльності — від освіти до державного управління і бізнесу.

## РОЗДІЛ 2. ПРОЕКТУВАННЯ СИСТЕМИ

### 2.1 Модель проекту

Модель проекту являє собою сучасну клієнт-серверну вебплатформу, що забезпечує повний цикл роботи з цифровими сертифікатами — їх створення, збереження, перегляд та перевірку автентичності. Головною особливістю системи є використання технології блокчейн, що гарантує незмінність і захищеність даних. Архітектура платформи поділена на дві основні частини: клієнтську (frontend) і серверну (backend), які взаємодіють між собою через чітко визначений API.

#### Клієнтська частина (Frontend)

Інтерфейс користувача розроблено за допомогою сучасного JavaScript-фреймворку React. Це дозволяє створити динамічний, інтуїтивно зрозумілий та адаптивний інтерфейс, який підтримує роботу на різних пристроях, включаючи десктопи, планшети та смартфони. Користувачі можуть легко реєструватися, авторизуватися, створювати нові сертифікати, переглядати існуючі, а також перевіряти їх автентичність.

Для обміну даними з сервером використовується бібліотека axios, що реалізує асинхронні HTTP-запити до REST API. Такий підхід дозволяє розділити логіку клієнта та сервера, покращуючи масштабованість і підтримуваність системи.

#### Серверна частина (Backend)

Сервер реалізовано на платформі Node.js із використанням фреймворка Express, що забезпечує високу продуктивність і простоту налаштувань маршрутизації запитів. Сервер виконує кілька основних функцій:

- **Обробка запитів** від клієнта (реєстрація, авторизація, створення сертифікатів, запити на перевірку);
- **Керування автентифікацією і авторизацією** користувачів за допомогою JWT (JSON Web Tokens), що забезпечує безпечне зберігання і передачу інформації про сесії;

- **Хешування паролів** із використанням bcryptjs для захисту облікових даних від несанкціонованого доступу;
- **Збереження даних** у документно-орієнтованій базі MongoDB через ODM-бібліотеку mongoose, що полегшує роботу зі складними структурами даних і підтримує гнучку схему;
- **Реалізація логіки формування блоків блокчейну**, що містять інформацію про сертифікати, їх хеші, а також посилання на попередні блоки.

Для зберігання та передачі JWT токенів сервер використовує cookie-parser, що дозволяє зручно та безпечно керувати сесіями користувачів.

## **Основні сутності системи**

### **1. Користувач (User)**

Кожен користувач має унікальний обліковий запис, що включає такі поля, як ім'я, email, пароль у хешованому вигляді, а також дату реєстрації.

### **2. Цифровий сертифікат (Certificate)**

Це основний цифровий документ, що підтверджує певний факт або подію (наприклад, проходження курсу, участь у заході тощо). Сертифікат містить назву, опис, дату створення, а також посилання на ініціатора (користувач, який його створив). Сертифікат зберігається у базі та стає частиною блокчейн-ланцюга для забезпечення його незмінності.

### **3. Блок (Block)**

Ключова складова блокчейн-системи. Кожен блок містить інформацію про один або кілька сертифікатів, а також метадані: індекс блоку, часову позначку (timestamp), хеш поточного блоку та хеш попереднього блоку. Хеші генеруються криптографічними алгоритмами (SHA-256), що створює захищений ланцюг, де зміна даних у будь-якому блоці змінює хеші всього ланцюга і робить підробку миттєво помітною.

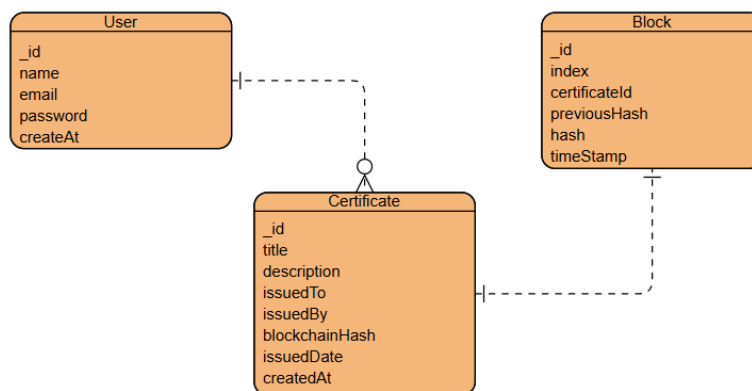


Рисунок 2.1 ER діаграма

### Реалізація блокчейн-архітектури

Блокчейн реалізований як послідовність документів у базі MongoDB. Після створення кожного нового сертифіката автоматично формується новий блок, що включає цей сертифікат. Для обчислення хешу блоку використовується інформація про вміст блоку та хеш попереднього, що забезпечує цілісність і послідовність ланцюга.

В разі будь-якої спроби зміни сертифіката або блоку у вже доданому ланцюзі, хеші перестануть співпадати, що дає можливість легко виявити спробу підробки чи несанкціонованого втручання.

### Безпека та авторизація

Для авторизації користувачів система застосовує JWT-токени, які зберігаються у HTTP-only cookie для захисту від атак типу XSS. Паролі користувачів хешуються за допомогою bcryptjs із сольовими значеннями, що забезпечує додатковий рівень безпеки збережених облікових даних.

### Технологічний стек

- **Frontend:** React, Axios
- **Backend:** Node.js, Express
- **База даних:** MongoDB, Mongoose
- **Безпека:** JWT, bcryptjs, cookie-parser

- **Криптографія:** SHA-256 (для хешування блоків)

## Висновок

Модель проекту об'єднує сучасні підходи веброзробки з технологією блокчейн, що робить систему не лише зручною для користувачів, а й надійною з точки зору безпеки та незмінності даних. Така архітектура дозволяє гарантувати автентичність цифрових сертифікатів, захищає від підробок і створює прозору систему перевірки інформації.

## 2.2 Архітектура системи

Розроблена система цифрової сертифікації базується на архітектурі типу «клієнт-сервер». Клієнтська частина створена за допомогою бібліотеки React, яка забезпечує динамічний інтерфейс користувача та взаємодіє із сервером через HTTP-запити. Серверна частина реалізована з використанням Node.js та Express.js, а база даних — MongoDB, яка є NoSQL-сховищем, з керуванням через бібліотеку Mongoose.

Блокчейн-структура інтегрована у backend та використовується для запису цифрових сертифікатів у вигляді блоків. Кожен блок містить криптографічний хеш попереднього блоку, тим самим забезпечуючи незмінність і достовірність даних.

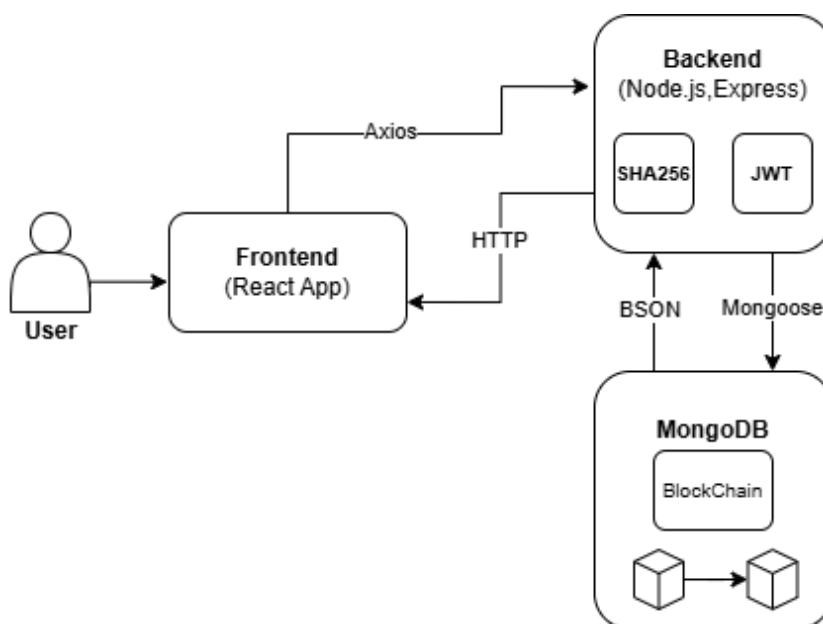


Рис. 2.2. Архітектура системи

## 2.3 Вибір технологій

Під час розробки системи цифрової сертифікації я свідомо обрав стек технологій, який відповідає сучасним вимогам до веб-розробки, забезпечує зручність у роботі з даними, масштабованість та безпеку. Основна мета полягала в тому, щоб створити платформу, яка буде зрозумілою для користувача, простою у підтримці та ефективною з технічної точки зору.

### Backend

- **Node.js** я обрав як серверне середовище, оскільки воно дозволяє писати весь застосунок однією мовою — JavaScript. Це спрощує розробку і дає змогу швидко переходити між фронтом і бекендом. Node.js має асинхронну неблокуючу модель, що забезпечує високу продуктивність і можливість обробляти велику кількість одночасних запитів.
- **Express.js** став основним фреймворком для побудови API, бо він легкий, добре документований і дає повну гнучкість у створенні RESTful-сервісів. З його допомогою я легко реалізував маршрути, обробку запитів, обмеження доступу та обробку помилок.
- **mongoose** — бібліотека, яку я використав для роботи з MongoDB, дозволяє створювати схеми для документів і забезпечує валідацію даних. Це значно зменшує ризик помилок та спрощує структурування коду.
- **bcryptjs** я використовую для хешування паролів користувачів перед збереженням у базі даних. Це необхідно для забезпечення базового рівня безпеки облікових записів.
- **cookie-parser** застосовується для обробки cookie-файлів. Це дозволяє передавати і зчитувати токени аутентифікації з cookies, що зручно в браузерному середовищі.
- **cors** необхідний для дозволу кросдоменної взаємодії між клієнтом (який запускається окремо) та сервером, особливо на етапі розробки.

