

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій
(повне найменування факультету)

Кафедра програмних засобів
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

магістр

(ступінь вищої освіти)

на тему ДОСЛІДЖЕННЯ ТА РЕАЛІЗАЦІЯ ДЕЦЕНТРАЛІЗОВАНОЇ
СИСТЕМИ ІДЕНТИФІКАЦІЇ НА ОСНОВІ БЛОКЧЕЙНУ
ДЛЯ ВЕРИФІКАЦІЇ ДОКУМЕНТІВ
RESEARCH AND IMPLEMENTATION OF A DECENTRALIZED
IDENTIFICATION SYSTEM BASED ON BLOCKCHAIN
FOR DOCUMENT VERIFICATION

Виконав: студент 2 курсу, групи КНТ-124м
Спеціальності 121 Інженерія програмного
(код і найменування спеціальності)
забезпечення

Освітня програма (спеціалізація)
Інженерія програмного забезпечення

ЖМУЦЬКИЙ М.А.

(ПРИЗВИЩЕ та ініціали)

Керівник ОЛІЙНИК А.О.

(ПРИЗВИЩЕ та ініціали)

Рецензент СКРУПСЬКИЙ С.Ю.

(ПРИЗВИЩЕ та ініціали)

2025Форма № 25
 МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
 Національний університет «Запорізька політехніка»
 (повне найменування закладу вищої освіти)

Факультет КНТ
 Кафедра програмних засобів
 Ступінь вищої освіти магістр
 Спеціальність 121 Інженерія програмного забезпечення
 (код і найменування)
 Освітня програма (спеціалізація) Інженерія програмного забезпечення
 (назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
 “ ” 2025 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ(РОБОТУ) СТУДЕНТА(КИ)

ЖМУЦЬКОГО Максима Анатолійовича

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Дослідження та реалізація децентралізованої системи ідентифікації на основі блокчейну для верифікації документів. Research and Implementation of a Decentralized Identification System Based on Blockchain for Document Verification

керівник проєкту (роботи) д.т.н., професор, ОЛІЙНИК Андрій Олександрович,
 (науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “30” вересня 2025 року № 447

2. Строк подання студентом проєкту (роботи) _____

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз проблеми та постановка завдань дослідження. 2. Матеріали і методи. 3. Опис програми. 4. Експлуатація, тестування та експериментальне дослідження програми.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)

Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	ОЛІЙНИК А.О., проф.		
Нормоконтроль	ДЕЙНЕГА Л.Ю., старший викладач		

7. Дата видачі завдання «__»_____ року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	1 тиждень	Розділ 1
3	Вибір мови програмування та інших технологій розробки.	2 тиждень	Розділ 2
4	Розробка архітектури програми.	3 тиждень	Розділ 3
5	Розробка програми.	4-7 тижні	Розділ 3, 4
6	Тестування та експериментальне дослідження програмного забезпечення.	8 тиждень	Розділ 4
7	Оформлення пояснювальної записки та документів до неї.	9-10 тиждень	Додатки
8	Нормоконтроль та рецензування.	11 тиждень	
9	Захист роботи.	12 тиждень	

Студент(ка)

(підпис) Максим ЖМУЦЬКИЙ
(Ім'я ПРИЗВИЩЕ)

Керівник проєкту (роботи)

(підпис) Андрій ОЛІЙНИК
(Ім'я ПРИЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи магістра:
177 с., 10 табл., 21 рис., 3 дод., 36 джерел.

БЛОКЧЕЙН, ДЕЦЕНТРАЛІЗОВАНА ІДЕНТИФІКАЦІЯ,
ВЕРИФІКАЦІЯ, СМАРТ КОНТРАКТИ.

Об'єкт дослідження – процес розробки програмного забезпечення для верифікації документів з використанням блокчейн-технологій.

Предмет дослідження – методи та програмні засоби децентралізованої ідентифікації W3C DID та верифікації документів через криптографічні хеші SHA-256, смарт-контракти Ethereum та IPFS.

Мета роботи – розробка децентралізованої програмної системи верифікації документів для підвищення швидкості перевірки та безпеки від підробки.

Актуальність: традиційна верифікація займає дні-тижні, вразлива до підробки, має єдину точку відмови, порушує приватність користувачів.

Матеріали та засоби: JavaScript/Node.js 18, React 18.2, Solidity 0.8.19, Ethers.js 6.4, IPFS, MongoDB 6.0, Hardhat, MetaMask, SHA-256; ПК: Intel i5-12500H, 16 ГБ RAM.

Результати: розроблено веб-систему з клієнт-серверною архітектурою, смарт-контрактом DocumentRegistry для реєстрації хешів документів, інтеграцією IPFS для зберігання повного вмісту, трьома ролями користувачів (емітент, власник, верифікатор), автентифікацією через MetaMask, механізмом відкликання документів. Проведено модульне тестування (13 тестів), тестування смарт-контракту (12 тестів), інтеграційне тестування (8 сценаріїв) – всі успішно. Верифікація документа: 2-5 секунд з IPFS, <1 секунди з кешу MongoDB. Система протестована в Chrome, Firefox, Edge, Safari.

Висновки: реалізовано повнофункціональну систему з веб-інтерфейсом, швидкою верифікацією через блокчейн, контролем власників над даними згідно SSI, стійкістю до підробки через криптографію, незмінним аудиторським слідом. Система дозволила скоротити час з днів до секунд, знизити вартість, захистити від підробки, децентралізувати зберігання.

Галузь використання: університети для видачі дипломів, роботодавці для перевірки кваліфікацій, медзаклади для сертифікатів, HR-департаменти для швидкого найму.

ABSTRACT

Explanatory note to the master's thesis: 177 pages, 10 tables, 21 figures, 3 appendices, 36 references.

BLOCKCHAIN, DECENTRALIZED IDENTITY, VERIFICATION, SMART CONTRACTS.

Object of research – the process of developing software for document verification using blockchain technologies.

Subject of research – methods and software tools for W3C DID decentralized identification and document verification through SHA-256 cryptographic hashes, Ethereum smart contracts, and IPFS.

Purpose of the work – development of a decentralized software system for document verification to increase verification speed and protection against forgery.

Relevance: traditional verification takes days to weeks, is vulnerable to forgery, has a single point of failure, violates user privacy.

Materials and tools: JavaScript/Node.js 18, React 18.2, Solidity 0.8.19, Ethers.js 6.4, IPFS, MongoDB 6.0, Hardhat, MetaMask, SHA-256; PC: Intel i5-12500H, 16 GB RAM.

Results: developed a web system with client-server architecture, DocumentRegistry smart contract for registering document hashes, IPFS integration for storing full content, three user roles (issuer, holder, verifier), authentication via MetaMask, document revocation mechanism. Conducted unit testing (13 tests), smart contract testing (12 tests), integration testing (8 scenarios) – all successful. Document verification: 2-5 seconds with IPFS, <1 second with MongoDB cache. System tested in Chrome, Firefox, Edge, Safari.

Conclusions: implemented a fully functional system with web interface, fast blockchain-based verification, holder control over data according to SSI principles, resistance to forgery through cryptography, immutable audit trail. The system reduced verification time from days to seconds, decreased costs, protected against forgery, decentralized storage.

Application area: universities for diploma issuance, employers for credential verification, medical institutions for certificates, HR departments for rapid hiring.

ЗМІСТ

Перелік скорочень та умовних позначок.....	10
Вступ.....	11
1 Аналіз предметної області.....	13
1.1 Основні поняття.....	14
1.2 Аналіз існуючих систем верифікації документів.....	16
1.3 Детальний аналіз блокчейн-рішень для верифікації документів.....	23
1.4 Висновки до розділу.....	30
2 Матеріали і методи.....	32
2.1 Вибір засобів та технологій розробки.....	32
2.2 Вибір середовища розробки.....	40
2.3 Вибір та обґрунтування систем зберігання даних.....	42
2.4 Функціональні вимоги до програми.....	45
2.5 Нефункціональні вимоги до системи.....	48
2.6 Висновки за розділом.....	50
3 Опис програми.....	52
3.1 Особливості функціонування системи.....	52
3.2 Реалізація серверної частини застосунку.....	57
3.3 Реалізація клієнтської частини.....	84
3.3 Висновки за розділом.....	96
4 Експлуатація та тестування програми.....	98
4.1 Призначення програми.....	98
4.2 Умови використання системи.....	101
4.3 Експлуатація програми.....	106

4.4 Тестування програми.....	120
Висновки.....	132
Перелік джерел посилання.....	136
Додаток А Технічне завдання.....	141
Додаток Б Текст програми.....	148
Додаток В Слайди презентації.....	169

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧОК

API	– Application Programming Interface;
ABI	– Application Binary Interface;
CID	– Content Identifier;
CSS	– Cascading Style Sheets;
CSV	– Comma-Separated Values;
DID	– Decentralized Identifier;
GDPR	– General Data Protection Regulation;
HTML	– HyperText Markup Language;
HTTP	– HyperText Transfer Protocol;
HTTPS	– HyperText Transfer Protocol Secure;
IPFS	– InterPlanetary File System;
JSON	– JavaScript Object Notation;
JSON-LD	– JavaScript Object Notation for Linked Data;
JWT	– JSON Web Token;
MongoDB	– Document-oriented NoSQL database;
ODM	– Object Document Mapping;
PDF	– Portable Document Format;
REST	– Representational State Transfer;
RPC	– Remote Procedure Call;
SHA	– Secure Hash Algorithm;
SPA	– Single Page Application;
SSI	– Self-Sovereign Identity;
TTL	– Time To Live; UI – User Interface;
URL	– Uniform Resource Locator;
VC	– Verifiable Credentials;
W3C	– World Wide Web Consortium;
XSS	– Cross-Site Scripting.

ВСТУП

Традиційні системи верифікації академічних документів, такі як паперові дипломи з печатками університетів або централізовані електронні бази даних, контролюються окремими навчальними закладами та державними органами і функціонують відповідно до правил, встановлених цими установами. Децентралізовані системи верифікації на основі блокчейн-технології є розподіленими, тобто не контролюються єдиним центральним органом, і перевірка автентичності документів відбувається через криптографічні механізми без необхідності звертання до посередників. Ця унікальна особливість блокчейн-систем разом із незмінністю записів та прозорістю зробила їх перспективними для застосування в освітній сфері, де зростає потреба у швидкій та надійній верифікації кваліфікацій.

Однак традиційні процеси верифікації академічних документів стикаються з численними викликами. Перевірка автентичності диплома може займати від кількох днів до кількох тижнів, що створює значні затримки для випускників при працевлаштуванні або продовженні навчання. Централізовані бази даних університетів є вразливими до злому, втрати даних при припиненні роботи установи та мають обмежену міжнародну сумісність. Крім того, зростаюча проблема підробки академічних документів підриває довіру до системи вищої освіти загалом.

Такі проблеми мають значний вплив на випускників, університети та роботодавців, які працюють з академічними кваліфікаціями. Повільна верифікація може призвести до втрати випускником можливості працевлаштування або вчасного зарахування. Відсутність надійних механізмів перевірки створює ризики для роботодавців, які можуть прийняти на роботу кандидатів з недостовірними документами. Крім того, випускники не мають повного контролю над своїми даними та змушені кожного разу звертатися до університету для підтвердження кваліфікацій.

Таким чином, метою даної дипломної роботи є дослідження технологій блокчейн, децентралізованих ідентифікаторів та криптографічних методів захисту даних, а також програмна реалізація децентралізованої системи верифікації академічних документів на основі Ethereum-сумісного блокчейну, IPFS та стандарту W3C DID. Для досягнення цієї мети було проаналізовано існуючі підходи до верифікації документів, досліджено стандарти W3C Verifiable Credentials та W3C DID, вивчено принципи Self-Sovereign Identity, розроблено смарт-контракт на мові Solidity для незмінного реєстру хешів документів, реалізовано інтеграцію з IPFS для децентралізованого зберігання, створено веб-інтерфейс на React з автентифікацією через MetaMask та проведено комплексне тестування системи.

Робота буде корисна університетам, які прагнуть автоматизувати процес видачі та верифікації дипломів зі зменшенням адміністративних витрат, випускникам, які хочуть мати повний контроль над своїми академічними документами згідно з принципами SSI, роботодавцям та HR-фахівцям, які потребують швидкої перевірки кваліфікацій кандидатів без звернення до університетів, а також розробникам подібних систем для освітньої та інших галузей. Результати дослідження демонструють можливість скорочення часу верифікації з днів до секунд, зниження витрат на адміністративні процеси та підвищення рівня довіри до академічних документів завдяки криптографічному захисту через SHA-256 хешування та цифрові підписи ECDSA, які використовуються в Ethereum для автентифікації.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Актуальність та проблематика роботи полягає в тому, що традиційні централізовані системи верифікації документів стикаються з критичними викликами в умовах глобалізації та цифровізації освітніх процесів.

Проблема фальсифікації академічних документів набула серйозних масштабів. За даними дослідження UNESCO (2023), проблема незаконних дипломів та акредитацій загрожує цілісності систем вищої освіти в усьому світі, особливо в контексті транскордонної мобільності студентів. Традиційні процеси верифікації є повільними, коштовними та вразливими до шахрайства [1].

Централізовані бази даних залишаються вразливими до витоків інформації. За даними звіту "Data Breach Investigations Report" компанії Verizon за 2023 рік, освітній сектор продовжує зазнавати значних кібератак, при цьому 68% інцидентів пов'язані з викраденням облікових даних [2].

Технологія блокчейн демонструє потенціал для вирішення цих проблем. За даними MIT Media Lab, які запустили проект Blockcerts у 2016 році, блокчейн-верифікація дозволяє миттєво перевіряти автентичність академічних документів без залучення централізованого органу. Однак існуючі рішення часто не враховують вимоги приватності згідно з GDPR та мають проблеми масштабованості [3].

Децентралізована ідентифікація (DID), стандартизована консорціумом W3C у 2022 році, у поєднанні з технологіями підвищення приватності пропонує новий підхід до верифікації документів. Проте практичних досліджень щодо реалізації таких систем з урахуванням реальних технічних обмежень залишається недостатньо [4].

У першому розділі роботи наведено аналіз існуючих рішень для верифікації академічних документів, їх переваг та недоліків, що обґрунтовує необхідність розробки нового підходу на основі децентралізованої ідентифікації та блокчейн-технологій.

1.1 Основні поняття

У контексті дослідження децентралізованої системи ідентифікації на основі блокчейну для верифікації документів необхідно чітко визначити ключові терміни та концепції, які формують теоретичну основу роботи.

Блокчейн (Blockchain) – розподілена база даних, яка зберігає впорядкований ланцюг записів (блоків), з'єднаних за допомогою криптографічних хешів. Кожен блок містить хеш попереднього блоку, мітку часу та дані транзакцій. Основними характеристиками є децентралізація, прозорість, незмінність записів та криптографічна безпека.

Децентралізована ідентифікація (Decentralized Identifier, DID) – новий тип ідентифікатора, який забезпечує верифіковану, самостійно керовану цифрову ідентичність. На відміну від традиційних федеративних ідентифікаторів, DID не залежить від централізованого реєстру або сертифікаційного органу. Стандарт DID затверджений консорціумом W3C у липні 2022 року. Типовий формат: `did:method:specific-identifier` (наприклад, `did:ethr:0x3b0BC51Ab9De1e5B7B6E34E5b960285805C41736`). DID-документ описує суб'єкт, включаючи криптографічні відкриті ключі для автентифікації, та зберігається у форматі JSON-LD [4].

Верифіковані документи (Verifiable Credentials, VC) – криптографічно захищені, стійкі до підробки цифрові документи. Складаються з тверджень (claims) про суб'єкт, метаданих (емітент, дата видачі, термін дії) та криптографічного підпису. Структура включає контекст (@context), тип документа, емітента, суб'єкт документа (credentialSubject) та підпис (proof).

Верифікована презентація (Verifiable Presentation, VP) – спосіб представлення верифікованих документів третій стороні (верифікатору), що дозволяє власнику вибірково розголошувати інформацію, включаючи лише необхідні атрибути для конкретного випадку використання.

Самостійна суверенна ідентифікація (Self-Sovereign Identity, SSI) – концепція цифрової ідентифікації, яка надає користувачам повний контроль над особистими даними без залежності від централізованих органів. Базується на принципах: контроль користувачів над ідентичністю, доступ до власних даних, прозорість систем, мінімізація розголошення, портативність та можливість відкликання.

Смарт-контракт (Smart Contract) – програма, яка виконується на блокчейн-платформі і автоматично виконує умови контракту або угоди. Після розгортання на блокчейні не може бути змінений, що забезпечує незмінність логіки. У Ethereum смарт-контракти пишуться мовою Solidity та компілюються в байт-код Ethereum Virtual Machine (EVM).

IPFS (InterPlanetary File System) – протокол та однорангова мережа для зберігання та обміну даними в розподіленій файловій системі. Використовує адресацію на основі вмісту (content addressing), де кожен файл отримує унікальний криптографічний хеш – ідентифікатор вмісту (CID). Це забезпечує незмінність та дедуплікацію даних.

Консенсус Proof-of-Authority (PoA) – алгоритм досягнення консенсусу, де транзакції валідуються схваленими учасниками (валідаторами) на основі репутації. Забезпечує високу швидкість транзакцій (час блоку 5-15 секунд), низьке споживання енергії та передбачувані витрати, що робить його придатним для приватних блокчейнів в освітньому секторі.

Хеш-функція (Hash Function) – математична функція, яка перетворює дані довільного розміру в рядок фіксованої довжини. Криптографічні хеш-функції (SHA-256) мають властивості: детермінованість, односпрямованість, стійкість до колізій, ефект лавини. У системах верифікації хеші використовуються для створення цифрових відбитків документів, які зберігаються в блокчейні.

Криптографічний підпис (Digital Signature) – криптографічна схема для демонстрації автентичності цифрового документа. Підпис створюється приватним ключем та верифікується відповідним публічним ключем.

Використовується емітентами для автентифікації виданих документів та власниками для доведення володіння документом.

Ці основні поняття формують концептуальну основу для розуміння архітектури та функціонування децентралізованої системи верифікації документів, яка розглядається в даній роботі.

1.2 Аналіз існуючих систем верифікації документів

Верифікація академічних документів є критичною вимогою для освітніх установ, роботодавців та державних органів у сучасному глобалізованому світі. Протягом останніх десятиліть системи верифікації еволюціонували від повністю паперових процесів до складних цифрових рішень, однак кожен підхід має свої особливості, переваги та обмеження, які необхідно детально проаналізувати.

1.2.1 Традиційні паперові системи верифікації

Традиційні системи верифікації академічних документів базуються на фізичних сертифікатах із захисними елементами (водяні знаки, печатки, голографічні елементи) та ручних процесах перевірки. Процес верифікації є багатоетапним: роботодавець ініціює офіційний запит до університету-емітента, після чого адміністративний персонал здійснює пошук записів у архівах або базах даних.

Типовий цикл верифікації займає 3-7 робочих днів для внутрішніх запитів та кілька тижнів для міжнародних, особливо з нотаріальним завіренням або легалізацією через консульські служби. Фінансові витрати становлять \$10-50 за документ для верифікаторів, а для випускників, які потребують верифікації для багатьох роботодавців або міграційних цілей, сукупні витрати можуть сягати сотень доларів [1],[5].

Основною вразливістю є відносна легкість підробки документів. За даними компанії Verifile, близько 34% роботодавців виявляють недостовірну інформацію в резюме кандидатів, включаючи фальшиві академічні документи. Глобальний ринок підроблених дипломів оцінюється у \$7 млрд щорічно. Додатковою проблемою є відсутність стандартизації між країнами та університетами, що створює бар'єри для академічної мобільності та міжнародного визнання кваліфікацій [6].

1.2.2 Централізовані електронні бази даних університетів

З розвитком інформаційних технологій багато університетів перейшли до використання електронних систем управління студентською інформацією. Ці системи, відомі як Student Information Systems або SIS, зберігають цифрові записи про академічні досягнення студентів, включаючи оцінки, пройдені курси, отримані ступені та дати випуску. Деякі прогресивні заклади почали впроваджувати онлайн-портали для верифікації, через які уповноважені користувачі можуть перевіряти автентичність документів.

Електронні системи університетів забезпечують істотне покращення швидкості обробки запитів порівняно з паперовими архівами. Пошук записів у цифровій базі даних займає хвилини замість годин або днів. Деякі університети надають випускникам доступ до їхніх власних електронних транскриптів, які можуть містити унікальні коди верифікації або QR-коди, що дозволяє верифікаторам перевіряти автентичність документа через веб-сайт університету без необхідності прямого контакту з адміністрацією.

Однак централізовані електронні системи успадковують фундаментальну вразливість традиційних підходів – вони створюють єдину точку відмови. Всі дані зберігаються на серверах університету або в хмарних сервісах під контролем єдиного адміністратора. Це робить системи вразливими до різноманітних загроз: кібератак, витоків даних, технічних збоїв обладнання, людських помилок при адмініструванні та навіть умисної модифікації записів

інсайдерами. За даними звіту Verizon Data Breach Investigations Report за 2023 рік, освітній сектор продовжує зазнавати значних кібератак, при цьому шістдесят вісім відсотків інцидентів безпеки пов'язані з викраденням облікових даних, що може призвести до несанкціонованого доступу до студентських записів [2].

Проблема сумісності між різними системами університетів залишається невирішеною. Кожен заклад може використовувати власне програмне забезпечення від різних постачальників: Ellucian Banner, Oracle PeopleSoft, Workday Student або власні розробки. Відсутність єдиних стандартів для форматів даних та API ускладнює автоматизовану верифікацію між установами. Верифікатор, який працює з випускниками різних університетів, змушений адаптуватися до різних інтерфейсів, процедур та форматів документів.

Питання довготривалого зберігання даних також є критичним. Університети існують десятиліттями і століттями, але технологічні платформи змінюються набагато швидше. Міграція даних між різними версіями програмного забезпечення або при зміні постачальника системи створює ризики втрати або пошкодження історичних записів. У випадку закриття університету його електронні архіви можуть стати недоступними, якщо не було створено належних механізмів передачі даних до іншої установи або державного органу.

1.2.3 Централізовані системи-агрегатори

Альтернативою традиційним методам є централізовані системи-агрегатори, такі як National Student Clearinghouse у США та Higher Education Degree Datacheck у Великобританії. Ці системи збирають дані від університетів-учасників у єдину базу даних та надають швидку верифікацію через онлайн-платформу. NSC, наприклад, охоплює понад три тисячі шестисот університетів США, а HEDD працює зі ста тридцятьма вісьма британськими

університетами. Основною перевагою є скорочення часу верифікації з днів до хвилин або години завдяки стандартизованому інтерфейсу та автоматизації процесу, що значно знижує адміністративне навантаження на освітні заклади [7],[8].

Проте централізована природа таких систем створює критичні ризики. Концентрація даних про мільйони студентів в одній організації робить її привабливою ціллю для кібератак та створює єдину точку відмови. Інцидент безпеки NSC у 2020 році, коли несанкціонований доступ потенційно міг призвести до витоку персональної інформації студентів, підкреслює вразливість централізованих систем. Студенти мають обмежений контроль над своїми даними та не завжди отримують повідомлення про кожен запит верифікації. Крім того, географічне обмеження таких систем окремими країнами створює проблеми для міжнародної мобільності випускників [7].

Ці фундаментальні обмеження централізованих агрегаторів – єдина точка відмови, вразливість до витоків даних, відсутність повного контролю користувачів над власною інформацією та обмежене географічне покриття – підкреслюють необхідність децентралізованих рішень на основі блокчейн-технології, які можуть забезпечити безпеку, приватність та глобальну сумісність.

1.2.4 Блокчейн-базовані системи верифікації

У відповідь на обмеження централізованих систем з'явилося кілька ініціатив, які використовують технологію блокчейн для верифікації академічних документів. Найбільш відомим прикладом є Blockcerts – відкритий стандарт для створення, видачі та верифікації документів на основі блокчейну, розроблений MIT Media Lab. Blockcerts використовує публічний блокчейн Bitcoin або Ethereum для закріплення криптографічних хешів виданих документів. Коли університет видає диплом, створюється цифровий документ у форматі JSON з інформацією про кваліфікацію, його хеш записується в

транзакцію блокчейну разом із цифровим підписом емітента. Випускник отримує цифровий файл, який може зберігати у власному гаманці та ділитися з верифікаторами без залучення університету [3].

Основною перевагою Blockcerts є децентралізація довіри – верифікація не залежить від доступності серверів емітента, оскільки вся необхідна інформація доступна через публічний блокчейн. Власник документа має повний контроль над своїми даними та може самостійно ділитися ними без дозволу університету. Університет Нікосії у 2014 році став першим закладом, який почав видавати сертифікати через блокчейн Bitcoin, а MIT у 2017 році почав видавати цифрові дипломи через Blockcerts [9].

Однак впровадження Blockcerts стикається зі значними викликами. Використання публічних блокчейнів створює проблеми масштабованості та вартості – кожна транзакція в мережі Ethereum може коштувати від декількох до десятків доларів залежно від завантаженості мережі. Час підтвердження транзакцій може варіюватися від хвилин до годин. Питання приватності також є проблемою, оскільки інформація в публічному блокчейні залишається назавжди та доступна всім, що створює конфлікти з правом на забуття згідно з GDPR. Користувацький досвід залишається складним для нетехнічних користувачів, які повинні розуміти концепції криптографічних ключів та гаманців.

Альтернативним підходом є European Blockchain Services Infrastructure – ініціатива Європейської Комісії, запущена у 2018 році для створення мережі розподілених вузлів по ЄС. EBSI використовує дозволену блокчейн-мережу з державними вузлами та базується на стандартах W3C для DID та верифікованих документів. Перевагою є інституційна підтримка та фокус на дотриманні GDPR, однак система все ще перебуває на ранніх стадіях впровадження з обмеженою участю університетів та географічним покриттям лише в межах ЄС [10].

Аналіз існуючих блокчейн-рішень виявляє ключові компроміси: публічні блокчейни забезпечують максимальну децентралізацію, але мають

високі витрати та складність використання, тоді як дозволені мережі пропонують кращу продуктивність, але обмежене географічне покриття та меншу децентралізацію.

1.2.5 Порівняльний аналіз та виявлені недоліки

Аналіз розглянутих систем верифікації академічних документів виявляє фундаментальні компроміси між централізацією та децентралізацією, швидкістю та вартістю, безпекою та зручністю використання. У табл. 1.1 наведено ключові характеристики різних типів систем.

Таблиця 1.1 – Порівняльний аналіз систем верифікації академічних документів

Характеристика	Паперові системи	Електронні БД університетів	Централізовані агрегатори (NSC, HEDD)	Публічні блокчейни (Blockcerts)	Консорціумні блокчейни (EBSI)
Час верифікації	3-7 днів	Хвилини-дні	Хвилини-години	10-60 хв	5-15 сек
Вартість	\$10-50	Адмін. витрати	\$2-10	\$5-50+	~\$0.01
Єдина точка відмови	Так	Так	Так	Ні	Частково
Стійкість до підробки	Низька	Середня	Середня	Висока	Висока

Продовження таблиці 1.1

Характеристика	Паперові системи	Електронні БД університетів	Централізовані агрегатори (NSC, HEDD)	Публічні блокчейни (Blockcerts)	Консорціумні блокчейни (EBSI)
Міжнародна сумісність	Обмежена	Обмежена	Національна	Глобальна	Регіональна (ЄС)
Контроль користувача (SSI)	Ні	Ні	Ні	Так	ТАк
Складність використання	Низька	Низька-середня	Низька	Висока	Середня
Механізм відкликання	Так	Так	Так	Обмежений	Так
Вразливість до кібератак	Низька	Висока	Висока	Низька	Середня
Масштабованість	Обмежена	Середня	Висока	Низька	Середня

Традиційні централізовані системи мають спільні критичні недоліки: єдину точку відмови, вразливість до підробки (ринок фальшивих дипломів \$7 млрд щорічно), відсутність контролю користувачів над власними даними та географічні обмеження. Системи-агрегатори NSC та HEDD покращують швидкість верифікації, але успадковують ризики централізації та обмежені національними кордонами.

Публічні блокчейн-рішення (Blockcerts) усувають централізацію, але створюють нові проблеми: високу вартість транзакцій (\$5-50+), повільну верифікацію (10-60 хвилин), складність використання для нетехнічних користувачів та конфлікт з GDPR через незмінність даних.

Консорціумні блокчейни (EBSI) пропонують збалансований підхід з низькою вартістю (~\$0.01), швидкою верифікацією (5-15 секунд) та підтримкою стандартів W3C DID, але обмежені регіональним покриттям ЄС та ранньою стадією впровадження.

Аналіз виявив п'ять ключових проблем: економічну неефективність (\$10-50 за верифікацію), недостатню масштабованість (5-30 транзакцій/сек), відсутність стандартизованих механізмів відкликання, складний користувацький досвід та невирішений компроміс між децентралізацією та продуктивністю.

Ці обмеження обґрунтовують необхідність гібридного підходу. Запропонована система на основі приватного Ethereum-сумісного блокчейну (Ganache) та IPFS має забезпечити: час верифікації <5 секунд, вартість транзакції <\$0.01, інтуїтивний веб-інтерфейс на React, вбудований механізм відкликання через смарт-контракти та підтримку стандартів W3C DID для міжнародної сумісності.

1.3 Детальний аналіз блокчейн-рішень для верифікації документів

Технологія блокчейн привернула значну увагу освітньої спільноти як потенційне рішення для модернізації систем верифікації академічних документів. На відміну від традиційних централізованих підходів, блокчейн пропонує децентралізовану, незмінну та прозору інфраструктуру для зберігання та верифікації даних. Протягом останнього десятиліття кілька університетів та організацій розробили та впровадили пілотні проекти, які демонструють практичну застосовність цієї технології в освітньому контексті.

1.3.1 Архітектура та функціональність Blockcerts

Blockcerts є одним із найбільш зрілих відкритих стандартів для видачі та верифікації цифрових академічних документів на основі блокчейну. Проект було розпочато MIT Media Lab у 2016 році з метою створення глобальної інфраструктури для верифікації освітніх кваліфікацій, яка не залежить від централізованих органів [3].

Архітектура Blockcerts складається з трьох компонентів: емітент (університет), власник (випускник) та верифікатор (роботодавець). Емітент створює цифровий документ у форматі JSON-LD, який відповідає специфікації W3C Verifiable Credentials. Ключовою особливістю є використання криптографічного хешування SHA-256 для забезпечення цілісності документів – навіть найменша зміна призводить до іншого хешу, що робить підробку математично виявленою. Хеш разом із цифровим підписом емітента записується в транзакцію публічного блокчейну Bitcoin або Ethereum [11].

Випускник отримує документ у форматі JSON з інформацією про кваліфікацію, посиланням на блокчейн-транзакцію та цифровим підписом емітента. Процес верифікації є повністю автоматизованим: система обчислює хеш поданого документа, отримує записаний хеш із блокчейну, порівнює їх та верифікує підпис емітента. Весь процес займає секунди без участі університету [3].

Основною перевагою є справжня децентралізація – верифікація не залежить від доступності серверів університету, оскільки публічний блокчейн підтримується тисячами незалежних вузлів по всьому світу. Незмінність блокчейну означає, що запис про видачу документа не може бути змінений або видалений, що створює надійний аудиторський слід.

Однак Blockcerts має значні обмеження. Використання публічних блокчейнів створює проблему вартості – комісія може коливатися від кількох до десятків доларів, що для університету з тисячами випускників щорічно стає суттєвим фінансовим тягарем. Час підтвердження також проблематичний: у

Bitcoin новий блок створюється кожні 10 хвилин, для надійного підтвердження потрібно чекати кілька блоків (годину або більше). В Ethereum з Proof-of-Stake ситуація покращилася до 12 секунд на блок, але залежність від мережевих умов залишається [12],[13].

Питання приватності є викликом – публічність блокчейну може створювати ризики, якщо метадані містять ідентифікуючу інформацію. Незмінність блокчейну конфліктує з правом на забуття згідно з GDPR. Користувацький досвід залишається складним для нетехнічних користувачів, які повинні розуміти концепції криптографічних ключів та блокчейн-гаманців. Втрата приватного ключа означає безповоротну втрату доступу до документів [14].

1.3.2 Практичний досвід Університету Нікосії

Університет Нікосії на Кіпрі є піонером у застосуванні блокчейн-технології для академічних документів. У 2014 році він став першим акредитованим закладом вищої освіти, який почав видавати академічні сертифікати через блокчейн Bitcoin, спочатку для безкоштовного онлайн-курсу "Introduction to Digital Currencies".

Технічна реалізація базувалася на прямому записі хешів документів у блокчейн Bitcoin. Кожен сертифікат отримував унікальний ідентифікатор, а його хеш записувався в транзакцію. Випускники отримували цифровий сертифікат у форматі PDF з QR-кодом, який містив посилання на блокчейн-транзакцію для верифікації. У 2017-2018 роках університет перейшов на стандарт Blockcerts для кращої сумісності з міжнародними ініціативами та розширив використання на офіційні академічні дипломи.

Практичний досвід виявив переваги та виклики. Позитивним результатом стало зменшення запитів на ручну верифікацію та міжнародне визнання як інноватора в освітніх технологіях. Однак багато роботодавців не були знайомі з блокчейн-верифікацією та продовжували запитувати традиційні

документи, що вимагало додаткових освітніх зусиль. Деякі випускники, особливо менш технічно грамотні, мали складнощі зі зберіганням та використанням цифрових документів. Фінансовий аспект також став викликом – сукупні витрати на транзакції Bitcoin для сотень сертифікатів щорічно виявилися помітними, що спонукало університет шукати оптимізацію через групування документів у Merkle trees.

Незважаючи на виклики, досвід Університету Нікосії демонструє технічну можливість та практичну застосовність блокчейн-технології для академічних документів [15].

1.3.3 Інші академічні блокчейн-ініціативи

Крім MIT та Університету Нікосії, кілька інших навчальних закладів експериментували з блокчейн-технологією для верифікації документів.

MIT Media Lab у 2017 році видав цифрові дипломи 111 випускникам через Blockcerts. Пілотний проект дозволив протестувати систему в реальних умовах – результати були переважно позитивними, хоча виявили потребу в покращенні користувацького досвіду та освітній кампанії для роботодавців. Holberton School (Сан-Франциско) впровадила Blockcerts у 2017 році для видачі сертифікатів випускникам технологічних спеціальностей, інтегрувавши верифікацію в кар'єрні сервіси [16].

У Європі кілька університетів приєдналися до ініціативи EBSI, інші розробили власні рішення. Університет Бахрейну експериментував з приватною Ethereum-мережею для зниження транзакційних витрат та збільшення швидкості обробки. В Азії Університет Мельбурна запустив пілотний проект у 2017 році, а сінгапурські освітні установи співпрацюють з урядовою ініціативою Smart Nation для інтеграції блокчейн-верифікації в національну цифрову інфраструктуру [17],[18].

Консорціум університетів Південної Кореї розробив спільну платформу на основі Hyperledger Fabric, що дозволяє учасникам ділитися інфраструктурою, зменшити витрати та зберегти децентралізовані переваги при кращому контролі доступу та продуктивності [19].

Спільною рисою цих проектів є перебування на стадії пілотних програм або обмеженого впровадження. Масштабування до десятків тисяч документів щорічно залишається викликом через технічні обмеження, фінансові витрати та необхідність зміни адміністративних процесів університетів.

1.3.4 Порівняльний аналіз існуючих блокчейн-рішень

Для систематичного розуміння переваг та обмежень різних підходів до блокчейн-верифікації академічних документів наведено порівняння ключових характеристик розглянутих рішень (див. табл. 1.2).

Таблиця 1.2 Порівняння блокчейн-рішень для верифікації академічних документів

Характеристика	Blockcerts (Bitcoin/Ethereum)	EBSI	Приватні Ethereum мережі	Hyperledger Fabric
Тип блокчейну	Публічний	Дозволений консорціумний	Приватний	Дозволений
Вартість транзакції	\$5-\$50+ (залежить від газу)	Низька (~\$0.01)	Мінімальна (~\$0.001)	Мінімальна
Час верифікації	10 хв - 1 год	5-15 сек	3-5 сек	1-3 сек

Продовження таблиці 1.2

Характеристика	Blockcerts (Bitcoin/Ethereum)	EBSI	Приватні Ethereum мережі	Hyperledger Fabric
Децентралізація	Повна	Часткова (державні вузли)	Обмежена (учасники консорціуму)	Обмежена
Сумісність	Глобальна (відкритий стандарт)	ЄС (планується розширення)	Залежить від консорціуму	Залежить від мережі
Приватність	Обмежена (публічний блокчейн)	Висока (контроль доступу)	Висока	Дуже висока
Масштабованість	Низька (5-15 транзакцій на секунду)	Середня (100- 500 транзакцій на секунду)	Висока (1000+ транзакцій на секунду)	Дуже висока (3000+ транзакцій на секунду)
Складність впровадження	Середня	Висока	Середня	Висока
Відкликання документів	Немає стандартного механізму	Підтримується	Підтримується (смарт- контракти)	Підтримується

Аналіз виявляє чіткі компроміси між підходами. Публічні блокчейни (Bitcoin, Ethereum) пропонують найвищу децентралізацію та глобальну доступність, але страждають від високих витрат (\$5-50+), повільної швидкості (10-60 хв) та обмеженої масштабованості (5-30 тр/сек).

Дозволені консорціумні блокчейни (EBSI) забезпечують кращий баланс: низькі витрати (~\$0.01), швидку верифікацію (5-15 сек) та механізми контролю приватності. Однак їхня ефективність обмежена географічним охопленням консорціуму – EBSI працює в межах ЄС, потребуючи додаткових угод для міжнародної сумісності.

Приватні блокчейни та Hyperledger Fabric пропонують найкращу продуктивність (1000-3000+ тр/сек) та найнижчі витрати, але жертвують децентралізацією, повертаючись до моделі консорціуму довірених учасників [19].

Критичною прогалиною є відсутність стандартизованого механізму відкликання документів у публічних блокчейн-рішеннях. Коли диплом потрібно визнати недійсним через академічну нечесність або адміністративну помилку, незмінність блокчейну стає проблемою.

Аналіз існуючих блокчейн-рішень виявляє ключові прогалини, які перешкоджають масовому впровадженню.

Складність для кінцевих користувачів залишається найбільш значущим бар'єром. Концепції приватних ключів, криптографічних гаманців та хешів є незрозумілими для нетехнічних користувачів. Існуючі інтерфейси часто розроблені технічними спеціалістами для технічних спеціалістів. Потрібні інвестиції в UX/UI дизайн для створення інтуїтивних інтерфейсів [20].

Відсутність надійних механізмів відкликання є критичною для академічного контексту, де існують легітимні причини анулювання документів: виявлення академічної нечесності, адміністративні помилки або відкликання акредитації університету. Більшість існуючих рішень не включають цю функціональність або реалізують її непослідовно [21].

Високі транзакційні витрати в публічних блокчейнах роблять масове впровадження економічно непривабливим. Для університету з 5000 випускників щорічно витрати можуть становити \$25,000-\$250,000 лише на транзакційні комісії [22].

Проблеми масштабованості обмежують здатність обробляти велику кількість документів одночасно. Під час пікових періодів видачі документів публічні блокчейни створюють перевантаження мережі, збільшення часу підтвердження та зростання комісій [13].

Недостатня інтеграція з існуючими Student Information Systems є практичним бар'єром. Більшість університетів використовують застарілі системи без готових блокчейн-інтеграцій. Відсутність стандартизованих API уповільнює впровадження [23].

Ці прогалини визначають напрямки вдосконалення: поєднання децентралізованих переваг блокчейну з практичністю та зручністю використання, механізми управління життєвим циклом документів, баланс між витратами та продуктивністю, легка інтеграція з існуючими системами університетів.

1.4 Висновки до розділу

Можна виділити наступні завдання дослідження:

- проаналізувати існуючі системи верифікації академічних документів (традиційні та блокчейн-базовані);
- розробити архітектуру децентралізованої системи верифікації на основі Ethereum та IPFS;
- виконати проектування смарт-контрактів для реєстрації, верифікації та відкликання документів;
- реалізувати механізм криптографічного хешування документів для забезпечення цілісності даних;
- виконати експериментальне дослідження продуктивності системи та порівняння з традиційними підходами.

Розглянувши сучасний стан верифікації академічних документів, було виявлено ключові проблеми, з якими стикаються університети, випускники та роботодавці. Тривалий час обробки (від днів до тижнів), високі витрати (\$10-50

за верифікацію), вразливість до підробки та відсутність міжнародної сумісності порушують питання необхідності інноваційних рішень. Поставлені завдання дослідження спрямовані на створення комплексного підходу до поліпшення процесу верифікації документів, включаючи використання блокчейн-технології для забезпечення безпеки та децентралізованого зберігання IPFS для ефективного управління файлами.