- **dotenv** дає змогу зберігати конфіденційні змінні (ключі доступу, URI бази даних, секрети JWT тощо) у окремому файлі `.env`, що підвищує безпеку та гнучкість конфігурації проекту.

- **jsonwebtoken** я застосовую для створення токенів аутентифікації. Це дозволяє реалізувати захищений доступ до частин системи — наприклад, тільки авторизовані користувачі можуть видавати або переглядати сертифікати.

## База даних

- **MongoDB** я вибрав як основну базу даних через її гнучкість і зручність роботи з JSON-подібними структурами. Вона чудово підходить для зберігання записів про користувачів, сертифікати, їх статус, ролі тощо. Крім того, MongoDB добре масштабується, що дозволяє розвивати систему в майбутньому.

## Frontend

- **React** я обрав як основну бібліотеку для побудови користувацького інтерфейсу. Завдяки компонентному підходу я міг створювати модульні, повторно використовувані частини UI, як-от форми реєстрації, панель викладача, перегляд сертифікатів. React добре підходить для створення SPA (Single Page Application), що забезпечує швидку і плавну взаємодію з користувачем.

- **Axios** використовується для виконання HTTP-запитів до API. Я надаю перевагу Axios через його зручний синтаксис, можливість налаштовувати глобальні заголовки (наприклад, токен авторизації) та обробку помилок.

## Додаткові приклади альтернатив

Під час вибору технологій я також розглядав інші варіанти:

- Замість **MongoDB** я розглядав **PostgreSQL**, але врешті-решт вирішив, що документоорієнтована структура MongoDB краще підходить для зберігання сертифікатів, які мають гнучку структуру.

- Для фронтенду я міг би використати **Vue.js** або **Angular**, але обрав React через його популярність, активну спільноту, широку базу компонентів і зручну інтеграцію з іншими бібліотеками.
- У сфері аутентифікації замість **jsonwebtoken** можна було використати **Passport.js**, однак JWT виявився простішим та ефективним для реалізації stateless-аутентифікації через API.
- Для серверної частини можна було використати **NestJS**, який базується на TypeScript і має інший підхід до архітектури, проте Express дав мені більше контролю над структурою та був простішим у реалізації для мого рівня задач.

Загалом, обрані технології утворюють **надійний та перевірений MERN-стек**, який повністю задовольняє потреби мого проекту. Вони забезпечують зручність розробки, хорошу продуктивність, масштабованість, а головне — дають змогу ефективно реалізувати функції цифрової сертифікації з використанням блокчейн-підходу.

## 2.4 Структура бази даних

У розробленій системі цифрової сертифікації використовується база даних **MongoDB**, яка є документо-орієнтованим NoSQL-сховищем. Це рішення було обране з огляду на його гнучкість, масштабованість та природну інтеграцію з JavaScript-орієнтованим стеком. Уся структура бази даних орієнтована на простоту розширення і логічне групування пов'язаних сутностей.

### Основні колекції системи:

#### 1. Users

Ця колекція містить облікові записи всіх зареєстрованих користувачів системи, які мають право створювати та видавати сертифікати. Для кожного користувача зберігається ім'я, електронна пошта, хешований пароль та дата реєстрації. Хешування паролів виконується для забезпечення безпеки персональних даних.



## 2. Certificates

У цій колекції зберігаються всі видані цифрові сертифікати. Кожен документ містить інформацію про курс (назву), опис, особу, якій видано сертифікат (власника), а також дату створення. Окрім цього, вказується автор сертифіката — викладач або організація, яка видала документ. Сертифікати пов'язані з користувачами за допомогою унікального ідентифікатора.

Це дозволяє:

- зберігати історію виданих документів;
- аналізувати активність викладачів;
- реалізовувати механізм пошуку за ім'ям власника або назвою курсу.

## 3. Blocks

Колекція Blocks реалізує логіку блокчейн-структури в межах системи. Кожен блок відповідає одному сертифікату, підтверджуючи факт його створення та незмінність. Основними елементами кожного блоку є:

- ідентифікатор сертифіката, до якого він належить;
- дата створення блоку (timestamp);
- хеш поточного блоку, який генерується на основі внутрішніх даних та попереднього хешу;
- хеш попереднього блоку в ланцюжку.

Цей механізм гарантує послідовність та захист від фальсифікацій: будь-яка зміна в одному блоці призведе до розриву цілісності всього ланцюга. Оскільки блок містить лише службову інформацію — без копіювання вмісту сертифіката — він виконує виключно функцію криптографічного підтвердження достовірності.

### Взаємозв'язки між колекціями

- Кожен сертифікат пов'язаний із власником через унікальний ідентифікатор користувача в полі **issuedTo**.

- Виданий документ також містить інформацію про викладача, який його створив, через поле **issuedBy**.
- Кожен блок у колекції **Blocks** пов'язаний із відповідним сертифікатом через поле **certificateId**, завдяки чому зберігається чітка прив'язка між записом у базі та його криптографічним підтвердженням.

## Переваги обраної структури

1. **Гнучкість** — дозволяє легко додавати нові атрибути до документів без потреби у зміні всієї схеми.
2. **Масштабованість** — кожна колекція функціонує автономно, що забезпечує стабільну роботу при збільшенні обсягів даних.
3. **Безпека** — хешування паролів та збереження сертифікатів у блокчейні гарантують захист від несанкціонованих змін.
4. **Достовірність** — завдяки використанню блокчейн-механізму можна перевірити справжність кожного сертифіката навіть без доступу до всієї бази даних.

## 2.5 Взаємодія компонентів системи

1. **Реєстрація користувача:** через форму на React frontend, дані відправляються на сервер, де пароль хешується з використанням **bcryptjs**, після чого користувач створюється в MongoDB.
2. **Логін:** користувач надсилає email і пароль, які перевіряються на сервері, а у випадку успішної аутентифікації повертається JWT-токен.
3. **Створення сертифікату:** заповнена форма сертифіката відправляється на сервер, де створюється документ у колекції **Certificates**, після чого автоматично генерується новий блок блокчейну.
4. **Зберігання блоків:** кожен блок містить хеш попереднього, тим самим формуючи ланцюжок.

## 2.6 Безпека та авторизація

Для забезпечення безпеки:

- Використовується **JWT** для авторизації запитів.
- Паролі зберігаються у хешованому вигляді за допомогою **bcryptjs**.
- Використовується **CORS** для контролю доступу між доменами.
- Cookies обробляються через **cookie-parser**, що дозволяє зберігати сесійні дані.

## 2.7 Висновки до розділу

Проект цифрової сертифікації був спроектований на основі принципів клієнт-серверної архітектури з чітким розділенням відповідальностей між frontend та backend. Використання стеку MERN (MongoDB, Express, React, Node.js) дало змогу реалізувати ефективну, гнучку та масштабовану систему. Інтеграція блокчейн-структури у вигляді зв'язаних блоків із криптографічними хешами гарантує незмінність даних та прозору перевірку автентичності цифрових сертифікатів. Вибрані технології відповідають сучасним вимогам до безпеки, продуктивності та зручності у розробці, що дозволяє системі бути конкурентоспроможною, безпечною та зручною в користуванні.

## РОЗДІЛ 3. РЕАЛІЗАЦІЯ ПРОЕКТУ

### 3.1 Реалізація серверної частини проекту

У цьому підрозділі детально описано всі основні етапи створення серверної частини вебзастосунку для цифрової сертифікації, що використовує блокчейн для перевірки автентичності виданих сертифікатів. Розробка реалізована з використанням стеку технологій **Node.js + Express + MongoDB**.

#### 3.1.1 Ініціалізація проекту

Розробка серверної частини почалася зі створення Node.js-проекту:

```
npm init -y
```

Основними залежностями, які було встановлено:

express — фреймворк для побудови REST API;

```
npm install express
```

mongoose — ODM для MongoDB;

```
npm install mongoose
```

dotenv — для роботи з конфігураційними змінними;

```
npm install dotenv
```

bcryptjs — для хешування паролів;

```
npm install bcryptjs
```

jsonwebtoken — для генерації та перевірки токенів авторизації;

```
npm install jsonwebtoken
```

cookie-parser — для роботи з куками;

```
npm install cookie-parser
```

cors — для налаштування міждоменої взаємодії.

```
npm install cors
```

#### 3.1.2 Ініціалізація серверу та підключення маршрутів

Файл **app.js** є головним серверним файлом Node.js-додатку, написаного з використанням фреймворку **Express.js**. Його основна мета — налаштування

та запуск сервера, підключення до бази даних, а також підключення маршрутів (routes) для обробки запитів.

У цьому файлі я створив сервер на базі Express для мого проекту. Спочатку я підключив усі необхідні модулі: Express для побудови сервера, CORS для обробки запитів з фронтенду, dotenv для роботи зі змінними середовища, cookie-parser для обробки куки, а також маршрути, які відповідають за автентифікацію, користувачів і сертифікати.

```
{
  const express = require('express');
  const cors = require('cors');
  const dotenv = require('dotenv');
  const cookieParser = require("cookie-parser");
  const userRoutes = require('./routes/user-routes');
  const certificateRoutes = require('./routes/certificate-routes');
  const authRoutes = require("./routes/auth-routes");
}
```

Я використовую функцію connectDB, щоб підключити базу даних, після чого ініціалізую Express-додаток і вказую порт для запуску сервера.