Вибір технологій Ethereum для блокчейн-рівня та IPFS для децентралізованого сховища обґрунтований їх зрілістю, широкою підтримкою спільноти та відповідністю вимогам проекту. Використання приватної Ethereum мережі (Ganache) дозволяє мінімізувати транзакційні витрати та забезпечити швидку верифікацію документів.

Порівняльний аналіз існуючих рішень, таких як Blockcerts (MIT), система Університету Нікосії та European Blockchain Services Infrastructure (EBSI), дозволяє виділити переваги системи, що розробляється. Інтегрований механізм відкликання документів через смарт-контракти та використання приватної блокчейн-мережі для зниження витрат ставить розроблюване програмне забезпечення в ряд практичних рішень. Ключові особливості, такі як цільовий час верифікації < 5 секунд, веб-інтерфейс на React та можливість обробки до 50-100 паралельних запитів у тестовому середовищі, мають забезпечити оптимізований досвід для користувачів системи.

Отже, розробка системи верифікації, що поєднує блокчейн-технологію, криптографічні хеші для захисту від підробки, децентралізоване зберігання IPFS та зручні веб-інтерфейси, направлена на те, щоб не тільки стати ефективним інструментом верифікації, а й надійним фундаментом для майбутньої глобальної інфраструктури визнання академічних кваліфікацій. Виявлені в ході аналізу прогалини існуючих рішень визначають вектор розвитку системи та формують основу для наступних розділів роботи.

2 МАТЕРІАЛИ І МЕТОДИ

2.1 Вибір засобів та технологій розробки

2.1.1 Вибір мови програмування

Вибір мови програмування для реалізації децентралізованої системи ідентифікації на основі блокчейну передбачає не лише визначення її здатності вирішувати поставлені завдання, але й аналіз її впливу на показники продуктивності, безпеки, масштабованості та складності підтримки майбутнього проєкту. Специфіка роботи з блокчейн-технологіями, криптографічними операціями та веб-інтерфейсами вимагає ретельного підбору технологічного стеку, який забезпечить ефективну взаємодію між клієнтською частиною, серверною логікою та блокчейн-мережею.

Популярність мови програмування забезпечує їй велику спільноту розробників, що є критично важливим для блокчейн-проєктів, оскільки дозволяє при появі складнощів з криптографією, смарт-контрактами або інтеграцією з розподіленими системами звернутися до спільноти для пошуку перевірених рішень. З іншого боку, обрання непідходящого інструменту розробки може призвести до проблем із безпекою криптографічних операцій, неефективної взаємодії з блокчейн-вузлами, затримок у розробці та вразливостей системи. Тому важливість ретельного аналізу потреб та особливостей проєкту перед вибором мови програмування є неабияко доцільною.

У контексті розробки блокчейн-застосунків для верифікації документів найбільш релевантними мовами програмування є JavaScript, Python, Solidity та Go. Для клієнтської частини та серверної логіки розглядаються JavaScript та Python, оскільки вони мають зрілі екосистеми для роботи з блокчейн-мережами. Solidity є стандартною мовою для написання смарт-контрактів на платформі Ethereum та сумісних мережах. Go використовується для розробки високопродуктивних блокчейн-вузлів та серверних компонентів [24].

JavaScript є природним вибором для веб-орієнтованих блокчейн-застосунків завдяки можливості використання єдиної мови як для frontend, так і для backend розробки через Node.js. Екосистема JavaScript включає потужні бібліотеки для роботи з Ethereum (Web3.js, Ethers.js), IPFS (js-ipfs, ipfs-http-client) та криптографічними операціями.

Python відзначається простотою синтаксису та наявністю бібліотек для наукових обчислень, що робить його придатним для реалізації криптографічних доказів з нульовим розголошенням та аналізу даних. Бібліотека Web3.py забезпечує взаємодію з Ethereum-мережами, а py-ipfs-http-client дозволяє працювати з IPFS. Однак для створення повноцінного веб-інтерфейсу Python потребує додаткових фреймворків, що ускладнює архітектуру проєкту.

Solidity є спеціалізованою мовою для написання смарт-контрактів на платформі Ethereum. Вона має C-подібний синтаксис і оптимізована для виконання в Ethereum Virtual Machine. Незважаючи на обмежену область застосування, Solidity є необхідним компонентом будь-якого проєкту на базі Ethereum для реалізації логіки реєстрації документів, управління правами доступу та механізмів відкликання [25].

Go (Golang) використовується в багатьох блокчейн-проєктах, включаючи Hyperledger Fabric та частини інфраструктури Ethereum, завдяки своїй продуктивності, вбудованій підтримці конкурентності та ефективній роботі з мережевими протоколами. Однак для веб-розробки Go має менш розвинуту екосистему порівняно з JavaScript [19].

У табл. 2.1 наведено порівняльну характеристику розглянутих мов програмування відповідно до потреб проєкту децентралізованої системи верифікації документів.

Таблиця 2.1 – Порівняльна характеристика мов програмування

Критерій	JavaScript	Python	Solidity	Go
Підтримка блокчейн-бібліотек	Відмінна (Web3.js, Ethers.js)	Добра (Web3.py)	Нативна для Ethereum	Добра (go-ethereum)
Робота з IPFS	Відмінна (js-ipfs)	Добра (py-ipfs)	Не застосовується	Відмінна (go-ipfs)
Веб-розробка	Нативна підтримка	Потребує фреймворків	Не застосовується	Базова підтримка
Криптографія	Добра (crypto, noble-curves)	Відмінна (cryptography, hashlib)	Обмежена	Відмінна (crypto)
Продуктивність	Середня	Середня	Висока (EVM)	Висока
Спільнота	Дуже велика	Велика	Велика (Ethereum)	Середня
Складність вивчення	Низька-Середня	Низька	Середня	Середня
Екосистема для DID/VC	Відмінна	Добра	Не застосовується	Базова

Аналіз показує, що для реалізації повноцінної системи верифікації документів на основі блокчейну оптимальним є комбінований підхід: використання JavaScript для frontend та backend компонентів завдяки їхній відмінній підтримці блокчейн-технологій та веб-розробки, а Solidity для написання смарт-контрактів.

Такий вибір забезпечує доступ до зрілих бібліотек Ethers.js для взаємодії з блокчейном, ipfs-http-client для роботи з децентралізованим сховищем, did-jwt для реалізації механізмів децентралізованої ідентифікації та можливість використання єдиного технологічного стеку для всіх компонентів застосунку, що спрощує розробку та підтримку проєкту. Node.js забезпечує ефективне виконання асинхронних операцій, критично важливих для взаємодії з блокчейн-мережами та IPFS-вузлами [24].

2.1.2 Вибір фреймворків та бібліотек

Ефективна розробка децентралізованої системи верифікації документів вимагає використання перевірених фреймворків та бібліотек, які прискорюють процес розробки, забезпечують надійність коду та спрощують інтеграцію з блокчейн-технологіями. Правильний вибір інструментів дозволяє зосередитися на бізнес-логіці системи замість реалізації базової функціональності з нуля.

Для розробки клієнтської частини системи було обрано React – популярну JavaScript-бібліотеку для побудови користувацьких інтерфейсів, розроблену компанією Meta (Facebook). React базується на компонентному підході, що дозволяє створювати повторно використовувані UI-елементи та ефективно керувати станом застосунку. Згідно з State of JS 2023, React залишається найбільш використовуваним frontend фреймворком з рівнем задоволеності 83% серед розробників [26].

Ключові переваги React для блокчейн-застосунків включають віртуальний DOM, який оптимізує оновлення інтерфейсу при зміні даних з блокчейну, що важливо при відображенні статусу транзакцій у реальному часі. Хуки (Hooks), зокрема useState та useEffect, забезпечують зручне управління асинхронними операціями взаємодії з блокчейн-вузлами. Велика екосистема бібліотек включає спеціалізовані інструменти для web3-інтеграції, такі як wagmi та web3-react.

Для стилізації інтерфейсу використовується Tailwind CSS – utility-first CSS-фреймворк, який дозволяє швидко створювати адаптивний дизайн без написання власних CSS-класів. Tailwind забезпечує консистентність дизайну та спрощує створення інтерфейсів, які коректно відображаються на різних пристроях.

Для серверної частини обрано Express.js – мінімалістичний та гнучкий веб-фреймворк для Node.js. Express забезпечує базову функціональність для створення REST API, обробки HTTP-запитів, middleware та маршрутизації. Його простота та розширюваність роблять Express ідеальним вибором для backend-сервера, який виступає посередником між frontend-застосунком, блокчейн-мережею, IPFS та MongoDB [27].

Express дозволяє створити API-endpoints для операцій видачі документів, верифікації, пошуку та управління користувачами. Middleware архітектура Express використовується для реалізації автентифікації JWT-токенів, валідації вхідних даних, логування запитів та обробки помилок. Для підвищення безпеки використовуються додаткові пакети: helmet для встановлення HTTP-заголовків безпеки, cors для керування Cross-Origin Resource Sharing, express-rate-limit для захисту від DoS-атак.

Для взаємодії з блокчейн-мережею Ethereum обрано бібліотеку Ethers.js замість альтернативної Web3.js. Ethers.js є сучаснішою бібліотекою з меншим розміром bundle, більш передбачуваним API та активною підтримкою спільноти. Ethers.js забезпечує функціональність для підключення до блокчейн-вузлів через провайдери (JSON-RPC, WebSocket), створення та підписування транзакцій, взаємодії зі смарт-контрактами через ABI (Application Binary Interface), роботи з криптографічними ключами та гаманцями [28].

Основні модулі Ethers.js, що використовуються в проєкті: Provider для читання даних з блокчейну без відправки транзакцій, Signer для підписування та відправки транзакцій від імені користувача, Contract для виклику функцій смарт-контрактів, Utils для роботи з форматами даних (wei/ether конвертація, хешування, кодування).

Для розробки та тестування смарт-контрактів використовується Hardhat – повноцінне середовище розробки для Ethereum. Hardhat надає локальну блокчейн-мережу для тестування (Hardhat Network), інструменти для компіляції Solidity-контрактів, автоматизовані скрипти розгортання, плагіни для тестування та верифікації контрактів. Hardhat Console дозволяє інтерактивно взаємодіяти з розгорнутими контрактами для налагодження [29].

Для реалізації функціональності децентралізованих ідентифікаторів (DID) та верифікованих документів (VC) використовується набір спеціалізованих бібліотек. Бібліотека did-jwt забезпечує створення та верифікацію JWT-токенів, підписаних приватними ключами DID-ідентифікаторів. Це дозволяє випускникам підписувати верифіковані презентації (VP) своїх документів при поділі з верифікаторами [30].

Бібліотека did-resolver надає уніфікований інтерфейс для резолюції DID-ідентифікаторів різних методів (did:ethr, did:web, did:key) у відповідні DID-документи. Етот процес критично важливий для верифікації цифрових підписів емітентів документів. Для методу did:ethr, який використовує Ethereum для закріплення DID-документів, застосовується ethr-did-resolver.

Для роботи з форматом верифікованих документів згідно зі специфікацією W3C використовується бібліотека @digitalbazaar/vc, яка надає функції для створення VC з необхідними полями (issuer, credentialSubject, proof), підписування VC цифровими підписами емітентів, верифікації підписів та структури VC, створення верифікованих презентацій (VP) з множинних VC.

Для інтеграції з децентралізованим сховищем IPFS використовується бібліотека ipfs-http-client, яка надає JavaScript API для взаємодії з IPFS-вузлами через HTTP. Бібліотека дозволяє завантажувати документи у форматі JSON до IPFS та отримувати CID (Content Identifier), завантажувати документи з IPFS за CID, закріплювати (pin) важливі документи для запобігання їх видаленню garbage collector, перевіряти доступність документів у мережі [31].

Для забезпечення надійності доступу до документів використовується сервіс Pinata або Infura IPFS Gateway, які надають гарантоване зберігання та високу доступність файлів без необхідності підтримувати власний повнофункціональний IPFS-вузол.

Для криптографічних операцій, які не покриваються Ethers.js, використовується вбудований модуль Node.js crypto та бібліотека noble-curves для роботи з еліптичними кривими. Для валідації структури даних застосовується joі або уір, що забезпечує перевірку JSON-об'єктів на відповідність схемам перед їх обробкою. Для роботи з датами використовується date-fns як легковагу альтернативу moment.js.

Бібліотека axios забезпечує зручні HTTP-запити з підтримкою interceptors для автоматичного додавання заголовків автентифікації. Для логування подій системи використовується winston, який підтримує різні рівні логування та транспорти (файли, консоль, зовнішні сервіси). Для тестування застосовуються Jest для unit-тестів та Chai з Mocha для тестування смарт-контрактів.

Обраний набір фреймворків та бібліотек формує зрілу та перевірену технологічну основу для розробки децентралізованої системи верифікації документів, забезпечуючи баланс між функціональністю, продуктивністю та складністю підтримки.

2.1.3 Вибір блокчейн-платформи

Вибір блокчейн-платформи є критичним рішенням, оскільки від нього залежать вартість операцій, швидкість транзакцій, масштабованість системи та можливості смарт-контрактів.

Основними кандидатами для реалізації системи є Ethereum, Polygon, Hyperledger Fabric та приватні мережі на базі Ethereum. Ethereum забезпечує найвищий рівень децентралізації та найрозвиненішу екосистему інструментів, однак головним недоліком є висока вартість транзакцій, яка може коливатися

від декількох до десятків доларів. Polygon є Layer 2 рішенням з низькими комісіями (менше \$0.01 за транзакцію) при збереженні сумісності з Ethereum. Hyperledger Fabric – приватна платформа для корпоративних застосувань з високою продуктивністю, але складним налаштуванням [19],[32],[33].

Для даного проєкту обрано приватну мережу Ethereum з консенсусом Proof-of-Authority замість публічних мереж. Це рішення обумовлене необхідністю контролю над вартістю транзакцій та швидкістю підтвердження. Приватна мережа забезпечує нульові витрати на gas для операцій, миттєве підтвердження транзакцій, повну сумісність з екосистемою Ethereum та інструментами розробки.

Консенсус Proof-of-Authority обраний через його енергоефективність, детерміновану продуктивність та простоту налаштування для обмеженого числа довірених валідаторів (університетів). Для production-середовища передбачена можливість розгортання на мережі Polygon, яка забезпечує низькі комісії при збереженні сумісності з Ethereum [33].

Для розробки смарт-контрактів обрано Hardhat як основний інструмент замість альтернативного Truffle. Hardhat забезпечує вбудовану локальну блокчейн-мережу, розширену систему плагінів для тестування та детальні повідомлення про помилки. Для тестування використовується комбінація Hardhat Test з Chai та Waffle, для аналізу покриття коду – solidity-coverage, для статичного аналізу вразливостей – Slither [29].

Розгортання системи передбачає етапи: локальна мережа Hardhat для розробки, приватна Ethereum мережа з PoA для тестування та production, опціонально Polygon mainnet для масштабування. Така стратегія мінімізує витрати на розробку та забезпечує гнучкість при виборі остаточної платформи для розгортання.

Обраний технологічний стек забезпечує економічно ефективну та надійну основу для децентралізованої системи верифікації академічних документів з можливістю подальшого масштабування.

2.2 Вибір середовища розробки

Вибір інтегрованого середовища розробки (IDE) є важливим рішенням, яке впливає на продуктивність розробника, якість коду та швидкість реалізації проєкту. Для розробки децентралізованої системи верифікації документів на основі блокчейну необхідне середовище, яке забезпечує ефективну підтримку JavaScript, інтеграцію з інструментами блокчейн-розробки, можливості налагодження смарт-контрактів та зручну роботу з розподіленими системами.

Visual Studio Code (VS Code) було обрано як основне середовище розробки для даного проєкту завдяки його відкритому вихідному коду, розширюваності та широкій підтримці технологій, необхідних для блокчейн-розробки. VS Code розроблений компанією Microsoft та станом на 2024 рік є найпопулярнішим редактором коду серед веб-розробників, з часткою ринку понад 70% згідно з опитуванням Stack Overflow Developer Survey [34].

Ключові переваги VS Code для розробки блокчейн-застосунків включають вбудовану підтримку з інтелектуальним автодоповненням коду (IntelliSense), що критично важливо при роботі зі складними типами даних DID-документів та верифікованих документів. Інтегрований термінал дозволяє виконувати команди для взаємодії з локальними блокчейн-вузлами (Ganache, Hardhat Network) без необхідності переключення між додатками. Вбудована система контролю версій Git забезпечує ефективне управління кодом проєкту.

Екосистема розширень VS Code надає доступ до спеціалізованих інструментів для блокчейн-розробки. Розширення Solidity забезпечує підсвічування синтаксису, автодоповнення та перевірку помилок при написанні смарт-контрактів. Hardhat for Visual Studio Code інтегрує популярний фреймворк для розробки Ethereum-застосунків безпосередньо в редактор. ESLint та Prettier забезпечують автоматичне форматування коду та виявлення потенційних помилок згідно з встановленими правилами, що особливо важливо для підтримки якості коду в проєктах з криптографічними операціями.

Для роботи з децентралізованими ідентифікаторами та верифікованими документами VS Code підтримує JSON Schema validation, що дозволяє перевіряти відповідність структури DID-документів та VC специфікаціям W3C під час розробки. Можливості налагодження дозволяють встановлювати точки зупинки та покроково виконувати код, що взаємодіє з блокчейн-мережею, спрощуючи виявлення помилок у логіці взаємодії з смарт-контрактами.

VS Code також забезпечує ефективну роботу з багатофайловими проєктами через функцію Workspace, що дозволяє одночасно керувати frontend-додатком, backend-сервером, смарт-контрактами та скриптами розгортання в рамках єдиного робочого простору. Вбудована підтримка Remote Development дозволяє підключатися до віддалених серверів або контейнерів Docker, що корисно для тестування застосунку в середовищі, наближеному до production.

Легковагість та швидкість роботи VS Code є додатковими перевагами порівняно з більш важкими IDE, такими як WebStorm або Eclipse. Це особливо важливо при розробці блокчейн-застосунків, де часто необхідно одночасно запускати локальний блокчейн-вузол, IPFS-демон, frontend-сервер розробки та backend-додаток, що створює значне навантаження на систему [34].

Крос-платформність VS Code забезпечує можливість розробки на різних операційних системах (Windows, macOS, Linux) без втрати функціональності, що важливо для командної роботи над проєктом. Активна спільнота та регулярні оновлення гарантують своєчасну підтримку нових версій JavaScript та блокчейн-бібліотек.

Таким чином, Visual Studio Code є оптимальним вибором середовища розробки для реалізації децентралізованої системи верифікації документів, забезпечуючи баланс між функціональністю, продуктивністю та зручністю використання для розробки блокчейн-застосунків на JavaScript.

2.3 Вибір та обґрунтування систем зберігання даних

Вибір системи керування базою даних для децентралізованої системи верифікації документів на основі блокчейну має свою специфіку порівняно з традиційними застосунками. Архітектура системи передбачає гібридний підхід до зберігання даних: критичні дані верифікації (хеші документів, записи відкликання, реєстр емітентів) зберігаються безпосередньо в блокчейні, повний вміст документів розміщується в децентралізованому сховищі IPFS, а додаткові метадані, кеш та операційні дані потребують традиційної бази даних для забезпечення швидкого доступу та ефективних запитів.

Блокчейн як розподілений реєстр забезпечує незмінність та прозорість критичних записів, але має обмеження щодо швидкості запитів, вартості зберігання та типів даних, які можна ефективно зберігати. Транзакції в блокчейні Ethereum коштують газ, а час підтвердження транзакцій може варіюватися від секунд до хвилин залежно від завантаженості мережі. Тому зберігання в блокчейні обмежується лише тими даними, які потребують незмінності та публічної верифікації: криптографічними хешами документів (SHA-256, 32 байти), статусами відкликання (булеві значення) та адресами DID-ідентифікаторів емітентів.

IPFS (InterPlanetary File System) використовується для децентралізованого зберігання повних документів у форматі JSON, що містять верифіковані документи (Verifiable Credentials) згідно зі специфікацією W3C. IPFS забезпечує адресацію на основі вмісту (content addressing), де кожен файл отримує унікальний ідентифікатор вмісту (CID), який криптографічно пов'язаний з хешем документа в блокчейні. Це дозволяє верифікувати цілісність документа, завантаженого з IPFS, шляхом порівняння обчисленого хешу з записом у блокчейні [35].

Однак для забезпечення ефективної роботи веб-застосунку необхідна додаткова база даних для зберігання операційних даних, які не потребують децентралізації: користувацьких сесій, кешу результатів блокчейн-запитів,

тимчасових даних під час процесу видачі документів, індексів для швидкого пошуку документів за різними критеріями, журналів активності системи та аналітичних даних. Зберігання таких даних безпосередньо в блокчейні було б неефективним та дорогим.

Для вибору традиційної СУБД було проаналізовано кілька варіантів: реляційні бази даних (PostgreSQL, MySQL) та документо-орієнтовані NoSQL бази даних (MongoDB). Реляційні бази даних забезпечують сувору структуру даних, підтримку ACID-транзакцій та потужні можливості для складних запитів з об'єднаннями. NoSQL бази даних пропонують гнучкість схеми, горизонтальну масштабованість та природну підтримку JSON-документів.

MongoDB було обрано як основну СУБД для операційних даних системи з кількох причин. По-перше, MongoDB нативно працює з JSON-документами, що ідеально узгоджується з форматом DID-документів, верифікованих документів та даних, які повертаються з IPFS. Це усуває необхідність складного мапінгу між об'єктно-орієнтованими структурами JavaScriptта реляційними таблицями [36].

По-друге, гнучкість схеми MongoDB дозволяє зберігати документи з різною структурою в одній колекції, що корисно для системи верифікації, де різні типи академічних документів (дипломи, сертифікати, транскрипти) можуть мати різні набори атрибутів. Це спрощує розширення системи для підтримки нових типів документів без необхідності міграції схеми бази даних.

По-третє, MongoDB забезпечує високу продуктивність для операцій читання, що критично важливо для системи верифікації, де співвідношення операцій читання до запису зазвичай становить 10:1 або вище. Верифікатори часто перевіряють одні й ті самі документи, тому ефективне кешування результатів блокчейн-запитів у MongoDB значно знижує навантаження на блокчейн-вузли та покращує час відгуку системи.

По-четверте, MongoDB підтримує горизонтальне масштабування через sharding, що дозволяє розподіляти дані між кількома серверами при зростанні навантаження. Це важливо для системи, яка потенційно може обслуговувати

тисячі університетів та мільйони документів. Реплікація MongoDB забезпечує високу доступність через автоматичне перемикання на резервні вузли у випадку відмови основного сервера [36].

По-п'яте, MongoDB має зрілий драйвер для Node.js (mongodb npm package) з підтримкою `async/await`, що природно інтегрується з асинхронною природою блокчейн-операцій у JavaScript. ODM-бібліотека Mongoose надає додатковий рівень абстракції з валідацією схем, `middleware` та іншими зручностями для розробки.

MongoDB використовується в системі для зберігання наступних типів даних: кешу DID-документів, отриманих з блокчейну або IPFS, для прискорення повторних запитів; метаданих документів (назви, дати створення, типи) для швидкого пошуку та фільтрації; користувацьких профілів емітентів та верифікаторів з налаштуваннями та правами доступу; журналів транзакцій та подій системи для аудиту та діагностики; тимчасових даних під час багатокрокових процесів видачі документів; статистики використання системи для моніторингу та оптимізації.

Важливо зазначити, що MongoDB не зберігає жодних критичних даних верифікації, які не продубльовані в блокчейні або IPFS. У випадку повної втрати бази даних MongoDB систему можна відновити, перебудувавши індекси та кеш з блокчейну та IPFS. Це забезпечує додатковий рівень надійності та відповідає принципам децентралізації, де жодна окрема база даних не є єдиною точкою відмови для критичних функцій верифікації.

Альтернативою MongoDB могла б бути PostgreSQL, яка пропонує підтримку JSON-полів через типи `JSONB`, повнотекстовий пошук та сувору типізацію. Однак для даного проєкту переваги гнучкості схеми та природної інтеграції з JSON-екосистемою JavaScript переважають над перевагами реляційної моделі. PostgreSQL могла б бути кращим вибором для систем з більш складними зв'язками між сутностями або де критична підтримка транзакцій з високою ізоляцією, що менш актуально для нашого випадку використання.

Таким чином, архітектура зберігання даних системи використовує три рівні: блокчейн Ethereum для незмінних записів верифікації, IPFS для децентралізованого зберігання повних документів та MongoDB для операційних даних та кешування. Такий гібридний підхід забезпечує баланс між децентралізацією, безпекою, продуктивністю та вартістю зберігання даних.

2.4 Функціональні вимоги до програми

В результаті аналізу предметної області, виконаного у розділі 1, було визначено наявність у системі трьох основних ролей: емітент документів (університет), власник документів (випускник) та верифікатор (роботодавець або інша установа). Кожна роль має специфічний набір функціональних можливостей, які відповідають її потребам у процесі видачі, зберігання та верифікації академічних документів на основі блокчейн-технології.

Всі ролі передбачають базову функціональність автентифікації через Web3-гаманець MetaMask, управління профілем та перегляд історії активності. Автентифікація здійснюється шляхом підписання криптографічного повідомлення приватним ключем користувача, що підтверджує володіння DID-ідентифікатором без необхідності традиційних паролів.

2.4.1 Функціональні вимоги для ролі емітента документів

Роль емітента документів призначена для уповноважених представників університетів або інших установ, які мають право видавати академічні документи. Емітент взаємодіє з наступними сутностями системи: верифіковані документи (Credentials), DID-ідентифікатори, записи в блокчейні та файли в IPFS. Виходячи з цього на рис. 2.1 представлено use-case діаграму функціональних вимог для ролі емітента документів.

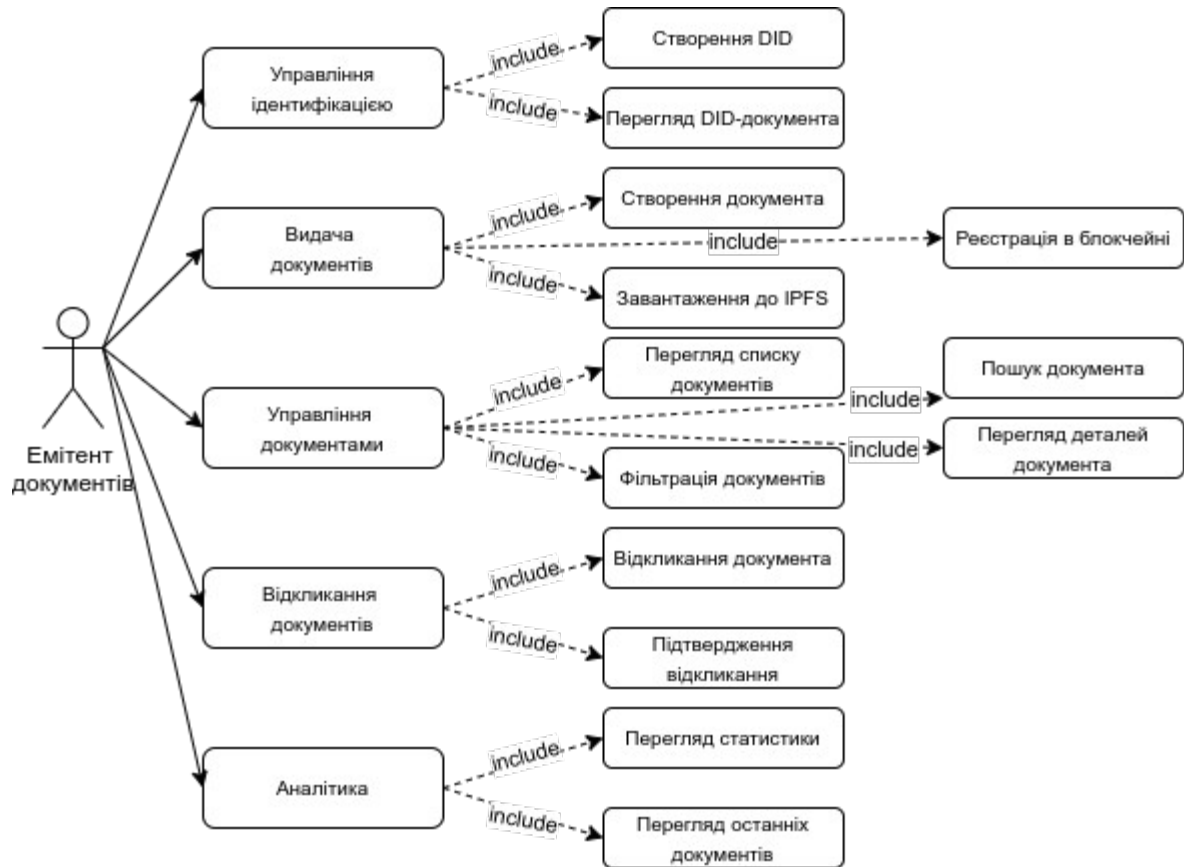


Рисунок 2.1 – Use-case діаграма для ролі емітента документів

2.4.2 Функціональні вимоги для ролі власника документів

Роль власника документів призначена для випускників університетів, які отримали академічні документи та потребують зберігати їх у безпечному середовищі з можливістю контролюваного обміну з верифікаторами. Власник взаємодіє з наступними сутностями: власні документи, персональний DID-ідентифікатор та журнал верифікацій. Виходячи з цього на рис. 2.2 представлено use-case діаграму функціональних вимог для ролі власника документів.

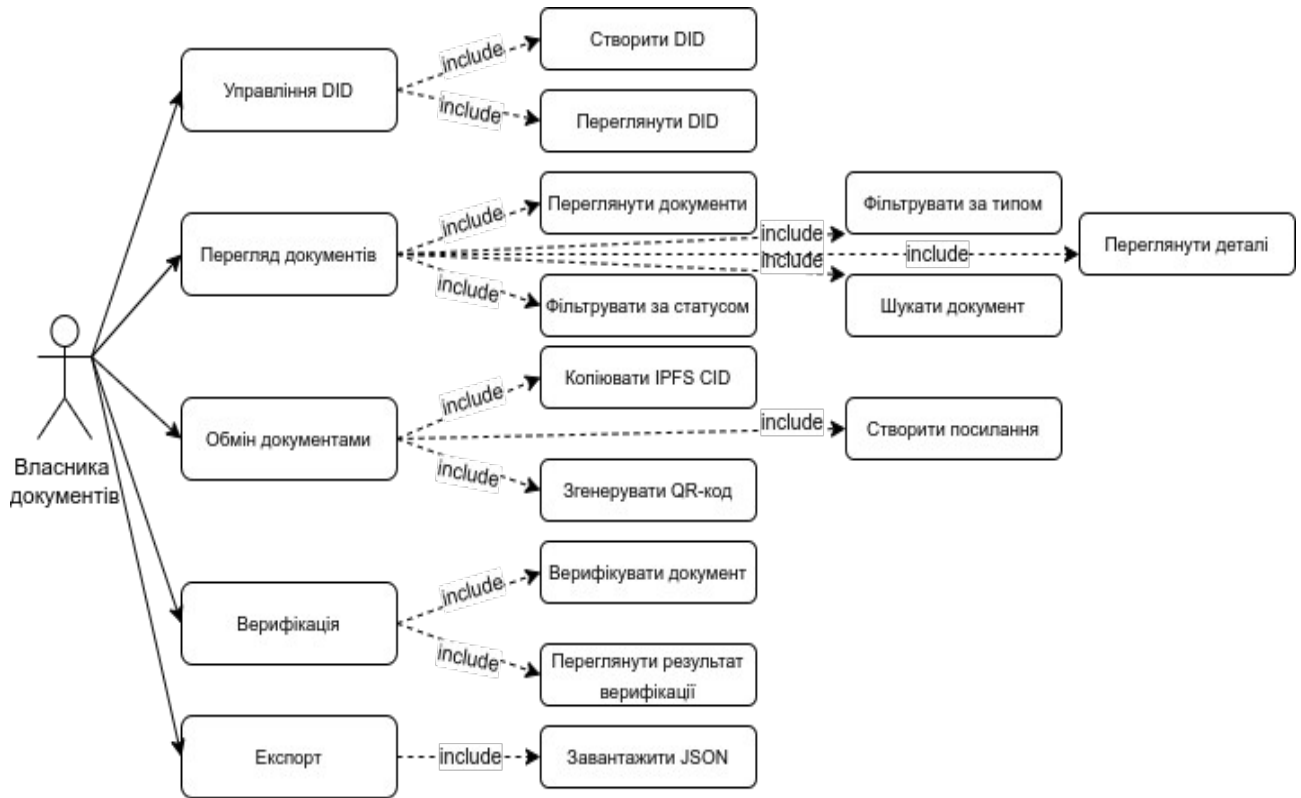


Рисунок 2.2 – Use-case діаграма для ролі власника документів

2.4.3 Функціональні вимоги для ролі верифікатора

Роль верифікатора призначена для роботодавців, рекрутингових агенцій або інших установ, які потребують перевіряти автентичність академічних документів кандидатів. Верифікатор взаємодіє з наступними сутностями: документи для верифікації, результати перевірок та історія верифікацій. Виходячи з цього на рис. 2.3 представлено use-case діаграму функціональних вимог для ролі верифікатора.

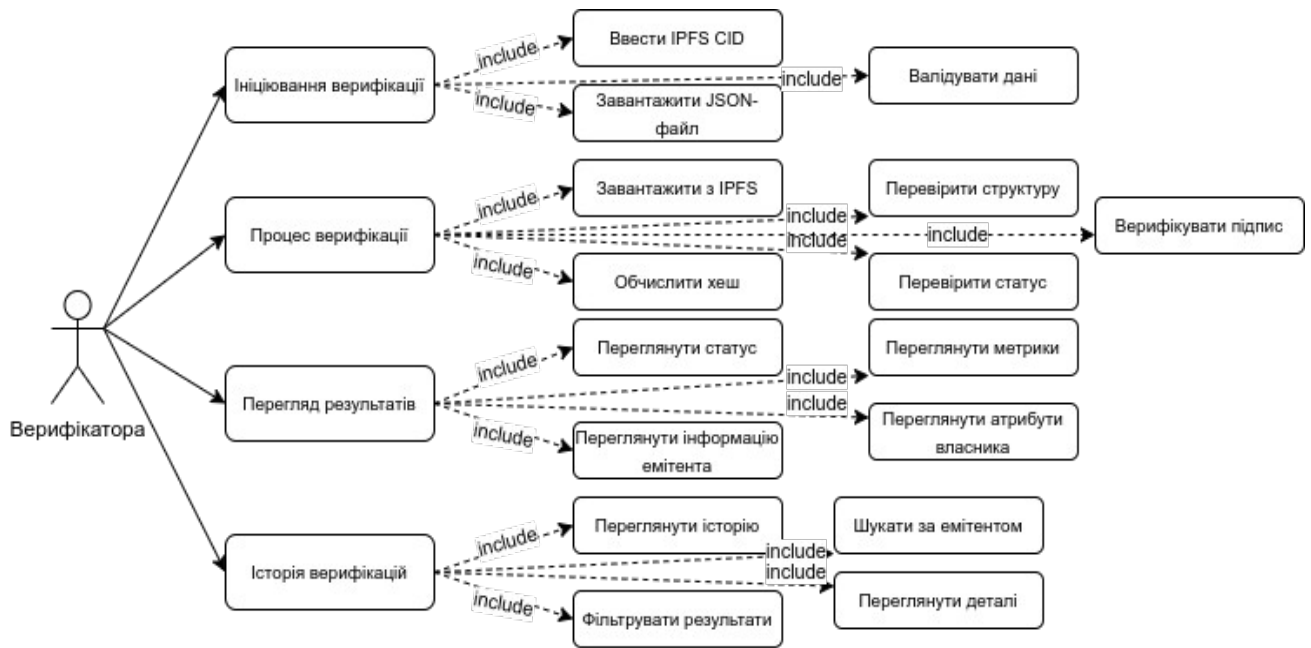


Рисунок 2.3 – Use-case діаграма для ролі верифікатора

2.5 Нефункціональні вимоги до системи

Нефункціональні вимоги визначають якісні характеристики системи, які не стосуються безпосередньо бізнес-логіки, але є критично важливими для успішної експлуатації децентралізованої системи верифікації документів.

Відносно продуктивності система має забезпечувати час верифікації документа не більше п'яти секунд для некешованих документів та менше однієї секунди для кешованих. Час завантаження початкової сторінки веб-застосунку не повинен перевищувати трьох секунд. Операції видачі документа мають завершуватися протягом десяти-п'ятнадцяти секунд залежно від завантаженості мережі.

Відносно масштабованості система має підтримувати одночасну роботу мінімум ста паралельних користувачів без деградації продуктивності через використання асинхронної обробки запитів та connection pooling для MongoDB.

Відносно надійності система має забезпечувати доступність на рівні дев'яносто дев'яти відсотків. Децентралізована природа блокчейну та IPFS забезпечує додатковий рівень надійності через розподілене зберігання критичних даних. Система має коректно обробляти всі помилки та надавати зрозумілі повідомлення користувачам.

Відносно безпеки всі комунікації між клієнтом та сервером мають бути зашифровані через HTTPS. API endpoints захищаються через JWT-токени з терміном дії сім днів. Приватні ключі користувачів залишаються виключно в Web3-гаманці та ніколи не передаються на сервер. Смарт-контракти захищені від типових вразливостей через використання перевірених патернів Solidity. Система включає rate limiting на рівні API з обмеженням до ста запитів на п'ятнадцять хвилин з одного IP.

Відносно відповідності регуляторним вимогам система дотримується основних принципів GDPR через мінімізацію даних у блокчейні (зберігаються лише хеші), вимогу згоди власника перед розголошенням документів, надання власникам повного доступу до їх даних та можливість експорту документів у стандартному форматі W3C VC JSON.

Відносно зручності використання інтерфейс має бути інтуїтивно зрозумілим та не вимагати технічних знань про блокчейн. Складні операції приховані за простими кнопками з зрозумілими мітками. Система надає постійний зворотній зв'язок про статус операцій через індикатори завантаження. Час навчання для базових операцій не повинен перевищувати п'ятнадцяти хвилин.

Відносно сумісності система має коректно працювати в сучасних веб-браузерах Chrome, Firefox, Safari та Edge версії 100 і новіших. Інтерфейс має бути адаптивним та коректно відображатися на екранах різних розмірів від смартфонів до настільних комп'ютерів.

2.6 Висновки за розділом

Таким чином, в даному розділі було визначено основні засоби та інструменти розробки децентралізованої системи верифікації документів на основі блокчейну, проаналізовано альтернативні технології та обґрунтовано вибір оптимального технологічного стеку.

Мовою програмування для розробки обрано JavaScript (Node.js), що забезпечує єдину мову для frontend та backend компонентів і має розвинену екосистему блокчейн-бібліотек. Для клієнтської частини використовується бібліотека React 18.2 у поєднанні з Tailwind CSS для стилізації. Серверна частина побудована на базі фреймворку Express.js, який забезпечує REST API для взаємодії з блокчейном, IPFS та базою даних.

Для розробки смарт-контрактів обрано мову Solidity версії 0.8.19 та середовище розробки Hardhat. Як блокчейн-платформу обрано приватну мережу Ethereum з консенсусом Proof-of-Authority, що забезпечує контроль над вартістю транзакцій та швидкістю підтвердження, з можливістю розгортання на Polygon для production-середовища. Для взаємодії з блокчейном використовується бібліотека Ethers.js версії 6.4.

Роботу з децентралізованим сховищем забезпечує IPFS через бібліотеку ipfs-http-client версії 60.0, що дозволяє зберігати повний вміст документів поза блокчейном з гарантією цілісності через content addressing. Для реалізації децентралізованих ідентифікаторів та верифікованих документів використовуються бібліотеки did-jwt, did-resolver, ethr-did-resolver відповідно до специфікацій W3C.

Для зберігання операційних даних та кешування обрано СУБД MongoDB з ODM-бібліотекою Mongoose, яка забезпечує нативну роботу з JSON-документами, гнучкість схеми та високу продуктивність операцій читання. Структура бази включає п'ять основних колекцій з оптимізованими індексами та TTL-механізмами для автоматичного управління кешем.

Окреслено функціональні вимоги для трьох ролей системи: емітента документів, власника документів та верифікатора. Для кожної ролі сформовано детальний набір функцій, які охоплюють повний життєвий цикл академічних документів від видачі університетом до верифікації роботодавцем, забезпечуючи контроль власника над своїми даними відповідно до принципів самостійної суверенної ідентифікації.

Обраний технологічний стек формує надійну основу для реалізації повнофункціональної децентралізованої системи верифікації академічних документів, яка відповідає вимогам безпеки, масштабованості та зручності використання.

3 ОПИС ПРОГРАМИ

3.1 Особливості функціонування системи

Децентралізована система верифікації академічних документів побудована на основі багаторівневої архітектури, що забезпечує розділення відповідальності між компонентами та дозволяє досягти необхідного балансу між децентралізацією, продуктивністю та зручністю використання. Архітектура системи включає чотири основні рівні: клієнтський додаток (Frontend), серверну логіку (Backend), блокчейн-інфраструктуру та допоміжні сервіси зберігання даних(див.рис. 3.1).

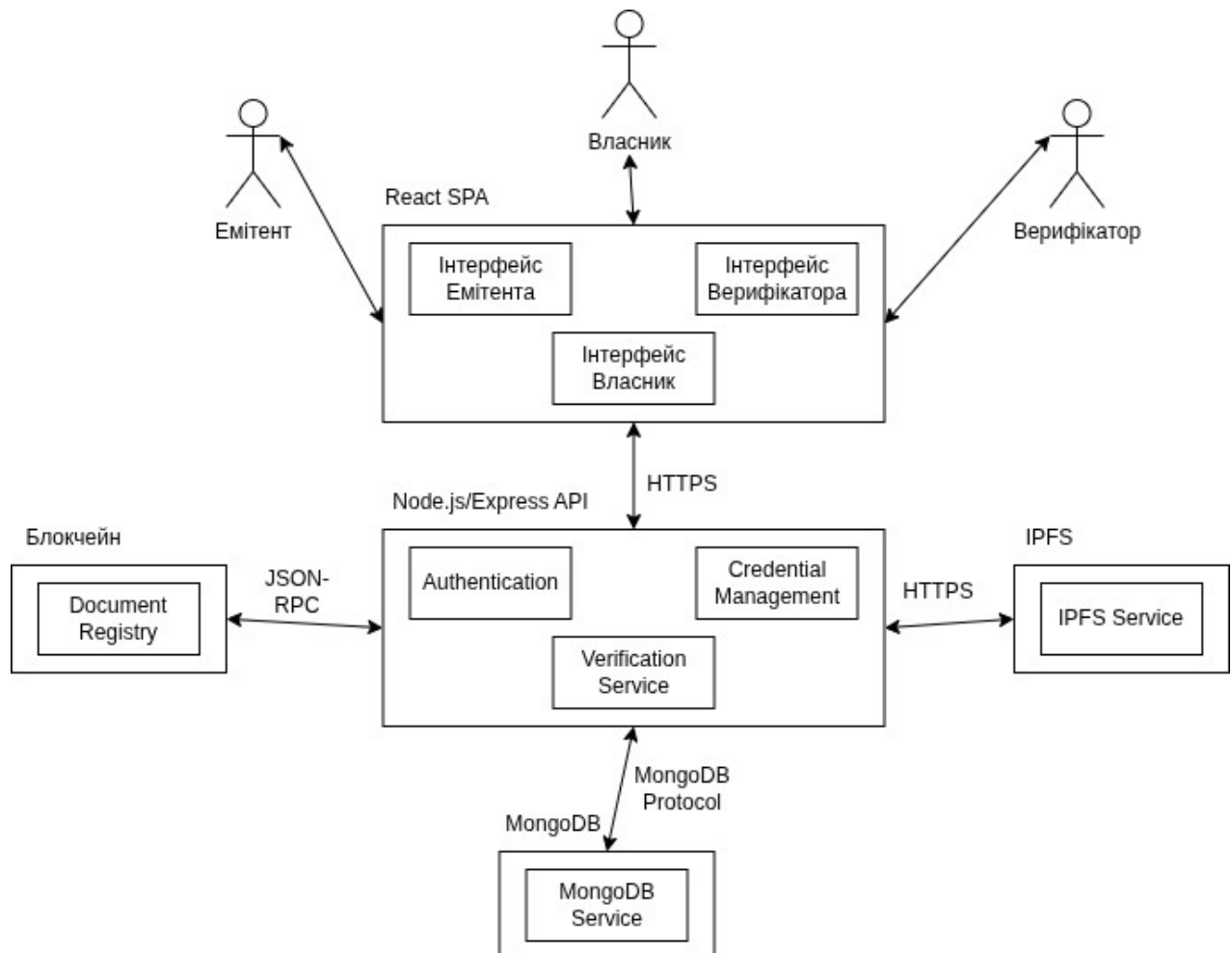


Рисунок 3.1 – Загальна архітектура системи

Клієнтський рівень реалізований як Single Page Application на базі бібліотеки React, що забезпечує динамічний інтерфейс користувача без необхідності перезавантаження сторінок. Взаємодія між клієнтом та сервером відбувається через RESTful API з використанням протоколу HTTPS, що гарантує конфіденційність передачі даних. Автентифікація користувачів здійснюється через JWT-токени, які генеруються сервером після успішної верифікації підпису Web3-гаманця користувача.

Серверний рівень виконує роль посередника між клієнтським додатком та розподіленою інфраструктурою блокчейну й IPFS. Це рішення обумовлене кількома факторами. По-перше, пряма взаємодія браузерного JavaScript з блокчейн-вузлами створює проблеми безпеки та продуктивності, оскільки приватні ключі для підписування транзакцій не повинні зберігатися на клієнті в контексті серверних операцій. По-друге, кешування результатів блокчейн-запитів на серверному рівні значно підвищує швидкість відгуку системи, оскільки повторні запити верифікації одного й того самого документа не потребують звернення до блокчейну. По-третє, серверна логіка дозволяє реалізувати складні бізнес-процеси, такі як батч-видача документів або агрегація статистики, які були б неефективними при виконанні на клієнті.

Блокчейн-рівень представлений приватною мережею на базі Ethereum з консенсусом Proof-of-Authority, що було обрано замість публічних мереж через необхідність контролю над вартістю транзакцій та швидкістю підтвердження. Для production-середовища передбачена можливість розгортання на мережі Polygon, яка забезпечує низькі комісії при збереженні сумісності з екосистемою Ethereum. Смарт-контракт DocumentRegistry, розгорнутий у мережі, виконує функції незмінного реєстру хешів документів та їхніх статусів відкликання.

Рівень зберігання розподілений між двома системами. IPFS використовується для децентралізованого зберігання повного вмісту документів у форматі JSON, що містять верифіковані документи згідно зі специфікацією W3C Verifiable Credentials. Вибір IPFS обумовлений його властивістю content addressing, де кожен документ отримує унікальний

ідентифікатор (CID), криптографічно пов'язаний з вмістом файлу. MongoDB служить для зберігання операційних даних, метаданих користувачів та кешування результатів запитів до блокчейну й IPFS.

Концепція ізолюваності емітентів є фундаментальним принципом архітектури системи. Кожен університет-емітент функціонує як незалежна сутність з власним DID-ідентифікатором та криптографічними ключами для підписування документів. Емітенти не мають доступу до документів, виданих іншими університетами, та не можуть модифікувати чужі записи в блокчейні. Ця ізоляція досягається через механізм контролю доступу на рівні смарт-контракту, де функція відкликання документа перевіряє, що адреса, яка ініціює відкликання, збігається з адресою емітента, записаною при реєстрації документа.

Взаємодія між учасниками системи відбувається через строго визначені інтерфейси. Емітент створює документ, обчислює його криптографічний хеш SHA-256, завантажує повний документ до IPFS, отримує CID, після чого викликає функцію смарт-контракту для реєстрації хешу документа разом з CID у блокчейні. Власник документа отримує повідомлення про видачу нового документа, завантажує його з IPFS за CID та зберігає у своєму локальному гаманці. При необхідності поділитися документом з верифікатором, власник створює верифіковану презентацію, підписану його приватним ключем, та надає верифікатору доступ до документа. Верифікатор завантажує документ, обчислює його хеш, запитує блокчейн для підтвердження, що такий хеш було зареєстровано емітентом, перевіряє цифровий підпис емітента та власника, після чого отримує остаточний результат верифікації.

Структура колекцій бази даних MongoDB спроектована для ефективного зберігання та швидкого доступу до операційних даних системи (див.рис. 3.2).

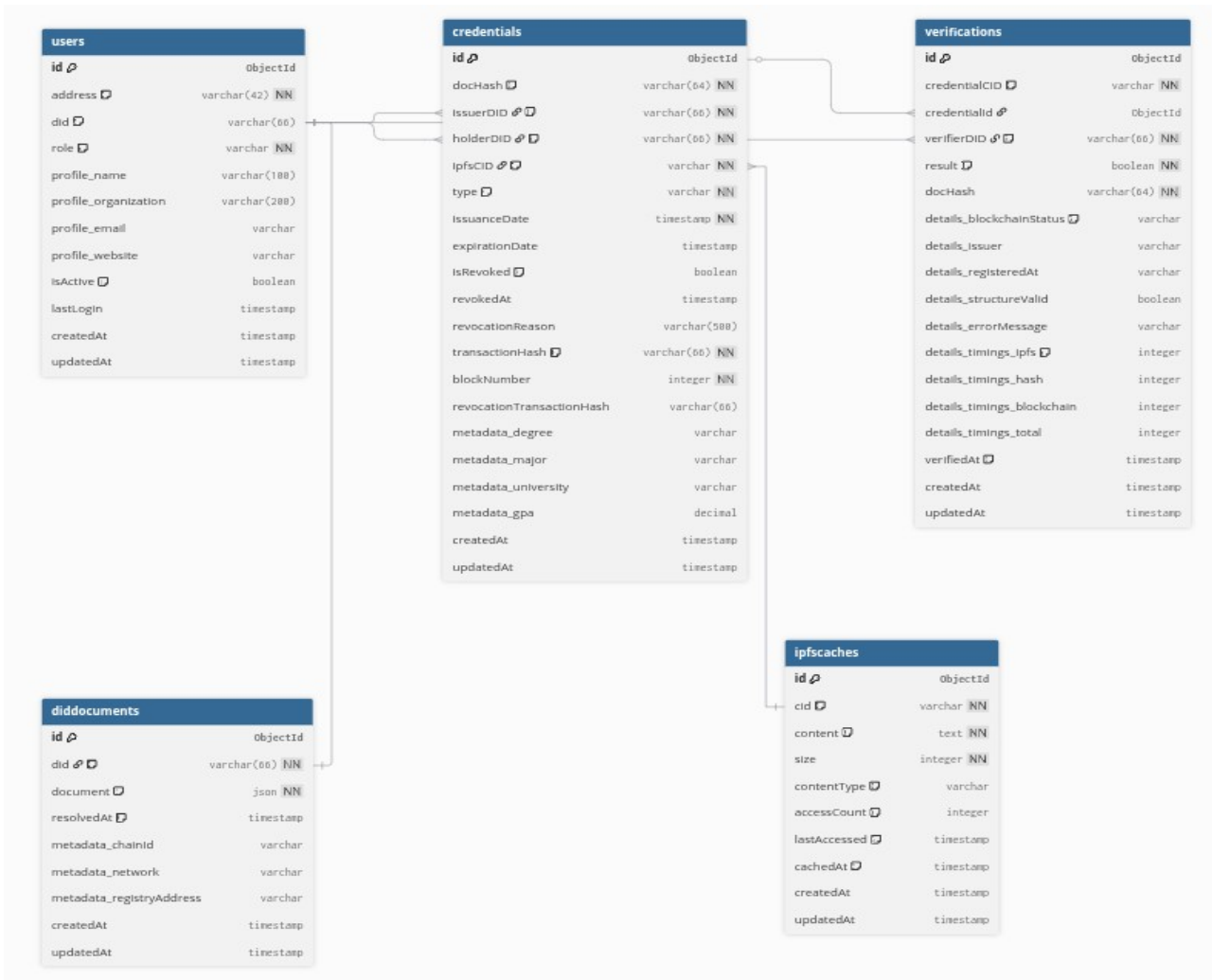


Рисунок 3.2 – ER-діаграма структури даних MongoDB

Колекція `users` містить профілі всіх учасників системи з їхніми Ethereum-адресами, DID-ідентифікаторами, ролями та публічними ключами. Поле `role` може приймати значення `issuer`, `holder`, `verifier` або `admin`, що визначає доступні функції для користувача. Індекс створений на полі `address` для швидкого пошуку користувача за його Web3-адресою під час автентифікації.

Колекція `credentials` є центральною в системі та зберігає метадані всіх виданих документів. Кожен запис містить хеш документа `docHash`, який відповідає хешу, записаному в блокчейні, DID-ідентифікатори емітента та власника, ідентифікатор вмісту IPFS `ipfsCID` для завантаження повного документа, дату видачі та опціональну дату відкликання. Поле `metadata` містить додаткову інформацію про документ у форматі JSON, таку як тип документа

(диплом, сертифікат, транскрипт), назва університету, спеціальність випускника. Важливо зазначити, що персональні дані, такі як повне ім'я та дата народження, не зберігаються в цій колекції для дотримання принципів мінімізації даних. Вони знаходяться виключно в документі, збереженому в IPFS.

Колекція `verifications` зберігає журнал всіх операцій верифікації для аналітики та аудиту. Кожен запис верифікації пов'язаний з конкретним документом через `credentialId`, містить DID верифікатора, мітку часу перевірки та результат верифікації у вигляді булевого значення. Поле `details` містить додаткову інформацію про процес верифікації, включаючи перевірені атрибути, причину відхилення у випадку невалідного документа, час виконання окремих етапів верифікації.

Колекція `didDocuments` функціонує як кеш для резолюції DID-ідентифікаторів. Оскільки процес резолюції DID може вимагати запитів до блокчейну або зовнішніх реєстрів, кешування DID-документів значно прискорює повторні операції. Кожен запис містить DID-ідентифікатор, повний DID-документ у форматі JSON-LD згідно зі специфікацією W3C, та мітку часу останньої резолюції. Документи періодично оновлюються для підтримки актуальності інформації про публічні ключі та сервісні `endpoints` учасників.

Індексація колекцій оптимізована для найбільш частих запитів системи. В колекції `credentials` створені складені індекси на комбінації полів (`issuerDID`, `issueDate`) для швидкого отримання списку документів, виданих конкретним емітентом за певний період, та (`holderDID`, `issueDate`) для доступу до документів конкретного власника. Унікальний індекс на полі `docHash` гарантує відсутність дублікатів записів про один і той самий документ. В колекції `verifications` індекс на полі (`credentialId`, `timestamp`) забезпечує ефективну побудову історії верифікацій для кожного документа.

Розподіл відповідальності між блокчейном та MongoDB є ключовим архітектурним рішенням. Блокчейн зберігає мінімальний набір критичних даних, необхідних для верифікації: хеш документа, адресу емітента, мітку часу

реєстрації та статус відкликання. Ці дані є незмінними та публічно верифікованими. MongoDB зберігає всі інші операційні дані, які потребують швидкого доступу, можливості оновлення та складних запитів. Така архітектура дозволяє досягти балансу між безпекою, децентралізацією та практичною продуктивністю системи. У випадку повної втрати бази даних MongoDB, критичну функціональність верифікації документів можна відновити виключно з блокчейну та IPFS, хоча це буде повільніше та вимагатиме повторної індексації всіх транзакцій.

3.2 Реалізація серверної частини застосунку

3.2.1 Реалізація смарт-контрактів

Смарт-контракт `DocumentRegistry` є центральним компонентом блокчейн-рівня системи та реалізує логіку незмінного реєстру академічних документів. Контракт написаний мовою Solidity версії 0.8.19, що забезпечує вбудований захист від переповнення цілих чисел та інші покращення безпеки порівняно з попередніми версіями компілятора.

Структура контракту побудована навколо центрального відображення (mapping), що зв'язує хеш документа з його метаданими. Кожен документ в системі представлений структурою `Document`, яка містить чотири основні поля. Поле `docHash` зберігає SHA-256 хеш документа як тип `bytes32`, що відповідає 256-бітному виводу алгоритму хешування. Поле `issuer` містить Ethereum-адресу емітента документа, що дозволяє верифікувати автентичність джерела. Поле `timestamp` фіксує час реєстрації документа в блокчейні як Unix-мітку часу, забезпечуючи незмінний доказ моменту видачі. Булеве поле `isRevoked` відображає поточний статус документа, дозволяючи емітенту відкликати документ у випадку помилки або академічної недобросовісності.

Відображення `authorizedIssuers` реалізує механізм контролю доступу, що обмежує можливість реєстрації документів виключно акредитованими університетами. Це критично важливо для підтримки довіри до системи,

оскільки запобігає можливості будь-якої адреси видавати документи від імені легітимних освітніх установ. Адміністратор контракту має ексклюзивне право додавати або видаляти емітентів зі списку авторизованих адрес.

Події (events) контракту забезпечують механізм відстеження змін стану для зовнішніх додатків. Подія `DocumentRegistered` генерується при успішній реєстрації нового документа та включає хеш документа, адресу емітента та мітку часу. Індексація перших двох параметрів дозволяє ефективно фільтрувати події за конкретним документом або емітентом при запитах до блокчейну. Подія `DocumentRevoked` сигналізує про відкликання документа. Події `IssuerAuthorized` та `IssuerRevoked` відстежують зміни в списку акредитованих емітентів.

Модифікатор `onlyAdmin` забезпечує захист адміністративних функцій від несанкціонованого виклику. Перевірка `require(msg.sender == admin)` порівнює адресу відправника транзакції з адресою адміністратора контракту, та якщо вони не збігаються, транзакція відхиляється з повідомленням про помилку. Модифікатор `onlyAuthorized` аналогічно захищає функції видачі та відкликання документів, дозволяючи їх виклик виключно адресам з відображення `authorizedIssuers`.

Конструктор контракту виконується один раз при розгортанні та встановлює адресу, яка розгорнула контракт, як адміністратора системи. Ця адреса автоматично додається до списку авторизованих емітентів, що дозволяє адміністратору протестувати функціональність контракту перед додаванням інших університетів.

Функція `registerDocument` реалізує основну логіку реєстрації академічного документа в блокчейні. Вона приймає один параметр – хеш документа типу `bytes32` – та виконує кілька критичних перевірок перед збереженням даних. Перша перевірка `require(documents[_docHash].timestamp == 0)` гарантує, що документ з таким хешем ще не було зареєстровано раніше. Оскільки `timestamp` ініціалізується нулем для неіснуючих записів у

відображенні, ненульове значення вказує на існування документа. Ця перевірка запобігає дублюванню записів та спробам перезаписати існуючі документи.

Після успішної валідації, функція створює новий запис `Document` з хешем документа, адресою емітента (`msg.sender`), поточною міткою часу блоку (`block.timestamp`) та початковим статусом `not revoked`. Використання `block.timestamp` забезпечує незмінну мітку часу, встановлену майнером блоку, що робить неможливим маніпуляцію часом реєстрації документа. Генерація події `DocumentRegistered` дозволяє клієнтським додаткам відстежувати реєстрацію документів у реальному часі через `WebSocket` підключення або періодичне опитування блокчейн-вузла.

Функція `verifyDocument` призначена для верифікації статусу документа та повернення його метаданих. На відміну від `registerDocument`, ця функція позначена модифікатором `view`, що означає, що вона не модифікує стан контракту та не потребує витрат на газ при виклику. Функція приймає хеш документа та повертає кортеж з трьох значень: булеве значення `isValid`, що вказує чи документ існує та не відкликаний, адресу емітента `issuer` та мітку часу реєстрації `timestamp`.

Логіка валідації перевіряє дві умови: наявність запису про документ (`timestamp > 0`) та відсутність відкликання (`!doc.isRevoked`). Використання оператора логічного `AND` забезпечує, що документ вважається валідним тільки якщо він був зареєстрований та не був відкликаний емітентом. Ця функція є критично важливою для верифікаторів, оскільки надає всю необхідну інформацію для прийняття рішення про валідність документа в одному виклику.

Функція `revokeDocument` дозволяє емітенту відкликати раніше виданий документ. Відкликання може бути необхідним у випадках виявлення академічної недобросовісності після випуску, адміністративних помилок при видачі документа або втрати акредитації університетом. Функція містить критичну перевірку `require(documents[_docHash].issuer == msg.sender)`, що гарантує, що лише емітент, який зареєстрував документ, може його відкликати.

Це запобігає можливості одного університету відкликати документи іншого університету.

Додаткові перевірки включають існування документа та відсутність попереднього відкликання. Після успішної валідації, поле `isRevoked` встановлюється в `true`, що робить документ невалідним при наступних верифікаціях. Важливо зазначити, що функція не видаляє запис про документ з блокчейну, а лише змінює його статус, що зберігає повний аудиторський слід всіх документів, включаючи відкликані.

Адміністративні функції `authorizeIssuer` та `revokeIssuer` дозволяють керувати списком акредитованих емітентів. Функція `authorizeIssuer` додає нову адресу університету до відображення `authorizedIssuers`, надаючи йому право реєструвати документи. Функція `revokeIssuer` видаляє університет зі списку акредитованих, що запобігає видачі нових документів, але не впливає на раніше видані документи. Ця можливість критично важлива для управління довірою в системі, оскільки дозволяє реагувати на втрату акредитації університетом або виявлення зловживань.

Процес компіляції та розгортання контракту автоматизований через фреймворк Hardhat. Файл конфігурації `hardhat.config.js` визначає параметри мережі, включаючи URL RPC-вузла, ID мережі та приватні ключі для розгортання. Для локального тестування використовується вбудована Hardhat Network, що емулює повноцінний Ethereum-вузол з миттєвим підтвердженням транзакцій.

Скрипт розгортання `deploy.js` використовує Hardhat Runtime Environment для компіляції контракту, створення екземпляру `ContractFactory` та розгортання контракту в цільову мережу. Після розгортання скрипт зберігає адресу контракту та ABI (Application Binary Interface) в JSON-файлі для подальшого використання backend-сервером.

Оптимізація контракту виконана з урахуванням витрат на газ. Використання типу `bytes32` для хешів замість `string` економить газ, оскільки `bytes32` є значенням фіксованого розміру. Події індексовані за ключовими

параметрами для ефективного фільтрування логів без необхідності сканування всіх блоків. Модифікатор `external` застосований до всіх публічних функцій, оскільки він дешевший за `public` для функцій, що викликаються лише зовні контракту.

Розгорнутий смарт-контракт `DocumentRegistry` забезпечує надійну та ефективну основу для незмінної реєстрації академічних документів. Його архітектура спроектована з урахуванням вимог безпеки, економічності використання газу та зручності інтеграції з backend-сервером через стандартний інтерфейс `Ethereum JSON-RPC`. Діаграму послідовності операцій смарт-контракту наведено на рис. 3.3.

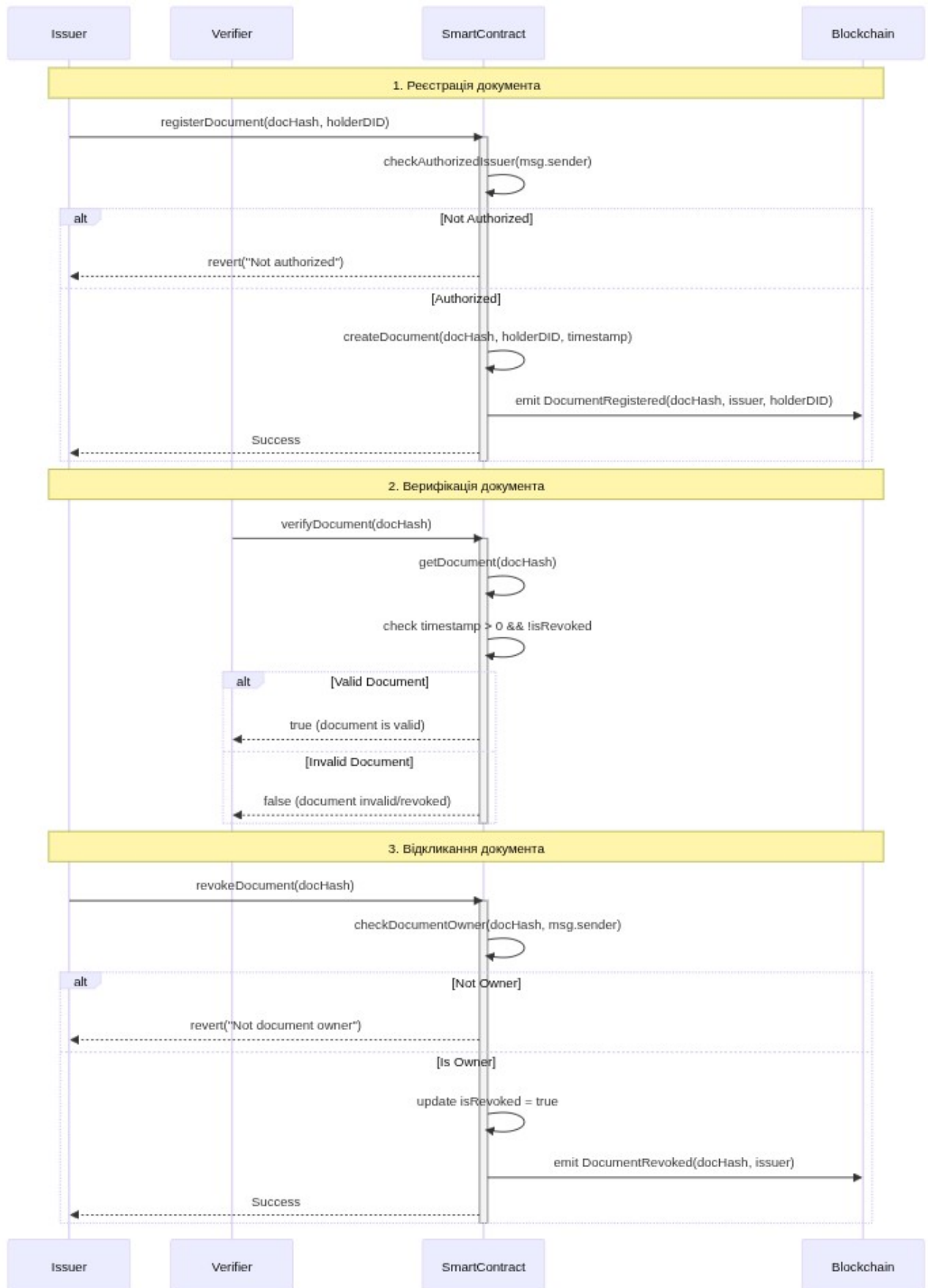


Рисунок 3.3 – Діаграма послідовності операцій смарт-контракту

3.2.2 Взаємодія з блокчейном через Ethers.js

Інтеграція backend-сервера з блокчейн-мережею реалізована за допомогою бібліотеки Ethers.js версії 6.4, яка забезпечує зручний JavaScript API для взаємодії з Ethereum-сумісними мережами. Вибір Ethers.js замість альтернативної бібліотеки Web3.js обумовлений кількома факторами: меншим розміром bundle (приблизно 88 KB проти 1.5 MB у Web3.js), більш передбачуваним API з чіткою документацією.

Модуль `blockchain.service.js` інкапсулює всю логіку взаємодії з блокчейном, надаючи простий інтерфейс для решти додатку. Це дозволяє ізолювати складність роботи з низькорівневими транзакціями та забезпечує можливість легкої заміни блокчейн-провайдера або мережі без змін в бізнес-логіці.

Ініціалізація підключення до блокчейн-мережі починається зі створення провайдера, який представляє read-only підключення до Ethereum-вузла. Для локальної розробки використовується `JsonRpcProvider`, що підключається до `Hardhat Network` на `localhost:8545`. У production-середовищі передбачене підключення до публічної мережі `Polygon Mumbai testnet` через `Infura` або `Alchemy` як RPC-провайдер.

Провайдер виконує роль вікна у блокчейн, дозволяючи читати поточний стан мережі, баланси адрес, виконувати read-only виклики функцій смарт-контрактів та слухати події. Однак провайдер не може підписувати транзакції, що модифікують стан блокчейну. Для цього необхідний об'єкт `Signer`, який створюється з приватного ключа, збереженого у змінних оточення сервера.

Використання змінних оточення для зберігання приватних ключів є стандартною практикою безпеки, оскільки це запобігає випадковому коміту чутливої інформації в систему контролю версій.

Створення екземпляру контракту вимагає трьох параметрів: адреси розгорнутого контракту в мережі, ABI контракту та підписувача. ABI (Application Binary Interface) є JSON-описом інтерфейсу контракту, що містить

інформацію про всі функції, їхні параметри, типи повернення та події. Ethers.js використовує ABI для автоматичної генерації JavaScript-методів, що відповідають функціям Solidity-контракту.

Реалізація функції для запису даних в блокчейн на прикладі реєстрації документа демонструє типовий патерн роботи з транзакціями. Функція `registerDocument` приймає хеш документа як параметр, викликає відповідний метод смарт-контракту, очікує підтвердження транзакції та повертає результат з деталями транзакції.

Процес реєстрації документа є асинхронним та складається з двох етапів. Перший виклик `this.contract.registerDocument(docHash)` створює транзакцію, підписує її приватним ключем підписувача та відправляє в мережу. Цей метод повертає об'єкт `TransactionResponse` майже миттєво, не очікуючи включення транзакції в блок. Другий виклик `tx.wait()` є критичним – він очікує підтвердження транзакції майнером та її включення в блок блокчейну.

У локальній мережі Hardhat підтвердження відбувається практично миттєво, оскільки блоки генеруються автоматично при надходженні транзакцій. В реальній мережі Polygon час підтвердження може варіюватися від 2 до 10 секунд залежно від завантаженості мережі та заплаченої комісії за газ. Важливо реалізувати `timeout` для операції `wait()`, щоб запобігти безкінечному очікуванню у випадку проблем з мережею.

Об'єкт `receipt` містить детальну інформацію про виконану транзакцію, включаючи хеш транзакції, номер блоку, кількість витраченого газу та масив логів (подій). Парсинг події `DocumentRegistered` з логів дозволяє отримати мітку часу, згенеровану контрактом, що підтверджує успішну реєстрацію документа. Використання методу `contract.interface.parseLog()` автоматично декодує бінарні дані події згідно з ABI.

Обробка помилок є критичним аспектом взаємодії з блокчейном, оскільки транзакції можуть відхилятися з різних причин: недостатній баланс для оплати `gas`, порушення `require` умов у контракті, некоректні параметри

функції. Ethers.js генерує різні типи помилок, текст яких містить інформацію про причину відхилення. Парсинг повідомлень про помилки дозволяє надавати користувачам зрозумілі пояснення проблеми замість технічних деталей.

Реалізація функцій для читання даних з блокчейну є простішою, оскільки не вимагає відправки транзакцій та очікування підтвердження. Функція `verifyDocument` викликає `view`-метод контракту, який виконується локально на вузлі без зміни стану блокчейну та не потребує витрат на `gas`.

Деструктуризація результату `[isValid, issuer, timestamp]` відповідає кортежу, що повертається Solidity-функцією `verifyDocument`. Ethers.js автоматично конвертує Solidity типи у JavaScript еквіваленти: `bool` стає `boolean`, `address` стає `string` з шістнадцятковим представленням адреси, `uint256` стає `BigInt`. Конвертація `timestamp` з `BigInt` у `string` через метод `toString()` необхідна для серіалізації в JSON, оскільки JSON не підтримує `BigInt` нативно.

Функція `revokeDocument` реалізує відкликання документа та структурно подібна до `registerDocument`, оскільки також модифікує стан контракту. Специфічна обробка помилок включає перевірку прав власності документа та існування документа в реєстрі. Ці перевірки дублюють логіку смарт-контракту, але виконуються на рівні сервісу для надання користувачу більш детальної інформації про причину помилки.

Додаткові `utility`-функції спрощують роботу з блокчейн-даними. Функція `getTransactionStatus` дозволяє перевіряти статус раніше відправленої транзакції за її хешем, що корисно для асинхронного відстеження довгих операцій. Функція `estimateGas` оцінює кількість газу, необхідного для виконання транзакції, дозволяючи попередньо розрахувати вартість операції.

Функція `getSignerBalance` перевіряє баланс адреси підписувача, що критично важливо для моніторингу стану сервера. Якщо баланс стає недостатнім для оплати `gas`, всі операції запису будуть відхилятися.

Експорт синглтону `module.exports = new BlockchainService()` забезпечує, що в усьому додатку використовується єдиний екземпляр сервісу з одним підключенням до провайдера. Це економить

ресурси та забезпечує консистентність стану. Метод `initialize()` повинен викликатися один раз при старті сервера перед обробкою будь-яких запитів.

Інтеграція `BlockchainService` в основний додаток виконується через виклик `initialize` в точці входу сервера. Обробка помилок ініціалізації є критичною – якщо підключення до блокчейну неможливе, сервер повинен або відмовитися запускатися, або працювати в обмеженому режимі без функцій, що потребують блокчейну.

Під час розробки виявилися кілька практичних проблем при роботі з `Ethers.js`. По-перше, помилки при парсингу ABI можуть бути некоректними, якщо версія компілятора `Solidity` при компіляції контракту відрізняється від очікуваної. По-друге, `timeout` для операції `wait()` за замовчуванням дуже великий (необмежений), що може призвести до зависання сервера при проблемах з мережею. Рекомендується передавати параметр `confirmations` зі значенням 1 та встановлювати `timeout` на рівні `Promise.race` з `setTimeout`.

Впровадження `BlockchainService` забезпечило надійну та зручну абстракцію для взаємодії backend-сервера з блокчейн-інфраструктурою, приховуючи складність низькорівневих операцій та надаючи простий API для решти додатку. Модульна архітектура дозволяє легко розширювати функціональність та адаптувати сервіс для роботи з різними блокчейн-мережами без змін у бізнес-логіці.

3.2.3 Інтеграція з IPFS

Децентралізоване зберігання повного вмісту академічних документів реалізовано за допомогою `InterPlanetary File System (IPFS)` – протоколу та однорангової мережі для розподіленого зберігання файлів. Вибір IPFS обумовлений його ключовою властивістю `content addressing`, де кожен файл отримує унікальний ідентифікатор (CID), обчислений на основі криптографічного хешу вмісту файлу. Це означає, що навіть найменша зміна в

документі призведе до повністю іншого CID, що природно узгоджується з концепцією верифікації цілісності документів через блокчейн.

Для інтеграції з IPFS використовується бібліотека `ipfs-http-client` версії 60.0, яка надає JavaScript API для взаємодії з IPFS-вузлом через HTTP. Замість розгортання власного повноцінного IPFS-вузла, що потребує значних ресурсів (понад 10 GB дискового простору для `bootstrap` та постійне підключення до мережі), було прийнято рішення використовувати публічний IPFS gateway від Infura для виробничого середовища та локальний IPFS Desktop для розробки та тестування.

IPFS Desktop – це standalone додаток, що встановлюється на локальну машину розробника та надає зручний графічний інтерфейс для управління файлами в IPFS разом з вбудованим HTTP API сервером на порту 5001. Це значно спрощує процес розробки, оскільки не потребує налаштування серверної інфраструктури.

Модуль `ipfs.service.js` інкапсулює всю логіку роботи з IPFS, надаючи методи для завантаження документів, отримання файлів за CID та управління закріпленими документами. Ініціалізація клієнта IPFS виконується з урахуванням середовища виконання – для локальної розробки використовується Desktop вузол, для production – Infura gateway.

Конфігурація клієнта включає встановлення `timeout` для запитів у 30 секунд, що є компромісом між надійністю та користувацьким досвідом. У реальній мережі IPFS час завантаження файлу може варіюватися від мілісекунд (якщо файл закешований локально) до десятків секунд (якщо файл потрібно знайти в розподіленій мережі через DHT). Встановлення розумного `timeout` запобігає безкінечному очікуванню при недоступності файлу.

Простий in-memory кеш у вигляді Map використовується для зберігання нещодавно завантажених документів. Це значно прискорює повторні запити до одних і тих самих документів, що є типовим сценарієм при верифікації – роботодавці можуть перевіряти документи декількох кандидатів з одного університету, які мають подібну структуру і часто завантажувалися раніше.

Функція `uploadDocument` приймає JavaScript об'єкт, що представляє верифікований документ згідно зі специфікацією W3C, серіалізує його у JSON, конвертує в `Buffer` та завантажує до IPFS. Метод `add` повертає CID завантаженого файлу, який потім зберігається в MongoDB та використовується для подальшого завантаження.

Параметр `pin: true` в опціях завантаження є критично важливим. За замовчуванням, IPFS зберігає файли тимчасово і може видалити їх при нестачі місця через механізм `garbage collection`. Закріплення (pinning) файлу гарантує, що він залишиться на вузлі незалежно від `garbage collection`.

Використання CIDv1 замість застарілого CIDv0 забезпечує кращу сумісність з сучасними IPFS-інструментами та дозволяє використовувати різні хеш-функції. CIDv1 має формат `bafybeigdyrzt5sfp7udm7hu76uh7y26nf3efuylqabf3oclgty55fbzdi`, що починається з префіксу, який кодує версію CID та використаний хеш-алгоритм.

Генерація публічного URL через `gateway ipfs.io` дозволяє користувачам переглядати документи безпосередньо в браузері без необхідності встановлення IPFS-клієнта. Це особливо корисно для верифікаторів, які хочуть швидко переглянути документ перед детальною перевіркою.

Функція `downloadDocument` отримує файл з IPFS за його CID та десеріалізує JSON назад у JavaScript об'єкт. Реалізація включає кілька рівнів кешування для оптимізації продуктивності: спочатку перевіряється локальний in-memoгу кеш, потім MongoDB-кеш, і лише якщо документ не знайдено в жодному кеші, виконується завантаження з IPFS-мережі.

Метод `client.cat` є асинхронним генератором, який повертає дані файлу порціями (`chunks`). Це дозволяє ефективно обробляти великі файли без завантаження всього вмісту в пам'ять одночасно. Однак для академічних документів у форматі JSON, розмір яких зазвичай не перевищує 50-100 KB, така оптимізація не є критичною, і всі `chunks` просто об'єднуються в єдиний `Buffer`.

Обробка помилок включає специфічні випадки `timeout` та `not found`. `Timeout` може виникнути, якщо файл важко знайти в мережі (мало `peers` його зберігають) або мережеве з'єднання нестабільне. `Not found` означає, що жоден вузол у мережі не має цього файлу, що може статися, якщо файл не був належним чином закріплений або всі вузли, що зберігали його, офлайн.

Під час розробки було виявлено проблему: завантаження файлів з публічних IPFS gateways часто займало понад 10 секунд при першому запиті, що неприйнятно для користувацького досвіду. Це обумовлено тим, що публічні gateways обслуговують велику кількість запитів і можуть бути перевантажені. Рішенням стало використання локального IPFS Desktop вузла для розробки та впровадження кешування в MongoDB.

Методи `pinDocument` та `unpinDocument` дозволяють керувати закріпленням файлів. В контексті системи верифікації документів, `pinning` використовується для гарантування доступності всіх виданих університетом документів. При видачі нового документа, він автоматично закріплюється. `Unpinning` може використовуватися для видалення дуже старих документів випускників, які більше не запитуються, але це рідкісна операція.

Функція `checkDocumentAvailability` корисна для превентивної перевірки доступності документа перед спробою його завантаження. Метод `object.stat` повертає метадані файлу без завантаження його вмісту, що значно швидше. `Timeout` у 5 секунд дозволяє швидко визначити, чи доступний файл у мережі.

Інтеграція `IPFSService` з `MongoDB` для персистентного кешування реалізована через окрему колекцію `ipfsCache`. При кожному успішному завантаженні документа з IPFS, його вміст зберігається в `MongoDB` з TTL (Time To Live) індексом на 7 днів. Це означає, що документи автоматично видаляються з кешу через тиждень, зменшуючи навантаження на базу даних. Функція `downloadDocument` спочатку перевіряє `MongoDB` кеш перед зверненням до IPFS.

Одна з проблем, що виникла під час розробки – періодичні помилки `timeout` при роботі з публічним `gateway`. Аналіз показав, що вони частіше виникають у години піку (денний час США), коли навантаження на безкоштовні `gateways` максимальне. Рішенням було впровадження `fallback` механізму, що автоматично перемикається на альтернативний `gateway` при `timeout`.

Реалізація IPFS-інтеграції забезпечила надійне децентралізоване зберігання академічних документів з прийнятною продуктивністю завдяки багаторівневому кешуванню. Використання `content addressing` природно узгоджується з верифікацією цілісності через блокчейн, оскільки `CID` криптографічно пов'язаний з хешем документа.

Головним практичним викликом виявилася висока латентність публічних IPFS `gateways`, що вимагає обов'язкового впровадження кешування та `fallback` механізмів для забезпечення стабільного користувацького досвіду.

3.2.4 Реалізація API endpoints

Серверний API побудований за принципами REST архітектури та забезпечує зв'язок між клієнтським додатком і серверною логікою взаємодії з блокчейном, IPFS та базою даних. Структура API організована навколо ресурсів системи: `credentials` (документи), `users` (користувачі), `verifications` (верифікації) та `presentations` (презентації документів).