Потім я налаштував проміжні обробники (middleware). Зокрема:

- cookieParser() дозволяє працювати з куками;
- cors налаштований так, щоб дозволити запити з клієнта, який працює на localhost:5173, і передавати куки;
- express.json() дозволяє серверу обробляти вхідні JSON-дані.

```
{
  app.use(cookieParser())
  app.use(cors({
    origin: 'http://localhost:5173',
    credentials: true
  }));
  app.use(express.json());
```

```
}
```

Після цього я підключив маршрути: спочатку для автентифікації, потім для користувачів і сертифікатів.

```
{  
  app.use(authRoutes);  
  app.use(userRoutes);  
  app.use(certificateRoutes);  
}
```

Наприкінці я запускаю сервер і вивожу повідомлення про успішний запуск.

```
{  
  app.listen(PORT, () => console.log(`Server running on port ${PORT}`));  
}
```

### **3.1.3 Реалізація підключення до бази даних MongoDB**

```
{  
  const mongoose = require('mongoose')  
  const connectDB = async () => {  
    try {  
      await mongoose.connect(process.env.MONGO_URI);  
      console.log('MongoDB connected');  
    } catch(err) {  
      console.error(err.message);  
      process.exit(1)  
    }  
  };  
  module.exports = connectDB;  
}
```

У цьому файлі я реалізував функцію підключення до бази даних MongoDB. Для цього я використовую бібліотеку Mongoose.

Функція `connectDB` асинхронна: вона намагається встановити з'єднання з базою за допомогою `URI`, який зберігається в змінних середовища. Якщо підключення проходить успішно — я виводжу повідомлення в консоль. Якщо стається помилка — виводжу її опис і зупиняю процес.

Цю функцію я експортував, щоб можна було викликати її в головному файлі сервера.

### 3.1.4 Реалізація функціоналу серверної частини

У цьому розділі представлено реалізацію ключових функціональних компонентів серверної частини вебсистеми цифрової сертифікації. Сервер побудований із використанням технології `Node.js` та фреймворку `Express`, а для зберігання даних застосовано `MongoDB`.

Основну увагу зосереджено на трьох важливих модулях:

- логіці автентифікації користувачів,
- обробці цифрових сертифікатів та побудові блокчейн-структури для забезпечення їх достовірності,
- а також на роботі з інформацією про зареєстрованих користувачів.

Усі функції написані з урахуванням базових принципів безпеки, включаючи хешування паролів, роботу з токенами доступу та обробку помилок. Наведені нижче підрозділи описують кожен з цих компонентів окремо.

У файлі **`auth-controller.js`** я реалізував основні функції для автентифікації користувачів. Вони обробляють реєстрацію, вхід, отримання профілю та оновлення токена доступу.

Функція **реєстрації** (`createUser`) приймає дані користувача, хешує пароль за допомогою окремої утиліти, створює об'єкт користувача та зберігає його в базі. У разі успіху повертаю об'єкт нового користувача, у разі помилки — обробляю її через загальну функцію `handleError`.

```
{  
  const createUser = async (req, res) => {
```

```
const { name, email, password } = req.body
```

```
try {  
  const hashedPassword = await hashPassword(password);  
  const user = new User({ name, email, password: hashedPassword, })  
  const result = await user.save()  
  res.status(200).json(result)  
} catch (err) { handleError(res, err) }  
};  
}
```

Функція **входу** (loginUser) перевіряє, чи існує користувач з таким email, і порівнює введений пароль з тим, що збережений у базі. Якщо паролі збігаються, я генерую токени — access і refresh — і зберігаю refresh токен у cookies. Користувачеві повертаю access токен.

```
{  
  const loginUser = async (req, res) => {  
    const { email, password } = req.body  
  
    try {  
      const user = await User.findOne({ email: email })  
      const isMatch = await bcrypt.compare(password, user.password)  
  
      if (isMatch) {  
        const token = generateAccessToken(user._id)  
        const refreshToken = generateRefreshToken(user._id)  
  
        res.cookie('refreshToken', refreshToken, {  
          httpOnly: true,  
          secure: false,  
          sameSite: 'Lax',  
        })  
      }  
    }  
  }  
}
```



```

        maxAge: 60 * 60 * 1000
    });

    res.status(200).json({ success: true, token:
token,refreshToken:refreshToken})
    }
    else {
        return res.status(200).json({ success: false, message: "Невірний
пароль" });
    }
} catch (err) { handleError(res, err) }
};
}

```

Функція **отримання профілю** (getProfile) перевіряє наявність токена в заголовку запиту, валідує його та повертає дані користувача, якого ідентифікує токен.

```

{
const getProfile = async (req, res) => {
    const authHeader = req.headers.authorization;
    if (!authHeader) return res.status(401).json({ error: 'No token provided' });

    try {
        const decoded = await verifyToken(authHeader)
        const user = await User.findById(decoded.userId)
        res.status(200).json(user)

    } catch (err) {
        res.status(401).json({ error: err });
    }
}
}

```

Функція **оновлення access токена** (refreshToken) перевіряє refresh token у cookies. Якщо він дійсний — генерує новий access token і відправляє його у відповідь.

Ці функції я експортував, щоб використовувати їх у маршрутах автентифікації.

```
{
  const refreshToken = async (req, res) => {
    const token = req.cookies.refreshToken;
    if (!token) return res.status(401).json({ error: 'No refresh token
provided', cookies: req.cookies })

    try {
      const decoded = verifyRefreshToken(token)
      const newAccessToken = generateAccessToken(decoded.userId)
      res.status(200).json({ token: newAccessToken })
    } catch (err) {
      res.status(401).json({ error: 'Invalid refresh token' });
    }
  }
}
```

У файлі **certificate-controller.js** я реалізував логіку для створення, перевірки та отримання цифрових сертифікатів, які зберігаються разом із блоками в системі, побудованій за принципом блокчейну.

Функція **створення сертифіката** (createNewCertificate) приймає дані сертифіката з тіла запиту, створює сертифікат у базі, а потім додає новий блок у ланцюжок. Для створення блоку я використовую SHA-256 хешування: у хеш входять індекс, дані сертифіката та його ідентифікатор. Також зберігається хеш попереднього блоку, що забезпечує послідовність блокчейну. Зрештою, хеш записується у поле blockchainHash сертифіката.

```
{
```

```

const createNewCertificate = async (req, res) => {
  const { title, description, issuedTo, issuedBy } = req.body

  try {
    const certificate = new Certificate({ title, description, issuedTo, issuedBy
  })

    const savedCertificate = await certificate.save()

    const lastBlock = await Block.findOne().sort({ index: -1 })

    const newIndex = lastBlock ? lastBlock.index + 1 : 0;
    const previousHash = lastBlock ? lastBlock.hash : '0'

    const dataToHash = newIndex + savedCertificate.title +
savedCertificate.description + savedCertificate._id.toString()

    const hash =
crypto.createHash('sha256').update(dataToHash).digest('hex');

    const newBlock = new Block({
      index: newIndex,
      certificateId: savedCertificate._id,
      previousHash,
      hash
    });
    await newBlock.save()

    savedCertificate.blockchainHash = hash;
    await savedCertificate.save()

    res.status(200).json({ certificate: savedCertificate, block: newBlock })
  }
}

```

```

    } catch (err) { handleError(res, err) }
  }
}

```

Функція **перегляду всіх сертифікатів** (getCertificates) дозволяє отримати список сертифікатів, які або видані певному користувачу, або створені ним. Це визначається через параметри запити.

Функція **перевірки одного сертифіката** (getOneCertificate) знаходить сертифікат та відповідний йому блок, повторно обчислює хеш з даних і порівнює його з тим, що збережений у блоці. Якщо хеш збігається — сертифікат вважається справжнім.

```

{
  const getOneCertificate = async (req, res) => {
    const { _id } = req.body

    try {
      let isChecked = false
      const certificate = await Certificate.findById(_id)
      const block = await Block.findOne({certificateId: _id})
      const dataToHash = block.index + certificate.title + certificate.description
+ certificate._id.toString()

      const hash =
crypto.createHash('sha256').update(dataToHash).digest('hex');

      if(block.hash == hash){
        isChecked = true
      }

      res.status(200).json({certificate:certificate,isChecked:isChecked})
    } catch (err) { handleError(res, err) }
  };
}

```

```
}
```

У файлі **user-controller.js** я реалізував функції для отримання інформації про користувачів з бази даних.

Функція **отримання всіх користувачів** (`getUsers`) виконує запит до бази даних і повертає повний список користувачів у форматі JSON. У випадку помилки вона обробляється стандартною функцією `handleError`.

Функція **отримання одного користувача** (`getUser`) отримує ідентифікатор з параметрів запиту та шукає користувача за цим ID.

### 3.1.5 Налаштування маршрутів серверної частини

У цьому розділі описано налаштування маршрутів для серверної частини вебзастосунку. Всі маршрути реалізовані за допомогою Express Router, що дозволяє логічно структурувати шляхи запитів до відповідних функціональних модулів.

Розділ охоплює три основні групи маршрутів:

- маршрути для автентифікації та авторизації користувачів;
- маршрути для роботи з цифровими сертифікатами;
- маршрути для отримання інформації про користувачів.

Кожен файл відповідає за окремий функціональний напрямок і підключає відповідні контролери. Такий підхід забезпечує модульність, зрозумілу структуру коду і спрощує підтримку та масштабування системи.

У файлі **auth-routes.js** я створив маршрути для обробки запитів, пов'язаних з автентифікацією користувачів. Для цього я використовую Express Router, що дозволяє логічно структурувати серверну частину проекту.

```
{  
  const express = require('express');  
  const {  
    createUser,  
    loginUser,  
    getProfile,
```

```

    refreshToken
  } = require('../controllers/auth-controller')
  const router = express.Router();

  router.post('/users', createUser);

  router.post('/users/login', loginUser);

  router.get('/users/auth/me', getProfile);

  router.post('/users/refresh', refreshToken)

  module.exports = router;
}

```

- Маршрут **POST /users** відповідає за реєстрацію нового користувача — він викликає функцію `createUser`.
- Маршрут **POST /users/login** дозволяє користувачу увійти в систему, передаючи дані у функцію `loginUser`.
- Маршрут **GET /users/auth/me** повертає інформацію про автентифікованого користувача за `access` токеном.
- Маршрут **POST /users/refresh** призначений для оновлення `access` токена за допомогою `refresh` токена.