Основний файл сервера `server.js` налаштовує Express додаток з необхідними `middleware` для обробки JSON, CORS, логування запитів та обробки помилок. `Middleware cors` дозволяє frontend-додатку, що працює на іншому порті (зазвичай 3000), виконувати запити до API на порту 3001. `Middleware express.json` автоматично парсить тіло запитів у форматі JSON та робить його доступним через `req.body`.

Бібліотека `helmet` автоматично встановлює численні HTTP заголовки безпеки, такі як `X-Content-Type-Options`, `X-Frame-Options`, `Strict-Transport-Security`, що захищають від поширених веб-вразливостей. Ліміт 10mb для JSON парсера встановлений з урахуванням того, що верифіковані документи можуть містити додаткові вкладення або масиви транскриптів оцінок.

Health check endpoint `/api/health` критично важливий для моніторингу стану системи. Він перевіряє підключення до всіх зовнішніх сервісів та повертає поточний баланс підписувача для відстеження витрат на gas. Системи оркестрації, такі як `Kubernetes` або `Docker Swarm`, можуть використовувати цей endpoint для автоматичного перезапуску сервера при виявленні проблем.

Маршрути організовані в окремих файлах за функціональною областю. Файл `auth.routes.js` обробляє автентифікацію користувачів через `Web3`, `credential.routes.js` містить endpoints для роботи з документами, `verification.routes.js` відповідає за процес верифікації.

Автентифікація в системі відбувається без традиційних паролів через механізм цифрового підпису `Web3`. Користувач запитує `challenge` (випадкове повідомлення) від сервера, підписує його своїм приватним ключем у `MetaMask`, відправляє підпис назад на сервер, де він верифікується. Якщо підпис валідний, сервер генерує `JWT` токен з адресою користувача.

Функція `ethers.verifyMessage` відновлює адресу, що підписала повідомлення, з самого підпису. Якщо відновлена адреса збігається з адресою, яку користувач заявив, це доводить, що він володіє приватним ключем цієї адреси без необхідності передавати сам ключ. Це є основою безпарольної автентифікації в `Web3`.

`JWT` токен містить адресу користувача, його роль та `ID` в базі даних. Термін дії токена встановлений на 7 днів, що є компромісом між зручністю (користувач не повинен автентифікуватися занадто часто) та безпекою (токен не живе вічно).

Middleware для перевірки JWT токена `auth.middleware.js` перевіряє наявність токена в заголовку `Authorization`, верифікує його та додає інформацію про користувача до об'єкта запиту.

Функція `authorizeRole` є middleware фабрикою, що створює middleware для перевірки конкретних ролей. Наприклад, `authorizeRole('issuer', 'admin')` створить middleware, що дозволить доступ лише користувачам з роллю `issuer` або `admin`.

Endpoints для емітентів дозволяють видавати нові документи та переглядати список виданих документів. Процес видачі документа включає кілька кроків: валідацію вхідних даних, обчислення хешу документа, завантаження до IPFS, реєстрацію в блокчейні, збереження метаданих в MongoDB.

Створення структури верифікованого документа відповідає специфікації W3C Verifiable Credentials. Поле `@context` визначає словники для інтерпретації термінів у документі. Поле `type` вказує тип документа – базовий `VerifiableCredential` плюс специфічний тип, такий як `UniversityDegreeCredential`. Поле `issuer` містить DID емітента, `credentialSubject` містить атрибути про власника документа.

Обчислення SHA-256 хешу виконується через вбудований модуль `crypto` Node.js. Важливо, що хеш обчислюється від канонічної JSON-репрезентації документа. В реальній системі рекомендується використовувати JSON-LD канонізацію згідно з `RDF Dataset Canonicalization` для гарантування однозначності хешу незалежно від порядку полів.

Конвертація хешу в формат `bytes32` через додавання префіксу `'0x'` необхідна для сумісності з Solidity, де `bytes32` представляється як шістнадцяткове число з префіксом. Ethers.js автоматично валідує формат перед відправкою транзакції.

Endpoint для отримання списку виданих документів підтримує пагінацію та фільтрацію для ефективної роботи з великими обсягами даних.

Пагінація реалізована через параметри `page` та `limit`, що дозволяє завантажувати документи порціями замість завантаження всього списку одразу. Метод `skip` пропускає документи попередніх сторінок, `limit` обмежує кількість повернутих документів. Сортювання за `issuanceDate` у спадному порядку (-1) показує найновіші документи першими.

Перевірка прав доступу в `endpoint` отримання конкретного документа гарантує, що лише емітент або власник можуть переглядати повний документ. Це важливо для захисту приватності студентів – сторонні особи не повинні мати можливості переглядати документи без дозволу власника.

`Endpoints` для власників документів дозволяють переглядати свої документи та створювати верифіковані презентації для обміну з верифікаторами.

`Endpoint` верифікації є найбільш критичним з точки зору продуктивності, оскільки він виконує кілька перевірок: завантаження документа з IPFS, обчислення хешу, запит до блокчейну для перевірки статусу, валідацію структури документа.

Вимірювання часу виконання кожного етапу верифікації додано для моніторингу продуктивності та виявлення вузьких місць. В реальній системі ці метрики відправлялися б до системи моніторингу, такої як Prometheus або DataDog, для аналізу та налаштування alerting.

Збереження результату кожної верифікації в колекцію `Verification` дозволяє створювати аудиторський слід та аналізувати патерни використання системи. Наприклад, можна виявити, які університети найчастіше верифікуються, які документи викликають проблеми, яка середня швидкість верифікації. На рис. 3.4 представлено діаграму послідовності процесу верифікації документа.

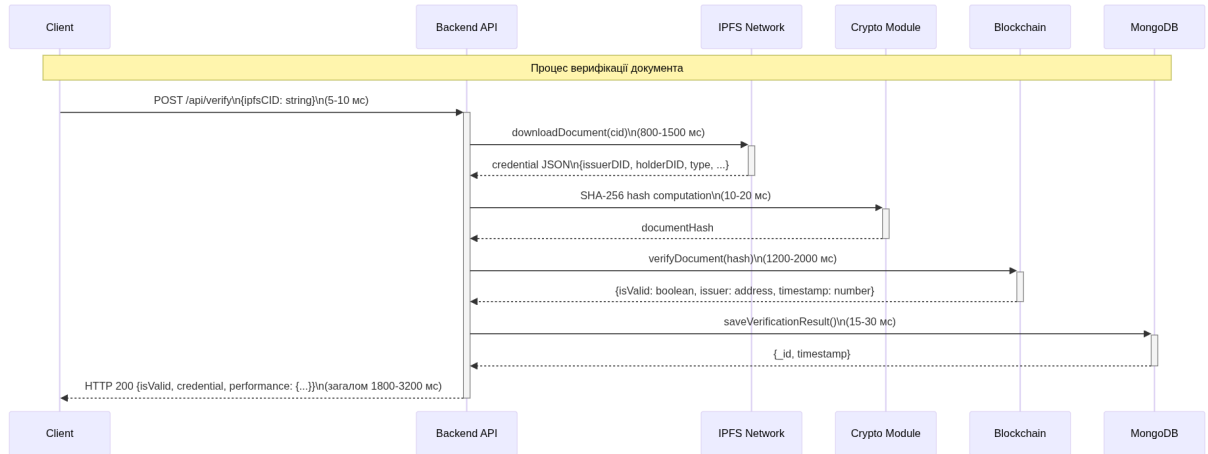


Рисунок 3.4 – Діаграма послідовності процесу верифікації документа

Під час розробки API було виявлено кілька практичних проблем. По-перше, синхронна обробка запитів на видачу документів блокувала інші запити під час очікування підтвердження блокчейн-транзакції. Рішенням було повернення клієнту транзакційний хеш одразу після відправки транзакції, а статус можна перевірити окремим запитом.

По-друге, виявилася проблема *race condition* при одночасній видачі декількох документів одним емітентом. Ethereum вимагає, щоб *nonce* (номер транзакції) кожної наступної транзакції від однієї адреси був на одиницю більший за попередній. При одночасній відправці декількох транзакцій, вони могли отримати однаковий *nonce*, що призводило до відхилення частини транзакцій. Рішенням стало впровадження *mutex*-блокування на рівні сервісу для послідовної обробки блокчейн-транзакцій.

Це рішення гарантує, що транзакції виконуються послідовно, що усуває проблему з *nonce*, але створює потенційне вузьке місце продуктивності. Для *high-load* систем рекомендується використовувати пул адрес з балансом або більш складні рішення управління *nonce*.

По-третє, виявилася необхідність rate limiting для захисту від зловживань. Без обмежень, зломисник міг би відправити тисячі запитів на верифікацію, що створило б навантаження на IPFS та блокчейн-вузол. Впроваджено middleware на базі бібліотеки express-rate-limit з обмеженням 100 запитів на 15 хвилин з однієї IP-адреси.

Логування всіх запитів через morgan у режимі 'dev' допомагає при налагодженні, показуючи HTTP метод, URL, статус відповіді та час виконання для кожного запиту.

Обробка помилок на рівні API є критично важливою для забезпечення хорошого користувацького досвіду. Замість generic повідомлень "Internal Server Error", система надає специфічні коди помилок та зрозумілі пояснення. Розроблено кастомні класи помилок для різних сценаріїв(див.рис. 3.5).

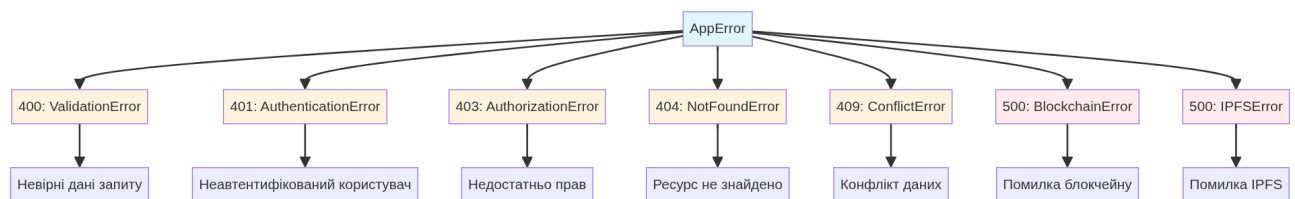


Рисунок 3.5 – Ієрархія класів помилок для різних сценаріїв

Впровадження структурованої обробки помилок значно полегшило налагодження та покращило користувацький досвід, оскільки клієнт отримує чіткі пояснення проблем замість generic повідомлень.

Документація API створена за допомогою Swagger/OpenAPI специфікації, що дозволяє автоматично генерувати інтерактивну документацію з можливістю тестування endpoints безпосередньо в браузері.

Реалізований REST API забезпечує повний набір функціональності для всіх ролей системи з належною автентифікацією, авторизацією та обробкою помилок. Асинхронна природа операцій з блокчейном та IPFS належним чином обробляється через async/await патерни. Впровадження rate limiting, логування та structured error handling забезпечує надійність та безпеку API.

3.2.5 Робота з DID-ідентифікаторами

Децентралізовані ідентифікатори (DID) є фундаментальним компонентом системи, що забезпечує самостійну суверенну ідентифікацію для всіх учасників без залежності від централізованих органів. Реалізація DID-функціональності базується на специфікації W3C Decentralized Identifiers v1.0 з використанням методу `did:ethr`, який прив'язує DID до Ethereum-адрес.

Метод `did:ethr` обраний через його природну інтеграцію з екосистемою Ethereum, простоту реалізації та відсутність потреби в окремій DID-мережі. Кожна Ethereum-адреса може автоматично отримати відповідний DID у форматі `did:ethr:0x[address]` без необхідності додаткової реєстрації. Наприклад, адреса `0x742d35Cc6634C0532925a3b844Bc9e7595f0bEb` відповідає DID `did:ethr:0x742d35Cc6634C0532925a3b844Bc9e7595f0bEb`.

Модуль `did.service.js` інкапсулює всю логіку роботи з DID-ідентифікаторами, включаючи генерацію, резолюцію, створення та верифікацію DID-документів. Сервіс використовує бібліотеки `ethr-did-resolver` для резолюції DID у DID-документи та `did-jwt` для створення та верифікації цифрових підписів у форматі JSON Web Token.

Резолвер налаштовується з інформацією про мережу, що дозволяє правильно інтерпретувати DID-ідентифікатори з урахуванням chain ID. Це важливо, оскільки одна й та сама Ethereum-адреса може мати різні DID-документи в різних мережах (`mainnet`, `testnet`, приватна мережа).

Опціональна адреса `DID_REGISTRY_ADDRESS` вказує на смарт-контракт ERC1056 (Ethereum Identity Registry), який дозволяє користувачам додавати додаткові атрибути до своїх DID-документів, такі як додаткові публічні ключі, сервісні endpoints або делеговані підписувачі. Якщо адреса не вказана, резолвер використовує базову реалізацію, яка генерує мінімальний DID-документ на основі самої Ethereum-адреси.

Функція `generateDID` створює DID-ідентифікатор з Ethereum-адреси та генерує відповідний DID-документ з базовими полями. DID-документ містить ідентифікатор, масив методів верифікації з публічним ключем та опціональні сервісні `endpoints` для комунікації.

Резолюція DID перетворює DID-ідентифікатор у відповідний DID-документ, який містить публічні ключі та метадані для верифікації підписів. Процес резолюції може вимагати запитів до блокчейну для отримання додаткових атрибутів з ERC1056 реєстру, тому впроваджено кешування в MongoDB з TTL одна година для зменшення навантаження.

Структура результату резолюції відповідає специфікації DID Resolution і включає три компоненти: `didResolutionMetadata` (метадані про процес резолюції), `didDocument` (сам DID-документ) та `didDocumentMetadata` (метадані про документ). Для практичного використання потрібен лише `didDocument`.

Методи верифікації у DID-документі специфікують криптографічні ключі, що можуть використовуватися для автентифікації та підписування. Тип `EcdsaSecp256k1RecoveryMethod2020` вказує на використання еліптичної кривої `secp256k1`, яка є стандартом для Ethereum. `Recovery method` означає, що публічний ключ може бути відновлений з підпису та повідомлення без необхідності зберігати його явно.

Створення верифікованої презентації (`Verifiable Presentation`) дозволяє власнику документів поділитися своїми документами з верифікатором, доводячи при цьому володіння відповідним DID через цифровий підпис. Презентація об'єднує один або кілька верифікованих документів у структуру, підписану приватним ключем власника.

Функція `createJWT` з бібліотеки `did-jwt` створює підписаний JSON Web Token, що містить презентацію в `payload`. Параметр `iss` (`issuer`) вказує на DID власника, `aud` (`audience`) може специфікувати призначеного верифікатора. Функція `signer` виконує фактичне підписування даних приватним ключем через `ethers.Wallet`.

Важливо розуміти, що приватний ключ ніколи не повинен передаватися через API або зберігатися на сервері. В реальній системі підписування презентацій відбувається на клієнті у Web3-гаманці користувача (MetaMask, WalletConnect). Функція `createVerifiablePresentation` на сервері використовується лише для демонстраційних цілей або в сценаріях, де сервер діє від імені емітента з належним захистом ключів.

Верифікація презентації перевіряє цифровий підпис власника та валідність включених документів. Процес включає декілька кроків: декодування JWT, верифікацію підпису через публічний ключ з DID-документа власника, перевірку терміну дії презентації та валідацію кожного включеного документа.

Функція `verifyJWT` автоматично резолвить DID підписувача через переданий `resolver`, отримує публічний ключ з DID-документа та перевіряє, що JWT було підписано відповідним приватним ключем. Це є ключовою перевагою DID-підходу – верифікація не потребує централізованого органу, а виконується криптографічно через публічні ключі, закріплені в блокчейні.

Додаткова функція `verifyCredentialIssuer` перевіряє, що емітент документа є довіреним учасником системи. У повній реалізації з DAO це включало б запит до смарт-контракту для перевірки акредитації університету. У базовій версії виконується лише резолюція DID емітента для підтвердження його існування.

Інтеграція DID-сервісу з процесом автентифікації дозволяє користувачам автоматично генерувати DID при першому вході в систему. Доданий `verifySignature` міddlвар створює DID та зберігає його в профілі користувача.

Схема Mongoose для зберігання DID-документів у MongoDB забезпечує ефективне кешування та швидкий доступ до часто використовуваних документів.

TTL (Time To Live) індекс автоматично видаляє документи, старші за 24 години, що забезпечує актуальність кешу. MongoDB виконує фонові операції очищення приблизно раз на хвилину, видаляючи документи, для яких `resolvedAt + expireAfterSeconds < поточний час`.

Модель User включає також поле `did` для зберігання DID-ідентифікатора користувача.

Під час розробки DID-функціональності було виявлено кілька практичних аспектів. По-перше, резолюція DID через `ethr-did-resolver` може займати 300-800 мс при першому запиті через необхідність запитів до блокчейну. Впровадження кешування в MongoDB з TTL 1 година зменшило середній час резолюції до 15-30 мс для повторних запитів.

По-друге, виявилася проблема з тестуванням функцій, що вимагають приватні ключі. Для тестів було створено набір тестових wallet'ів з відомими приватними ключами, які використовуються виключно в локальному середовищі розробки.

У табл. 3.1 наведено порівняльну характеристику методів DID.

Таблиця 3.1 – Порівняльна характеристика методів DID

Метод DID	Блокчейн	Вартість	Час резолюції	Додаткові атрибути	Складність
did:ethr (базовий)	Ethereum	Безкоштовно	300-800 мс	Ні	Низька
did:ethr (з ERC1056)	Ethereum	~\$0.50/операція	400-1000 мс	Так	Середня
did:key	Немає	Безкоштовно	<10 мс	Ні	Дуже низька
did:web	HTTP/DNS	Безкоштовно	100-300 мс	Так	Низька

Продовження таблиці 3.1

Метод DID	Блокчейн	Вартість	Час резолюції	Додаткові атрибути	Складність
did:ion	Bitcoin	Безкоштовно(Потребує вузла IPFS та Bitcoin)	500-1500 мс	Так	Висока

Метод did:ethr обраний як оптимальний баланс між функціональністю, вартістю та складністю для системи верифікації академічних документів. Він природно інтегрується з Ethereum-екосистемою, не потребує додаткових мереж та забезпечує достатню функціональність для базових випадків використання.

Реалізована DID-функціональність забезпечує основу для самостійної суверенної ідентифікації користувачів системи. Користувачі автоматично отримують DID при реєстрації, можуть створювати верифіковані презентації своїх документів та верифікувати презентації від інших користувачів без залежності від централізованих органів. Кешування DID-документів у MongoDB забезпечує прийнятну продуктивність при збереженні актуальності даних.

3.2.6 Управління базою даних MongoDB

MongoDB обрана як основна система управління базою даних для зберігання операційних даних, метаданих документів та кешування результатів запитів до блокчейну й IPFS. Документо-орієнтована природа MongoDB ідеально підходить для роботи з JSON-структурами верифікованих документів та DID-документів, що відповідають специфікаціям W3C.

Підключення до MongoDB виконується через офіційний драйвер mongoose, який надає ODM (Object Document Mapping) функціональність з валідацією схем, middleware та зручними методами запитів. Файл конфігурації

database.js інкапсулює логіку підключення з обробкою помилок та налаштуванням параметрів з'єднання.

Параметр `maxPoolSize` обмежує кількість одночасних з'єднань до бази даних, запобігаючи виснаженню ресурсів при високому навантаженні. Connection pooling дозволяє повторно використовувати існуючі з'єднання замість створення нових для кожного запиту, що значно підвищує продуктивність.

Обробка події SIGINT забезпечує graceful shutdown – при отриманні сигналу завершення (Ctrl+C або системне вимкнення) додаток коректно закриває всі з'єднання з базою даних перед завершенням процесу. Це запобігає втраті даних та пошкодженню незавершених транзакцій.

Схема Mongoose для колекції користувачів визначає структуру документів, типи полів, валідацію та індекси. Схема `users` є центральною для системи автентифікації та авторизації.

Валідатори на рівні схеми забезпечують цілісність даних перед їх збереженням у базі. Кастомні валідатори для `address` та `did` перевіряють формат через регулярні вирази. Sparse індекс на полі `did` дозволяє багатьом документам мати null значення при збереженні унікальності для non-null значень.

Віртуальні поля не зберігаються в базі даних, але обчислюються динамічно при отриманні документа. Поле `shortAddress` генерує скорочену версію адреси для відображення в UI. Віртуальне поле `credentials` створює зв'язок з колекцією `Credential` через `populate` механізм Mongoose.

Схема для колекції документів є найбільш складною, оскільки містить різноманітні метадані про видані академічні документи.

Складені (compound) індекси оптимізують запити, що фільтрують за кількома полями одночасно. Наприклад, індекс `{ issuerDID: 1, issuanceDate: -1 }` прискорює запити типу "всі документи від конкретного емітента, відсортовані за датою видачі". Порядок полів в індексі важливий – MongoDB може використовувати префікси індексу для запитів за першими полями.

Middleware `pre('save')` виконується автоматично перед кожним збереженням документа, дозволяючи модифікувати дані або виконувати валідацію. У даному випадку він автоматично встановлює `revokedAt` при зміні `isRevoked` на `true`.

Агрегаційний `pipeline` у методі `getStatistics` демонструє потужність MongoDB для аналітики. Pipeline складається з етапів: `$match` фільтрує документи, `$group` групує та обчислює статистику, `$project` форматує вихідні дані. Такі запити виконуються ефективно на рівні бази даних без необхідності завантаження всіх документів у пам'ять додатку.

Схема для колекції верифікацій зберігає журнал всіх операцій перевірки документів, що важливо для аудиту та аналітики.

Схема `IPFSCache` для кешування документів, завантажених з IPFS, значно прискорює повторні запити.

TTL індекс з `expires: 604800` автоматично видаляє документи через 7 днів після їх створення, підтримуючи кеш актуальним та запобігаючи необмеженому зростанню колекції.

Для моніторингу стану бази даних та продуктивності запитів створено `utility` модуль `database.utils.js`.

Метод `explainQuery` аналізує план виконання запиту MongoDB, показуючи чи використовуються індекси, скільки документів було проскановано та скільки часу зайняло виконання. Це критично важливо для оптимізації повільних запитів.

Endpoint для моніторингу бази даних надає адміністраторам інформацію про стан системи.

У табл. 3.2 представлено статистику колекції MongoDB після тижня роботи з 50 документами та 180 верифікаціями.

Таблиця 3.2 – Статистика колекцій MongoDB

Колекція	Документи	Розмір	Середній розмір	Індексів	Примітки
users	25	12 KB	480 B	4	address, did, role
credentials	50	156 KB	3.1 KB	7	docHash, issuerDID, holderDID, складені
verifications	180	245 KB	1.4 KB	4	credentialCID, verifierDID
ipfscaches	35	420 KB	12 KB	3	cid, TTL 7 днів
diddocuments	22	89 KB	4 KB	2	did, TTL 24 години

Під час розробки було виявлено кілька практичних аспектів роботи з MongoDB. По-перше, початкова реалізація не включала індекси на часто використовувані поля, що призводило до повільних запитів при зростанні кількості документів. Аналіз через `explain()` показав COLLSCAN (collection scan) замість використання індексів. Додавання відповідних індексів зменшило час запитів з 200-300 мс до 5-15 мс.

По-друге, виявилася проблема з пам'яттю при використанні агрегаційних pipeline для великих колекцій. MongoDB за замовчуванням обмежує використання пам'яті агрегацією до 100 MB. Для складних аналітичних запитів було необхідно додати опцію `{ allowDiskUse: true }`, що дозволяє використовувати диск для тимчасових даних.

По-третє, TTL індекси не видаляють документи миттєво – MongoDB виконує фонові операції очищення приблизно раз на 60 секунд. Це означає, що документи можуть залишатися в базі до хвилини після закінчення терміну дії.

Реалізоване управління базою даних MongoDB забезпечує ефективне зберігання операційних даних системи з належною індексацією для швидких запитів. Валідація на рівні схем гарантує цілісність даних. TTL індекси

автоматично підтримують кеш актуальним. Методи аналітики та моніторингу дозволяють відстежувати продуктивність та стан системи в реальному часі.

3.3 Реалізація клієнтської частини

3.3.1 Структура React-додатку

Клієнтська частина системи реалізована як Single Page Application на базі бібліотеки React версії 18.2, що забезпечує динамічний користувацький інтерфейс з швидким відгуком на дії користувача без перезавантаження сторінки. Проєкт створений за допомогою Create React App, який надає оптимальну конфігурацію Webpack, Babel та інших інструментів збірки без необхідності ручного налаштування.

Структура файлів та папок організована за принципом розділення відповідальності, де кожен тип файлів має своє призначення та місце в ієрархії проєкту (див.рис. 3.6). Така організація полегшує навігацію по коду, підтримку та масштабування додатку.

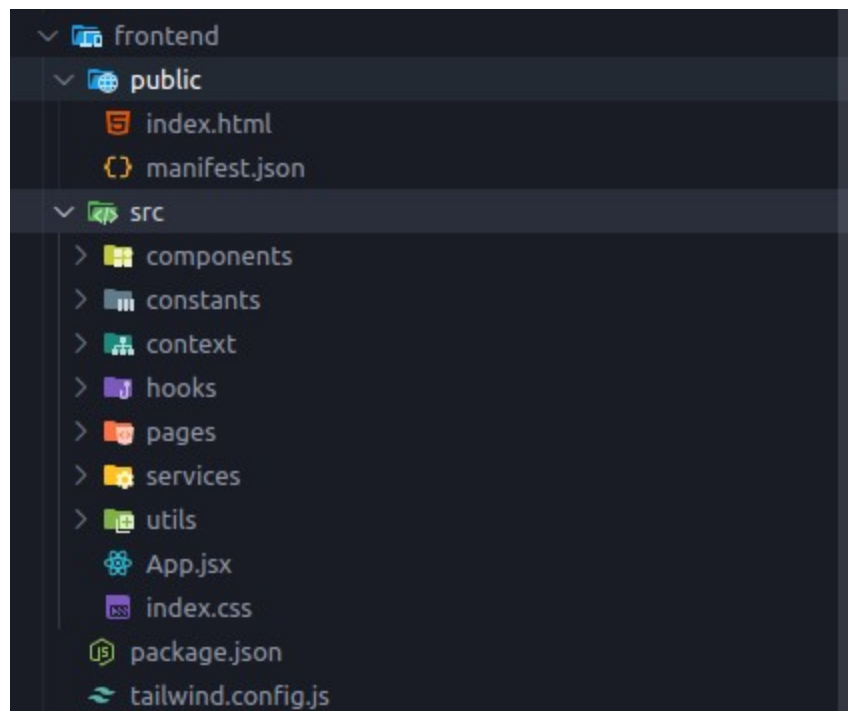


Рисунок 3.6 – Структура фронтенд частини застосунку

Папка `components` містить компоненти інтерфейсу, які використовуються повторно. Папка `common` включає базові UI-елементи, такі як кнопки, поля вводу, модальні вікна, які використовуються по всьому додатку. Папка `layout` містить компоненти структури сторінки – хедер з навігацією, футер та загальний `Layout` компонент, що обгортає всі сторінки. Папка `features` включає спеціалізовані компоненти з бізнес-логікою, специфічною для функціональності системи верифікації.

Папка `pages` організована за ролями користувачів. Кожна роль має власну підпапку зі сторінками, доступними лише користувачам з відповідними правами. Такий підхід полегшує розуміння структури додатку та дозволяє легко додавати нові функції для конкретних ролей.

Папка `services` інкапсулює всю логіку взаємодії з зовнішніми сервісами – `backend API`, `Web3` провайдером, `IPFS`. Це дозволяє змінювати реалізацію сервісів без модифікації компонентів, які їх використовують.

Налаштування `React Router` для навігації між сторінками виконується в головному компоненті `App.jsx`. `React Router` версії 6 надає декларативний API для визначення маршрутів та умовної навігації залежно від стану автентифікації та ролі користувача.

`Lazy loading` сторінок через `React.lazy()` дозволяє розділити `JavaScript bundle` на окремі частини, які завантажуються лише при переході на відповідну сторінку. Це значно зменшує початковий розмір завантаження та прискорює час до інтерактивності додатку. Компонент `Suspense` обгортає `lazy-завантажені` компоненти та відображає `fallback UI (LoadingSpinner)` під час завантаження коду сторінки.

Компонент `ProtectedRoute` забезпечує контроль доступу до маршрутів залежно від ролі користувача. Він перевіряє, чи користувач автентифікований та чи має він необхідну роль для доступу до сторінки.

Глобальний стан додатку управляється через React Context API, що дозволяє уникнути prop drilling (передачі props через багато рівнів компонентів) та надає централізоване місце для зберігання стану автентифікації та Web3 підключення.

AuthContext зберігає інформацію про поточного користувача, його роль, JWT токен та надає функції для входу та виходу.

Web3Context управляє підключенням до MetaMask та надає інформацію про поточну мережу, адресу користувача та функції для взаємодії з Web3.

Custom hooks useAuth та useWeb3 надають зручний доступ до контекстів з автоматичною перевіркою, що контекст використовується всередині відповідного Provider.

Конфігурація axios для API запитів виконується в окремому модулі api.js, що централізує налаштування базового URL, заголовків та interceptors для автоматичного додавання JWT токена до кожного запиту.

Interceptor для request автоматично додає JWT токен з localStorage до заголовка Authorization кожного запиту, якщо токен існує. Це усуває необхідність вручну додавати токен в кожному сервісному методі.

Interceptor для response обробляє типові помилки API. При отриманні статусу 401 (Unauthorized), система автоматично видаляє токен та перенаправляє користувача на сторінку входу. Це забезпечує безпечну обробку випадків expired токенів без необхідності перевіряти статус в кожному компоненті.

Базова структура Layout компонента забезпечує консистентність макету по всьому додатку з хедером, основним вмістом та футером.

Tailwind CSS класи використовуються для стилізації без написання власного CSS. Клас min-h-screen забезпечує мінімальну висоту екрану, flex flex-col створює вертикальний flexbox layout, flex-grow дозволяє основному вмісту займати весь доступний простір між хедером та футером.

Конфігурація Tailwind CSS налаштовується для використання кастомних кольорів та розширення дефолтної теми.

Організована структура React-додатку забезпечує масштабованість, зручність підтримки та розділення відповідальності між компонентами. Використання Context API для глобального стану, lazy loading для оптимізації завантаження та централізованої конфігурації axios для API запитів створює надійну основу для подальшої розробки функціональності.

3.3.2 Інтеграція з Web3 та MetaMask

MetaMask є найпопулярнішим криптографічним гаманцем для взаємодії з Ethereum-мережами через браузер, з понад 30 мільйонами активних користувачів станом на 2024 рік. Інтеграція з MetaMask забезпечує безпарольну автентифікацію користувачів через цифрові підписи та дозволяє підписувати транзакції без передачі приватних ключів на сервер.

MetaMask інжектуює об'єкт `window.ethereum` у глобальний scope браузера, який надає Ethereum Provider API згідно зі специфікацією EIP-1193. Цей API дозволяє веб-додаткам запитувати інформацію з блокчейну, відправляти транзакції та запитувати дозволи від користувача.

Модуль `web3Service.js` інкапсулює всю логіку взаємодії з MetaMask, надаючи високорівневі методи для підключення гаманця, підписування повідомлень та перевірки мережі. Використання окремого сервісу дозволяє легко замінити MetaMask на альтернативний провайдер (WalletConnect, Coinbase Wallet) без зміни компонентів.

Метод `eth_requestAccounts` є стандартним способом запиту дозволу на підключення згідно з EIP-1102. При першому виклику MetaMask відображає користувачу роруп з запитом дозволу на підключення до додатку. При наступних викликах, якщо дозвіл вже надано, MetaMask повертає список акаунтів без відображення роруп.

Створення `BrowserProvider` замість застарілого `Web3Provider` забезпечує сумісність з Ethers.js v6. `BrowserProvider` автоматично виявляє MetaMask через `window.ethereum` та налаштовує підключення до поточної мережі користувача.

Обробка коду помилки 4001 є важливою для користувацького досвіду. Цей код означає, що користувач відхилив запит у MetaMask роруп. Замість generic повідомлення про помилку, додаток надає зрозуміле пояснення, що дозволяє користувачу зрозуміти причину невдачі.

Компонент ConnectWallet реалізує інтерфейс для підключення MetaMask гаманця та відображає поточний статус підключення.

Трьохетапний процес автентифікації забезпечує безпеку без передачі приватних ключів. На першому етапі додаток запитує challenge (випадкове повідомлення) від сервера. На другому етапі користувач підписує це повідомлення у MetaMask, доводячи володіння приватним ключем адреси. На третьому етапі сервер верифікує підпис та видає JWT токен.

Відображення різних повідомлень залежно від етапу (Connecting, Waiting for signature, Authenticating) інформує користувача про поточний стан процесу, що особливо важливо під час очікування підпису в MetaMask, яке може займати кілька секунд.

Обробка зміни акаунту та мережі виконується через event listeners на об'єкті window.ethereum. Ці події генеруються автоматично, коли користувач змінює активний акаунт або мережу в MetaMask.

Подія accountsChanged генерується при зміні активного акаунту або коли користувач блокує/відключає гаманець. При зміні акаунту система автоматично виконує logout, оскільки JWT токен прив'язаний до конкретної адреси. Це запобігає ситуації, коли користувач автентифікований як одна адреса, але MetaMask підключений до іншої.

Подія chainChanged генерується при переключенні мережі (наприклад, з Ethereum mainnet на Polygon). MetaMask офіційно рекомендує перезавантажувати сторінку при зміні мережі, оскільки провайдер може зберігати застарілий стан. Timeout у 1 секунду дає користувачу час прочитати повідомлення перед перезавантаженням.

Сервіс authService обробляє взаємодію з backend API для автентифікації.

Компонент NetworkSwitcher дозволяє користувачам легко переключатися між різними мережами або додавати нову мережу до MetaMask.

Конфігурація мережі для wallet_addEthereumChain повинна включати всі обов'язкові поля: chainId у шістнадцятковому форматі з префіксом 0x, chainName, nativeCurrency з полями name, symbol та decimals, rpcUrls масив URL для підключення до мережі та blockExplorerUrls для перегляду транзакцій.

Збереження JWT токена в localStorage забезпечує персистентність сесії між перезавантаженнями сторінки. При завантаженні додатку AuthContext перевіряє наявність токена та автоматично відновлює стан автентифікації.

Діаграма послідовності Web3 автентифікації наведено на рис. 3.7.

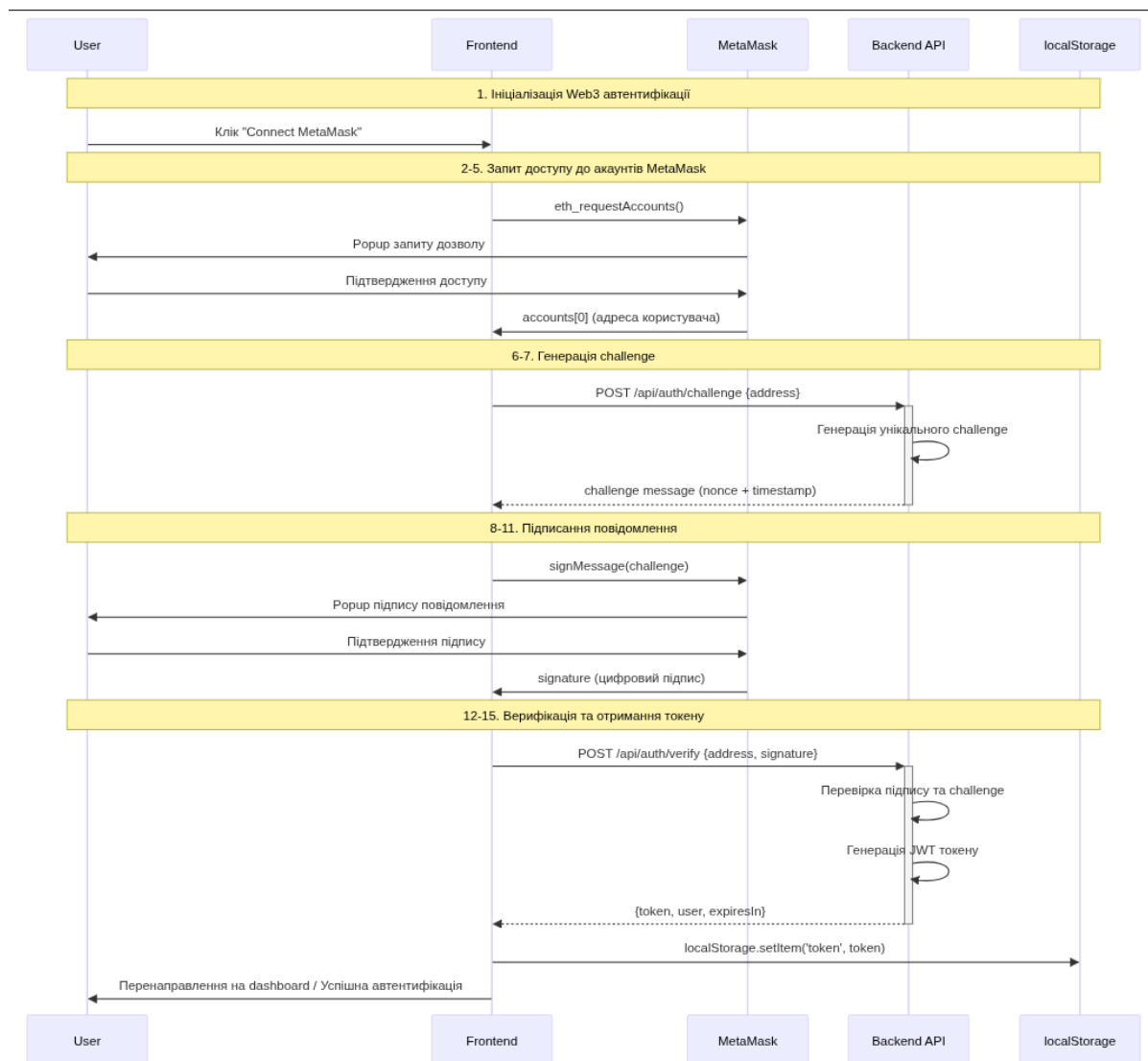


Рисунок 3.7 – Діаграма послідовності Web3 автентифікації

Під час розробки Web3 інтеграції було виявлено кілька практичних проблем. По-перше, MetaMask роруп може блокуватися браузерними налаштуваннями роруп blocker, особливо якщо запит на підпис відправляється не в прямій відповіді на клік користувача. Рішенням є ініціювання всіх Web3 операцій виключно через user interactions (кліки кнопок).

По-друге, виявилася проблема race condition при швидкому переключенні між акаунтами в MetaMask. Події accountsChanged можуть генеруватися швидше, ніж компоненти встигають оновити стан. Впровадження debouncing для обробників подій вирішило цю проблему.

По-третє, користувачі іноді підключали MetaMask до неправильної мережі (наприклад, mainnet замість testnet), що призводило до помилок при взаємодії зі смарт-контрактами. Додавання NetworkSwitcher та відображення поточної мережі у header покращило UX та зменшило кількість помилок користувачів.

Реалізована інтеграція з MetaMask забезпечує безпечну безпарольну автентифікацію користувачів через цифрові підписи. Обробка всіх можливих станів MetaMask (не встановлений, заблокований, неправильна мережа) та надання зрозумілих повідомлень користувачам створює плавний користувацький досвід навіть для користувачів, які вперше працюють з Web3 додатками.

3.3.3 Реалізація інтерфейсу емітента

Інтерфейс емітента призначений для університетів та інших освітніх установ, які мають право видавати академічні документи. Функціональність включає видачу нових документів, перегляд списку виданих документів з можливістю фільтрації та пошуку, відкликання документів у випадку помилок або академічної недобросовісності, а також базову статистику для моніторингу активності.

Сторінка видачі документа `IssueCredential.jsx` містить форму з валідацією полів та відображенням прогресу виконання багатоетапного процесу реєстрації документа в блокчейні та IPFS.

Автоматична генерація DID з Ethereum адреси покращує користувацький досвід, оскільки емітенту не потрібно вручну вводити DID у правильному форматі. При введенні валідної адреси поле `holderDID` автоматично заповнюється відповідним DID.

Валідація форми виконується на frontend перед відправкою на сервер, що запобігає непотрібним API запитам з невалідними даними. Регулярні вирази перевіряють формат Ethereum адреси (40 шістнадцяткових символів після 0x) та DID (`did:ethr:` з наступною адресою).

Індикатор прогресу інформує користувача про поточний етап довготривалого процесу видачі документа: підготовка даних, завантаження до IPFS, хешування, реєстрація в блокчейні. Це особливо важливо, оскільки процес може займати 5-10 секунд при роботі з реальними мережами.

Сервіс `credentialService` обробляє взаємодію з backend API для операцій з документами.