Ці маршрути я експортував, щоб використовувати в основному файлі сервера.

У файлі **certificate-routes.js** я налаштував маршрути для роботи з цифровими сертифікатами. Всі запити обробляються за допомогою `Express Router`, а відповідні функції підключені з контролера `certificate-controller`.

```

{
  const express = require('express');

```

```

    const {getCertificates, createNewCertificate, getOneCertificate} =
require('../controllers/certificate-controller');

const router = express.Router();

router.get('/certificates',getCertificates)

router.post('/certificates',createNewCertificate)

router.post('/certificates/certificate', getOneCertificate)

module.exports = router
}

```

- Маршрут **GET /certificates** повертає список сертифікатів — або виданих користувачу, або створених ним, залежно від параметрів запити.
- Маршрут **POST /certificates** обробляє створення нового сертифіката, а також формує новий блок для блокчейн-структури.
- Маршрут **POST /certificates/certificate** призначений для перевірки автентичності одного конкретного сертифіката.

У файлі **user-routes.js** я налаштував маршрути, що дозволяють отримувати інформацію про користувачів із бази даних. Для цього використовую Express Router і підключаю відповідні функції з контролера `user-controller`.

```

{
const express = require('express');
const {
  getUsers,

```

```

    getOneUser

    } = require('../controllers/user-controller')
    const router = express.Router();

    router.get('/users', getUsers);

    router.get('/users/user', getOneUser)

    module.exports = router;
  }

```

- Маршрут **GET /users** повертає повний список усіх користувачів, що зберігаються в системі.
- Маршрут **GET /users/user** дозволяє отримати дані одного конкретного користувача за його ідентифікатором, переданим через параметри запиту.

Ці маршрути я експортував, щоб додати їх до основного серверного файлу.

### 3.1.6 Реалізація допоміжних функцій безпеки та автентифікації

У цьому розділі розглядаються допоміжні модулі, які забезпечують безпеку та коректну роботу механізмів автентифікації в системі. Зокрема, описано функції для хешування паролів, генерації та перевірки JWT-токенів, а також централізовану обробку помилок.

Такі модулі є фундаментальними для забезпечення захисту даних користувачів, підтримки сесій та стабільної роботи серверної частини. Централізований підхід до обробки помилок і розділення логіки безпеки сприяють підвищенню якості коду, полегшують його підтримку та подальший розвиток.



У файлі **handleError.js** я створив просту універсальну функцію для обробки помилок, яка використовується у всій серверній частині застосунку.

Функція приймає відповідь (res) і об'єкт помилки (err), після чого надсилає клієнту відповідь з кодом помилки 500 та самою помилкою у форматі JSON.

Це дозволяє централізовано обробляти винятки в асинхронних функціях і уникати дублювання коду в кожному маршруті або контролері.

Цю функцію я експортував для використання в інших файлах проекту.

```
{
const handleError = (res, err) => {
  res.status(500).json({ err });
}
module.exports = handleError
}
```

У файлі **hashPassword.js** я реалізував функцію для хешування паролів, яка використовується при реєстрації нових користувачів.

Функція приймає на вхід пароль, перевіряє його на наявність, а потім за допомогою бібліотеки bcryptjs виконує хешування з використанням 10 раундів солі. Це забезпечує захист збережених паролів у базі даних від прямого доступу навіть у разі витоку.

Функція є асинхронною, і я експортував її для подальшого використання в контролерах, пов'язаних із реєстрацією користувачів.

```
{
const bcrypt = require('bcryptjs')
const hashPassword = async (password) => {
  if (!password) throw new Error('Password is required');
  const saltRounds = 10;
  return await bcrypt.hash(password, saltRounds);
};
module.exports = hashPassword;
```

```
}
```

У файлі **jwt.js** я реалізував функції для роботи з JWT-токенами, які використовуються для автентифікації користувачів.

Функції **generateAccessToken** і **generateRefreshToken** створюють токени з ідентифікатором користувача (**userId**), використовуючи секретний ключ з файлу конфігурації. Access токен діє 15 хвилин, а refresh токен — 1 годину.

Функції **verifyToken** і **verifyRefreshToken** перевіряють валідність відповідних токенів. Для **verifyToken** токен отримується з заголовку авторизації, а для **verifyRefreshToken** — безпосередньо з куків або запиту.

Ці функції я експортував, щоб використовувати їх у контролерах автентифікації.

```
{  
  const jwt = require('jsonwebtoken')  
  const dotenv = require('dotenv')  
  dotenv.config()  
  const SECRET = process.env.JWT_SECRET  
  
  const generateAccessToken = (userId) => {  
    return jwt.sign({ userId }, SECRET, { expiresIn: '15m' });  
  };  
  const generateRefreshToken = (userId) => {  
    return jwt.sign({ userId }, SECRET, { expiresIn: '1h' });  
  };  
  const verifyToken = (authHeader) => {  
    const result = jwt.verify(authHeader.split(' ')[1], SECRET)  
    return result  
  }  
}
```

```

const verifyRefreshToken = (token)=>{
  const result = jwt.verify(token,SECRET)
  return result
}

module.exports = {
  generateRefreshToken,generateAccessToken,verifyToken,verifyRefreshToken
}

```

### **Висновок до розділу**

У цьому розділі я детально описав реалізацію серверної частини вебзастосунку для цифрової сертифікації, який складається з кількох взаємопов'язаних модулів та компонентів.

Спочатку було забезпечено надійне підключення до бази даних MongoDB, що є основним сховищем для збереження користувачів, сертифікатів та блоків блокчейну. Це дозволило організувати зручний і швидкий доступ до даних.

Далі я реалізував повноцінну систему автентифікації користувачів. Для безпеки паролі хешуються за допомогою бібліотеки bcrypt із використанням солі, що запобігає прямому доступу до реальних паролів у разі компрометації бази. Для керування сесіями застосовуються JWT-токени: короткотривалі access токени для авторизації запитів і refresh токени для оновлення сесії без повторного введення пароля. Всі ці функції інкапсульовані у окремі модулі, що підвищує читабельність і повторне використання коду.

Важливою частиною є організація REST API із чітким розподілом маршрутів, які поділені на логічні блоки: автентифікація, робота з користувачами, управління цифровими сертифікатами. Така структура дозволяє легко масштабувати застосунок і додавати нові функції без порушення існуючої логіки.

Особливу увагу приділено реалізації цифрових сертифікатів із застосуванням блокчейн-технології. При створенні сертифіката формується

новий блок, що містить індекс, хеш попереднього блоку і хеш даних поточного сертифіката. Це дозволяє перевіряти цілісність і автентичність сертифікатів, забезпечуючи захист від підробок і змін.

Також було впроваджено централізований механізм обробки помилок, який стандартизує відповіді сервера при виникненні виняткових ситуацій, полегшуючи діагностику та підтримку.

Загалом, описані в розділі рішення формують надійну, безпечну та масштабовану серверну інфраструктуру, що відповідає сучасним стандартам веб-розробки. Цей фундамент дозволяє інтегрувати серверну частину з фронтендом, підвищує зручність підтримки й розвитку системи цифрової сертифікації, а також гарантує захист даних користувачів та достовірність виданих сертифікатів.

### 3.2 Реалізація клієнтської частини

Клієнтська частина вебплатформи реалізована за допомогою бібліотеки **React**, яка забезпечує побудову динамічного, компонентно-орієнтованого інтерфейсу користувача. React дозволяє реалізовувати швидкий обмін даними між клієнтом і сервером без необхідності перезавантаження сторінки, що позитивно впливає на зручність користування.

Комунікація з серверною частиною здійснюється за допомогою бібліотеки **Axios**, яка забезпечує зручний інтерфейс для надсилання HTTP-запитів до REST API.

#### 3.2.1 Ініціалізація проекту

У терміналі було виконано команду:

```
npm create vite@latest frontend -- --template react
```

Це створює базову структуру React-застосунку з інтегрованим Vite.

```
npm install axios
```

**Axios** — для HTTP-запитів до бекенду;

**Vite** — це інструмент для швидкої та ефективної розробки фронтенду, який забезпечує миттєвий запуск, гаряче оновлення компонентів і швидко збірку проекту.

### 3.2.2 Реалізація маршрутизації та перевірки автентифікації користувача у клієнтській частині.

У файлі **app.jsx** я створив головний компонент мого React-додатку — App. Він відповідає за навігацію між сторінками та за перевірку, чи авторизований користувач.

#### 1. Імпорт

На початку я імпортую всі необхідні компоненти: Routes, Route, Link з react-router, а також сторінки додатку (такі як Home, Login, Register тощо). Також підключаю контекст автентифікації через useAuth і функцію authUser, яка перевіряє токен.

```
{
  import { Routes, Route, Link } from "react-router"
  import { useAuth } from "../context/authContext"
  import { authUser } from "../api/authApi"
  import { useEffect } from 'react'

  import Home from "../pages/Home"
  import Login from "../pages/Login"
  import Register from "../pages/Register"
  import ...

  ...
}
```

#### 2. Перевірка токена

У цьому компоненті я використовую хук useEffect, щоб при **першому завантаженні** мого додатку автоматично перевірити, чи вже є авторизований користувач. У useEffect я перевіряю, чи є токен у localStorage. Якщо токен знайдено і користувач ще не авторизований, я викликаю authUser() — вона звертається до API та повертає інформацію про користувача.

Потім якщо все добре, я зберігаю ці дані в контексті й встановлюю isAuth в true. Якщо ж ні — скидаю ці значення.

```

{
const { isAuth,setIsAuth,setUser } = useAuth()
useEffect(() => {
  const token = localStorage.getItem('token');
  if (token&& !isAuth) {
    const fetchUser = async () => {
      try {
        const userData = await authUser();
        setUser(userData);
        setIsAuth(true);
      } catch (err) {
        console.error('Помилка авторизації:', err);
        setIsAuth(false);
        setUser(null);
      }
    };
    fetchUser();
  } else {
    setIsAuth(false);
    setUser(null);
  }
}, [setIsAuth, setUser]);
}

```

### 3. Маршрутизація

За допомогою Routes я визначаю, яка сторінка повинна відкриватися при відповідному шляху. Наприклад:

- /login відкриває сторінку входу
- /createcertificate — сторінку створення сертифіката

- `/usercertificates/:id` — сторінку перегляду конкретного сертифіката

Це дозволяє легко переміщатися між різними частинами додатку.

```
{
  <Routes>
    <Route path="/" element={<Home></Home>}></Route>
    <Route path="/login" element={<Login></Login>}></Route>
    <Route path="/register" element={<Register></Register>}></Route>
    <Route path="/account" element={<Account></Account>}></Route>
    ...
  </Routes>
}
```

### 3. Розмітка

У функції `App` я повертаю всю основну структуру мого додатку. Вона складається з меню та блоку маршрутів.

Спочатку я створюю верхнє меню з кнопками для переходу між сторінками. У цьому меню я показую посилання на головну, на перевірку сертифіката, а також — залежно від того, авторизований користувач чи ні — або кнопку “Профіль”, або “Вхід / Реєстрація”. Це дозволяє адаптувати інтерфейс під різні стани користувача.

Після цього я повертаю блок `Routes`, де описую всі можливі маршрути мого застосунку. Це означає, що коли користувач переходить на певну адресу (наприклад, `/login` або `/createcertificate`), відповідна сторінка автоматично завантажується без перезавантаження всього сайту.

### 5. Підсумок

Отже, в цьому компоненті я:

- Перевіряю токен при завантаженні сторінки
- Керую станом авторизації
- Відображаю меню, яке змінюється залежно від статусу користувача

- Налаштовую всі маршрути для переходу між сторінками

Це основа мого додатку — тут я з'єднаю все в одне ціле.

### 3.2.3 Реалізація клієнтської логіки авторизації та роботи з сертифікатами

У цьому розділі я описую реалізацію ключових функцій на стороні клієнта, які відповідають за авторизацію користувача та взаємодію з сертифікатами. Було створено набір допоміжних модулів, що забезпечують централізовану та безпечну роботу з API сервера.

Зокрема, у файлі `axiosInstance.js` я налаштував інтерсептори для автоматичного додавання токена до кожного запиту та обробки ситуацій з його протермінуванням. У файлі `api.js` зібрано функції для реєстрації, входу та перевірки автентичності користувача. А в `certificateApi.js` (умовна назва файлу з функціями роботи з сертифікатами) реалізовано запити для створення, отримання та перегляду сертифікатів.

Такий підхід дозволив мені спростити логіку у компонентах інтерфейсу та забезпечити єдину точку взаємодії з бекендом. Це не лише покращує читабельність коду, а й спрощує подальше масштабування та обслуговування системи.

У файлі **`axiosInstance.js`** я створив власний екземпляр `axios`, який автоматично додає токен до кожного запиту, якщо користувач авторизований. Це значно спрощує роботу з API, бо мені не треба вручну додавати заголовок `Authorization` у кожному запиті.

```
{
  axiosInstance.interceptors.request.use(
    (config) => {
      const token = localStorage.getItem("token");
      if (token) {
        config.headers.Authorization = `Bearer ${token}`;
      }
      return config;
    }
  );
}
```



```

    },
    (err) => Promise.reject(err)
  );
  axiosInstance.interceptors.response.use(
    (response) => response,
    async (err) => {
      const originalRequest = err.config;

      if(err.response?.status === 401 && !originalRequest._retry){
        originalRequest._retry = true;

        try {
          const res = await axios.post(`${API_URL}users/refresh`, {}, {
withCredentials: true});

          const newAccessToken = res.data.token
          console.log('Новий токен був створений');
          localStorage.setItem('token',newAccessToken)
          originalRequest.headers.Authorization = `Bearer ${newAccessToken}`;
          return axiosInstance(originalRequest);
        } catch (refreshError) {
          console.error('Помилка оновлення токена:', refreshError);
          localStorage.removeItem('token');
        }
      }
      return Promise.reject(err);
    }
  )
}

```

## 1. Налаштування базової URL-адреси

Я вказав, що всі запити мають відправлятися на сервер за адресою `http://localhost:5000/`, щоб не дублювати цю адресу у кожному запиті.

## 2. Інтерсептор запитів

Я додав перехоплювач запитів (request interceptor), який спрацьовує **перед кожним запитом**. У ньому я:

- перевіряю, чи є токен в localStorage
- якщо є — додаю його до заголовків (Authorization: Bearer ...)

Це потрібно для того, щоб бекенд знав, що користувач авторизований.

### 3. Інтерсептор відповідей

Далі я налаштував перехоплювач відповідей (response interceptor). Він обробляє **відповіді від сервера**, і якщо бачить помилку 401 (тобто "не авторизований"), тоді:

1. Я перевіряю, чи це перший раз, коли запит "впав" (щоб уникнути нескінченного циклу).
2. Якщо так — я відправляю запит на users/refresh, щоб отримати **новий access token**.
3. Якщо отримую новий токен — зберігаю його в localStorage і повторюю той самий запит, який не пройшов.
4. Якщо оновлення не вдається — видаляю токен, і користувач буде змушений залогінитись знову.

### 5. Підсумок

Цей файл допомагає мені автоматизувати дві важливі речі:

- автоматичне додавання токена до кожного запиту
- автоматичне оновлення токена при його закінченні

Завдяки цьому я не думаю про авторизацію в кожному окремому компоненті — усе це обробляється централізовано тут, через axiosInstance.

У файлі **authApi.js** я зібрав основні функції для взаємодії з API, пов'язані з користувачами — реєстрація, вхід і перевірка автентичності. Я використовую налаштований axiosInstance, який автоматично додає токен до кожного запиту, якщо він є в localStorage.

#### 1. loginUser(email, password)

Ця функція відповідає за вхід користувача в систему. Я надсилаю POST-запит на маршрут users/login, передаючи електронну пошту

і пароль. Додатково додаю `{ withCredentials: true }`, щоб сервер міг працювати з cookie (наприклад, для refresh-токена).

Якщо запит успішний — я зберігаю токен у `localStorage` і повертаю дані користувача. Якщо сталася помилка — повертаю `null`.

```
{
  export const loginUser = async (email, password) => {
    try {
      const res = await axiosInstance.post("users/login", { email, password }, {
withCredentials: true });
      localStorage.setItem("token", res.data.token);
      console.log("Token:", res.data.token);
      return res.data;
    } catch (err) {
      console.error("Error logging user:", err);
      return null;
    }
  };
}
```

## **2. createUser(name, email, password)**

Ця функція викликається при реєстрації нового користувача. Я відправляю POST-запит на `users`, передаючи ім'я, пошту і пароль.

Якщо все пройшло успішно — я повертаю відповідь сервера. У разі помилки — виводжу її в консоль і повертаю `null`.

```
{
  export const createUser = async (name, email, password) => {
    try {
      const res = await axiosInstance.post("users", { name, email, password });
      return res.data;
    } catch (err) {
      console.error("Error creating user:", err);
    }
  }
}
```

```

    return null;
  }
};
}

```

### 3. authUser()

Це функція для перевірки, чи користувач ще авторизований. Я відправляю GET-запит на `users/auth/me`, який повертає дані користувача, якщо токен у заголовках дійсний.

Цю функцію я використовую, наприклад, після перезавантаження сторінки — щоб перевірити, чи залишився користувач у системі.

```

{
export const authUser = async () => {
  try {
    const res = await axiosInstance.get("users/auth/me");
    return res.data;
  } catch (err) {
    console.error("Error authenticating user:", err);
    return null;
  }
};
}

```

### 4. Загалом

Цей файл дозволяє мені централізовано працювати з авторизацією і користувачами. Завдяки йому я не дублюю код у компонентах і просто викликаю потрібну функцію: для входу, реєстрації або перевірки автентичності. Це спрощує логіку і робить код більш читабельним.

У файлі **`certificatesApi.js`** я створив набір функцій для роботи з сертифікатами через API.

#### 1. getOneCertificate(id)

Ця функція надсилає POST-запит на бекенд, щоб отримати один сертифікат за його ID.

Я передаю `_id` в `data`, бо сервер очікує його в тілі запиту.

Якщо все проходить успішно — повертаю дані сертифіката. Якщо виникає помилка — вивожу її в консоль і повертаю `null`.

```
{  
  
export const getOneCertificate = async (id) => {  
  
  try {  
  
    const response = await axios({  
  
      url: `${API_URL}certificates/certificate`,  
  
      method: "POST",  
  
      data: { _id: id }  
  
    });  
  
    return response.data;  
  
  } catch (err) {  
  
    console.error("Error fetching user:", err);  
  
    return null;  
  
  }  
  
};  
  
}
```

## 2. `getCertificates(id, isOwn)`

Це функція для отримання списку сертифікатів.

Я використовую GET-запит і передаю id користувача та прапорець isOwn (чи це власні сертифікати, чи отримані від інших) як params, тобто через URL. Так само, як і в інших функціях, у разі помилки повертаю null.

```
{  
  
export const getCertificates = async (id,isOwn) => {  
  
  try {  
  
    const response = await axios({  
  
      url: `${API_URL}certificates`,  
  
      method: "GET",  
  
      params: {id: id,isOwn: isOwn }  
  
    });  
  
    return response.data;  
  
  } catch (err) {  
  
    console.error("Error fetching user:", err);  
  
    return null;  
  
  }  
};}
```

### 3. createCertificate(data)

Ця функція дозволяє створити новий сертифікат.