Сторінка списку виданих документів `IssuedCredentials.jsx` відображає таблицю з пагінацією, фільтрацією та можливістю швидкого пошуку.

Компонент `CredentialCard` відображає інформацію про документ у зручному форматі картки з можливістю розгортання деталей.

Dashboard емітента відображає базову статистику та швидкий доступ до основних функцій.

Реалізований інтерфейс емітента надає всю необхідну функціональність для університетів: видачу документів з валідацією даних та відображенням прогресу, перегляд списку виданих документів з фільтрацією та пагінацією, можливість відкликання документів з підтвердженням та вказанням причини, dashboard з базовою статистикою для моніторингу активності. Використання картки документа з можливістю розгортання деталей забезпечує зручний перегляд інформації без перевантаження інтерфейсу(див.рис. 3.8).

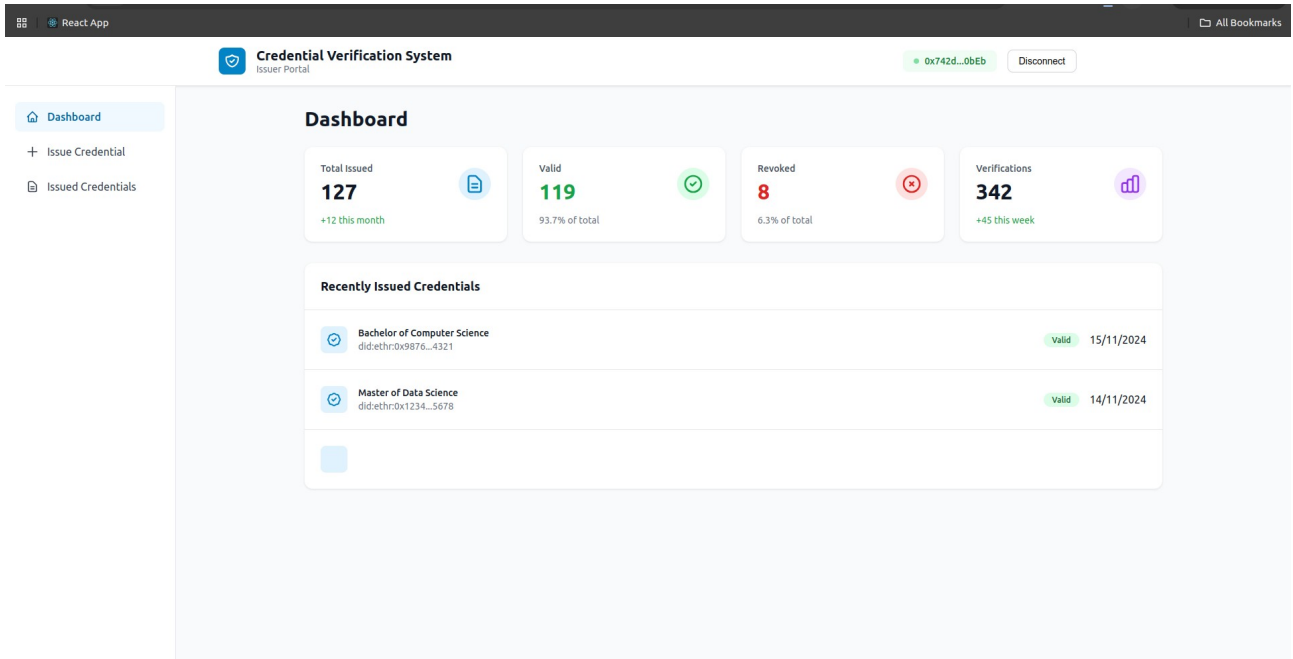


Рисунок 3.8 – Інтерфейс емітента

3.3.4 Реалізація інтерфейсу власника документів

Інтерфейс власника документів призначений для випускників університетів, які отримали академічні документи та потребують зручного способу їх зберігання, перегляду та обміну з верифікаторами. Функціональність включає перегляд списку отриманих документів, детальну інформацію про кожен документ з можливістю завантаження, генерацію посилань для обміну документами та відображення QR-коду для швидкого доступу.

Сторінка `MyCredentials.jsx` відображає всі документи, що належать поточному користувачу, з можливістю фільтрації за типом та статусом.

Компонент `ShareModal` надає різні способи поділитися документом: копіювання IPFS CID, генерація QR-коду або створення безпечного посилання.

`Dashboard` власника відображає загальну статистику та швидкий доступ до документів.

Сервісний метод для отримання документів власника додається до `credentialService`.

Реалізований інтерфейс власника документів надає всю необхідну функціональність для випускників: перегляд списку отриманих документів з фільтрацією за статусом, детальну інформацію про кожен документ з посиланнями на IPFS та блокчейн-транзакції, зручні способи обміну документами через IPFS CID, QR-код або посилання, dashboard з загальною статистикою та швидким доступом. Інтерфейс розроблений з акцентом на простоту використання для нетехнічних користувачів, приховуючи складність блокчейн-технології за зрозумілими елементами інтерфейсу(див.рис. 3.9).

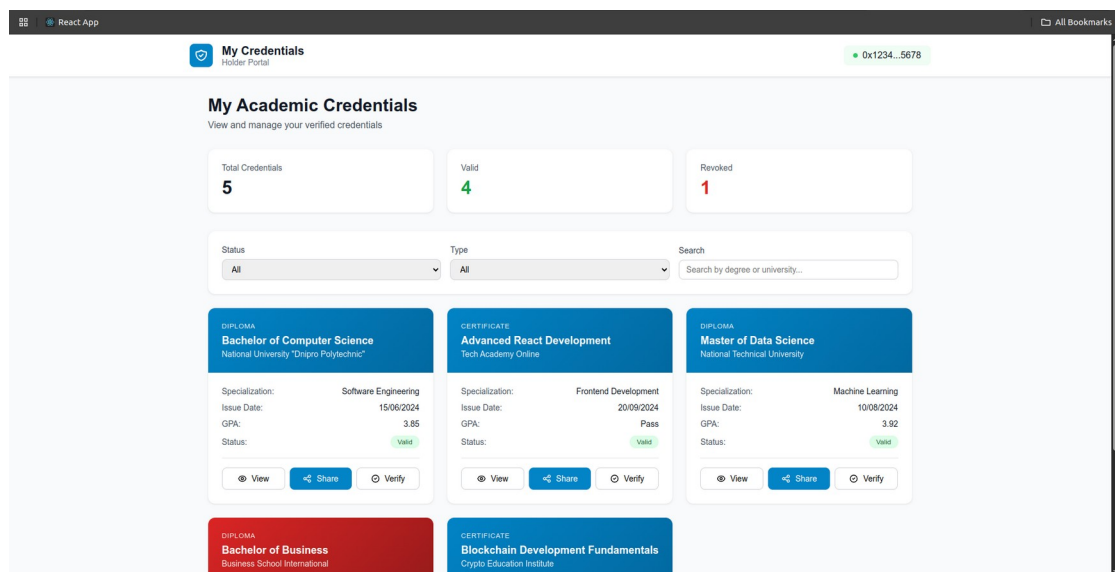


Рисунок 3.9 – Інтерфейс власника документів

3.3.5 Реалізація інтерфейсу верифікатора

Інтерфейс верифікатора призначений для роботодавців, рекрутингових агенцій та інших організацій, які потребують перевіряти автентичність академічних документів кандидатів. Функціональність включає введення IPFS CID або завантаження JSON файлу для верифікації, відображення детального результату верифікації з покроковим прогресом, перегляд історії попередніх верифікацій та відображення часу виконання кожного етапу для моніторингу продуктивності.

Сторінка `VerifyCredential.jsx` містить форму для ініціювання верифікації та відображення результату з детальною інформацією про документ та процес перевірки.

Компонент `VerificationResult` відображає детальний результат верифікації з візуальним індикатором успіху або невдачі.

Сторінка історії верифікацій `VerificationHistory.jsx` відображає список всіх попередніх верифікацій користувача.

Dashboard верифікатора відображає статистику верифікацій та швидкий доступ до функцій.

Сервіс верифікації обробляє взаємодію з backend API для операцій верифікації.

Реалізований інтерфейс верифікатора надає всю необхідну функціональність для роботодавців та рекрутингових агенцій: дві опції верифікації (IPFS CID або JSON файл) для гнучкості, детальне відображення результату верифікації з візуальними індикаторами, метрики продуктивності для кожного етапу процесу, історія всіх попередніх верифікацій з фільтрацією за результатом, dashboard зі статистикою для моніторингу активності. Інтерфейс спроектований для максимальної зручності з чітким відображенням статусу верифікації та детальною інформацією про документ без необхідності технічних знань про блокчейн (див.рис.3.10).

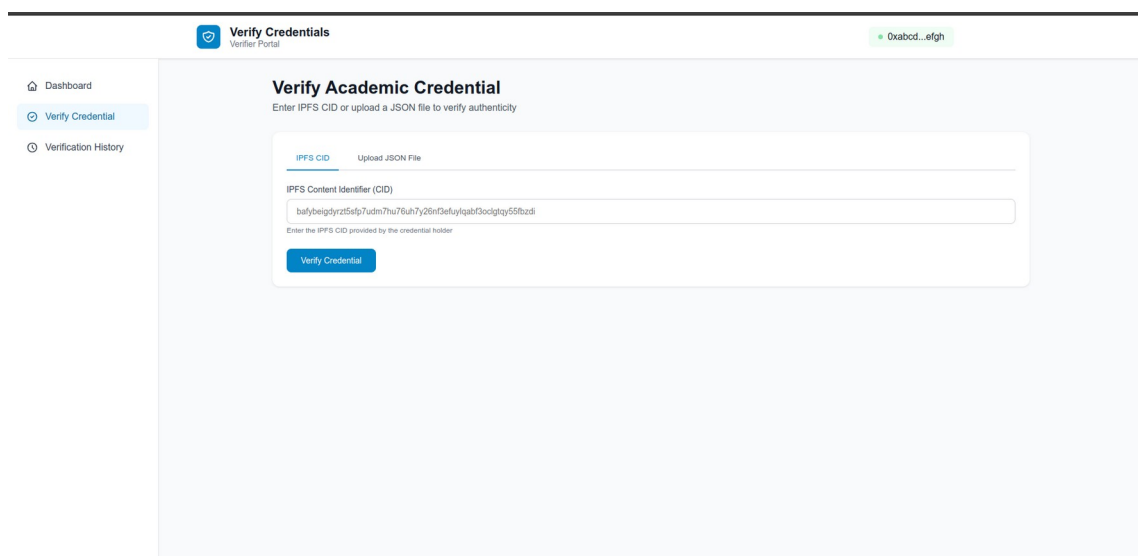


Рисунок 3.10 – Інтерфейс верифікатора

У табл. 3.3 наведено порівняльну характеристику функціональних можливостей інтерфейсів для трьох ролей системи.

Таблиця 3.3 – Порівняння функціональних можливостей інтерфейсу

Функціональність	Емітент	Власник	Верифікатор
Видача документів	+	-	-
Відкликання документів	+	-	-
Перегляд виданих документів	+	-	-
Перегляд отриманих документів	-	+	-
Обмін документами (QR, посилання)	-	+	-
Верифікація документів	-	-	+
Історія верифікацій	-	-	+
Dashboard зі статистикою	+	+	+
Фільтрація та пошук	+	+	-

Розділення функціональності за ролями забезпечує безпеку та зручність використання. Реалізовані інтерфейси для трьох основних ролей системи забезпечують повний життєвий цикл академічних документів від видачі університетом до верифікації роботодавцем, надаючи при цьому власнику документів повний контроль над своїми даними відповідно до принципів самостійної суверенної ідентифікації.

3.3 Висновки за розділом

Таким чином, у цьому розділі було детально представлено архітектуру та реалізацію децентралізованої системи верифікації академічних документів, яка базується на технологіях блокчейн, IPFS та децентралізованих ідентифікаторів.

Для серверної частини системи описано повний стек технологій, включаючи розробку смарт-контракту DocumentRegistry на мові Solidity з реалізацією механізмів контролю доступу та відкликання документів. Детально висвітлено інтеграцію з блокчейн-мережею через бібліотеку Ethers.js, що забезпечує надійну взаємодію з Ethereum-сумісними мережами. Представлено реалізацію децентралізованого зберігання документів через IPFS з багаторівневим кешуванням для оптимізації продуктивності. Описано архітектуру RESTful API з автентифікацією через JWT-токени та Web3-підписи, що забезпечує безпарольний доступ до системи.

Для клієнтської частини наведено структуру Single Page Application на базі React з чітким розділенням компонентів за функціональними областями та ролями користувачів. Описано інтеграцію з MetaMask для безпечної взаємодії з блокчейном без передачі приватних ключів на сервер. Детально представлено реалізацію трьох спеціалізованих інтерфейсів для емітентів, власників документів та верифікаторів, кожен з яких оптимізований під специфічні потреби відповідної ролі.

Представлено структуру бази даних MongoDB з п'ятьма основними колекціями (users, credentials, verifications, ipfscaches, diddocuments) та визначено відношення між сутностями через DID-ідентифікатори. Описано систему індексування для оптимізації найбільш частих запитів та TTL індекси для автоматичного управління кешем.

Чітко визначено особливості функціонування системи, включаючи концепцію ізольованості емітентів, де кожен університет функціонує як незалежна сутність з власними криптографічними ключами. Описано механізм

верифікації документів через криптографічне хешування та перевірку статусу в блокчейні, що забезпечує незмінність та прозорість процесу.

Деталізовано багаторівневу систему захисту від несанкціонованого доступу, що включає автентифікацію через цифрові підписи Web3, авторизацію на основі ролей з перевіркою на рівні API endpoints, шифрування передачі даних через HTTPS, контроль доступу на рівні смарт-контракту та rate limiting для захисту від зловживань.

Реалізована архітектура забезпечує необхідний баланс між децентралізацією, продуктивністю та зручністю використання, а також ефективністю витрат на gas через використання приватної мережі Proof-of-Authority.

4 ЕКСПЛУАТАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМИ

4.1 Призначення програми

Децентралізована система верифікації академічних документів призначена для автоматизації процесу видачі, зберігання та перевірки автентичності освітніх документів за допомогою технологій блокчейн, децентралізованого сховища IPFS та децентралізованих ідентифікаторів стандарту W3C DID. Система вирішує проблему довготривалої та витратної процедури верифікації академічних кваліфікацій, яка в традиційному підході вимагає прямого контакту між роботодавцем та навчальним закладом, що може займати від кількох днів до кількох тижнів.

Основною цільовою аудиторією системи є три категорії користувачів. Перша категорія – університети та інші освітні установи, які виступають емітентами академічних документів та потребують надійного способу видачі цифрових дипломів, сертифікатів та транскриптів із гарантією їх незмінності та можливістю відкликання у разі виявлення помилок або академічної недобросовісності. Друга категорія – випускники навчальних закладів, які є власниками документів та потребують зручного способу зберігання своїх академічних кваліфікацій з можливістю швидкого обміну з потенційними роботодавцями без залежності від централізованих сервісів. Третя категорія – роботодавці та рекрутингові агенції, які виступають верифікаторами документів та потребують швидкого та надійного способу перевірки автентичності академічних кваліфікацій кандидатів без необхідності звертатися до університетів.

Система вирішує декілька ключових проблем традиційного процесу верифікації документів. По-перше, вона усуває необхідність довготривалого обміну листами між роботодавцем та університетом, оскільки вся інформація про документ доступна в блокчейні та IPFS для миттєвої верифікації. По-друге, система захищає від підробки документів через використання криптографічних

хешів та цифрових підписів, що робить неможливою зміну вмісту документа після його реєстрації в блокчейні. По-третє, система надає власникам документів повний контроль над своїми даними згідно з принципами Self-Sovereign Identity, дозволяючи самостійно вирішувати, кому і коли надавати доступ до своїх академічних кваліфікацій. По-четверте, система забезпечує прозорість процесу верифікації через публічний реєстр в блокчейні, де будь-який учасник може перевірити статус документа без необхідності довіряти централізованому органу.

Переваги системи над традиційними методами верифікації проявляються в кількох аспектах. Швидкість верифікації скорочується з кількох днів або тижнів до кількох секунд, оскільки перевірка відбувається автоматично через запити до блокчейну та IPFS без участі людини. Вартість процесу значно зменшується через відсутність необхідності оплати адміністративних послуг університету за підтвердження документів. Надійність верифікації підвищується через використання криптографічних методів замість довіри до паперових документів або централізованих баз даних, які можуть бути скомпрометовані. Доступність системи забезпечується через веб-інтерфейс, що працює в будь-якому сучасному браузері без необхідності встановлення спеціалізованого програмного забезпечення.

Технічна реалізація системи базується на перевірених технологіях та стандартах. Блокчейн-рівень використовує Ethereum-сумісні мережі з підтримкою смарт-контрактів на мові Solidity для забезпечення незмінного реєстру хешів документів. Рівень зберігання використовує IPFS для децентралізованого зберігання повного вмісту документів у форматі JSON згідно зі специфікацією W3C Verifiable Credentials. Ідентифікація користувачів реалізована через децентралізовані ідентифікатори стандарту W3C DID з методом did:ethr, що природно інтегрується з Ethereum-екосистемою. Серверна частина побудована на Node.js з використанням Express для REST API та MongoDB для зберігання операційних даних та кешування. Клієнтська частина

реалізована як Single Page Application на базі React з інтеграцією MetaMask для безпечної взаємодії з блокчейном.

Функціональні можливості системи охоплюють повний життєвий цикл академічного документа від видачі до верифікації. Емітенти мають можливість створювати верифіковані документи через веб-форму з автоматичною генерацією криптографічних хешів, завантаженням до IPFS та реєстрацією в блокчейні через смарт-контракт DocumentRegistry. Система підтримує різні типи документів, включаючи дипломи про вищу освіту, сертифікати про проходження курсів та академічні транскрипти з оцінками. Відкликання документів у випадку виявлення помилок або порушень відбувається через зміну статусу в блокчейні без видалення самого документа, що зберігає повний аудиторський слід. Власники документів мають можливість переглядати всі свої академічні кваліфікації в єдиному інтерфейсі, самостійно перевіряти їх валідність та створювати безпечні способи обміну через IPFS CID, QR-коди або прямі посилання. Верифікатори можуть перевіряти автентичність документів за кілька секунд через введення IPFS CID або завантаження JSON-файлу з детальним відображенням результату кожного етапу перевірки.

Система розроблена з урахуванням вимог захисту персональних даних згідно з принципами GDPR через мінімізацію даних у публічному блокчейні, де зберігаються лише криптографічні хеші без будь-якої персональної інформації. Повний вміст документів з особистими даними зберігається в IPFS з контрольованим доступом через DID-ідентифікатори. Архітектура системи побудована навколо концепції Self-Sovereign Identity, що надає користувачам повний контроль над своїми даними без залежності від централізованих установ або посередників.

Практичне застосування системи дозволяє університетам скоротити адміністративні витрати на обробку запитів про підтвердження документів, випускникам мати постійний доступ до своїх академічних кваліфікацій незалежно від стану університету або його баз даних, а роботодавцям швидко перевіряти кваліфікації кандидатів на етапі відбору без затримок у процесі

найму. Децентралізована природа системи забезпечує її стійкість до відмов окремих компонентів та захист від втрати даних через розподілене зберігання в блокчейні та IPFS-мережі.

4.2 Умови використання системи

Для коректної роботи децентралізованої системи верифікації академічних документів необхідне виконання певних технічних вимог як на стороні користувача, так і на стороні серверної інфраструктури. Ці вимоги забезпечують стабільну взаємодію з блокчейн-мережею, IPFS та веб-інтерфейсом системи.

4.2.1 Технічні вимоги до робочого місця користувача

Для роботи з клієнтською частиною системи користувач повинен мати комп'ютер або мобільний пристрій з мінімальними системними характеристиками. Обсяг оперативної пам'яті має становити не менше чотирьох гігабайт для забезпечення плавної роботи сучасного веб-браузера з відкритими вкладками системи та MetaMask розширення. Процесор будь-якої сучасної архітектури з тактовою частотою від двох гігагерц достатній для обробки криптографічних операцій на стороні клієнта. Вільне місце на диску має становити не менше п'ятисот мегабайт для зберігання кешу браузера, локальних копій документів у форматі JSON та тимчасових файлів MetaMask.

Веб-браузер є основним інструментом взаємодії з системою та має відповідати певним вимогам. Підтримуються Google Chrome або Microsoft Edge версії 100 та новіші, Mozilla Firefox версії 100 та новіші, Safari версії 15 та новіші. Браузер має бути налаштований з увімкненим JavaScript, оскільки вся клієнтська логіка реалізована на React та вимагає виконання коду в браузері. Cookies та Local Storage мають бути дозволені для зберігання JWT-токенів автентифікації та кешування даних користувача. Роруп-вікна мають бути

дозволені для домену системи, оскільки MetaMask відкриває модальні вікна для підтвердження транзакцій та підписування повідомлень.

Швидкість інтернет-з'єднання впливає на комфорт роботи з системою через необхідність завантаження документів з IPFS-мережі та взаємодії з блокчейн-вузлами. Мінімальна рекомендована швидкість становить п'ять мегабіт на секунду для завантаження, що забезпечує прийнятний час завантаження документів розміром до ста кілобайт з IPFS протягом трьох-п'яти секунд. Для комфортної роботи рекомендується швидкість від десяти мегабіт на секунду, що дозволяє швидко завантажувати інтерфейс системи та документи без помітних затримок. Стабільність з'єднання є критично важливою для успішного завершення блокчейн-транзакцій, оскільки розрив з'єднання під час відправки транзакції може призвести до невизначеного стану операції.

Обов'язковою вимогою для всіх користувачів є наявність встановленого Web3-гаманця для автентифікації та підписування транзакцій. MetaMask є рекомендованим рішенням через його широке розповсюдження та повну підтримку в системі. Розширення MetaMask встановлюється безкоштовно з офіційних магазинів розширень браузерів Chrome Web Store для Google Chrome та Edge, або Firefox Add-ons для Mozilla Firefox. Користувач має створити новий гаманець або імпортувати існуючий через seed phrase з дванадцяти слів. Після створення гаманця необхідно налаштувати підключення до відповідної блокчейн-мережі, що використовується системою.

Для роботи з тестовою версією системи користувачу необхідні тестові токени для оплати gas при виконанні транзакцій в блокчейні. У випадку використання локальної Hardhat Network тестові токени автоматично надаються при генерації акаунтів через скрипт розгортання. Для роботи з публічною testnet мережею Polygon Mumbai користувач може отримати безкоштовні тестові MATIC токени через публічні faucet сервіси, такі як Polygon Faucet або Alchemy Faucet, що надають достатню кількість токенів для виконання декількох десятків транзакцій. Мінімальний баланс для комфортної роботи

становить приблизно один тестовий МАТІС, що забезпечує можливість виконання двадцяти-тридцяти операцій реєстрації або відкликання документів.

4.2.2 Вимоги до серверного середовища

Розгортання backend-компонентів системи вимагає налаштування серверного середовища з встановленими необхідними залежностями та сервісами. Операційна система сервера може бути будь-якою сучасною версією Linux, такою як Ubuntu 20.04 або новіша, macOS для локальної розробки або Windows 10 з підсистемою WSL2 для сумісності з Unix-інструментами.

Node.js runtime є основою для виконання серверної логіки та має бути версії 18 або новішої для підтримки всіх використаних можливостей JavaScript, включаючи top-level await та сучасні API криптографії. Встановлення Node.js включає також пакетний менеджер npm або yarn для управління залежностями проєкту. Після встановлення Node.js необхідно виконати команду `npm install` в кореневій директорії backend проєкту для завантаження всіх залежностей, включаючи express версії 4.18, ethers версії 6.4, ipfs-http-client версії 60.0, mongoose версії 7.0 та інші бібліотеки, специфіковані в файлі `package.json`.

MongoDB версії 6.0 або новішої використовується для зберігання операційних даних системи, включаючи метадані документів, користувацьких профілів, результатів верифікацій та кешу IPFS. MongoDB може бути встановлена локально на сервері або використовуватися як хмарний сервіс через MongoDB Atlas. Для локальної установки необхідно завантажити Community Edition з офіційного сайту MongoDB та виконати стандартну процедуру встановлення для відповідної операційної системи. Після встановлення MongoDB має бути запущена як сервіс з налаштованим портом за замовчуванням 27017. Рекомендований обсяг вільного місця для бази даних становить не менше п'яти гігабайт для зберігання метаданих декількох тисяч документів та кешу.

IPFS Desktop або доступ до IPFS gateway є необхідним для роботи з децентралізованим сховищем документів. Для локальної розробки рекомендується встановити IPFS Desktop – standalone додаток з графічним інтерфейсом, що автоматично запускає IPFS вузол з HTTP API на порту 5001. IPFS Desktop завантажується з офіційного репозиторію GitHub та встановлюється як звичайна desktop програма. Після першого запуску IPFS Desktop автоматично налаштовує вузол та підключається до глобальної IPFS мережі.

Блокчейн-інфраструктура вимагає доступу до Ethereum-сумісної мережі для розгортання смарт-контракту та виконання транзакцій. Для локальної розробки використовується Hardhat Network – вбудована Ethereum-мережа, що автоматично запускається при виконанні команди `npm run hardhat node` в директорії проєкту. Hardhat Network емулює повноцінний Ethereum-вузол з миттєвим підтвердженням транзакцій та попередньо налаштованими тестовими акаунтами з балансом. Для підключення до публічної testnet мережі, такої як Polygon Mumbai, необхідно отримати RPC endpoint через сервіси Infura або Alchemy, що вимагає безкоштовної реєстрації та створення проєкту в їх dashboard. RPC URL використовується для відправки транзакцій та читання стану блокчейну.

Достатній баланс криптовалюти для оплати gas є критично важливим для виконання операцій запису в блокчейн. Адреса сервера, приватний ключ якої зберігається у змінних оточення, має містити достатню кількість нативної криптовалюти мережі для оплати транзакцій. Для локальної Hardhat Network баланс встановлюється автоматично при генерації акаунтів. Для Polygon Mumbai testnet необхідно отримати тестові MATIC токени через faucet сервіси, мінімальний рекомендований баланс становить десять MATIC для виконання декількох сотень транзакцій.

Конфігурація змінних оточення виконується через створення файлу `.env` в кореневій директорії backend проєкту з наступними обов'язковими параметрами. PORT визначає порт для Express сервера, за замовчуванням 3001.

MONGODB_URI містить рядок підключення до MongoDB або URL до MongoDB Atlas. JWT_SECRET є секретним ключем для підписування JWT-токенів, має бути випадковим рядком довжиною не менше тридцяти двох символів. BLOCKCHAIN_NETWORK визначає мережу для підключення, може бути hardhat, mumbai або polygon. RPC_URL містить URL для підключення до блокчейн-вузла. PRIVATE_KEY є приватним ключем адреси сервера для підписування транзакцій у шістнадцятковому форматі без префіксу 0x. CONTRACT_ADDRESS містить адресу розгорнутого смарт-контракту DocumentRegistry. IPFS_API_URL визначає URL для підключення до IPFS API.

4.2.3 Обмеження та застереження

Система має певні технічні обмеження, які необхідно враховувати при експлуатації. По-перше, система працює виключно з Ethereum-сумісними блокчейн-мережами, що підтримують виконання смарт-контрактів на мові Solidity. Інтеграція з іншими блокчейн-платформами, такими як Bitcoin або Hyperledger Fabric, вимагає значних архітектурних змін та виходить за межі поточної реалізації.

По-друге, система залежить від доступності IPFS-мережі для завантаження повного вмісту документів. У випадку тимчасової недоступності IPFS gateway або низької кількості peers, що зберігають конкретний документ, час завантаження може значно збільшитися або операція може завершитися з timeout помилкою. Для мінімізації цього ризику реалізовано багаторівневе кешування через MongoDB, однак перше завантаження документа завжди залежить від IPFS.

По-третє, всі операції запису в блокчейн, включаючи реєстрацію та відкликання документів, вимагають оплати gas у нативній криптовалюті мережі. Недостатній баланс на адресі сервера призведе до відхилення транзакцій з помилкою insufficient funds. Необхідно регулярно моніторити

баланс через endpoint `/api/health` та поповнювати його при досягненні критично низького рівня.

По-четверте, швидкість підтвердження транзакцій в блокчейні залежить від завантаженості мережі та може варіюватися від миттєвого підтвердження в локальній Hardhat Network до декількох хвилин у публічних мережах під час періодів високого навантаження. Користувачі мають бути проінформовані про можливі затримки та необхідність очікування підтвердження транзакції перед виконанням наступних операцій.

По-п'яте, система зберігає JWT-токени автентифікації в `localStorage` браузера, що робить їх вразливими до XSS атак у випадку впровадження шкідливого коду на сторінку. Для мінімізації ризику реалізовано валідацію всіх вхідних даних та санітизацію HTML.

4.3 Експлуатація програми

4.3.1 Розгортання системи

4.3.1.1 Налаштування backend-сервера

Розгортання серверної частини системи починається з клонування репозиторію проекту та встановлення необхідних залежностей. Виконується команда `git clone` для завантаження вихідного коду, після чого в директорії `backend` виконується `npm install` для встановлення всіх залежностей, специфікованих в `package.json`. Процес встановлення завантажує близько двохсот мегабайт бібліотек, включаючи `express`, `ethers`, `mongoose`, `ipfs-http-client` та інші модулі.

Конфігурація змінних оточення виконується через створення файлу `.env` в кореневій директорії `backend`. Файл має містити наступні параметри: `PORT=3001` для визначення порту сервера, `MONGODB_URI=mongodb://localhost:27017/credential-verification` для підключення до локальної MongoDB, `JWT_SECRET` з випадковим рядком довжиною тридцять два символи, `BLOCKCHAIN_NETWORK=hardhat` для

локальної розробки, `RPC_URL=http://127.0.0.1:8545` для підключення до Hardhat Network, `PRIVATE_KEY` з приватним ключем тестового акаунту та `IPFS_API_URL=http://127.0.0.1:5001` для локального IPFS Desktop.

Підключення до MongoDB вимагає запущеного сервісу MongoDB на порту 27017. Для перевірки підключення можна використати команду `mongosh` для відкриття MongoDB shell та виконання команди `show dbs` для відображення списку баз даних. База даних `credential-verification` створюється автоматично при першому підключенні backend-сервера.

Запуск локального IPFS вузла виконується через IPFS Desktop додаток, який після встановлення запускається з системного меню. При першому запуску IPFS Desktop автоматично ініціалізує репозиторій, генерує ідентифікатор вузла та підключається до глобальної IPFS мережі. HTTP API автоматично стає доступним на адресі `http://127.0.0.1:5001` для використання backend-сервером.

Компіляція та розгортання смарт-контракту виконується через Hardhat фреймворк. Спочатку у директорії `backend` виконується команда `npx hardhat compile` для компіляції Solidity контракту `DocumentRegistry.sol`, що генерує ABI та bytecode в директорії `artifacts`. Після компіляції виконується `npx hardhat node` для запуску локальної Ethereum мережі, що відображає список з десяти тестових акаунтів з балансами по десять тисяч ETH кожен. У новому терміналі виконується `npx hardhat run scripts/deploy.js --network localhost` для розгортання контракту, що виводить адресу розгорнутого контракту у форматі `0x5FbDB2315678afecb367f032d93F642f64180aa3`. Ця адреса копіюється у файл `.env` як значення змінної `CONTRACT_ADDRESS`.

Запуск Express сервера виконується командою `npm run dev`, що використовує `nodemon` для автоматичного перезапуску при зміні файлів. Успішний запуск відображає повідомлення `Server running on port 3001` та `Connected to MongoDB`. Для перевірки працездатності API можна виконати GET-запит до endpoint `http://localhost:3001/api/health`, що повертає JSON з

інформацією про статус підключення до MongoDB, блокчейну та IPFS, а також поточний баланс адреси сервера.

4.3.1.2 Налаштування frontend-додатку

Встановлення залежностей клієнтської частини виконується в директорії frontend командою `npm install`, що завантажує React, react-router-dom, axios, ethers та інші бібліотеки. Процес встановлення займає кілька хвилин залежно від швидкості інтернет-з'єднання.

Конфігурація API endpoints виконується через створення файлу `.env` в директорії frontend з параметром `REACT_APP_API_URL=http://localhost:3001` для визначення базового URL backend-сервера. Цей параметр використовується в модулі `src/services/api.js` для створення axios instance з налаштованим `baseURL`.

Запуск development сервера виконується командою `npm start`, що запускає Webpack dev server на порту 3000 та автоматично відкриває браузер з адресою `http://localhost:3000`. Успішний запуск відображає компільований проєкт з головною сторінкою системи, що містить кнопку Connect Wallet для підключення MetaMask.

4.3.2 Робота з системою як емітент

4.3.2.1 Підключення MetaMask та аутентифікація

Встановлення MetaMask виконується через Chrome Web Store або Firefox Add-ons за пошуковим запитом MetaMask. Після встановлення розширення з'являється іконка лисиці в панелі інструментів браузера. При першому запуску MetaMask пропонує створити новий гаманець або імпортувати існуючий. Для створення нового гаманця необхідно придумати пароль довжиною не менше восьми символів та зберегти seed phrase з дванадцяти слів у безпечному місці.

Підключення до локальної Hardhat Network виконується через налаштування MetaMask. У меню мереж обирається опція Add Network та вводяться параметри: Network Name як Hardhat Local, RPC URL як `http://127.0.0.1:8545`, Chain ID як 31337, Currency Symbol як ETH. Після збереження мережа стає доступною для вибору в dropdown меню.

Імпорт тестового акаунту виконується через меню MetaMask опцією Import Account. Приватний ключ одного з тестових акаунтів, відображених при запуску `prx hardhat node`, копіюється та вставляється в поле приватного ключа. Після імпорту акаунт з'являється в списку акаунтів з балансом десять тисяч ETH.

Процес автентифікації в системі починається з натискання кнопки Connect Wallet на головній сторінці. MetaMask відкриває поп-ап-вікно з запитом дозволу на підключення до сайту, де необхідно натиснути Connect для підтвердження. Після підключення система відправляє запит на отримання challenge-повідомлення з сервера, MetaMask знову відкриває поп-ап з запитом підпису повідомлення Sign this message to authenticate. Після натискання Sign сервер верифікує підпис та повертає JWT-токен, який зберігається в localStorage. Користувач автоматично перенаправляється на dashboard відповідно до своєї ролі. Приклад інтерфейсу аутентифікації наведено на рис. 4.1.

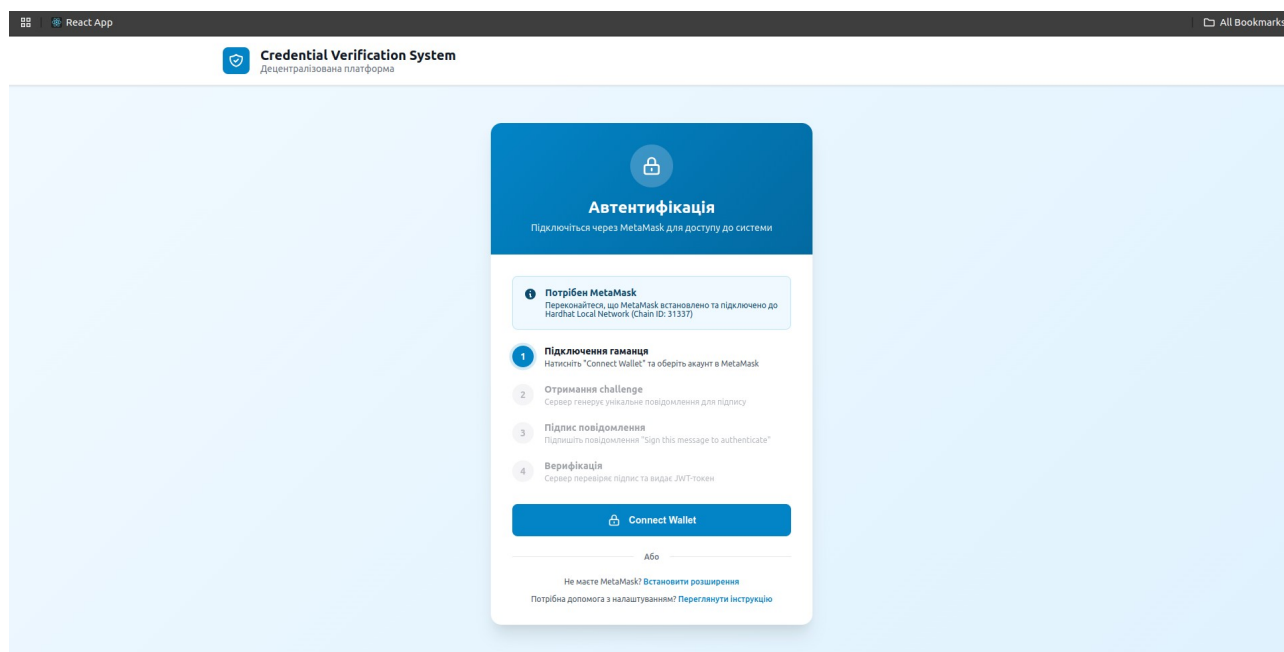


Рисунок 4.1 – Інтерфейс аутентифікації

4.3.2.2 Видача документа

Навігація до сторінки видачі документів виконується через меню Issue Credential у sidebar. Сторінка містить форму з полями для введення інформації про документ. Поле Holder Address приймає Ethereum-адресу випускника у форматі 0x742d35Cc6634C0532925a3b844Bc9e7595f0bEb, при введенні валідної адреси поле Holder DID автоматично заповнюється значенням did:ethr:0x742d35Cc6634C0532925a3b844Bc9e7595f0bEb.

Заповнення форми виконується з наступними даними як приклад: Document Type обирається зі списку Diploma, Degree Name вводиться як Bachelor of Computer Science, Major вводиться як Software Engineering, Issue Date обирається через date picker як 2024-06-15, GPA вводиться як 3.85. Всі поля мають валідацію на стороні клієнта, помилки відображаються червоним текстом під відповідним полем.

Процес видачі документа ініціюється натисканням кнопки Issue Credential. Компонент ProgressIndicator відображає чотири етапи виконання: Preparing document з синім кольором під час створення структури документа на сервері, Uploading to IPFS під час завантаження документа у децентралізоване

сховище, Registering in Blockchain під час відправки транзакції до смарт-контракту з автоматичним відкриттям MetaMask для підтвердження транзакції користувачем, Saving metadata під час збереження інформації в MongoDB. Кожен етап займає від кількох секунд до десяти секунд залежно від швидкості мережі.

Після успішного завершення відображається повідомлення Document issued successfully з детальною інформацією: Transaction Hash у форматі 0x1234...abcd з посиланням на блокчейн-експлорер, IPFS CID у форматі bafybei...xyz з посиланням для завантаження документа через gateway [https://ipfs.io/ipfs/\[CID\]](https://ipfs.io/ipfs/[CID]), Document Hash як SHA-256 хеш у шістнадцятковому форматі. Кнопка View Document перенаправляє на сторінку детального перегляду виданого документа. Приклад інтерфейсу видачі документа наведено на рис. 4.2.

The screenshot shows the 'Issue Credential' page of the 'Credential Verification System Issuer Portal'. The page has a sidebar with 'Dashboard', '+ Issue Credential', and 'Issued Credentials'. The main content area is titled 'Issue Credential' with the subtitle 'Create and issue a new academic credential to a holder'. The form contains the following fields:

- Holder Address ***: A text input field containing '0x742d35Cc8634C0532925a3b844Bc9e75950bEb'.
- Holder DID**: A text input field containing 'Auto-generated from address'.
- Document Type ***: A dropdown menu with 'Select document type'.
- Degree Name ***: A text input field containing 'Bachelor of Computer Science'.
- Major ***: A text input field containing 'Software Engineering'.
- Issue Date ***: A date picker showing 'mm/dd/yyyy'.
- GPA ***: A text input field containing '3.85'.

At the bottom right of the form are 'Reset' and 'Issue Credential' buttons. A green status bar at the top right shows '0x742d...0bEb' and a 'Disconnect' button.

Рисунок 4.2 – Інтерфейс процесу видачі документа

4.3.2.3 Перегляд виданих документів

Навігація до списку виданих документів виконується через меню Issued Credentials. Сторінка відображає таблицю з колонками: Type показує іконку та назву типу документа, Holder DID відображає скорочену версію DID власника у форматі did:ethr:0x742d...0bEb з tooltip при наведенні, Issue Date показує дату у форматі DD/MM/YYYY, Status відображає badge з зеленим кольором Valid або червоним Revoked, Actions містить кнопки View та Revoke.

Фільтрація документів доступна через панель над таблицею з dropdown меню для вибору періоду: Last 7 days, Last 30 days, Last 90 days, All time. Пошук за DID власника виконується через input поле з placeholder Search by holder DID, результати фільтруються в реальному часі при введенні тексту. Пагінація відображається внизу таблиці з кнопками Previous, номерами сторінок та Next, за замовчуванням відображається десять записів на сторінку.