Я надсилаю POST-запит з даними сертифіката: назвою, описом, ким видано і кому видано. Після створення сертифіката виводжу відповідь сервера в консоль і повертаю її.

```
export const getCertificates = async (id,isOwn) => {  
  
  try {
```

```

const response = await axios({
  url: `${API_URL}certificates`,
  method: "GET",
  params: {id: id,isOwn: isOwn }
});

return response.data;

} catch (err) {

  console.error("Error fetching user:", err);

  return null;

}};

```

#### 4. Загалом

Цей файл — це мій невеликий "API-сервіс" для роботи з сертифікатами. Він дозволяє мені:

- отримати конкретний сертифікат,
- отримати список сертифікатів для користувача,
- створити новий сертифікат.

Завдяки цьому я можу використовувати ці функції в будь-якій частині застосунку, не дублюючи код запитів. Це зручно й підтримує чисту архітектуру.

#### 3.2.4 Розробка користувацького інтерфейсу для роботи з цифровими сертифікатами.

У цьому розділі описується процес створення основних компонентів користувацького інтерфейсу веб-додатку для управління цифровими сертифікатами на основі блокчейн-технологій. Представлено реалізацію

функціоналу реєстрації, авторизації, перегляду особистого кабінету користувача, створення, перевірки та відображення сертифікатів.

Окрему увагу приділено навігації між сторінками та захищеному доступу до персональних даних користувача. Розглянуто механізми асинхронного завантаження інформації через API, обробки помилок і взаємодії з серверною частиною.

Реалізовані компоненти забезпечують інтуїтивний і зручний інтерфейс, що дозволяє користувачам легко створювати, перевіряти, переглядати та управляти цифровими сертифікатами, які зберігаються в блокчейні, забезпечуючи прозорість і безпеку операцій.

Компонент **Home.jsx** — це головна сторінка мого застосунку **BlockCerts**. Вона виконує роль вітальної сторінки, яка знайомить користувача з функціоналом платформи та пропонує перейти далі.

## 1. Логіка переходу

Я використовую хук `useNavigate` з `react-router` для переходу між сторінками, а також `useAuth`, щоб дізнатися, чи користувач авторизований.

У функції `handleNavigate` реалізована перевірка:

- якщо користувач авторизований (`isAuth` дорівнює `true`), я перенаправляю його на сторінку профілю (`/account`);
- якщо ні — на сторінку реєстрації (`/register`).

## 2. Інтерфейс

Компонент містить:

- заголовок із назвою платформи **BlockCerts**;
- опис, який коротко пояснює призначення сервісу — створення, збереження та обмін цифровими сертифікатами на основі блокчейну;



- кнопку з написом "Почати користуватися", яка при натисканні викликає функцію `handleNavigate`.

Компонент також підключає зовнішній CSS-файл `Home.css`, де я визначив стилі для заголовка, опису та кнопки.

```
const navigate = useNavigate()

const {isAuth} = useAuth()

const handleNavigate = ()=>{

  if(isAuth){

    navigate('/account')

  }else{

    navigate('/register')

  }}

```

### 3. Призначення

Цей компонент є першою точкою входу в систему. Його завдання — ознайомити користувача з можливостями платформи та направити його далі відповідно до статусу авторизації. Це дозволяє зробити перехід до роботи з сервісом максимально швидким і зручним.

Компонент **Register.jsx** відповідає за сторінку реєстрації користувача в моєму застосунку.

#### 1. Управління станом

Я використовую React-хуки `useState`, щоб зберігати значення полів форми: ім'я, електронну пошту, пароль, підтвердження пароля, а також повідомлення для користувача.

## 2. Логіка реєстрації

Функція `handleRegister` виконує кілька основних кроків:

- Спершу я викликаю функцію `validateRegister`, яка перевіряє коректність введених даних (наприклад, чи паролі співпадають, чи email має правильний формат).
- Якщо є помилка валідації, я встановлюю відповідне повідомлення і припиняю подальше виконання.
- Якщо валідація пройшла успішно, я викликаю асинхронну функцію `createUser` з API для створення нового користувача.
- Якщо реєстрація пройшла успішно, я виводжу повідомлення про успіх і через 2 секунди автоматично перенаправляю користувача на сторінку входу.
- Якщо сталася помилка — показую повідомлення про помилку.

```
{const navigate = useNavigate();

const handleRegister = async () => {

    const validationError = validateRegister({ name, email, password,
checkPassword });

    if (validationError) {

        setMessage(validationError);

        return;

    }

    const result = await createUser(name, email, password);

    if (result) {
```

```
setMessage("✅ Реєстрація успішна!");

setTimeout(() => {

  navigate("/login");

}, 2000);

} else {

  setMessage("❌ Помилка під час реєстрації. Спробуйте ще раз.");
};}
```

### 3. Відображення

У рендері я створив форму з полями вводу для імені, email, пароля і підтвердження пароля. Кнопка реєстрації запускає `handleRegister`. Під формою я виводжу повідомлення про помилки чи успіх, а також даю посилання на сторінку входу для тих, хто вже має акаунт.

### Підсумок

Цей компонент допомагає мені організувати процес реєстрації просто і зрозуміло. Я контролюю введення користувача, валідацію даних і логіку сповіщень, що покращує користувацький досвід і безпеку.

Компонент **Login.jsx** відповідає за сторінку входу користувача у мій застосунок.

### 1. Управління станом

Я використовую React-хуки `useState` для зберігання значень полів вводу — логіна, пароля — і повідомлення для користувача.

Також застосовую `useAuth` для роботи з глобальним станом автентифікації: отримую `isAuth`, `setIsAuth` і `setUser`.

### 2. Перевірка авторизації

За допомогою `useEffect` я стежу за змінною `isAuth`. Якщо користувач вже авторизований, одразу перенаправляю його на сторінку профілю (`/account`). Це дозволяє уникнути повторного входу.

### 3. Логіка входу

Функція `handleLogin` працює так:

- Викликає асинхронну функцію `loginUser` з API, передаючи логін і пароль.
- Якщо відповідь свідчить про успішний вхід (`result.success`), я додатково підтверджую авторизацію через `authUser`, щоб отримати актуальні дані користувача.
- Після цього оновлюю контекст автентифікації, викликаючи `setUser` та `setIsAuth(true)`.
- Виводжу повідомлення про успішний вхід і через секунду перенаправляю користувача на сторінку профілю.
- Якщо вхід не вдался, показую відповідне повідомлення про помилку.

```
{  useEffect(()=>{  
    if(isAuth){  
      navigate('/account')}}  
  ,[isAuth])  
  
  const handleLogin = async () => {  
    const result = await loginUser(login, password);  
    console.log(result);  
    if (result?.success) {
```

```

const authResult = await authUser(localStorage.getItem('token'))

try {

  setUser(authResult)

  setIsAuth(true)

} catch (err) {console.log(err);}

setMessage("✅ Вхід успішний");

setTimeout(() => {

  navigate('/account')

}, 1000)

} else {

  setMessage("❌ Помилка входу:", result?.message || "Невідома помилка");}}}}

```

#### 4. Відображення

У візуальній частині я зробив форму з двома полями: логін і пароль, а також кнопку, яка запускає `handleLogin`. Під формою відображається повідомлення, якщо воно є.

#### Підсумок

Цей компонент дозволяє користувачу увійти в систему з валідацією на стороні сервера і підтримкою глобального стану автентифікації. Я організував логіку так, щоб забезпечити зручний і безпечний процес входу.

Компонент **Account.jsx** відповідає за сторінку профілю користувача в моєму застосунку.

#### 1. Отримання даних користувача

Я беру з контексту автентифікації об'єкт `user`, а також функції `setIsAuth`, `setUser` і стан `isAuth`. Це дозволяє мені отримати інформацію про поточного користувача та керувати станом автентифікації.

## 2. Навігація

За допомогою хука `useNavigate` я отримую функцію `navigate`, щоб переходити між сторінками.

## 3. Логіка виходу

Функція `handleLogout` виконує:

- Очищення токена з локального сховища браузера.
- Оновлення стану автентифікації — встановлення `isAuth` в `false` і видалення даних користувача (`setUser(null)`).
- Перенаправлення на головну сторінку.

```
{  const handleLogout = () => {  
  
    localStorage.clear('token')  
  
    setIsAuth(false);  
  
    setUser(null);  
  
    navigate('/')  
  
  }}
```

## 4. Навігація по функціях профілю

Я створив дві функції, які перенаправляють користувача:

- `handleNavigateToUS` — переходить до списку сертифікатів користувача.
- `handleNavigateToCC` — переходить на сторінку створення нового сертифікату.

```
{  const handleNavigateToUS = () => {  
    navigate("/usercertificates")  }  
  
const handleNavigateToCC = () => {  
    navigate('/createcertificate')}}}
```

## 5. Відображення

Якщо дані користувача завантажені, я показую ім'я, email і ідентифікатор користувача. Також виводжу кнопки для переходу до сертифікатів, створення сертифікату та виходу з акаунту.

Якщо ж інформація ще не завантажена, виводиться повідомлення “Завантаження...”.

### Підсумок

Цей компонент дає змогу користувачу переглянути свої основні дані, перейти до керування сертифікатами або вийти з системи. Він простий і зручний, і допомагає мені організувати роботу з профілем у застосунку.

Компонент **CheckCertificates.jsx** я зробив для того, щоб користувач міг перевірити дійсність сертифікату за його унікальним ідентифікатором.

### 1. Стан компонента

Я використовую три стани:

- `certificateId` — для збереження введеного користувачем ID сертифікату.
- `certificate` — для збереження інформації про знайдений сертифікат.
- `notFound` — щоб показувати повідомлення, якщо сертифікат не знайдено.