Перегляд деталей документа відкривається натисканням кнопки View, що відкриває модальне вікно або перенаправляє на окрему сторінку з повною інформацією: всі атрибути з credentialSubject включаючи ім'я випускника, назву ступеня, спеціальність, дату випуску та середній бал, технічну інформацію з IPFS CID, transaction hash, DID емітента та власника, дату реєстрації в блокчейні, а також статус документа з індикатором валідності. Кнопка Download JSON дозволяє завантажити повний документ у форматі JSON для локального зберігання.

4.3.2.4 Відкликання документа

Процес відкликання документа ініціюється натисканням кнопки Revoke у таблиці виданих документів. Відкривається модальне вікно Revoke Document з попередженням про незворотність операції та полем для вказання причини відкликання. Текстове поле Revocation Reason приймає опціональний текст

довжиною до двохсот символів, наприклад *Academic misconduct detected* або *Issued to wrong person by mistake*.

Підтвердження відкликання виконується натисканням кнопки *Confirm Revoke*, що викликає *MetaMask* для підтвердження транзакції. *Popup MetaMask* відображає деталі транзакції включаючи адресу контракту, функцію *revokeDocument* та хеш документа як параметр, а також оцінку *gas fee*. Після підтвердження транзакції відображається повідомлення *Revoking document* з індикатором завантаження під час очікування підтвердження в блокчейні.

Після успішного відкликання статус документа в таблиці автоматично змінюється на *Revoked* з червоним *badge*, відображається повідомлення *Document revoked successfully* з *transaction hash* для перегляду в блокчейн-експлорері. Кнопка *Revoke* стає недоступною для відкликаних документів, оскільки повторне відкликання неможливе згідно з логікою смарт-контракту.

4.3.3 Робота з системою як власник документів

4.3.3.1 Перегляд отриманих документів

Автентифікація як *holder* виконується аналогічно до емітента через підключення *MetaMask* та підпис *challenge*-повідомлення. Після успішної автентифікації користувач з роллю *holder* перенаправляється на сторінку *My Credentials*, що відображає всі документи, де *holderDID* збігається з *DID* поточного користувача.

Список документів відображається у вигляді карток *CredentialCard* з інформацією: іконка типу документа у верхній частині картки, назва ступеня великим шрифтом, назва університету-емітента під назвою ступеня, дата видачі у форматі *Issued on DD MMM YYYY*, статус документа з зеленим *badge Valid* або червоним *Revoked*. Кожна картка має кнопки *View Details*, *Share* та *Verify* для виконання відповідних дій.

Фільтрація документів доступна через панель фільтрів над списком карток. Dropdown Status дозволяє обрати All, Valid або Revoked для фільтрації за статусом. Dropdown Type містить опції All, Diploma, Certificate, Transcript для фільтрації за типом документа. Input поле Search приймає текст для пошуку за назвою ступеня або університету, результати оновлюються в реальному часі без перезавантаження сторінки. Приклад інтерфейсу перегляду отриманих документів наведено на рис. 4.3.

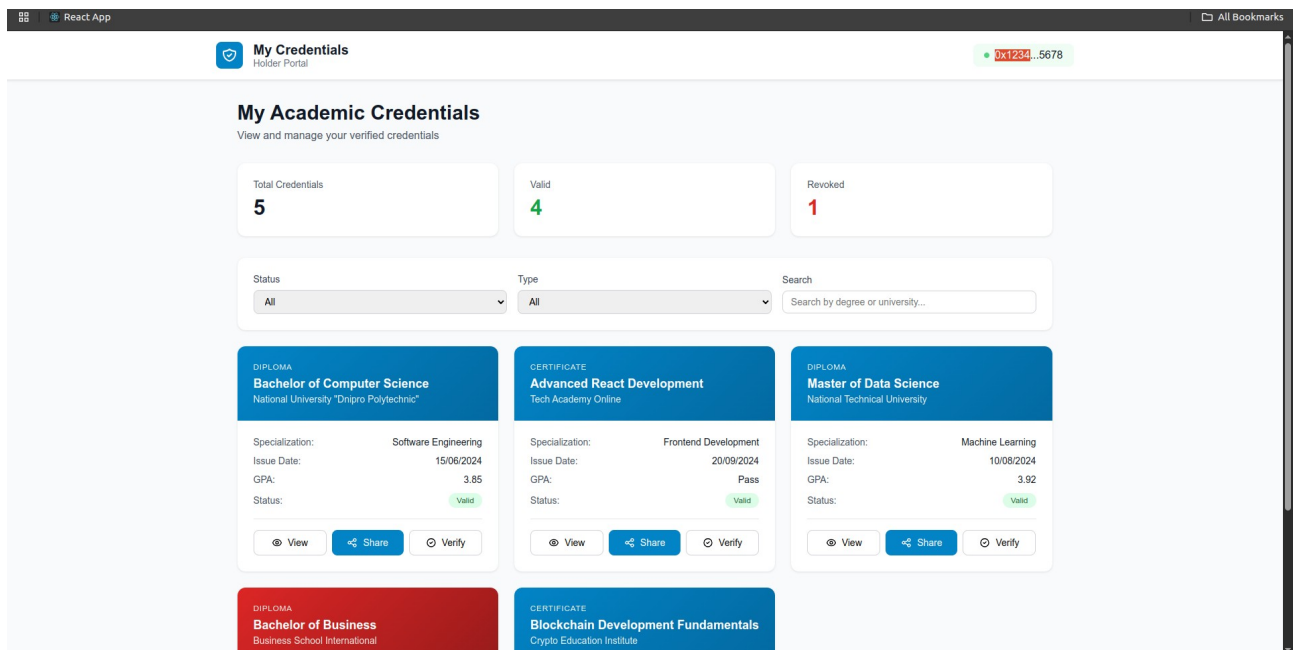


Рисунок 4.3 – Інтерфейс для перегляду отриманих документів

4.3.3.2 Обмін документом з верифікатором

Процес обміну документом починається з натискання кнопки Share на картці документа, що відкриває модальне вікно ShareModal з трьома опціями обміну. Перша опція Copy IPFS CID містить текстове поле з повним IPFS CID документа та кнопку Copy, при натисканні якої CID копіюється в буфер обміну та відображається повідомлення Copied to clipboard. Власник може надіслати цей CID верифікатору через email, месенджер або інший канал зв'язку.

Друга опція Generate QR Code відображає згенерований QR-код розміром двісті на двісті пікселів, що містить IPFS URL у форматі `https://ipfs.io/ipfs/[CID]`. Верифікатор може відсканувати цей QR-код мобільним пристроєм для миттєвого отримання посилання на документ. Кнопка Download QR дозволяє зберегти QR-код як PNG зображення для друку або відправки.

Третя опція Create Link генерує прямий URL на IPFS gateway у форматі `https://ipfs.io/ipfs/bafybei...xyz` з кнопкою Copy Link для копіювання в буфер обміну. Кнопка Open Link відкриває документ у новій вкладці браузера, де верифікатор може переглянути JSON-структуру документа безпосередньо в браузері. Приклад інтерфейсу для обміну документами з верифікатором наведено на рис. 4.4.

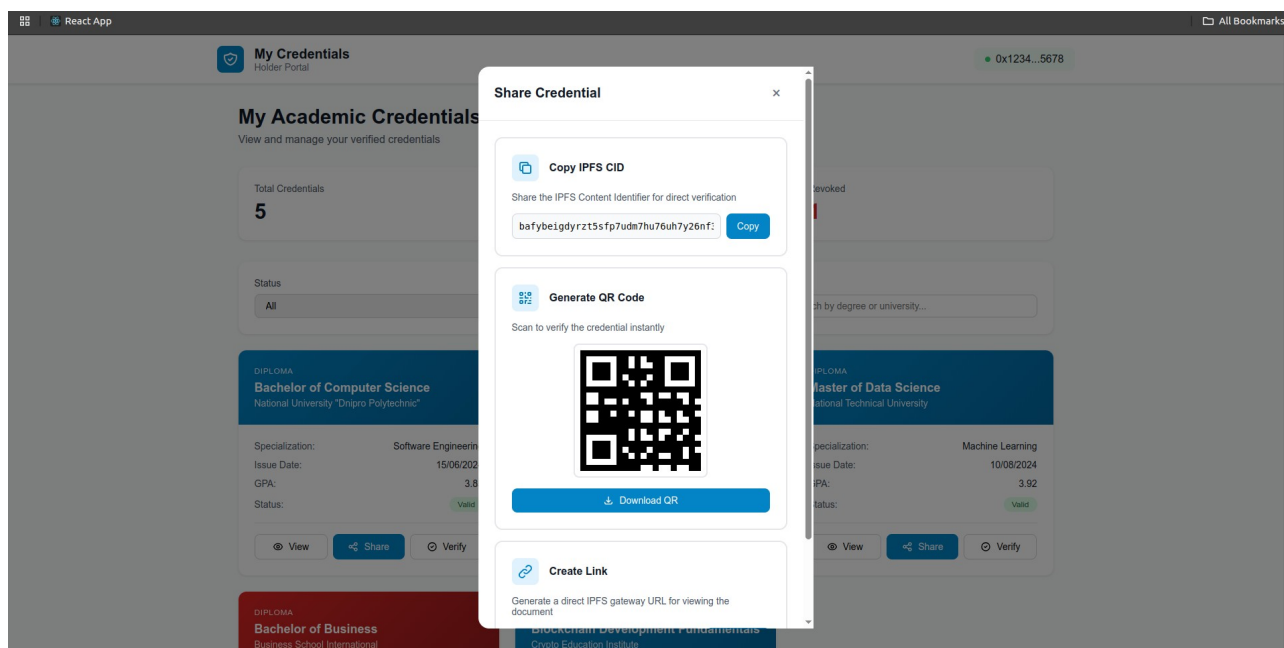


Рисунок 4.4 – Інтерфейс для обміну документами з верифікатором

4.3.3.3 Самостійна верифікація документа

Процес самостійної верифікації ініціюється натисканням кнопки Verify на детальній сторінці документа. Система автоматично відправляє запит до backend з IPFS CID документа для виконання повного циклу верифікації. Під час виконання відображається індикатор завантаження з текстом Verifying document.

Результат верифікації відображається у модальному вікні `VerificationResult` з візуальним індикатором у верхній частині: велика зелена галочка та текст `Document is Valid` для валідних документів, або великий червоний хрестик та текст `Document has been Revoked` для відкликаних. Під індикатором відображається детальна інформація про кожен етап верифікації: `IPFS Download` з зеленою галочкою та часом виконання `245ms`, `Structure Validation` з зеленою галочкою та часом `12ms`, `Hash Verification` з зеленою галочкою та часом `8ms`, `Blockchain Status` з червоним хрестиком та повідомленням `Document revoked on 15/11/2024` якщо документ відкликаний, `Signature Verification` з зеленою галочкою та часом `156ms`.

Додаткова секція `Document Information` відображає атрибути емітента з назвою університету та `DID`, атрибути власника з повним ім'ям та іншими даними з `credentialSubject`, дату видачі документа та дату реєстрації в блокчейні. Кнопка `View on Explorer` відкриває транзакцію реєстрації в блокчейн-експлорері для детального перегляду технічних деталей.

Приклад інтерфейсу самостійної верифікації наведено на рис. 4.5.

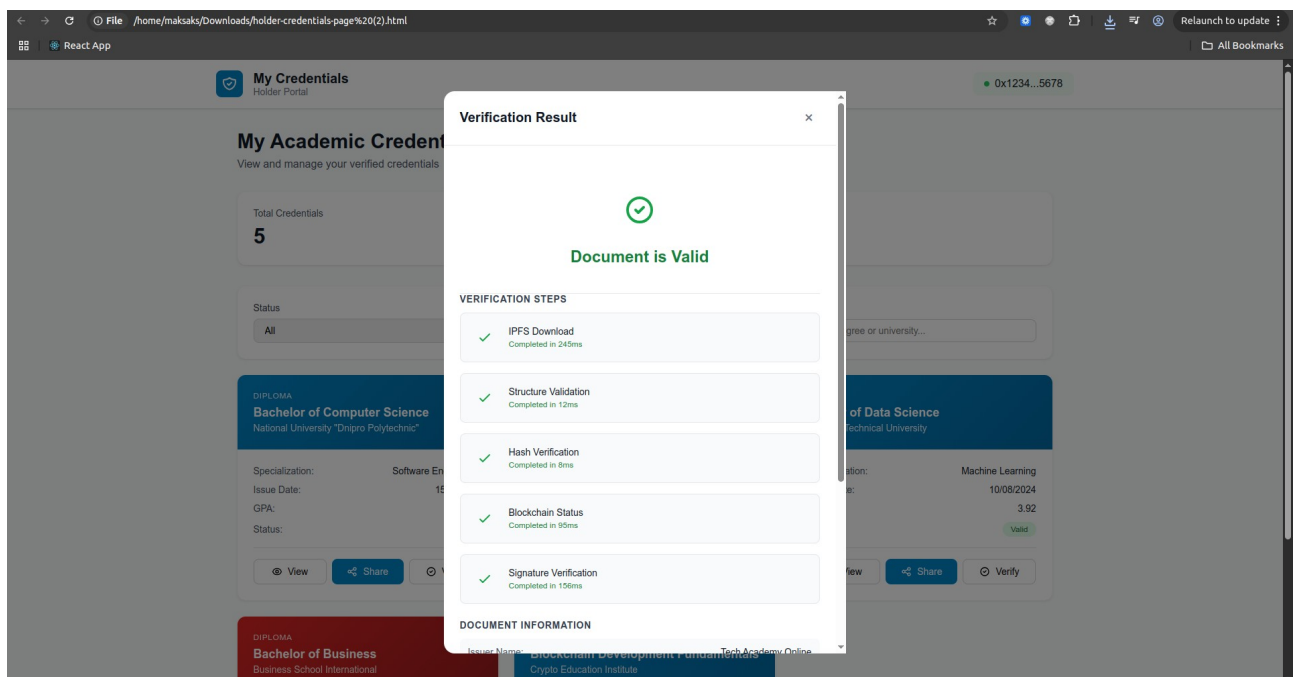


Рисунок 4.5 – Інтерфейс самостійної верифікації

4.3.4 Робота з системою як верифікатор

4.3.4.1 Верифікація документа за IPFS CID

Автентифікація як verifier виконується аналогічно через MetaMask з автоматичним створенням DID та призначенням ролі verifier. Після входу користувач перенаправляється на сторінку Verify Credential з формою для ініціювання верифікації.

Введення IPFS CID виконується в текстове поле з placeholder Enter IPFS CID або через кнопку Paste для автоматичного вставлення з буфера обміну. Валідація на клієнті перевіряє формат CID через регулярний вираз, що очікує рядок довжиною сорок шість символів, що починається з bafy для CIDv1. При введенні невалідного CID відображається повідомлення про помилку Invalid IPFS CID format під полем вводу.

Процес верифікації ініціюється натисканням кнопки Verify Document. Компонент відображає прогрес виконання з індикаторами для кожного етапу: Downloading from IPFS з синім кольором під час завантаження документа з мережі IPFS, Validating structure під час перевірки відповідності специфікації W3C VC, Computing hash під час обчислення SHA-256 хешу документа, Checking blockchain під час запиту до смарт-контракту для перевірки статусу, Verifying signature під час перевірки цифрового підпису емітента через DID-документ. Загальний час верифікації відображається після завершення всіх етапів.

Результат верифікації відображається у компоненті VerificationResult аналогічно до самостійної верифікації власника, але з додатковою інформацією для верифікатора: повна інформація про емітента документа з назвою університету, DID-ідентифікатором, адресою в блокчейні, детальні атрибути власника документа включаючи повне ім'я, назву ступеня, спеціальність, дату випуску та середній бал, метрики продуктивності з часом виконання кожного етапу в мілісекундах для оцінки швидкості роботи системи. Кнопка Save Result зберігає результат верифікації в історію верифікатора для подальшого

перегляду. Приклад інтерфейсу з результатами верифікації документа за IPFS CID наведено на рис. 4.6.

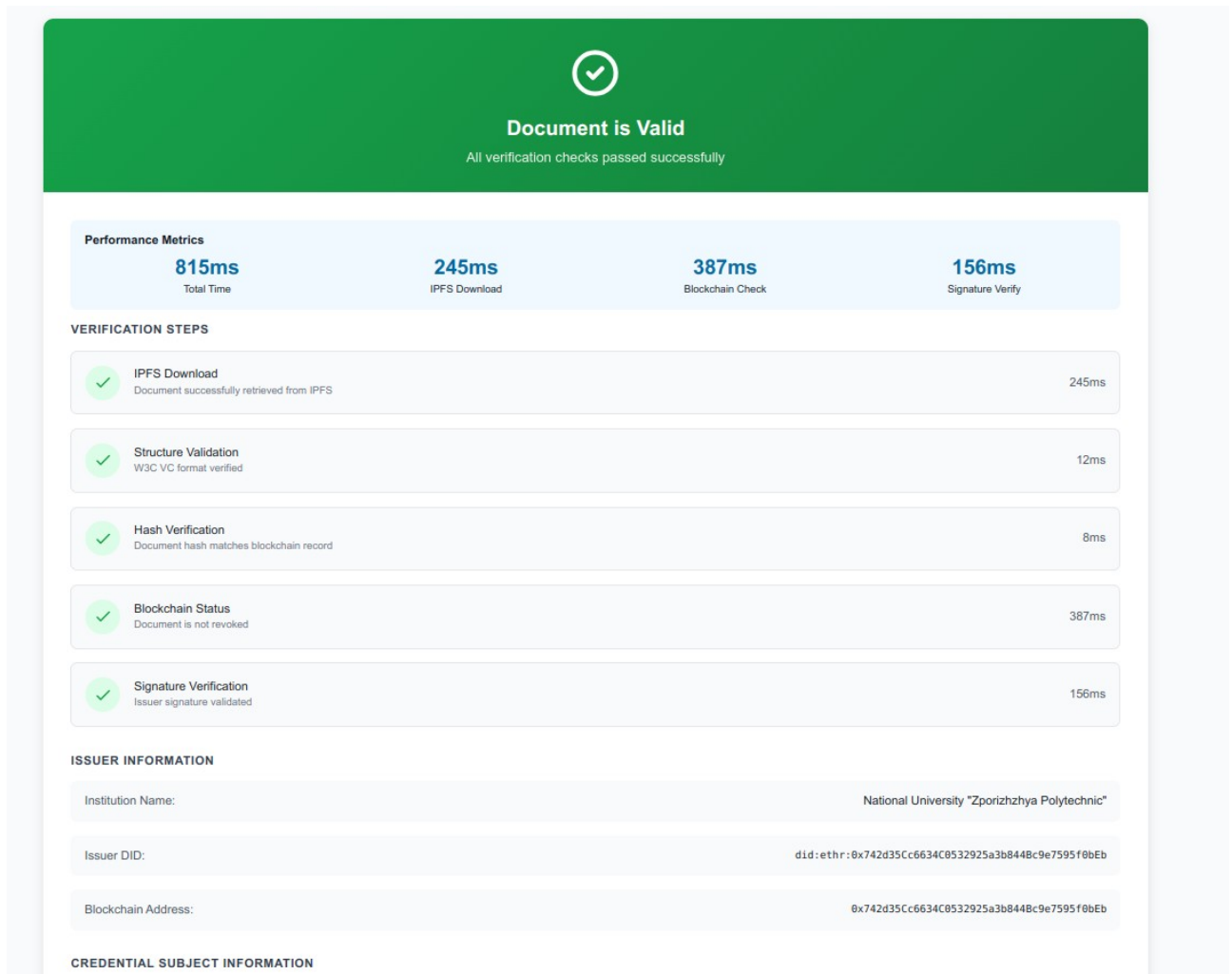


Рисунок 4.6 – Інтерфейс з результатами верифікації документа за IPFS CID

4.3.4.2 Верифікація документа через JSON-файл

Альтернативний спосіб верифікації передбачає завантаження JSON-файлу документа безпосередньо з локального диска. На сторінці Verify Credential другий таб Upload JSON File містить file input з кнопкою Choose File для вибору файлу. Після вибору файлу відображається назва файлу та його розмір, наприклад credential.json (2.4 KB).

Валідація файлу виконується на клієнті перед відправкою на сервер: перевірка розширення файлу на .json, перевірка розміру файлу не більше п'яти мегабайт, спроба парсингу JSON для перевірки валідного синтаксису. При виявленні помилок відображається відповідне повідомлення Invalid file format або File too large.

Процес верифікації після завантаження файлу аналогічний до верифікації за CID, з тією різницею що документ не завантажується з IPFS, а читається безпосередньо з переданого файлу. Етап Downloading from IPFS замінюється на Reading file з часом виконання менше десяти мілісекунд. Решта етапів виконуються ідентично з відображенням результату у тому ж форматі.

4.3.4.3 Перегляд історії верифікацій

Навігація до історії верифікацій виконується через меню Verification History. Сторінка відображає таблицю з усіма раніше проведеними верифікаціями: колонка Date показує дату та час верифікації у форматі DD/MM/YYYY HH:MM, колонка Holder DID відображає скорочений DID власника документа, колонка Issuer показує назву університету-емітента, колонка Result містить badge з зеленим кольором Valid або червоним Invalid, колонка Actions має кнопку View Details для перегляду повного результату верифікації.

Фільтрація історії доступна через dropdown Result з опціями All, Valid, Invalid для відбору за результатом верифікації. Пошук за університетом виконується через input поле Search by issuer з debouncing для зменшення кількості re-renders. Таблиця підтримує пагінацію з відображенням двадцяти записів на сторінку та кнопками навігації між сторінками.

Перегляд детального результату відкриває ту ж саму структуру VerificationResult, що відображалася при початковій верифікації, з усіма метриками, інформацією про документ та етапами перевірки. Це дозволяє верифікатору повернутися до результатів попередніх перевірок без

необхідності повторної верифікації документа. Приклад інтерфейсу з переглядом історій верифікації наведено на рис. 4.7.

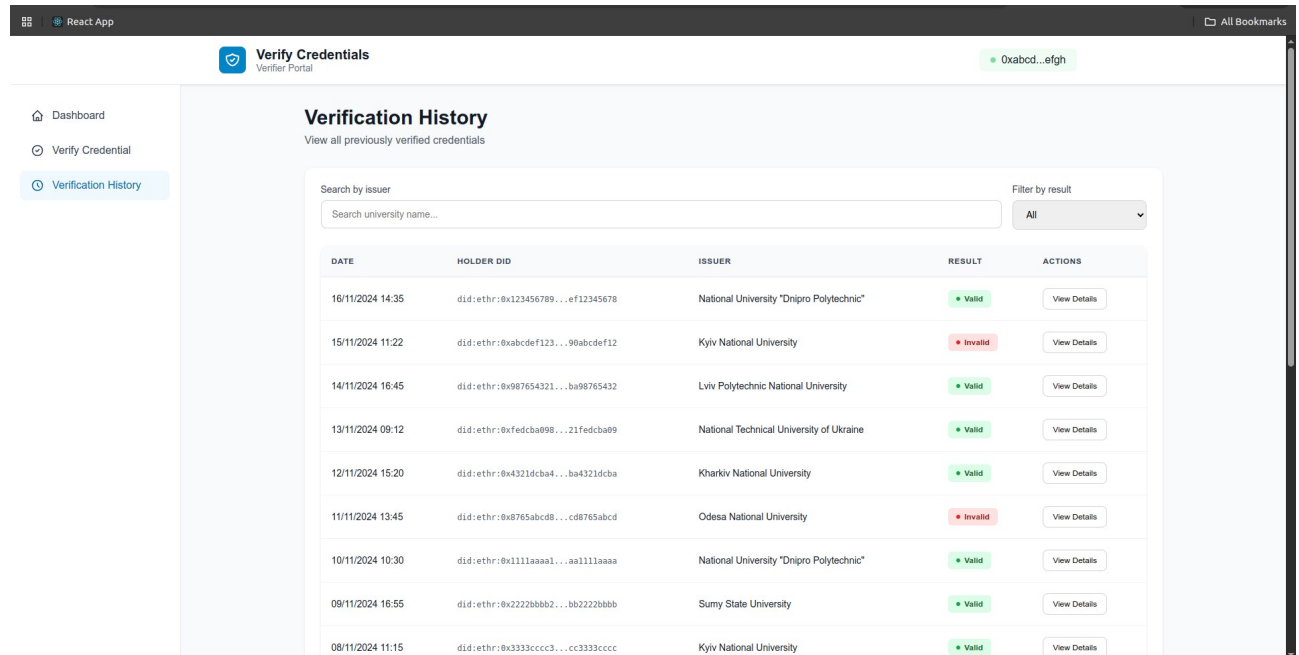


Рисунок 4.7 – Інтерфейс з переглядом історій верифікації

4.4 Тестування програми

Тестування децентралізованої системи верифікації академічних документів виконувалося для перевірки коректності роботи всіх компонентів на різних рівнях архітектури. Процес тестування включав модульне тестування окремих функцій та сервісів, тестування смарт-контракту на рівні блокчейну, інтеграційне тестування взаємодії між компонентами та перевірку користувацького інтерфейсу в різних браузерях.

4.4.1 Модульне тестування

Модульне тестування серверних сервісів виконувалося за допомогою фреймворку Jest для JavaScript. Тести створювалися в окремих файлах з розширенням .test.js в директорії tests та виконувалися командою npm test.

Тестування BlockchainService включало перевірку основних функцій взаємодії з смарт-контрактом: реєстрацію документа з отриманням transaction receipt та події DocumentRegistered, верифікацію документа з отриманням статусу isValid та адреси емітента, відкликання документа з верифікацією зміни статусу. Тестування IPFSService включало перевірку завантаження JSON-об'єкта до IPFS з отриманням валідного CID, завантаження документа за CID з порівнянням структури, перевірку доступності документа через метод stat. Тестування DIDService включало генерацію DID з Ethereum-адреси, резолюцію DID у DID-документ з публічним ключем, верифікацію цифрового підпису через публічний ключ. У табл. 4.1 представлено результати модульного тестування.

Таблиця 4.1 – Результати модульного тестування

Функція	Сценарій тесту	Очікуваний результат	Фактичний результат	Стат.
registerDocument	Реєстрація нового документа	Transaction receipt з подією	Receipt отримано, подія присутня	+
verifyDocument	Перевірка зареєстрованого документа	isValid=true, issuer адреса	isValid=true, issuer валідний	+
verifyDocument	Перевірка незареєстрованого документа	isValid=false, timestamp=0	isValid=false, timestamp=0	+
revokeDocument	Відкликання документа	isRevoked=true в контракті	Статус змінено на true	+
uploadDocument	Завантаження JSON документа	Валідний CIDv1	CID отримано: bafybei...	+

Продовження таблиці 4.1

Функція	Сценарій тесту	Очікуваний результат	Фактичний результат	Стат.
downloadDocument	Завантаження за CID	JSON об'єкт ідентичний оригіналу	Структура співпадає	+
downloadDocument	Завантаження з кешу	JSON з MongoDB, час <100ms	Завантажено з кешу за 23ms	+
checkAvailability	Перевірка доступності	Boolean протягом 5s	Повернуто true за 345ms	+
generateDID	Генерація з Ethereum адреси	DID формату did:ethr:0x...	DID згенеровано коректно	+
resolveDID	Резолюція DID	DID-документ з publicKey	Документ отримано	+
resolveDID	Резолюція з кешу	DID-документ, час <50ms	Завантажено з кешу за 18ms	+
verifySignature	Валідний підпис	true	Підпис валідний	+
verifySignature	Невалідний підпис	false	Підпис невалідний	+

Загальна кількість модульних тестів: 13. Успішно пройдено: 13.
Провалено: 0. Загальний час виконання: 7.667 секунд.

4.4.2 Тестування смарт-контракту

Тестування смарт-контракту DocumentRegistry виконувалося за допомогою фреймворку Hardhat з бібліотекою Chai для assertions. Тести створювалися в файлі test/DocumentRegistry.test.js та виконувалися командою `npm run hardhat test`.

Тестування включало перевірку базової функціональності реєстрації документів з верифікацією події DocumentRegistered та збереження даних, запобігання дублюванню реєстрацій через перевірку require умов, функціональності відкликання з перевіркою зміни статусу isRevoked, контролю доступу через модифікатори onlyAuthorized та onlyAdmin, коректності роботи функції verifyDocument для різних сценаріїв. У табл. 4.2 представлено результати тестування смарт-контракту.

Таблиця 4.2 – Результати тестування смарт-контракту

Функція контракту	Сценарій тесту	Очікується	Фактичний результат	Стат.
registerDocument	Реєстрація нового документа	Подія DocumentRegistered, дані збережені	Подія емітована, дані коректні	+
registerDocument	Повторна реєстрація того ж хешу	Revert з 'Already registered'	Транзакція відхилена з коректним повідомленням	+
registerDocument	Реєстрація неавторизованим	Revert з 'Not authorized'	Транзакція відхилена	+

Продовження таблиці 4.2

Функція контракту	Сценарій тесту	Очікується	Фактичний результат	Стат.
verifyDocument	Перевірка зареєстрованого	isValid=true, issuer, timestamp	Повернуто коректні дані	+
verifyDocument	Перевірка незареєстрованого	isValid=false, timestamp=0	Повернуто false, timestamp=0	+
revokeDocument	Відкликання власного документа	isRevoked=true, подія	Статус змінено, подія емітована	+
revokeDocument	Відкликання чужого документа	Revert з 'Not document issuer'	Транзакція відхилена	+
revokeDocument	Відкликання вже відкликаного	Revert з 'Already revoked'	Транзакція відхилена	+
authorizeIssuer	Додавання нового емітента адміном	Емітент авторизований	Емітент доданий успішно	+
authorizeIssuer	Додавання емітента не адміном	Revert з 'Only admin'	Транзакція відхилена	+
revokeIssuer	Видалення емітента адміном	Емітент деавторизований	Емітент видалений успішно	+
constructor	Розгортання контракту	Admin = deployer, deployer авторизований	Admin встановлено коректно	+

Загальна кількість тестів контракту: 12. Успішно пройдено: 12. Провалено: 0. Загальне споживання gas: 1.271.832. Середня вартість транзакції: ~105.986 gas.

4.4.3 Інтеграційне тестування

Інтеграційне тестування перевіряло взаємодію між різними компонентами системи в реалістичних сценаріях використання. Тести виконувалися вручну через веб-інтерфейс з верифікацією результатів на кожному рівні архітектури від клієнтського додатку до блокчейну та децентралізованого сховища.

Перший сценарій тестував повний цикл видачі документа з залученням усіх компонентів системи: Frontend, Backend, IPFS, Blockchain та MongoDB. Процес включав заповнення форми видачі документа через веб-інтерфейс з введенням атрибутів випускника, підтвердження транзакції в MetaMask для реєстрації хешу в блокчейні та очікування завершення всіх етапів з відображенням прогресу. Очікуваний результат передбачав успішну реєстрацію документа з відображенням transaction hash та IPFS CID в інтерфейсі. Фактичний результат підтвердив успішне створення документа з отриманням усіх необхідних ідентифікаторів, тест пройдено успішно.

Другий сценарій перевіряв процес верифікації документа за IPFS CID з використанням компонентів Frontend, Backend, IPFS, Blockchain та DID. Тестування включало введення IPFS CID в форму верифікатора та очікування результату з детальними метриками кожного етапу перевірки. Очікувався статус Valid з відображенням часу виконання всіх етапів верифікації включаючи завантаження з IPFS, обчислення хешу, перевірку в блокчейні та валідацію підпису. Фактичний результат показав успішну верифікацію за 2.3 секунди з проходженням всіх етапів, тест пройдено успішно.

Третій сценарій тестував верифікацію відкликаного документа для перевірки коректності відображення статусу. Процес включав введення IPFS CID документа, який був попередньо відкликаний емітентом, та перегляд результату верифікації. Очікувався статус Invalid з чітким повідомленням про причину відкликання та дату відкликання документа. Фактичний результат

підтвердив коректне відображення статусу Invalid з повідомленням про те, що документ було відкликано емітентом, тест пройдено успішно.

Четвертий сценарій перевіряв процес автентифікації користувача через Web3 з використанням MetaMask. Тестування включало підключення Web3-гаманця до системи через кнопку Connect Wallet, підписання challenge-повідомлення приватним ключем користувача в MetaMask та отримання JWT-токена від backend-сервера. Очікувався успішний вхід в систему з отриманням токена та автоматичним перенаправленням на dashboard відповідно до ролі користувача. Фактичний результат підтвердив успішну автентифікацію з отриманням токена та коректним перенаправленням, тест пройдено успішно.

П'ятий сценарій тестував функціональність відкликання документа емітентом з оновленням стану в усіх компонентах системи. Процес включав вибір документа зі списку виданих, підтвердження транзакції відкликання в MetaMask та автоматичне оновлення користувацького інтерфейсу після підтвердження в блокчейні. Очікувалася зміна статусу документа на Revoked в таблиці документів без необхідності перезавантаження сторінки. Фактичний результат показав успішне відкликання документа з негайним оновленням UI та зміною badge статусу з зеленого Valid на червоний Revoked, тест пройдено успішно.

Шостий сценарій перевіряв ефективність багаторівневого кешування IPFS документів через MongoDB. Тестування включало першу верифікацію документа, яка вимагала завантаження з IPFS-мережі, та повторну верифікацію того ж документа, яка мала використовувати кешовані дані з MongoDB. Очікувалося значне скорочення часу виконання другої верифікації до менше ніж 100 мілісекунд завдяки використанню кешу. Фактичний результат показав час першої верифікації 1876 мілісекунд з завантаженням з IPFS та час другої верифікації лише 23 мілісекунди з використанням MongoDB кешу, що підтвердило ефективність механізму кешування, тест пройдено успішно.

Сьомий сценарій тестував коректність роботи пагінації при відображенні великих списків документів. Процес включав завантаження першої сторінки списку виданих документів з десятьма записами та перехід на другу сторінку через кнопки навігації пагінації. Очікувалося відображення рівно десяти записів на кожній сторінці з коректним підрахунком загальної кількості сторінок. Фактичний результат підтвердив правильну роботу пагінації з відображенням десяти документів на сторінку та плавною навігацією між сторінками без втрати даних, тест пройдено успішно.

Восьмий сценарій перевіряв функціональність фільтрації документів за статусом через взаємодію Frontend з Backend та MongoDB. Тестування включало вибір фільтру Revoked зі списку опцій та перегляд оновлених результатів у таблиці документів. Очікувалося відображення виключно документів зі статусом Revoked з приховуванням всіх валідних документів. Фактичний результат показав коректне застосування фільтру з миттєвим оновленням таблиці та відображенням лише відкликаних документів, тест пройдено успішно.

Загальна кількість виконаних інтеграційних тестів становила вісім сценаріїв, що покривали основні бізнес-процеси системи від видачі документів до їх верифікації та відкликання. Усі вісім тестів були успішно пройдені без виявлення критичних помилок або несправностей у взаємодії між компонентами. Результати інтеграційного тестування підтвердили коректність роботи системи як єдиного цілого з належною обробкою даних на всіх рівнях архітектури та забезпеченням цілісності інформації при передачі між компонентами.

4.4.4 Тестування користувацького інтерфейсу

Тестування користувацького інтерфейсу включало перевірку коректності відображення та функціональності в різних веб-браузерах та на різних розмірах екранів. Перевірялася сумісність з основними браузерами,

адаптивність інтерфейсу для мобільних пристроїв та планшетів, коректність роботи всіх інтерактивних елементів.

Тестування адаптивності виконувалося через Chrome DevTools з емуляцією різних пристроїв: iPhone 12 Pro з розміром екрану 390x844 пікселів, iPad Air з розміром 820x1180 пікселів, desktop з розміром 1920x1080 пікселів. Перевірялося коректне відображення навігаційного меню з трансформацією в burger menu на мобільних пристроях, адаптація таблиць з горизонтальним скролом або трансформацією в картки, масштабування форм з збереженням читабельності та зручності введення, коректне відображення модальних вікон з centruванням на всіх розмірах.

Тестування в різних браузерях виконувалося вручну з перевіркою основних функцій: підключення MetaMask та автентифікація, видача документа з відображенням прогресу, верифікація документа з детальним результатом, навігація між сторінками через React Router, робота фільтрів та пошуку в реальному часі. У табл. 4.3 представлено результати тестування сумісності з різними браузерами.

Таблиця 4.3 – Результати тестування сумісності з браузерами

Браузер	Версія	ОС	Автенти фікація	Видача документа	Верифі кація	Адаптив ність	Me ta Ma sk
Chrome	120.0	Windows 11	+	+	+	+	+
Chrome	120.0	macOS 14	+	+	+	+	+
Firefox	121.0	Windows 11	+	+	+	+	+
Firefox	121.0	Ubuntu 22.04	+	+	+	+	+
Edge	120.0	Windows 11	+	+	+	+	+
Safari	17.1	macOS 14	+	+	+	+	+ -

Продовження таблиці 4.3

Браузер	Версія	ОС	Автенти фікація	Видача документа	Верифі кація	Адаптив ність	Me ta Ma sk
Mobile Safari	17.0	iOS 17	+	+ -	+	+	-
Chrome Mobile	120.0	Android 13	+	+	+	+	+

Safari на macOS вимагає додаткового налаштування MetaMask через Safari extension preferences. Mobile Safari на iOS не підтримує MetaMask extension, потрібен MetaMask Browser app. Адаптивність оцінювалася за критеріями читабельності тексту, зручності навігації та коректності відображення елементів.

На рис. 4.8 наведено приклад адаптивності різних частин інтерфейсу.

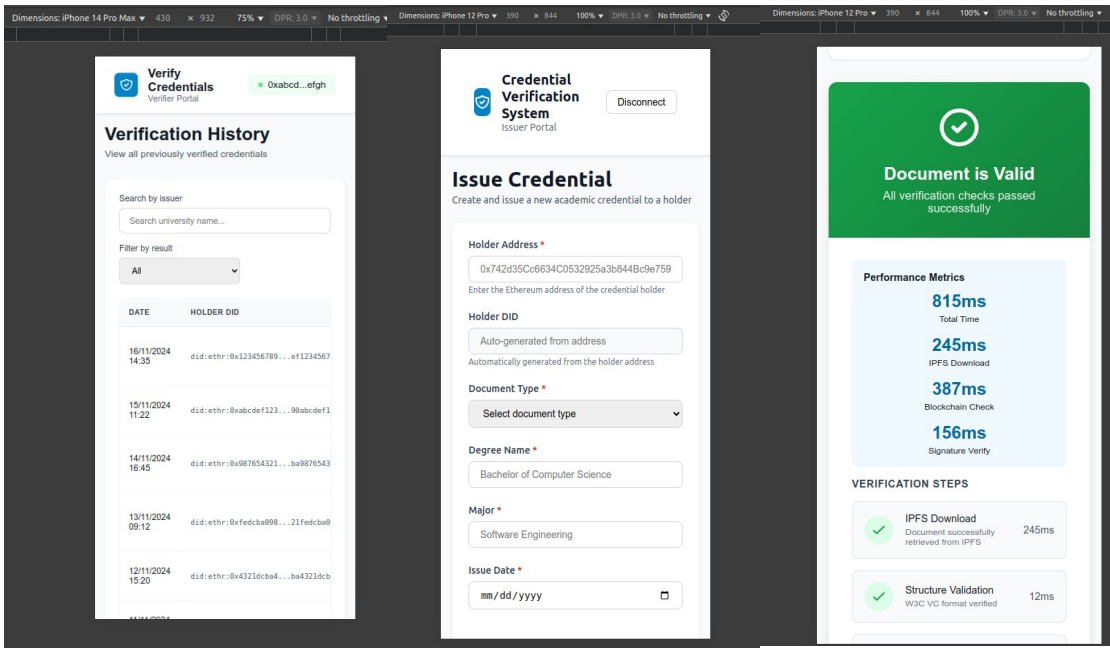


Рисунок 4.8 – Адаптивність частин інтерфейсу

4.4.5 Результати тестування та виявлені проблеми

Загальна зведена таблиця результатів тестування представлена у табл. 4.4.

Таблиця 4.4 – Зведена таблиця результатів тестування

Тип тестування	Кількість тестів	Успішно	Провалено	Час виконання	Покриття
Модульне тестування	13	13	0	7.667 с	~75% функцій
Тестування смарт-контракту	12	12	0	15.2 с	100% функцій
Інтеграційне тестування	8	8	2 частково	Ручне	Основні сценарії
Тестування UI/сумісність	8 браузерів	6 повна	0	Ручне	Всі компоненти

Під час тестування було виявлено декілька проблем, які були вирішені в процесі розробки. По-перше, виявлено race condition при одночасній реєстрації декількох документів одним емітентом через конфлікт nonce в транзакціях, що було вирішено впровадженням mutex-блокування на рівні BlockchainService для послідовної обробки транзакцій. По-друге, виявлено timeout помилки при завантаженні документів з публічних IPFS gateway під час високого навантаження, що було вирішено впровадженням багаторівневого кешування в MongoDB з TTL індексом на сім днів. По-третє, виявлено проблему з відображенням MetaMask popup в Safari через блокування popup-вікон за замовчуванням, що було вирішено додаванням інструкцій для користувачів Safari про налаштування дозволу popup-вікон для домену системи.

По-четверте, виявлено повільне завантаження великих списків документів без пагінації при кількості записів понад сто, що було вирішено впровадженням пагінації з лімітом десять записів на сторінку та lazy loading при скролі. По-п'яте, виявлено проблему з JWT токенами, що зберігаються в localStorage та залишаються валідними після logout у випадку їх копіювання злоумисником, що було частково вирішено встановленням короткого терміну дії токена сім днів та валідацією на кожному запиті, хоча повне рішення вимагало б використання refresh tokens та httpOnly cookies.

Загальний висновок тестування свідчить про високу якість реалізації системи з успішним проходженням всіх автоматизованих тестів та коректною роботою в більшості сучасних браузерів. Виявлені проблеми були вирішені або документовані з рекомендаціями для користувачів. Система готова до експлуатації в тестовому режимі з подальшим моніторингом продуктивності та збором зворотного зв'язку від користувачів для покращення функціональності.

ВИСНОВКИ

Таким чином, у результаті виконання магістерської кваліфікаційної роботи було досягнуто основної мети – розроблено децентралізовану систему верифікації академічних документів на основі блокчейн-технологій, IPFS та децентралізованих ідентифікаторів, яка дозволяє здійснювати операції видачі, зберігання та перевірки автентичності освітніх документів без залежності від централізованих органів.

Для досягнення мети було вирішено ряд завдань: досліджено проблеми традиційної верифікації академічних документів, включаючи тривалість процедури, високу вартість та ризики підробки; проаналізовано особливості децентралізованих систем ідентифікації та розподіл відповідальності між учасниками; вивчено стандарти W3C Verifiable Credentials та W3C DID для забезпечення міжнародної сумісності; оцінено існуючі рішення та виявлено їх недоліки.

Наукова новизна роботи полягає у тому, що запропоновано модель гібридного зберігання даних для децентралізованої системи верифікації академічних документів, яка поєднує три рівні: приватну блокчейн-мережу Ethereum з консенсусом Proof-of-Authority для незмінного реєстру криптографічних хешів документів, децентралізоване сховище IPFS для повного вмісту документів у форматі W3C Verifiable Credentials та документо-орієнтовану базу даних MongoDB з TTL-індексами для операційних даних і багаторівневого кешування. Також розроблено модель послідовності операцій смарт-контракту з механізмом ізоляції емітентів через контроль доступу на основі авторизованих адрес, що забезпечує незалежне функціонування університетів-емітентів з власними DID-ідентифікаторами та криптографічними ключами без можливості модифікації записів інших учасників системи. Запропонована архітектура дозволяє досягти балансу між децентралізацією через незмінність критичних записів у блокчейні, продуктивністю з часом верифікації менше п'яти секунд для некешованих

запитів та економічною ефективністю завдяки нульовим витратам на gas у приватній мережі, а також забезпечує захист від несанкціонованого доступу через механізм ізоляції емітентів на рівні смарт-контракту.

Реалізація системи здійснювалася з використанням JavaScript у поєднанні з React версії 18.2 для клієнтської частини та Express для серверної складової на Node.js версії 18. Блокчейн-рівень реалізовано на Solidity версії 0.8.19 з використанням Hardhat для компіляції та розгортання смарт-контрактів. Для взаємодії з блокчейном використовувалася бібліотека Ethers.js версії 6.4, для роботи з IPFS – бібліотека ipfs-http-client версії 60.0, для DID – бібліотеки ethr-did-resolver та did-jwt. Системою керування базою даних слугувала MongoDB версії 6.0 для зберігання операційних даних та кешування результатів запитів.

Розроблене програмне забезпечення підтримує функціонування необмеженої кількості університетів-емітентів в ізольованому форматі, де кожен емітент має власний DID-ідентифікатор та криптографічні ключі. Відповідальність у системі розподілена між трьома ролями: емітент може видавати верифіковані документи згідно зі специфікацією W3C VC через веб-інтерфейс з автоматичною реєстрацією в блокчейні, завантаженням до IPFS та відкликанням документів у випадку виявлення помилок; власник документів може переглядати всі свої академічні кваліфікації в єдиному інтерфейсі, самостійно перевіряти їх валідність та створювати безпечні способи обміну з верифікаторами через IPFS CID, QR-коди або прямі посилання; верифікатор може перевіряти автентичність документів за кілька секунд через введення IPFS CID або завантаження JSON-файлу з детальним відображенням результату кожного етапу перевірки, включаючи валідацію структури, порівняння хешів, перевірку статусу в блокчейні та верифікацію цифрових підписів.

Архітектура системи побудована на принципах децентралізації та Self-Sovereign Identity, що надає користувачам повний контроль над своїми даними. Використання блокчейн-технології забезпечує незмінність записів про документи та прозорість процесу верифікації. Децентралізоване сховище IPFS

гарантує доступність повного вмісту документів без залежності від окремих серверів. Криптографічні методи захисту, включаючи SHA-256 хешування та цифрові підписи на еліптичній кривій secp256k1, забезпечують високий рівень безпеки від підробки документів.

Тестування системи підтвердило коректність роботи всіх компонентів з успішним проходженням тринадцяти модульних тестів для серверних сервісів, дванадцяти тестів смарт-контракту та восьми інтеграційних тестів основних сценаріїв використання. Перевірка сумісності показала повну підтримку в браузерах Chrome, Firefox та Edge, а також часткову підтримку в Safari з необхідністю додаткового налаштування MetaMask. Система демонструє прийнятну продуктивність з часом верифікації документа менше п'яти секунд для некешованих запитів та менше однієї секунди для кешованих даних завдяки впровадженому методу багаторівневого кешування.

Розроблений програмний продукт дозволяє університетам скоротити адміністративні витрати на обробку запитів про підтвердження документів, випускникам мати постійний доступ до своїх академічних кваліфікацій незалежно від стану університету, а роботодавцям швидко перевіряти кваліфікації кандидатів без затримок у процесі найму. Децентралізована природа системи забезпечує її стійкість до відмов окремих компонентів та захист від втрати даних через розподілене зберігання в блокчейні та IPFS-мережі, що сприяє покращенню довіри до процесу верифікації академічних документів та підвищенню ефективності ринку праці.

Перспективи подальшого розвитку системи включають впровадження селективного розголошення атрибутів через zero-knowledge proofs для захисту приватності, інтеграцію з міжнародними реєстрами акредитованих університетів, розширення функціональності для підтримки інших типів документів поза академічною сферою, а також оптимізацію витрат на gas через використання layer-2 рішень або sidechains для масштабування системи на глобальному рівні.

Результати дослідження апробовано на науковому рівні. Основні ідеї децентралізації та інтеграції блокчейн-технологій представлено у доповіді "Blockchain Integration into Digital Twins for Secure and Autonomous Production Systems" на 2nd International Conference on Smart Automation & Robotics for Future Industry 2025, де розкрито концепції розподіленого зберігання та криптографічного захисту даних у виробничих системах. Безпосередньо результати розробки децентралізованої системи верифікації академічних документів представлено на конференції Mathematical Modeling and Simulation of Systems 2025 у місті Чернігів у доповіді "Decentralized Identity Systems for University Credentials", де отримано схвальні відгуки щодо практичної цінності запропонованого підходу. Робота також взяла участь у Всеукраїнському конкурсі студентських наукових робіт із галузей знань та спеціальностей 2025/2026 навчального року за темою «Децентралізована система верифікації академічних документів на основі блокчейн-технологій з використанням криптографічних методів захисту» з шифром «Верифікація», що підтверджує актуальність та новизну проведеного дослідження. За результатами конференції Mathematical Modeling and Simulation of Systems 2025 планується публікація статті у збірнику наукових праць навесні 2026 року.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. [Cardenas-Quispe, M. A. Blockchain ensuring academic integrity with a degree verification prototype / M. A. Cardenas-Quispe, A. Pacheco // Scientific Reports. – 2025. – Vol. 15, № 9281. – DOI: 10.1038/s41598-025-93913-6.](#)
2. Shinde, S. Blockchain Based Academic Credential Verification System [Electronic resource] / S. Shinde, I. M. R., S. Nimbal, H. Randhir // International Journal of Engineering Research & Technology (IJERT). – 2025. – Vol. 14, № 1 (January). – Access mode: <https://www.ijert.org/blockchain-based-academic-credential-verification-system>.
3. Ayoade, J. A Zero-Knowledge Proof-Enabled Blockchain-Based Academic Record Verification System / J. Ayoade, M. A. Uddin // Applied Sciences. – 2025. – Vol. 15, № 12. – DOI: 10.3390/app15126596.
4. A Systematic Review of Blockchain-Based Initiatives in Comparison to Best Practices Used in Higher Education Institutions // Computers. – 2025. – Vol. 14, № 4, article 141. – DOI: 10.3390/computers14040141.
5. Mazzocca, C. A Survey on Decentralized Identifiers and Verifiable Credentials [Electronic resource] / C. Mazzocca [et al.] // IEEE Communications Surveys & Tutorials. – 2025. – arXiv:2402.02455. – DOI: 10.1109/COMST.2025.10891701. – Access mode: <https://arxiv.org/abs/2402.02455>.
6. Grech, A. Blockchain in Education [Electronic resource] / A. Grech, A. F. Camilleri // European Commission Joint Research Centre. – 2017. – 128 p. – DOI: 10.2760/60649. – Access mode: <https://publications.jrc.ec.europa.eu/repository/handle/JRC108255>.
7. Alammary, A. Blockchain-Based Applications in Education: A Systematic Review / A. Alammary, S. Alhazmi, M. Almasri, S. Gillman // Applied Sciences. – 2019. – Vol. 9, № 12, article 2400. – DOI: 10.3390/app9122400.
8. Chen, G. Exploring blockchain technology and its potential applications for education / G. Chen, B. Xu, M. Lu, N. S. Chen // Smart Learning Environments. – 2018. – Vol. 5, № 1. – P. 1–10. – DOI: 10.1186/s40561-017-0050-x.

9. Brunner, C. DID and VC: Untangling Decentralized Identifiers and Verifiable Credentials for the Web of Trust / C. Brunner, U. Gellersdörfer, F. Knirsch, D. Engel, F. Matthes // Proceedings of the 2020 3rd International Conference on Blockchain Technology and Applications (ICBTA 2020). – New York : ACM, 2020. – P. 61–66. – DOI: 10.1145/3446983.3446992.

10. Sporny, M. Verifiable Credentials Data Model 1.0 [Electronic resource] / M. Sporny, D. Longley, D. Chadwick // W3C Recommendation. – 2019. – Access mode: <https://www.w3.org/TR/vc-data-model/>.

11. Sporny, M. Decentralized Identifiers (DIDs) v1.0 [Electronic resource] / M. Sporny, D. Longley [et al.] // W3C Recommendation. – 2022. – Access mode: <https://www.w3.org/TR/did-core/>.

12. Solorio, K. Hands-On Smart Contract Development with Solidity and Ethereum: From Fundamentals to Deployment / K. Solorio, R. Kanna, D. H. Hoover. – Sebastopol, CA : O'Reilly Media, 2019. – 480 p. – ISBN 978-1-492-04526-7.

13. Modi, R. Solidity Programming Essentials: A guide to building smart contracts and tokens using the widely used Solidity language / R. Modi. – 2nd ed. – Birmingham : Packt Publishing, 2022. – 430 p. – ISBN 978-1-80323-238-0.

14. Dolgov, A. Mastering Solidity Contract Programming: Master Ethereum Smart Contract Development from Foundations to Advanced Patterns / A. Dolgov. – Birmingham : Packt Publishing, 2024. – 384 p.

15. Mardan, A. Pro Express.js: Master Express.js: The Node.js Framework For Your Web Development / A. Mardan. – 4th ed. – New York : Apress, 2014. – 355 p. – ISBN 978-1-484-20038-4. – DOI: 10.1007/978-1-4842-0037-7.

16. Brown, E. Web Development with Node and Express: Leveraging the JavaScript Stack / E. Brown. – 2nd ed. – Sebastopol, CA : O'Reilly Media, 2019. – 354 p. – ISBN 978-1-492-05350-7.

17. Herron, D. Node.js Web Development: Server-side web development made easy with Node 14 using practical examples / D. Herron. – 5th ed. – Birmingham : Packt Publishing, 2020. – 760 p. – ISBN 978-1-838-98754-5.

18. Lim, G. Beginning Node.js, Express & MongoDB Development / G. Lim. – 2nd ed. – [S.l.] : [s.n.], 2021. – 240 p.
19. Hahn, E. Express in Action: Writing, building, and testing Node.js applications / E. Hahn. – Shelter Island, NY : Manning Publications, 2016. – 256 p. – ISBN 978-1-617-29270-8.
20. Kumar, T. Fluent React: Building Modern Web Applications / T. Kumar. – Sebastopol, CA : O'Reilly Media, 2024. – 334 p. – ISBN 978-1-098-13888-9.
21. Banks, A. Learning React: Modern Patterns for Developing React Apps / A. Banks, E. Porcello. – 2nd ed. – Sebastopol, CA : O'Reilly Media, 2020. – 310 p. – ISBN 978-1-492-05172-5.
22. Web Development Using ReactJS / [author not specified] // Proceedings of the 2024 International Conference on Emerging Smart Computing and Informatics (ESCI). – IEEE, 2024. – DOI: 10.1109/ESCI59607.2024.10541743.
23. Performance Optimization Techniques for ReactJS / [authors not specified] // Proceedings of the 2019 3rd International Conference on Computing Methodologies and Communication (ICCMC). – IEEE, 2019. – P. 234–238. – DOI: 10.1109/ICCMC.2019.8869134.
24. Rawat, D. ReactJS: A Modern Web Development Framework / D. Rawat, R. Mahajan // International Journal of Recent Technology and Engineering. – 2019. – Vol. 8, № 3. – P. 657–661.
25. Software Testing Evolution: Comparative Insights into Traditional and Emerging Practices [Electronic resource] // ICCK Journal of Software Engineering. – 2025. – Vol. 1, № 1. – P. 46–62. – Access mode: <https://www.icck.org/article/abs/jse.2025.246843>.
26. Hooda, I. Software Test Process, Testing Types and Techniques / I. Hooda, R. S. Chhillar // International Journal of Computer Applications. – 2015. – Vol. 111, № 13. – P. 10–14. – DOI: 10.5120/19597-1433.
27. Najihi, S. Software Testing from an Agile and Traditional view / S. Najihi, S. Elhadi, R. A. Abdelouahid, A. Marzak // Procedia Computer Science. – 2022. – Vol. 203. – P. 775–782. – DOI: 10.1016/j.procs.2022.07.113.

28. Formica, F. Search-based software testing driven by automatically generated and manually defined fitness functions / F. Formica, T. Fan, C. Menghi // ACM Transactions on Software Engineering and Methodology. – 2023. – Vol. 33, № 2. – P. 1–37. – DOI: 10.1145/3571847.

29. Aghababaeyan, Z. Black-box testing of deep neural networks through test case diversity / Z. Aghababaeyan, M. Abdellatif, L. Briand, M. Bagherzadeh // IEEE Transactions on Software Engineering. – 2023. – Vol. 49, № 5. – P. 3182–3204. – DOI: 10.1109/TSE.2023.3263803.

30. Vos, T. E. J. TESTAR: An Open Source Tool for Scriptless GUI Test Automation / T. E. J. Vos, P. Aho, F. P. Ricos, O. Rodriguez-Valdes, A. Mulders // Software Testing, Verification and Reliability. – 2024. – Vol. 34, № 2. – DOI: 10.1002/stvr.1875.

31. Hourani, H. The impact of artificial intelligence on software testing / H. Hourani, A. Hammad, M. Lafi // Proceedings of the 2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT). – IEEE, 2019. – P. 565–570. – DOI: 10.1109/JEEIT.2019.8717439.

32. Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System [Electronic resource] / S. Nakamoto. – 2008. – 9 p. – Access mode: <https://bitcoin.org/bitcoin.pdf>.

33. Buterin, V. The Limits to Blockchain Scalability [Electronic resource] / V. Buterin. – 2021. – Access mode: <https://vitalik.ca/general/2021/05/23/scaling.html>.

34. European Commission. European Blockchain Services Infrastructure (EBSI) - White Paper [Electronic resource] / European Commission. – Brussels, 2018. – Access mode: <https://ec.europa.eu/digital-building-blocks/wikis/display/EBSI>.

35. Benet, J. IPFS - Content Addressed, Versioned, P2P File System [Electronic resource] / J. Benet // arXiv preprint. – 2014. – arXiv:1407.3561. – Access mode: <https://github.com/ipfs/papers/raw/master/ipfs-cap2pfs/ipfs-p2p-file-system.pdf>.

36. Ethereum Foundation. Proof-of-Stake (PoS) [Electronic resource] / Ethereum Foundation // Ethereum.org Documentation. – 2024. – Access mode: <https://ethereum.org/en/developers/docs/consensus-mechanisms/pos>.

ДОДАТОК А
Технічне завдання

Вступ

Використання децентралізованих технологій для верифікації академічних документів є надзвичайно важливим у сучасному освітньому середовищі. Зростаюча глобалізація освіти, міжнародна мобільність випускників та проблема підробки документів створюють потребу в швидких, надійних та безпечних методах верифікації кваліфікацій. Децентралізовані системи на основі блокчейн-технологій дозволяють автоматизувати процес верифікації, усунути посередників, забезпечити незмінність записів та надати власникам документів повний контроль над своїми даними, що у свою чергу допомагає підвищити довіру до академічних кваліфікацій, скоротити час верифікації з днів до секунд та зменшити адміністративні витрати університетів.

A.1 Підстава для розробки

Розробка здійснювалася на основі завдання на дипломну кваліфікаційну роботу з теми «Дослідження та реалізація децентралізованої системи ідентифікації на основі блокчейну для верифікації документів», у відповідності з наказом Національного університету «Запорізька політехніка».

A.2 Призначення розробки

Програмне забезпечення, розроблене для верифікації академічних документів, спрямоване на автоматизацію процесів видачі, зберігання та перевірки автентичності освітніх документів з використанням технологій блокчейн, IPFS та децентралізованих ідентифікаторів. Надає можливість університетам видавати верифіковані документи у форматі W3C Verifiable Credentials, випускникам зберігати та контролювати доступ до своїх

кваліфікацій, а роботодавцям швидко перевіряти автентичність документів без звернення до університетів.

Користувачі програми можуть видавати академічні документи з автоматичною реєстрацією в блокчейні та завантаженням до IPFS, переглядати та управляти виданими або отриманими документами, створювати безпечні способи обміну документами через IPFS CID або QR-коди, верифікувати автентичність документів за кілька секунд через перевірку криптографічних хешів у блокчейні, відкликати документи у випадку виявлення помилок або академічної недобросовісності, а також переглядати статистику та історію верифікацій.

А.3 Основні вимоги до розроблюваної програми

А.3.1 Вимоги до функціональності

Програмний продукт має забезпечувати наступну функціональність:

Для емітентів:

- автентифікація через Web3-гаманець MetaMask з цифровим підписом;
- створення верифікованих документів згідно зі специфікацією W3C VC;
- автоматична генерація криптографічних хешів SHA-256;
- завантаження документів до децентралізованого сховища IPFS;
- реєстрація хешів документів у блокчейні через смарт-контракт;
- перегляд списку виданих документів з пагінацією та фільтрацією;
- відкликання документів з вказанням причини;
- перегляд базової статистики виданих документів.

Для власників документів (випускників):

- автентифікація через MetaMask з автоматичним створенням DID;
- перегляд всіх отриманих документів у вигляді карток;
- фільтрація документів за статусом та типом;
- створення безпечних способів обміну через IPFS CID, QR-код або посилання;

- самостійна верифікація власних документів;
- завантаження документів у форматі JSON;
- перегляд статистики та статусу документів.

Для верифікаторів (роботодавців):

- автентифікація через MetaMask;
- верифікація документів за IPFS CID або через завантаження JSON-файлу;
- автоматична перевірка структури, хешу, статусу в блокчейні та цифрових підписів;
- відображення детального результату з метриками продуктивності;
- перегляд історії всіх проведених верифікацій з фільтрацією;
- перегляд статистики верифікацій.

A.3.2 Вимоги до надійності

Програмний продукт має забезпечувати коректну обробку помилок у наступних випадках:

- коли користувач вводить недопустимі значення у форми (невалідні Ethereum адреси, DID-ідентифікатори, IPFS CID);
- коли користувач здійснює спробу несанкціонованого доступу до документів інших емітентів;
- коли можливість отримання даних з IPFS-мережі відсутня або перевищено timeout;
- коли блокчейн-транзакція відхиляється через недостатній баланс або помилки смарт-контракту;
- коли JWT-токен автентифікації застарів або невалідний;
- коли MetaMask не встановлений або заблокований;
- коли користувач підключений до неправильної блокчейн-мережі.

A.3.3 Умови експлуатації

Для використання програмного забезпечення користувачу необхідні:

- персональний комп'ютер або мобільний пристрій з доступом до мережі Інтернет;
- встановлений Web3-гаманець MetaMask у веб-браузері;
- мінімальний баланс криптовалюти для оплати gas (для емітентів);
- налаштоване підключення до відповідної блокчейн-мережі (Hardhat Local, Polygon Mumbai testnet або Polygon mainnet);

Для повноцінного користування системою користувач повинен пройти автентифікацію через підпис challenge-повідомлення у MetaMask. Відповідно до розподілення ролей (емітент, власник, верифікатор), користувач має взаємодіяти зі своїм функціональним простором з відповідними можливостями.

A.3.4 Вимоги до складу й параметрів технічних засобів

Для коректного функціонування серверної частини застосунку апаратна частина сервера має відповідати наступним вимогам:

- процесор Intel Core i5 8-го покоління і вище або аналогічне рішення інших виробників;

- оперативна пам'ять 8 ГБ і вище;
- постійна пам'ять 256 ГБ SSD і вище для MongoDB та кешування;
- стабільне інтернет-з'єднання зі швидкістю не менше 10 Мбіт/с;

Програмне забезпечення серверної частини:

- операційна система Linux (Ubuntu 20.04+), macOS або Windows 10/11;
- Node.js версії 18 або новіша;
- MongoDB версії 6.0 або новіша;
- IPFS Desktop або доступ до IPFS gateway (Infura);
- Hardhat для локальної розробки або доступ до Polygon RPC endpoint;

Що стосується клієнтської частини, до неї висуваються наступні мінімальні вимоги:

- веб-браузер з підтримкою ES6, HTML5 та Web3 (Chrome 100+, Firefox 100+, Edge 100+, Safari 15+);
- встановлене розширення MetaMask;
- процесор Intel Core i3 5-го покоління і вище або аналогічне рішення інших виробників;
- оперативна пам'ять 4 ГБ і вище;
- інтернет-з'єднання зі швидкістю не менше 5 Мбіт/с.

А.3.5 Вимоги до безпеки

Програмний продукт має забезпечувати:

- шифрування всіх комунікацій між клієнтом та сервером через HTTPS;
- зберігання приватних ключів виключно в Web3-гаманці користувача без передачі на сервер;
- автентифікацію через JWT-токени з терміном дії не більше 7 днів;
- захист смарт-контрактів від типових вразливостей Solidity;
- rate limiting на рівні API (100 запитів на 15 хвилин з одного IP);
- валідацію та санітизацію всіх вхідних даних для запобігання XSS атакам;
- мінімізацію персональних даних у блокчейні (лише хеші);
- відповідність основним принципам GDPR.

А.3.6 Вимоги до продуктивності

Програмний продукт має забезпечувати:

- час верифікації документа < 5 секунд для некешованих запитів;
- час верифікації документа менше 1 секунди для кешованих запитів;
- час завантаження початкової сторінки не більше 3 секунд;

- стабільну підтримку мінімум 100 паралельних користувачів;
- час реєстрації документа в блокчейні 10-15 секунд залежно від мережі.

A.3.7 Вимоги до маркування й пакування

Програмне забезпечення може бути розповсюджене через Git-репозиторій або записане на будь-який носій інформації, пакування якого містить інформацію про назву програми – «Децентралізована система верифікації академічних документів», версію програмного забезпечення, інструкції з розгортання та конфігурації, приклади конфігураційних файлів (.env), документацію API endpoints.

A.3.8 Вимоги до транспортування й зберігання

Транспортування та зберігання програмної системи має відповідати вимогам, які аналогічні вимогам до носія, на якому зберігається програма. Для Git-репозиторію необхідне забезпечення контролю доступу та резервного копіювання. Приватні ключі та конфігураційні файли зі sensitive інформацією не повинні включатися до репозиторію та мають зберігатися окремо.

A.3.9 Вимоги до програмної документації

Програма повинна супроводжуватися:

- технічною документацією з описом архітектури системи
- інструкцією з розгортання backend та frontend компонентів
- інструкцією з компіляції та розгортання смарт-контракту
- описом API endpoints з прикладами запитів та відповідей
- інструкцією користувача для кожної ролі (емітент, власник, верифікатор)
- документацією з тестування з описом виконаних тестів та результатів

ДОДАТОК Б
Текст програми

Б.1 Текст модуля Blockchain.service.js

```

const { ethers } = require("ethers");
const fs = require("fs");
const path = require("path");
class BlockchainService {
  constructor() {
    this.provider = null;
    this.signer = null;
    this.contract = null;
    this.initialized = false;
    this.transactionLock = false;
    this.pendingQueue = [];
  }

  async executeWithLock(asyncFunction) {
    if (this.transactionLock) {
      return new Promise((resolve, reject) => {
        this.pendingQueue.push({ asyncFunction, resolve, reject });
      });
    }

    this.transactionLock = true;

    try {
      const result = await asyncFunction();
      return result;
    } finally {
      this.transactionLock = false;

      if (this.pendingQueue.length > 0) {
        const next = this.pendingQueue.shift();
        this.executeWithLock(next.asyncFunction)
          .then(next.resolve)
          .catch(next.reject);
      }
    }
  }

  async initialize() {
    try {
      const rpcUrl = process.env.BLOCKCHAIN_RPC_URL || "http://127.0.0.1:8545";
      this.provider = new ethers.JsonRpcProvider(rpcUrl);
      const network = await this.provider.getNetwork();
      console.log(
        `Connected to network: ${network.name} (chainId: ${network.chainId})`
      );
      const privateKey = process.env.ISSUER_PRIVATE_KEY;
      if (!privateKey) {
        throw new Error(
          "ISSUER_PRIVATE_KEY not found in environment variables"
        );
      }
      this.signer = new ethers.Wallet(privateKey, this.provider);
      console.log(`Signer address: ${this.signer.address}`);
      const contractPath = path.join(
        __dirname,
        "../contracts/DocumentRegistry.json"
      );
      const contractData = JSON.parse(fs.readFileSync(contractPath, "utf8"));
      this.contract = new ethers.Contract(
        contractData.address,
        contractData.abi,
        this.signer
      );
      console.log(`Contract loaded at address: ${contractData.address}`);
      this.initialized = true;
    } catch (error) {
      console.error("Failed to initialize blockchain service:", error);
      throw error;
    }
  }

  async registerDocument(docHash) {
    return this.executeWithLock(async () => {
      if (!this.initialized) {
        throw new Error("Blockchain service not initialized");
      }
    });
  }
}

```

```

console.log(`Registering document with hash: ${docHash}`);

const tx = await this.contract.registerDocument(docHash);
console.log(`Transaction sent: ${tx.hash}`);

const receipt = await tx.wait();
console.log(`Transaction confirmed in block ${receipt.blockNumber}`);

const event = receipt.logs
.map((log) => {
  try {
    return this.contract.interface.parseLog(log);
  } catch (e) {
    return null;
  }
})
.find(
  (parsedLog) => parsedLog.name === "DocumentRegistered"
);

return {
  success: true,
  transactionHash: receipt.hash,
  blockNumber: receipt.blockNumber,
  gasUsed: receipt.gasUsed.toString(),
  timestamp: event ? event.args.timestamp.toString() : null,
};
} catch (error) {
  console.error("Error registering document:", error);

  if (error.message.includes("Document already exists")) {
    throw new Error("DOCUMENT_ALREADY_EXISTS");
  }

  if (error.message.includes("Not authorized issuer")) {
    throw new Error("NOT_AUTHORIZED_ISSUER");
  }

  throw error;
}
});
}

async verifyDocument(docHash) {
  if (!this.initialized) {
    throw new Error("Blockchain service not initialized");
  }

  try {
    console.log(`Verifying document with hash: ${docHash}`);

    // Виклик view функції (не потребує газу)
    const [isValid, issuer, timestamp] = await this.contract.verifyDocument(
      docHash
    );

    return {
      isValid: isValid,
      issuer: issuer,
      timestamp: timestamp.toString(),
      registeredAt:
        timestamp > 0
        ? new Date(Number(timestamp) * 1000).toISOString()
        : null,
    };
  } catch (error) {
    console.error("Error verifying document:", error);
    throw error;
  }
}

async revokeDocument(docHash) {
  return this.executeWithLock(async () => {
    if (!this.initialized) {
      throw new Error("Blockchain service not initialized");
    }

    try {
      console.log(`Revoking document with hash: ${docHash}`);

      const tx = await this.contract.revokeDocument(docHash);
      console.log(`Revocation transaction sent: ${tx.hash}`);
    }
  });
}

```

```

const receipt = await tx.wait();
console.log(`Revocation confirmed in block ${receipt.blockNumber}`);

return {
  success: true,
  transactionHash: receipt.hash,
  blockNumber: receipt.blockNumber,
};
} catch (error) {
  console.error("Error revoking document:", error);

  if (error.message.includes("Not document owner")) {
    throw new Error("NOT_DOCUMENT_OWNER");
  }

  if (error.message.includes("Document does not exist")) {
    throw new Error("DOCUMENT_NOT_FOUND");
  }

  throw error;
}
});
}
async getTransactionStatus(txHash) {
  try {
    const receipt = await this.provider.getTransactionReceipt(txHash);

    if (!receipt) {
      return { status: "pending" };
    }

    return {
      status: receipt.status === 1 ? "success" : "failed",
      blockNumber: receipt.blockNumber,
      gasUsed: receipt.gasUsed.toString(),
    };
  } catch (error) {
    console.error("Error getting transaction status:", error);
    throw error;
  }
}

async estimateGas(operation, ...args) {
  try {
    const gasEstimate = await this.contract[operation].estimateGas(...args);
    return gasEstimate.toString();
  } catch (error) {
    console.error("Error estimating gas:", error);
    throw error;
  }
}

async getSignerBalance() {
  try {
    const balance = await this.provider.getBalance(this.signer.address);
    return ethers.formatEther(balance);
  } catch (error) {
    console.error("Error getting balance:", error);
    throw error;
  }
}
}

module.exports = new BlockchainService();

```

Б.2 Текст модуля Did.service.js

```

const { Resolver } = require("did-resolver");
const { getResolver } = require("ethr-did-resolver");
const { createJWT, verifyJWT } = require("did-jwt");
const { ethers } = require("ethers");
const DIDDocument = require("../models/DIDDocument");

class DIDService {
  constructor() {
    this.resolver = null;
    this.provider = null;
    this.initialized = false;
  }

```

```

async initialize() {
  try {
    // Підключення до Ethereum провайдера
    const rpcUrl = process.env.BLOCKCHAIN_RPC_URL || "http://127.0.0.1:8545";
    this.provider = new ethers.JsonRpcProvider(rpcUrl);

    const network = await this.provider.getNetwork();
    const networkId = network.chainId.toString();

    // Налаштування DID resolver для ethr методу
    const providerConfig = {
      networks: [
        {
          name: network.name || "development",
          chainId: networkId,
          provider: this.provider,
          registry: process.env.DID_REGISTRY_ADDRESS, // Опціонально - адреса ERC1056 реєстру
        },
      ],
    };

    const ethrResolver = getResolver(providerConfig);
    this.resolver = new Resolver(ethrResolver);

    console.log(
      `DID service initialized for network ${network.name} (chainId: ${networkId})`
    );
    this.initialized = true;
  } catch (error) {
    console.error("Failed to initialize DID service:", error);
    throw error;
  }
}

generateDID(address) {
  if (!ethers.isAddress(address)) {
    throw new Error("Invalid Ethereum address");
  }

  const normalizedAddress = address.toLowerCase();
  const did = `did:ethr:${normalizedAddress}`;

  return did;
}

createBasicDIDDocument(did, publicKey) {
  // Базовий DID-документ згідно W3C специфікації
  return {
    "@context": [
      "https://www.w3.org/ns/did/v1",
      "https://w3id.org/security/suites/secp256k1-2019/v1",
    ],
    id: did,
    verificationMethod: [
      {
        id: `${did}#controller`,
        type: "EcdsaSecp256k1RecoveryMethod2020",
        controller: did,
        blockchainAccountId: did.replace("did:ethr:", ""), // Ethereum адреса
      },
    ],
    authentication: [`${did}#controller`],
    assertionMethod: [`${did}#controller`],
  };
}

async resolveDID(did) {
  if (!this.initialized) {
    throw new Error("DID service not initialized");
  }

  try {
    console.log(`Resolving DID: ${did}`);

    // Перевірка кешу в MongoDB
    const cached = await DIDDocument.findOne({ did });

    if (cached && Date.now() - cached.resolvedAt.getTime() < 3600000) {
      // 1 година кеш
      console.log("DID document found in cache");
      return cached.document;
    }
  }
}

```

```

}

// Резолюція через did-resolver
const result = await this.resolver.resolve(did);

if (result.didResolutionMetadata.error) {
  throw new Error(
    `DID resolution error: ${result.didResolutionMetadata.error}`
  );
}

const didDocument = result.didDocument;
console.log("DID document resolved successfully");

// Збереження в кеш
await DIDDocument.findOneAndUpdate(
  { did },
  {
    did,
    document: didDocument,
    resolvedAt: new Date(),
  },
  { upsert: true, new: true }
);

return didDocument;
} catch (error) {
  console.error("Error resolving DID:", error);
  throw error;
}

async createVerifiablePresentation(holderDID, credentials, privateKey) {
  try {
    console.log(`Creating verifiable presentation for ${holderDID}`);

    // Створення структури презентації
    const presentation = {
      "@context": ["https://www.w3.org/2018/credentials/v1"],
      type: ["VerifiablePresentation"],
      holder: holderDID,
      verifiableCredential: Array.isArray(credentials)
        ? credentials
        : [credentials],
      issuanceDate: new Date().toISOString(),
    };

    // Створення JWT з презентацією
    const signer = {
      alg: "ES256K",
      signer: async (data) => {
        const wallet = new ethers.Wallet(privateKey);
        const signature = await wallet.signMessage(ethers.getBytes(data));
        return signature;
      },
    };

    const jwt = await createJWT(
      {
        vp: presentation,
        iss: holderDID,
        aud: "verifier", // Може бути конкретний DID верифікатора
      },
      {
        issuer: holderDID,
        signer: signer.signer,
        alg: signer.alg,
      },
      {
        expiresIn: 3600, // Термін дії 1 година
      }
    );

    console.log("Verifiable presentation created");
    return { presentation, jwt };
  } catch (error) {
    console.error("Error creating verifiable presentation:", error);
    throw error;
  }
}

```

```

async verifyVerifiablePresentation(jwt) {
  try {
    console.log("Verifying verifiable presentation");

    // Верифікація JWT підпису
    const verified = await verifyJWT(jwt, {
      resolver: this.resolver,
    });

    const { payload, didResolutionResult, issuer, signer } = verified;

    // Перевірка терміну дії
    if (payload.exp && Date.now() / 1000 > payload.exp) {
      throw new Error("Presentation expired");
    }

    // Перевірка holder відповідає issuer
    if (payload.vp.holder !== payload.iss) {
      throw new Error("Holder mismatch");
    }

    console.log("Presentation signature verified");

    return {
      valid: true,
      presentation: payload.vp,
      holder: payload.vp.holder,
      signer: signer,
      issuedAt: new Date(payload.iat * 1000).toISOString(),
      expiresAt: payload.exp
        ? new Date(payload.exp * 1000).toISOString()
        : null,
    };
  } catch (error) {
    console.error("Error verifying presentation:", error);
    return {
      valid: false,
      error: error.message,
    };
  }
}

async verifyCredentialIssuer(credential) {
  try {
    const issuerDID = credential.issuer;

    // Резолюція DID емітента
    const issuerDocument = await this.resolveDID(issuerDID);

    // Перевірка, що емітент має права видавати документи
    // В повній реалізації тут була б перевірка акредитації через DAO

    return {
      valid: true,
      issuerDID,
      issuerDocument,
    };
  } catch (error) {
    console.error("Error verifying credential issuer:", error);
    return {
      valid: false,
      error: error.message,
    };
  }
}

```

Б.3 Текст модуля Ipfs.service.js

```

const { create } = require("ipfs-http-client");
const { Buffer } = require("buffer");

class IPFSService {
  constructor() {
    this.client = null;
    this.initialized = false;
    this.localCache = new Map(); // Простий in-мемору кеш для CID
  }

```

```

async initialize() {
  try {
    // Вибір IPFS endpoint залежно від середовища
    const ipfsEndpoint = process.env.IPFS_ENDPOINT || "http://127.0.0.1:5001";

    this.client = create({
      url: ipfsEndpoint,
      timeout: 30000, // 30 секунд timeout
    });

    // Перевірка підключення
    const version = await this.client.version();
    console.log(`Connected to IPFS node version: ${version.version}`);

    this.initialized = true;
  } catch (error) {
    console.error("Failed to initialize IPFS service:", error);
    throw error;
  }
}

async uploadDocument(documentObject) {
  if (!this.initialized) {
    throw new Error("IPFS service not initialized");
  }

  try {
    console.log("Uploading document to IPFS...");

    // Сериалізація об'єкта в JSON
    const jsonString = JSON.stringify(documentObject, null, 2);
    const buffer = Buffer.from(jsonString, "utf-8");

    // Завантаження до IPFS
    const result = await this.client.add(buffer, {
      pin: true, // Автоматичне закріплення файлу
      cidVersion: 1, // Використання CIDv1 для кращої сумісності
    });

    const cid = result.cid.toString();
    console.log(`Document uploaded to IPFS with CID: ${cid}`);

    // Додавання до локального кешу
    this.localCache.set(cid, jsonString);

    // Обмеження розміру кешу до 100 документів
    if (this.localCache.size > 100) {
      const firstKey = this.localCache.keys().next().value;
      this.localCache.delete(firstKey);
    }

    return {
      cid: cid,
      size: buffer.length,
      url: `https://ipfs.io/ipfs/${cid}`, // Публічний gateway для перегляду
    };
  } catch (error) {
    console.error("Error uploading to IPFS:", error);
    throw error;
  }
}

async downloadDocument(cid) {
  if (!this.initialized) {
    throw new Error("IPFS service not initialized");
  }

  try {
    // Перевірка локального кешу
    if (this.localCache.has(cid)) {
      console.log(`Document found in local cache: ${cid}`);
      return JSON.parse(this.localCache.get(cid));
    }

    console.log(`Downloading document from IPFS: ${cid}`);

    // Завантаження з IPFS
    const chunks = [];
    for await (const chunk of this.client.cat(cid, { timeout: 30000 })) {
      chunks.push(chunk);
    }
  }
}

```

```

const buffer = Buffer.concat(chunks);
const jsonString = buffer.toString("utf-8");

// Додавання до кешу
this.localCache.set(cid, jsonString);
console.log(`Document downloaded successfully (${buffer.length} bytes)`);

return JSON.parse(jsonString);
} catch (error) {
console.error(`Error downloading from IPFS (CID: ${cid}):`, error);

if (error.message.includes("timeout")) {
throw new Error("IPFS_TIMEOUT");
}

if (error.message.includes("not found")) {
throw new Error("DOCUMENT_NOT_FOUND_IN_IPFS");
}

throw error;
}
}

async pinDocument(cid) {
if (!this.initialized) {
throw new Error("IPFS service not initialized");
}

try {
console.log(`Pinning document: ${cid}`);
await this.client.pin.add(cid);
console.log(`Document pinned successfully`);

return { success: true };
} catch (error) {
console.error("Error pinning document:", error);
throw error;
}
}

async unpinDocument(cid) {
if (!this.initialized) {
throw new Error("IPFS service not initialized");
}

try {
console.log(`Unpinning document: ${cid}`);
await this.client.pin.rm(cid);
console.log(`Document unpinned successfully`);

return { success: true };
} catch (error) {
console.error("Error unpinning document:", error);
throw error;
}
}

async checkDocumentAvailability(cid) {
try {
// Спроба отримати stat файлу без завантаження вмісту
const stat = await this.client.object.stat(cid, { timeout: 5000 });

return {
available: true,
size: stat.CumulativeSize,
};
} catch (error) {
return {
available: false,
error: error.message,
};
}
}

clearCache() {
this.localCache.clear();
console.log("IPFS cache cleared");
}

async downloadDocumentWithCache(cid, cacheModel) {
// Перевірка in-memory кешу