### 2. Перевірка сертифікату

Коли користувач натискає кнопку перевірки, запускається функція `handleCheck`. Вона:

- Викликає API-функцію `getOneCertificate`, передаючи введений ID.
- Якщо сертифікат знайдений, зберігаю його дані у `certificate` та скидаю `notFound`.
- Якщо ні — очищаю сертифікат і ставлю `notFound` у `true`.

```
{  const handleCheck = async () => {  
    const result = await getOneCertificate(certificateId);  
    if (result) {  
        setCertificate(result);  
        setNotFound(false);  
    } else {  
        setCertificate(null);  
        setNotFound(true);  
    }  
  }  
};
```

### 3. Відображення

- Я показую поле вводу для ID сертифікату.
- Кнопку для запуску перевірки.
- Якщо сертифікат знайдено, виводжу деталі: заголовок, опис, дату видачі, хеш.
- Також виводжу повідомлення, чи співпадає хеш із блокчейном.



- Якщо сертифікат не знайдено, показую відповідне повідомлення.

## Підсумок

Цей компонент допомагає швидко і зручно перевірити автентичність цифрового сертифікату, що є важливою частиною мого проекту для підтримки безпеки і довіри користувачів.

Компонент **CreateCertificate.jsx** я створив, щоб користувач міг заповнити форму і створити новий цифровий сертифікат, який буде збережений у системі.

### 1. Початковий стан

Я створив стан form, де зберігаю всі необхідні поля сертифікату:

- заголовок (title)
- опис (description)
- кому видано (issuedTo)
- ким видано (issuedBy)

### 2. Автоматичне встановлення автора сертифікату

Використовуючи `useEffect`, я встановлюю поле `issuedBy` з ID поточного користувача (`user._id`), якщо він авторизований. Це зручно, бо користувачу не потрібно вводити себе вручну.

```
{ useEffect(()=>{  
  if(!isAuth){  
    return }  
  console.log(user);  
  setForm(prev => ( {...prev,issuedBy:user._id } ))
```

```
},[isAuth,user]))}
```

### 3. Обробка введення

Коли користувач вводить дані в форму, функція `handleChange` оновлює відповідне поле у стані `form`.

```
{ const handleChange = (e) => {  
  const { name, value } = e.target;  
  setForm(prev => ({ ...prev, [name]: value }));  
};}
```

### 4. Створення сертифікату

При натисканні на кнопку «Створити» запускається `handleCreateCertificate`, де я передаю поточний стан форми у функцію API `createCertificate`, щоб зберегти сертифікат на сервері.

```
{ const handleCreateCertificate = () => {  
  createCertificate(form) }}}
```

### 5. Відображення

- Якщо дані користувача завантажені, я показую форму з полями для введення.
- Якщо ні — показую повідомлення про завантаження.

### Підсумок

Цей компонент допомагає мені надати користувачам простий і зручний інтерфейс для створення власних сертифікатів, автоматично пов'язуючи їх із автором, що полегшує керування і підвищує безпеку.

Компонент `UserCertificates.jsx` я зробив для зручної навігації користувача між двома категоріями сертифікатів:

## 1. Основна ідея

Я хочу дати користувачу можливість швидко переключатися між переглядом сертифікатів, які він отримав, та тими, які він надіслав іншим.

## 2. Реалізація

- Я використовую два посилання (Link) для навігації:
  - Перше веде на сторінку **Отримані сертифікати**
  - Друге — на сторінку **Надіслані сертифікати**

## 3. Відображення

Всі посилання оформлені як кнопки з класом "button-27", щоб було зрозуміло і зручно для користувача.

```
function UserCertificates() {  
  return (  
    <div className="user-certificates">  
      <Link to={"/usercertificates/receivedcertificates"}  
className="button-27" > 📄 Отримані сертифікати</Link>  
      <Link to={"/usercertificates/sentcertificates"} className="button-  
27"> 📧 Надіслані сертифікати</Link>  
    </div> );}
```

## 4. Підсумок

Таким чином, я створив простий і зрозумілий інтерфейс для користувача, щоб він міг легко керувати своїми сертифікатами за двома напрямками — отримані та надіслані.

Компонент **ReceivedCertificates.jsx** було створено щоб користувач міг переглянути всі сертифікати, які він отримав.

## 1. Реалізація

- Я отримую інформацію про поточного користувача через контекст `useAuth`.
- Потім у `useEffect` запускаю функцію для завантаження сертифікатів з сервера, використовуючи API `getCertificates` з параметром `isOwn = true` (щоб отримати саме отримані сертифікати).

## 2. Робота з даними

- Отримані сертифікати зберігаю у стані `certificates`.
- Якщо сертифікати є, я їх виводжу у вигляді карток.
- Кожна картка містить назву, опис і кнопку "Подробиці", яка веде на сторінку з детальним переглядом цього сертифікату

## 3. Відсутність сертифікатів

- Якщо користувач ще не отримав жодного сертифікату, я виводжу повідомлення про це і даю кнопку повернення на попередню сторінку.

## 4. Навігація

- У кінці сторінки розміщую кнопку "Назад", щоб користувач міг легко повернутися до загального списку категорій сертифікатів.

## Підсумок

Таким чином я організував логіку відображення отриманих сертифікатів і зробив інтерфейс зрозумілим і простим для користувача.

Компонент **SentCertificates.jsx** показує всі сертифікати, які користувач надіслав іншим.

## 1. Як це працює

- Використовую контекст `useAuth`, щоб отримати інформацію про поточного користувача.
- У `useEffect` запускаю функцію, що звертається до API `getCertificates` з параметром `false`, щоб завантажити саме надіслані сертифікати.

## 2. Обробка даних

- Отримані сертифікати зберігаються у стані `certificates`.
- Якщо сертифікати є, виводжу їх у вигляді карток із назвою, описом і кнопкою "Подробиці", що веде на детальну сторінку сертифікату.

## 3. Відсутність даних

- Якщо жодного надісланого сертифікату немає, виводжу відповідне повідомлення та кнопку для повернення до списку категорій.

## 4. Навігація

- У нижній частині сторінки є кнопка "Назад" для повернення до загального розділу користувача сертифікатів.

```
{ useEffect(() => {  
  if (!user?._id) return;  
  
  const fetchCertificates = async () => {  
  
    try {  
  
      const result = await getCertificates(user._id, false);  
  
      console.log(result);  
    }  
  }  
})
```

```

        setCertificates(result); // save result in state if you want to display
it

        } catch (err) {

            console.error("Failed to fetch certificates", err);

        }

    };

    fetchCertificates();

}, [user?._id]);}

```

### Підсумок

Компонент забезпечує зручний перегляд усіх надісланих сертифікатів користувачем і надає простий інтерфейс для навігації.

Компонент **Certificate.jsx** відповідає за відображення детальної інформації про конкретний сертифікат за його унікальним ідентифікатором.

### Як працює компонент:

1. З URL отримуємо параметр `id` за допомогою хука `useParams` з бібліотеки `react-router`.
2. Використовуємо хук `useEffect`, щоб при завантаженні компонента зробити асинхронний виклик API `getOneCertificate`, передаючи `id`.
3. Якщо запит успішний, дані сертифікату зберігаються в локальному стані `certificate`.
4. Якщо дані є, виводимо заголовок, опис, унікальний ідентифікатор і хеш блокчейну сертифікату.
5. Якщо дані ще завантажуються, показуємо повідомлення "Завантаження...".

6. Є кнопка "Назад", яка повертає користувача до загального списку сертифікатів.

### **Підсумок**

Компонент забезпечує простий та зрозумілий інтерфейс для перегляду повної інформації про конкретний сертифікат, забезпечуючи користувача всіма необхідними даними у зручному вигляді.

### **3.2.5 Контекст автентифікації та валідація реєстрації користувача.**

У сучасних веб-додатках безпека та зручність користувача є одними з ключових аспектів розробки. Для ефективного управління станом автентифікації та забезпечення коректного введення даних користувачами застосовуються спеціальні механізми. У цьому розділі розглядаються два важливі компоненти системи аутентифікації:

1. Контекст автентифікації (AuthContext), який забезпечує глобальний доступ до стану користувача та статусу автентифікації в додатку, дозволяючи керувати цими даними у різних компонентах без необхідності передавати пропси вручну.

2. Валідація реєстраційних даних (validateRegister), що відповідає за перевірку коректності інформації, введеної користувачем при реєстрації, включаючи перевірку формату email, правил формування імені користувача та пароля, а також узгодженості паролів.

Ці дві частини відіграють фундаментальну роль у побудові надійної та зручної системи реєстрації та аутентифікації користувачів, забезпечуючи належний контроль даних і підтримку безпеки на рівні клієнтської частини додатку.

### **Контекст аутентифікації (AuthContext.jsx)**

Для управління станом аутентифікації користувача у додатку реалізовано React Context — AuthContext. Він надає доступ до даних користувача (user),

стану авторизації (isAuth), а також функцій оновлення цих значень (setUser, setIsAuth) у будь-якому компоненті дерева. Контекст інкапсулює логіку зберігання та передачі інформації про поточного користувача, що дозволяє централізовано керувати сесією.

Компонент-провайдер AuthProvider обгортає додаток, надаючи спільний доступ до цих даних та методів усім дочірнім компонентам.

```
{const AuthContext = createContext();  
  
export const useAuth = ()=> useContext(AuthContext);  
  
export const AuthProvider = ({children})=>{  
  
  const [user, setUser] = useState(null);  
  
  const [isAuth, setIsAuth] = useState(false);  
  
  const value = {isAuth, user, setUser, setIsAuth}  
  
  return <AuthContext.Provider value={value}>  
  
    {children}  
  
  </AuthContext.Provider>}
```

### Валідація даних реєстрації користувача

Функція validateRegister відповідає за перевірку коректності даних, які вводить користувач під час реєстрації. Вона приймає об'єкт із полями: name, email, password, checkPassword, і послідовно перевіряє їх на відповідність певним правилам:

1. **Заповненість полів** — всі поля мають бути заповнені.
2. **Формат email** — має відповідати стандартній структурі електронної адреси.
3. **Ім'я користувача** — може містити тільки латинські літери, цифри та символи @, ., \_, -.



4. **Пароль** — може містити ті ж символи, що й ім'я, з мінімальною довжиною 6 символів.

5. **Підтвердження пароля** — поле підтвердження пароля має співпадати з паролем.

У разі виявлення помилки функція повертає відповідне повідомлення з описом проблеми. Якщо всі умови виконані, повертається null, що означає відсутність помилок і коректність введених даних.

### **Висновок до розділу**

У даному розділі було розглянуто реалізацію клієнтської частини системи цифрової сертифікації, створеної за допомогою бібліотеки React. Основна мета цього етапу — створити функціональний та зручний інтерфейс користувача, що дозволяє ефективно працювати з цифровими сертифікатами: створювати, переглядати, перевіряти, а також керувати власними сертифікатами.

Особливу увагу було приділено реалізації авторизації. Завдяки контексту авторизації (AuthContext), стан користувача та його права доступу доступні у будь-якому компоненті застосунку. Це значно спрощує контроль доступу та дозволяє забезпечити належну безпеку при роботі з персональними сертифікатами.

Функціонал створення сертифікатів реалізовано у вигляді окремого компонента. Після автентифікації користувач може заповнити форму, вказавши ключові параметри сертифіката, такі як заголовок, опис, отримувач та хеш блоку. При цьому відправник підставляється автоматично на основі даних авторизованого користувача. Це дозволяє уникнути фальсифікації інформації про джерело сертифіката.

Користувач має можливість перевірити достовірність сертифіката за його унікальним ідентифікатором. Такий функціонал забезпечує прозорість

системи та дозволяє перевірити інформацію, збережену в базі, з урахуванням блокчейн-хешу, що підтверджує цілісність запису.

Окремо реалізовані компоненти для перегляду надісланих і отриманих сертифікатів. Залежно від типу доступу (відправник або одержувач), система завантажує відповідні дані та відображає їх у вигляді карток. Кожен сертифікат містить базову інформацію та посилання на сторінку з розширеними деталями.

Загальна архітектура клієнтської частини побудована модульно: кожен компонент відповідає за окрему частину функціоналу, що полегшує супровід, масштабування і подальший розвиток застосунку. Стилі зосереджені в окремих CSS-файлах для кращої ізоляції та підтримки.

Крім того, було реалізовано перевірку введених даних під час реєстрації. Окрема функція валідації аналізує правильність email-адреси, склад пароля та відповідність підтвердження пароля, що сприяє підвищенню безпеки збереження облікових даних.

У підсумку, клієнтська частина системи цифрової сертифікації забезпечує повноцінну взаємодію з користувачем, реалізує ключові функції керування сертифікатами та формує зручну й безпечну платформу для роботи з цифровими документами. Такий підхід дозволяє застосовувати систему в реальних умовах, а також слугує основою для її подальшого вдосконалення.

### **3.3 Тестування**

Під час розробки системи цифрової сертифікації я приділяв велику увагу тестуванню як клієнтської, так і серверної частин, оскільки від їхньої коректної роботи залежить надійність та безпека всієї системи.

У клієнтській частині я вручну перевіряв усі ключові функції: реєстрацію та авторизацію користувачів, створення сертифікатів, перегляд отриманих і надісланих сертифікатів, а також перевірку окремого сертифікату за його ідентифікатором. Особливо важливою була перевірка валідації

введених даних, щоб виключити можливість відправлення некоректної або неповної інформації, яка могла б призвести до помилок на сервері чи порушення цілісності бази даних.

Щодо серверної частини, я проводив тестування API вручну за допомогою інструментів для відправки HTTP-запитів (наприклад, Postman). Перевіряв коректність роботи ендпоінтів для реєстрації, авторизації, створення та отримання сертифікатів. Особливу увагу приділяв перевірці авторизації та аутентифікації користувачів, а також валідації даних на сервері. Це дозволило впевнитися, що сервер не приймає некоректні запити і коректно обробляє помилки.

Інтеграційне тестування полягало у перевірці взаємодії між клієнтською частиною і сервером. Я перевіряв, що дані, які відправляються з клієнта, правильно отримуються і обробляються сервером, а відповіді сервера коректно відображаються в інтерфейсі користувача. Це допомогло виявити та усунути проблеми із синхронізацією даних та обробкою помилок.

Також я вручну тестував систему на надійність під час різних сценаріїв: спроби вводу некоректних даних, повторного надсилання запитів, роботи з токенами авторизації, а також перевіряв, як система поводить себе при відсутності мережевого з'єднання.

Завдяки такому детальному ручному тестуванню я забезпечив стабільність і безпеку системи, що є критично важливим для роботи з чутливими даними та цифровими сертифікатами. Такий підхід дозволив мені вчасно виявляти помилки та усувати їх, підвищуючи якість кінцевого продукту.

## ВИСНОВКИ

У результаті виконаної роботи було створено комплексну систему цифрової сертифікації, яка включає як клієнтську, так і серверну частини, інтегровані між собою для забезпечення ефективної та безпечної роботи з цифровими сертифікатами.

Клієнтська частина реалізована з використанням сучасної бібліотеки React, що дозволило створити зручний і інтуїтивно зрозумілий інтерфейс користувача з функціоналом для створення, перегляду, перевірки та управління сертифікатами. Забезпечена надійна авторизація з контролем доступу на основі контексту, що підвищує безпеку роботи з персональними даними.

Серверна частина, побудована на основі Node.js та MongoDB, забезпечує стабільне збереження і обробку даних, реалізує надійну систему автентифікації користувачів із застосуванням bcrypt та JWT, а також надає REST API для взаємодії клієнта з сервером. Впроваджено механізм формування цифрових сертифікатів на основі блокчейн-технології, що гарантує їхню цілісність та достовірність.

Під час розробки системи приділялась значна увага тестуванню як клієнтської, так і серверної частин, адже від їхньої коректної роботи залежить надійність і безпека всієї системи. У клієнтській частині вручну перевірялися всі ключові функції: реєстрація та авторизація користувачів, створення сертифікатів, перегляд отриманих і надісланих сертифікатів, а також перевірка сертифікатів за їх унікальними ідентифікаторами. Особливу увагу приділяли валідації введених даних, що дозволяло виключити відправлення некоректної чи неповної інформації.

Тестування серверної частини здійснювалось за допомогою інструментів для відправки HTTP-запитів (наприклад, Postman), що дозволило перевірити коректність роботи API, зокрема ендпоінтів реєстрації, авторизації, створення та отримання сертифікатів. Було детально протестовано механізми

автентифікації, авторизації та валідації даних, що гарантує стабільність і безпеку обробки запитів.

Інтеграційне тестування забезпечило перевірку взаємодії клієнтської частини із сервером: коректність передачі даних, обробку відповідей і відображення інформації в інтерфейсі користувача. Виявлені проблеми зі синхронізацією та обробкою помилок були своєчасно усунені.

Окрім цього, було проведено тестування на надійність у різних сценаріях, зокрема при вводі некоректних даних, повторних запитах, роботі з токенами авторизації та відсутності мережевого з'єднання.

Загальна архітектура проекту відповідає сучасним вимогам масштабованості, безпеки та зручності підтримки, що створює міцний фундамент для подальшого розвитку системи та її застосування у реальних умовах.

Таким чином, реалізований проект успішно виконує поставлені завдання і може слугувати ефективним інструментом для цифрової сертифікації, забезпечуючи надійність, прозорість і безпеку роботи з цифровими документами.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Накамото С. Біткойн: однорангова електронна грошова система. – 2008. – URL: <https://bitcoin.org/bitcoin.pdf>
2. Вуд Г. Ethereum: Захищений децентралізований реєстр транзакцій загального призначення. – 2014.
3. Тапскотт Д., Тапскотт А. Революція блокчейну: Як технологія змінює гроші, бізнес і світ. – К.: Наш формат, 2018. – 368 с.
4. Node.js Documentation. – URL: <https://nodejs.org/en/docs/>
5. Express.js Documentation. – URL: <https://expressjs.com/>
6. MongoDB Documentation. – URL: <https://www.mongodb.com/docs/>
7. Mongoose ODM Documentation. – URL: <https://mongoosejs.com/docs/>
8. React Documentation. – URL: <https://reactjs.org/docs/getting-started.html>
9. Vite Documentation. – URL: <https://vitejs.dev/guide/>
10. Axios Documentation. – URL: <https://axios-http.com/docs/intro>
11. bcryptjs NPM Documentation. – URL: <https://www.npmjs.com/package/bcryptjs>
12. jsonwebtoken NPM Documentation. – URL: <https://www.npmjs.com/package/jsonwebtoken>
13. cookie-parser NPM Documentation. – URL: <https://www.npmjs.com/package/cookie-parser>
14. CORS Middleware for Express. – URL: <https://expressjs.com/en/resources/middleware/cors.html>
15. Blockcerts – Відкриті цифрові сертифікати на блокчейні. – URL: <https://github.com/blockchain-certificates>
16. Що таке REST API? // Red Hat Developer. – URL: <https://developers.redhat.com/articles/understanding-rest-api>

17. Об'єктно-орієнтоване програмування в JavaScript // Mozilla MDN. — URL: <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects>
18. Credly — Платформа цифрової сертифікації — URL: <https://www.credly.com>
19. OpenCerts — Відкрита система перевірки освітніх сертифікатів на базі блокчейну — URL: <https://www.opencerts.io>
20. Postman — Документація платформи для тестування API — URL: <https://learning.postman.com/docs/>