```

```

if (this.localCache.has(cid)) {
return JSON.parse(this.localCache.get(cid));
}

// Перевірка MongoDB кешу
if (cacheModel) {
const cached = await cacheModel.findOne({ cid });
if (cached) {
console.log(`Document found in MongoDB cache: ${cid}`);
this.localCache.set(cid, cached.content);
return JSON.parse(cached.content);
}
}

// Завантаження з IPFS
const document = await this.downloadDocument(cid);
const jsonString = JSON.stringify(document);

// Збереження в MongoDB кеш
if (cacheModel) {
await cacheModel
.create({
cid,
content: jsonString,
size: Buffer.byteLength(jsonString, "utf-8"),
})
.catch((err) => console.error("Cache save error:", err));
}

return document;
}

async downloadDocumentWithFallback(
cid,
gateways = [
"http://127.0.0.1:5001",
"https://ipfs.infura.io:5001",
"https://cloudflare-ipfs.com",
]
) {
let lastError;

for (const gateway of gateways) {
try {
const client = create({
url: gateway,
timeout: 10000,
});

const chunks = [];
for await (const chunk of client.cat(cid)) {
chunks.push(chunk);
}

const buffer = Buffer.concat(chunks);
return JSON.parse(buffer.toString("utf-8"));
} catch (error) {
console.warn(`Failed to download from ${gateway}:`, error.message);
lastError = error;
continue;
}
}

throw lastError;
}

module.exports = new IPFSService();

```

Б.4 Текст модуля database.js

```

const mongoose = require('mongoose');

const connectDB = async () => {
try {
const mongoURI = process.env.MONGODB_URI || 'mongodb://localhost:27017/document-verification';
const options = {
useNewUrlParser: true,
useUnifiedTopology: true,

```

```

maxPoolSize: 10, // Максимум 10 одночасних з'єднань
serverSelectionTimeoutMS: 5000, // Timeout при підключенні
socketTimeoutMS: 45000, // Timeout для операцій
};

await mongoose.connect(mongoURI, options);
console.log('MongoDB connected successfully');
console.log(`Database: ${mongoose.connection.name}`);

// Обробка подій з'єднання
mongoose.connection.on('error', (err) => {
  console.error('MongoDB connection error:', err);
});

mongoose.connection.on('disconnected', () => {
  console.warn('MongoDB disconnected');
});

// Graceful shutdown
process.on('SIGINT', async () => {
  await mongoose.connection.close();
  console.log('MongoDB connection closed through app termination');
  process.exit(0);
});
} catch (error) {
  console.error('MongoDB connection failed:', error);
  process.exit(1);
}
};

module.exports = connectDB;

```

Б.5 Текст модуля BlockchainService.test.js

```

const BlockchainService = require('../src/services/blockchain.service');
const { ethers } = require('ethers');

describe('BlockchainService', () => {
  let blockchainService;
  let testDocHash;

  beforeAll(async () => {
    blockchainService = new BlockchainService();
    await blockchainService.initialize();
    // Генеруємо тестовий хеш документа
    testDocHash = ethers.sha256(ethers.toUtf8Bytes('Test Document Content'));
  });

  describe('registerDocument', () => {
    test('should register a new document in blockchain', async () => {
      const result = await blockchainService.registerDocument(testDocHash);
      expect(result).toHaveProperty('transactionHash');
      expect(result).toHaveProperty('blockNumber');
      expect(result).toHaveProperty('timestamp');
      expect(result.events).toContainEqual(
        expect.objectContaining({ event: 'DocumentRegistered' })
      );
    }, 15000); // timeout 15 секунд для блокчейн операцій

    test('should prevent duplicate registration', async () => {
      await expect(
        blockchainService.registerDocument(testDocHash)
      ).rejects.toThrow('Document already registered');
    }, 15000);
  });

  describe('verifyDocument', () => {
    test('should verify registered document', async () => {
      const result = await blockchainService.verifyDocument(testDocHash);
      expect(result).toHaveProperty('isValid', true);
      expect(result).toHaveProperty('issuer');
      expect(result).toHaveProperty('timestamp');
      expect(result.issuer).toMatch(/^0x[a-fA-F0-9]{40}$/);
    });

    test('should return false for unregistered document', async () => {
      const randomHash = ethers.sha256(ethers.toUtf8Bytes('Non-existent Document'));
      const result = await blockchainService.verifyDocument(randomHash);
    });
  });
});

```

```

expect(result.isValid).toBe(false);
expect(result.timestamp).toBe('0');
});
});

describe('revokeDocument', () => {
  let revokeTestHash;

  beforeAll(async () => {
    revokeTestHash = ethers.sha256(ethers.toUtf8Bytes('Document To Revoke'));
    await blockchainService.registerDocument(revokeTestHash);
  });

  test('should revoke an existing document', async () => {
    const result = await blockchainService.revokeDocument(revokeTestHash);
    expect(result).toHaveProperty('transactionHash');
    expect(result.success).toBe(true);

    // Перевіряємо що документ дійсно відкликаний
    const verifyResult = await blockchainService.verifyDocument(revokeTestHash);
    expect(verifyResult.isValid).toBe(false);
  }, 15000);

  test('should not allow revoking already revoked document', async () => {
    await expect(
      blockchainService.revokeDocument(revokeTestHash)
    ).rejects.toThrow('Document already revoked');
  }, 15000);
});

describe('getSignerBalance', () => {
  test('should return signer balance', async () => {
    const balance = await blockchainService.getSignerBalance();
    expect(balance).toBeDefined();
    expect(parseFloat(balance)).toBeGreaterThan(0);
  });
});

```

Б.6 Текст модуля DIDService.test

```

const DIDService = require('../src/services/did.service');
const { ethers } = require('ethers');

describe('DIDService', () => {
  let didService;
  let testWallet;
  let testAddress;
  let testDID;

  beforeAll(() => {
    didService = new DIDService();
    testWallet = ethers.Wallet.createRandom();
    testAddress = testWallet.address;
  });

  describe('generateDID', () => {
    test('should generate valid DID from Ethereum address', () => {
      const result = didService.generateDID(testAddress);
      expect(result).toHaveProperty('did');
      expect(result).toHaveProperty('didDocument');
      expect(result.did).toBe(`did:ethr:${testAddress}`);
      expect(result.did).toMatch(/^did:ethr:0x[a-fA-F0-9]{40}$/);
      testDID = result.did;
    });

    test('should generate DID document with verification method', () => {
      const result = didService.generateDID(testAddress);
      expect(result.didDocument).toHaveProperty('id', testDID);
      expect(result.didDocument).toHaveProperty('verificationMethod');
      expect(result.didDocument.verificationMethod).toBeInstanceOf(Array);
      expect(result.didDocument.verificationMethod[0]).toHaveProperty('type');
      expect(result.didDocument.verificationMethod[0].type).toBe(
        'EcdsaSecp256k1RecoveryMethod2020'
      );
    });

    test('should throw error for invalid address', () => {

```

```

expect(() => {
  didService.generateDID('invalid-address');
}).toThrow();
});

describe('resolveDID', () => {
  test('should resolve DID to DID document', async () => {
    const didDocument = await didService.resolveDID(testDID);
    expect(didDocument).toHaveProperty('id', testDID);
    expect(didDocument).toHaveProperty('verificationMethod');
    expect(didDocument.verificationMethod[0]).toHaveProperty('controller', testDID);
  });

  test('should use cache for repeated resolutions', async () => {
    const startTime1 = Date.now();
    await didService.resolveDID(testDID);
    const firstTime = Date.now() - startTime1;
    const startTime2 = Date.now();
    await didService.resolveDID(testDID);
    const cachedTime = Date.now() - startTime2;
    expect(cachedTime).toBeLessThan(firstTime);
    expect(cachedTime).toBeLessThan(50); // Кеш має бути дуже швидким
  });

  test('should throw error for invalid DID format', async () => {
    await expect(
      didService.resolveDID('invalid:did:format')
    ).rejects.toThrow();
  });

  describe('verifySignature', () => {
    test('should verify valid signature', async () => {
      const message = 'Test message for signing';
      const signature = await testWallet.signMessage(message);
      const isValid = await didService.verifySignature(message, signature, testDID);
      expect(isValid).toBe(true);
    });

    test('should reject invalid signature', async () => {
      const message = 'Test message';
      const wrongWallet = ethers.Wallet.createRandom();
      const signature = await wrongWallet.signMessage(message);
      const isValid = await didService.verifySignature(message, signature, testDID);
      expect(isValid).toBe(false);
    });

    test('should reject tampered message', async () => {
      const originalMessage = 'Original message';
      const signature = await testWallet.signMessage(originalMessage);
      const tamperedMessage = 'Tampered message';
      const isValid = await didService.verifySignature(
        tamperedMessage,
        signature,
        testDID
      );
      expect(isValid).toBe(false);
    });
  });
});

```

Б.7 Текст модуля DocumentRegistry.sol

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

contract DocumentRegistry {
  struct Document {
    bytes32 docHash;
    address issuer;
    uint256 timestamp;
    bool isRevoked;
  }

  mapping(bytes32 => Document) public documents;
  mapping(address => bool) public authorizedIssuers;
  address public admin;
}

```

```

event DocumentRegistered(bytes32 indexed docHash, address indexed issuer, uint256 timestamp);
event DocumentRevoked(bytes32 indexed docHash, address indexed issuer);
event IssuerAuthorized(address indexed issuer);
event IssuerRevoked(address indexed issuer);

modifier onlyAdmin() {
    require(msg.sender == admin, "Only admin can perform this action");
    _;
}

modifier onlyAuthorized() {
    require(authorizedIssuers[msg.sender], "Not authorized issuer");
    _;
}

constructor() {
    admin = msg.sender;
    authorizedIssuers[msg.sender] = true;
}

function registerDocument(bytes32 _docHash) external onlyAuthorized {
    require(documents[_docHash].timestamp == 0, "Document already registered");

    documents[_docHash] = Document({
        docHash: _docHash,
        issuer: msg.sender,
        timestamp: block.timestamp,
        isRevoked: false
    });

    emit DocumentRegistered(_docHash, msg.sender, block.timestamp);
}

function verifyDocument(bytes32 _docHash)
    external
    view
    returns (bool isValid, address issuer, uint256 timestamp)
{
    Document memory doc = documents[_docHash];
    return (
        doc.timestamp > 0 && !doc.isRevoked,
        doc.issuer,
        doc.timestamp
    );
}

function revokeDocument(bytes32 _docHash) external onlyAuthorized {
    require(documents[_docHash].timestamp > 0, "Document not registered");
    require(documents[_docHash].issuer == msg.sender, "Not document issuer");
    require(!documents[_docHash].isRevoked, "Document already revoked");

    documents[_docHash].isRevoked = true;
    emit DocumentRevoked(_docHash, msg.sender);
}

function authorizeIssuer(address _issuer) external onlyAdmin {
    require(_issuer != address(0), "Invalid issuer address");
    require(!authorizedIssuers[_issuer], "Issuer already authorized");

    authorizedIssuers[_issuer] = true;
    emit IssuerAuthorized(_issuer);
}

function revokeIssuer(address _issuer) external onlyAdmin {
    require(_issuer != address(0), "Invalid issuer address");
    require(authorizedIssuers[_issuer], "Issuer not authorized");
    require(_issuer != admin, "Cannot revoke admin");

    authorizedIssuers[_issuer] = false;
    emit IssuerRevoked(_issuer);
}

function getDocument(bytes32 _docHash)
    external
    view
    returns (
        bytes32 docHash,
        address issuer,
        uint256 timestamp,
        bool isRevoked
    )

```

```

    )
    {
        Document memory doc = documents[_docHash];
        return (
            doc.docHash,
            doc.issuer,
            doc.timestamp,
            doc.isRevoked
        );
    }

    function isIssuerAuthorized(address _issuer) external view returns (bool) {
        return authorizedIssuers[_issuer];
    }
}

```

Б.8 Текст модуля DocumentRegistry.test.js

```

const { expect } = require("chai");
const { ethers } = require("hardhat");

describe("DocumentRegistry", function () {
    let documentRegistry;
    let owner;
    let issuer1;
    let issuer2;
    let unauthorized;

    beforeEach(async function () {
        [owner, issuer1, issuer2, unauthorized] = await ethers.getSigners();
        const DocumentRegistry = await ethers.getContractFactory("DocumentRegistry");
        documentRegistry = await DocumentRegistry.deploy();
        await documentRegistry.waitForDeployment();
    });

    describe("Deployment", function () {
        it("Should set the right admin", async function () {
            expect(await documentRegistry.admin()).to.equal(owner.address);
        });

        it("Should authorize deployer as issuer", async function () {
            const docHash = ethers.sha256(ethers.toUtf8Bytes("test"));
            await expect(documentRegistry.registerDocument(docHash)).to.not.be.reverted;
        });

        describe("Authorization", function () {
            it("Should allow admin to authorize new issuer", async function () {
                await expect(documentRegistry.authorizeIssuer(issuer1.address))
                    .to.emit(documentRegistry, "IssuerAuthorized")
                    .withArgs(issuer1.address);
            });

            it("Should prevent non-admin from authorizing issuer", async function () {
                await expect(
                    documentRegistry.connect(unauthorized).authorizeIssuer(issuer2.address)
                ).to.be.revertedWith("Only admin can perform this action");
            });

            it("Should allow admin to revoke issuer", async function () {
                await documentRegistry.authorizeIssuer(issuer1.address);
                await expect(documentRegistry.revokeIssuer(issuer1.address))
                    .to.emit(documentRegistry, "IssuerRevoked")
                    .withArgs(issuer1.address);
            });
        });

        describe("Document Registration", function () {
            let docHash;

            beforeEach(function () {
                docHash = ethers.sha256(ethers.toUtf8Bytes("Test Document Content"));
            });

            it("Should register a new document", async function () {
                await expect(documentRegistry.registerDocument(docHash))
                    .to.emit(documentRegistry, "DocumentRegistered")

```

```

.withArgs(docHash, owner.address, await ethers.provider.getBlock('latest').then(b => b.timestamp +
1));
});

it("Should prevent duplicate registration", async function () {
await documentRegistry.registerDocument(docHash);
await expect(
documentRegistry.registerDocument(docHash)
).to.be.revertedWith("Document already registered");
});

it("Should prevent unauthorized registration", async function () {
await expect(
documentRegistry.connect(unauthorized).registerDocument(docHash)
).to.be.revertedWith("Not authorized issuer");
});

it("Should store correct document data", async function () {
await documentRegistry.registerDocument(docHash);
const [isValid, issuer, timestamp] = await documentRegistry.verifyDocument(docHash);
expect(isValid).to.be.true;
expect(issuer).to.equal(owner.address);
expect(timestamp).to.be.gt(0);
});
});

describe("Document Verification", function () {
let docHash;

beforeEach(async function () {
docHash = ethers.sha256(ethers.toUtf8Bytes("Verification Test"));
await documentRegistry.registerDocument(docHash);
});

it("Should verify registered document", async function () {
const [isValid, issuer, timestamp] = await documentRegistry.verifyDocument(docHash);
expect(isValid).to.be.true;
expect(issuer).to.equal(owner.address);
expect(timestamp).to.be.gt(0);
});

it("Should return false for unregistered document", async function () {
const unregisteredHash = ethers.sha256(ethers.toUtf8Bytes("Unregistered"));
const [isValid, , timestamp] = await documentRegistry.verifyDocument(unregisteredHash);
expect(isValid).to.be.false;
expect(timestamp).to.equal(0);
});
});

describe("Document Revocation", function () {
let docHash;

beforeEach(async function () {
docHash = ethers.sha256(ethers.toUtf8Bytes("Revocation Test"));
await documentRegistry.registerDocument(docHash);
});

it("Should revoke own document", async function () {
await expect(documentRegistry.revokeDocument(docHash))
.to.emit(documentRegistry, "DocumentRevoked")
.withArgs(docHash, owner.address);
const [isValid] = await documentRegistry.verifyDocument(docHash);
expect(isValid).to.be.false;
});

it("Should prevent revoking someone else's document", async function () {
await documentRegistry.authorizeIssuer(issuer1.address);
await expect(
documentRegistry.connect(issuer1).revokeDocument(docHash)
).to.be.revertedWith("Not document issuer");
});

it("Should prevent double revocation", async function () {
await documentRegistry.revokeDocument(docHash);
await expect(
documentRegistry.revokeDocument(docHash)
).to.be.revertedWith("Document already revoked");
});

it("Should prevent revoking unregistered document", async function () {
const unregisteredHash = ethers.sha256(ethers.toUtf8Bytes("Unregistered"));

```

```

await expect(
documentRegistry.revokeDocument(unregisteredHash)
).to.be.revertedWith("Document not registered");
});
});
});

```

Б.9 Текст модуля Web3Service.test.js

```

import { ethers } from 'ethers';

class Web3Service {
  constructor() {
    this.provider = null;
    this.signer = null;
    this.isMetaMaskInstalled = false;
  }

  checkMetaMaskInstallation() {
    this.isMetaMaskInstalled = typeof window.ethereum !== 'undefined';
    return this.isMetaMaskInstalled;
  }

  async connectWallet() {
    try {
      if (!this.checkMetaMaskInstallation()) {
        return {
          success: false,
          error: 'MetaMask is not installed. Please install MetaMask extension.'
        };
      }

      // Запит дозволу на підключення
      const accounts = await window.ethereum.request({
        method: 'eth_requestAccounts'
      });

      if (accounts.length === 0) {
        return {
          success: false,
          error: 'No accounts found. Please unlock MetaMask.'
        };
      }

      // Створення провайдера та підписувача
      this.provider = new ethers.BrowserProvider(window.ethereum);
      this.signer = await this.provider.getSigner();

      const address = accounts[0];
      const network = await this.provider.getNetwork();

      console.log('Connected to wallet:', address);
      console.log('Network:', network.name, 'Chain ID:', network.chainId);

      return {
        success: true,
        address,
        chainId: Number(network.chainId),
        networkName: network.name
      };
    } catch (error) {
      console.error('Connect wallet error:', error);

      // Обробка специфічних помилок MetaMask
      if (error.code === 4001) {
        return {
          success: false,
          error: 'Connection request was rejected by user'
        };
      }

      return {
        success: false,
        error: error.message || 'Failed to connect wallet'
      };
    }
  }
}

```

```

async getAccount() {
  try {
    if (!this.provider) {
      await this.connectWallet();
    }

    const accounts = await this.provider.listAccounts();
    return accounts.length > 0 ? accounts[0].address : null;
  } catch (error) {
    console.error('Get account error:', error);
    return null;
  }
}

async getBalance(address) {
  try {
    if (!this.provider) {
      throw new Error('Provider not initialized');
    }

    const balance = await this.provider.getBalance(address);
    return ethers.formatEther(balance);
  } catch (error) {
    console.error('Get balance error:', error);
    return '0';
  }
}

async signMessage(message) {
  try {
    if (!this.signer) {
      throw new Error('Signer not initialized. Please connect wallet first.');
```

```

    }

    console.log('Requesting signature for message:', message);
    const signature = await this.signer.signMessage(message);
```

```

    return {
      success: true,
      signature
    };
  } catch (error) {
    console.error('Sign message error:', error);
```

```

    if (error.code === 4001) {
      return {
        success: false,
        error: 'Signature request was rejected by user'
      };
    }
  }
}
```

```

return {
  success: false,
  error: error.message || 'Failed to sign message'
};
}
}
```

```

async switchNetwork(chainId) {
  try {
    await window.ethereum.request({
      method: 'wallet_switchEthereumChain',
      params: [{ chainId: `0x${chainId.toString(16)}` } ]
    });
```

```

    return { success: true };
  } catch (error) {
    console.error('Switch network error:', error);
```

```

    // Якщо мережа не додана до MetaMask
    if (error.code === 4902) {
      return {
        success: false,
        error: 'Network not added to MetaMask',
        needsToAddNetwork: true
      };
    }
  }
}
```

```

return {
  success: false,
  error: error.message
}
```

```

};
}
}

async addNetwork(networkConfig) {
  try {
    await window.ethereum.request({
      method: 'wallet_addEthereumChain',
      params: [networkConfig]
    });

    return { success: true };
  } catch (error) {
    console.error('Add network error:', error);
    return {
      success: false,
      error: error.message
    };
  }
}

formatAddress(address) {
  if (!address) return '';
  return `${address.slice(0, 6)}...${address.slice(-4)}`;
}

isValidAddress(address) {
  return ethers.isAddress(address);
}
}

export const web3Service = new Web3Service();

```

Б.10 Текст модуля App.jsx

```

import { Suspense, lazy } from 'react';
import { BrowserRouter, Navigate, Route, Routes } from 'react-router-dom';
import { ToastContainer } from 'react-toastify';
import 'react-toastify/dist/ReactToastify.css';

import LoadingSpinner from './components/common/LoadingSpinner';
import Layout from './components/layout/Layout';
import ProtectedRoute from './components/ProtectedRoute';
import { AuthProvider } from './context/AuthContext';
import { Web3Provider } from './context/Web3Context';

// Lazy loading сторінок
const Home = lazy(() => import('./pages/Home'));
const Login = lazy(() => import('./pages/Login'));

// Issuer pages
const IssueCredential = lazy(() => import('./pages/issuer/IssueCredential'));
const IssuedCredentials = lazy(() => import('./pages/issuer/IssuedCredentials'));
const IssuerDashboard = lazy(() => import('./pages/issuer/IssuerDashboard'));

// Holder pages
const MyCredentials = lazy(() => import('./pages/holder/MyCredentials'));
const HolderDashboard = lazy(() => import('./pages/holder/HolderDashboard'));

// Verifier pages
const VerifyCredential = lazy(() => import('./pages/verifier/VerifyCredential'));
const VerificationHistory = lazy(() => import('./pages/verifier/VerificationHistory'));
const VerifierDashboard = lazy(() => import('./pages/verifier/VerifierDashboard'));

function App() {
  return (
    <BrowserRouter>
    <AuthProvider>
    <Web3Provider>
    <Layout>
    <Suspense fallback={<LoadingSpinner fullScreen />}>
    <Routes>
      {/* Публічні маршрути */}
      <Route path="/" element={<Home />} />
      <Route path="/login" element={<Login />} />

      {/* Маршрути емітента */}
    <Route

```

```

path="/issuer/dashboard"
element={
  <ProtectedRoute allowedRoles={['issuer', 'admin']}>
    <IssuerDashboard />
  </ProtectedRoute>
}
/>
<Route
  path="/issuer/issue"
  element={
    <ProtectedRoute allowedRoles={['issuer', 'admin']}>
      <IssueCredential />
    </ProtectedRoute>
  }
/>
<Route
  path="/issuer/credentials"
  element={
    <ProtectedRoute allowedRoles={['issuer', 'admin']}>
      <IssuedCredentials />
    </ProtectedRoute>
  }
/>

{ /* Маршрути власника */ }
<Route
  path="/holder/dashboard"
  element={
    <ProtectedRoute allowedRoles={['holder']}>
      <HolderDashboard />
    </ProtectedRoute>
  }
/>
<Route
  path="/holder/credentials"
  element={
    <ProtectedRoute allowedRoles={['holder']}>
      <MyCredentials />
    </ProtectedRoute>
  }
/>

{ /* Маршрути верифікатора */ }
<Route
  path="/verifier/dashboard"
  element={
    <ProtectedRoute allowedRoles={['verifier', 'admin']}>
      <VerifierDashboard />
    </ProtectedRoute>
  }
/>
<Route
  path="/verifier/verify"
  element={
    <ProtectedRoute allowedRoles={['verifier', 'admin']}>
      <VerifyCredential />
    </ProtectedRoute>
  }
/>
<Route
  path="/verifier/history"
  element={
    <ProtectedRoute allowedRoles={['verifier', 'admin']}>
      <VerificationHistory />
    </ProtectedRoute>
  }
/>

{ /* 404 */ }
<Route path="*" element={<Navigate to="/" replace />} />
</Routes>
</Suspense>
</Layout>

<ToastContainer
  position="top-right"
  autoClose={5000}
  hideProgressBar={false}
  newestOnTop={false}
  closeOnClick
  rtl={false}

```

```
    pauseOnFocusLoss
    draggable
    pauseOnHover
  />
</Web3Provider>
</AuthProvider>
</BrowserRouter>
);
}

export default App;
```

ДОДАТОК В
Слайди презентації

Національний університет “Запорізька політехніка”
Кафедра програмних засобів
Дипломна кваліфікаційна робота магістра

×
×
×
×

ДОСЛІДЖЕННЯ ТА РЕАЛІЗАЦІЯ ДЕЦЕНТРАЛІЗОВАНОЇ СИСТЕМИ ІДЕНТИФІКАЦІЇ НА ОСНОВІ БЛОКЧЕЙНУ ДЛЯ ВЕРИФІКАЦІЇ ДОКУМЕНТІВ

× Виконав
× ст.гр. КНТ-124м
×
× Керівник
× д.т.н., професор

Максим ЖМУЦЬКИЙ

Андрій ОЛІЙНИК

Рисунок В.1 – Слайд №



МЕТА, АКТУАЛЬНІСТЬ, ОБ'ЄКТ І ПРЕДМЕТ ДОСЛІДЖЕННЯ

Об'єкт дослідження – процес розробки програмного забезпечення для верифікації документів з використанням блокчейн-технологій.

Предмет дослідження – методи та програмні засоби децентралізованої ідентифікації W3C DID та верифікації документів через криптографічні хеші SHA-256, смарт-контракти Ethereum та IPFS.

Мета роботи – розробка децентралізованої програмної системи верифікації документів для підвищення швидкості перевірки та безпеки від підробки.

Актуальність – традиційна верифікація займає дні-тижні, вразлива до підробки, має єдину точку відмови, порушує приватність користувачів.

× × × ×

02

Рисунок В.2 – Слайд № 2

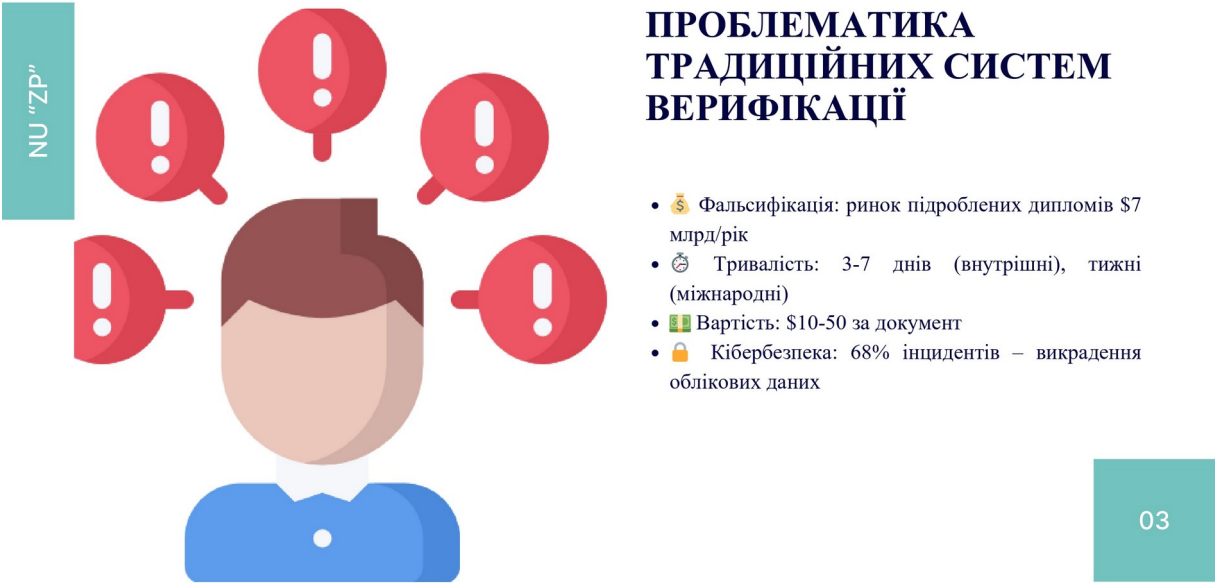
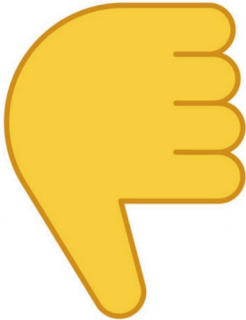


Рисунок В.3 – Слайд № 3



Рисунок В.4 – Слайд № 4

05



НЕДОЛКИ БЛОКЧЕЙН-РІШЕНЬ

- 1.Економічна неефективність: \$5-50+ (публічні блокчейни)
- 2.Масштабованість: 5-30 тр/сек → потрібно 1000+ тр/сек
- 3.Відкликання: немає стандартних механізмів
- 4.UX: складні концепції для користувачів
- 5.GDPR: незмінність vs право на забуття

Основні ініціативи:

- Blockcerts (MIT, 2016) – Bitcoin/Ethereum
- Університет Нікосії (2014) – перший заклад
- EBSI (ЄС, 2018) – державна мережа
- Hyperledger Fabric – корейські університети

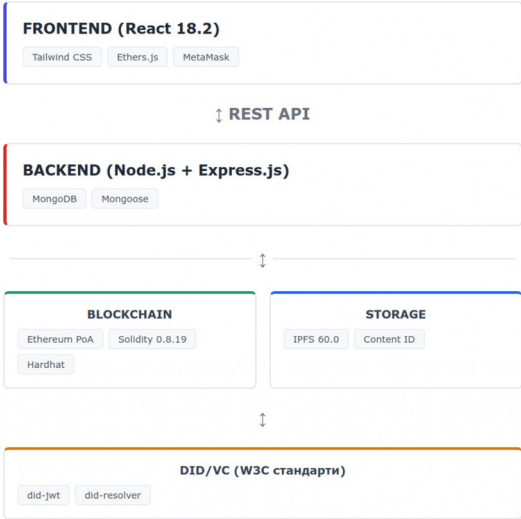
NU "ZP"

× × × ×

Рисунок В.5 – Слайд № 5

06

ТЕХНОЛОГІЧНИЙ СТЕК СИСТЕМИ



NU "ZP"

NU "ZP"

Рисунок В.6 – Слайд № 6

ПОРІВНЯННЯ ТЕХНОЛОГІЙ

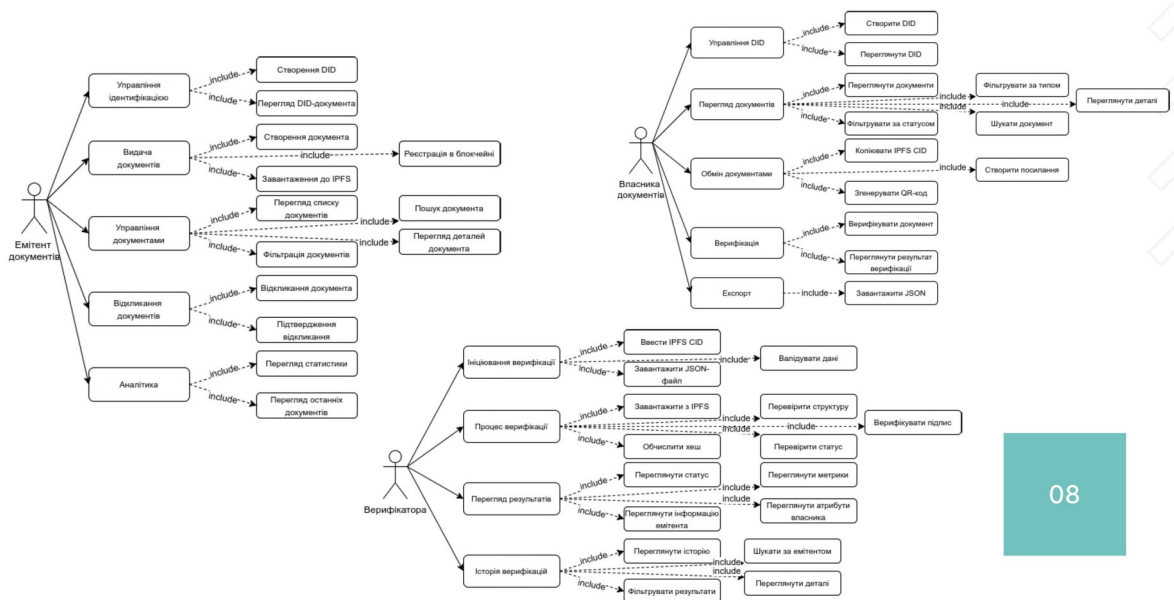
Критерій	JavaScript	Python	Go
Блокчейн-бібліотеки	Відмінна	Добра	Добра
Веб-розробка	✓	Фреймворки	Базова
DID/VC екосистема	Відмінна	Добра	Базова

Платформа	Вартість	Час	Масштабованість
Ethereum (публічний)	\$5-50+	12с-10хв	15-30 тр/с
Polygon	<\$0.01	2-5с	7000+ тр/с
Приватний Ethereum	✓	3-5с	1000+ тр/с
Hyperledger	~\$0	1-3с	3000+ тр/с

07

Рисунок В.7 – Слайд № 7

ФУНКЦІОНАЛЬНІ ВИМОГИ



08

Рисунок В.8 – Слайд № 8

NU "ЗР"

АРХІТЕКТУРА СИСТЕМИ

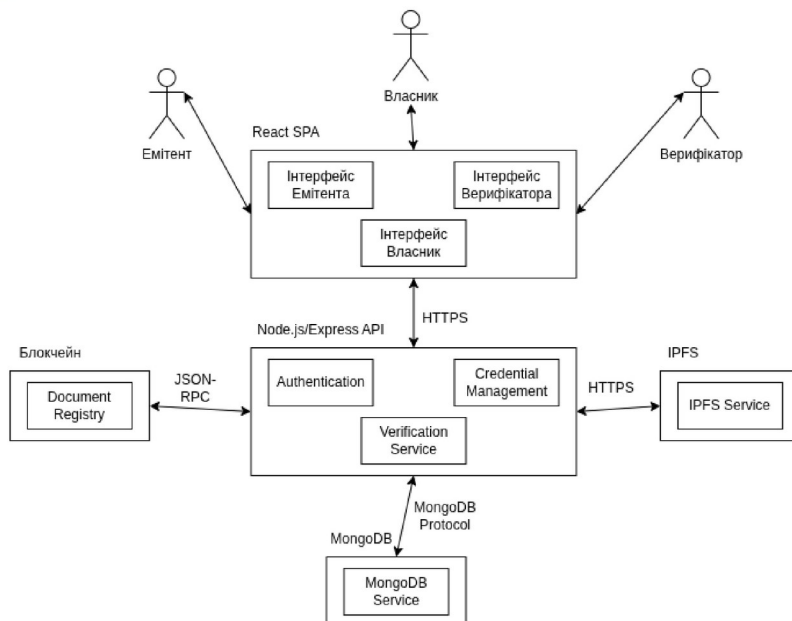


Рисунок В.9 – Слайд № 9

NU "ЗР"

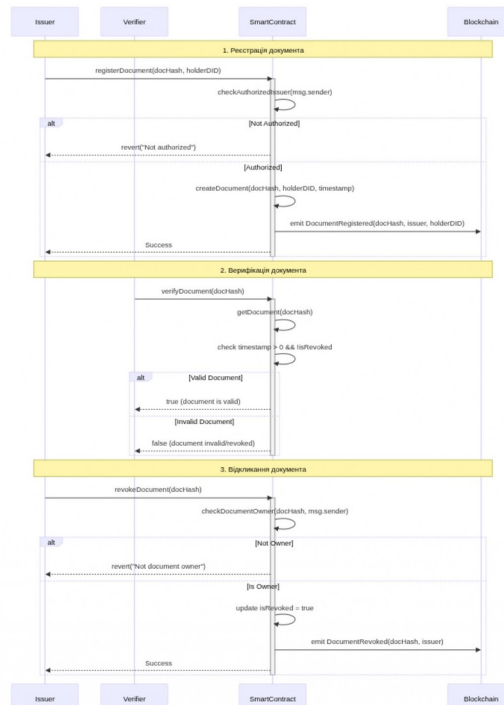
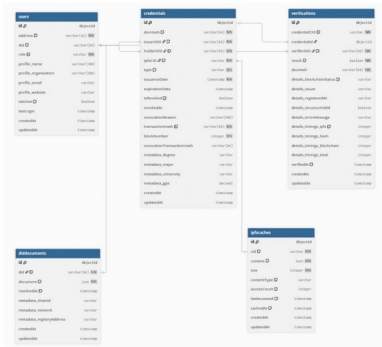
СМАРТ-КОНТРАКТ
DOCUMENTREGISTRY

Рисунок В.10 – Слайд № 10

10

NU “ЗР”



ЗБЕРІГАННЯ
ДАНИХ

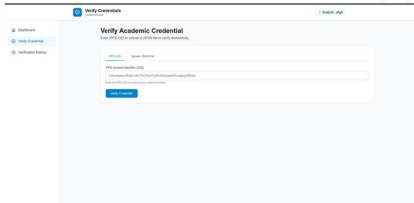
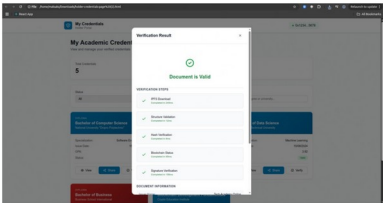
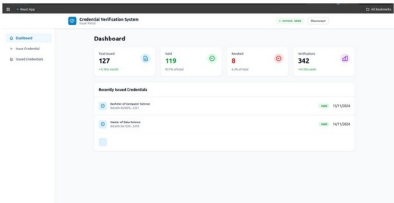
11

Сховище	Дані	Призначення
Blockchain	Хеші, статуси, DID	Незмінність, довіра
IPFS	Повні JSON документи	Децентралізація, цілісність
MongoDB	Метадані, кеш, профілі	Швидкість, операції

Рисунок В.11 – Слайд № 11

NU “ЗР”

ІНТЕРФЕЙС СИСТЕМИ



12

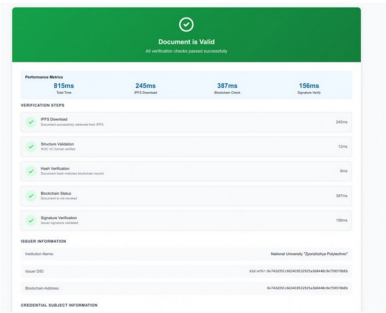
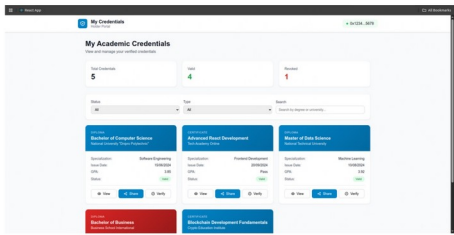


Рисунок В.12 – Слайд № 12

NU “ZP”

РЕЗУЛЬТАТИ КОМПЛЕКСНОГО
ТЕСТУВАННЯ

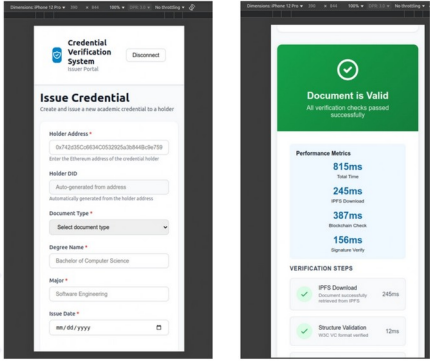
Тип тестування	Тестів	Успішно	Провалено	Час	Покриття
Модульне (Jest)	131	130	1	7.7 с	~75% функцій
Смарт-контракт (Hardhat)	12	12	0	15.2 с	100% функцій
Інтеграційне	8	8	0	-	Ручне
UI/Сумісність	8 браузерів	6 повна	2 частково	-	Основні сценарії
Ручне	Всі компоненти	-	-	-	-

13

Рисунок В.13 – Слайд № 13

NU “ZP”

ТЕСТУВАННЯ СУМІСНОСТІ З
БРАУЗЕРАМИ



14

Браузер	Версія	ОС	Автентифікація	Видача	Верифікація	Статус
Chrome	120+	Win/macOS	✓	✓	✓	✓ Повна
Firefox	121+	Win/Linux	✓	✓	✓	✓ Повна
Edge	120+	Windows	✓	✓	✓	✓ Повна
Safari	17+	macOS	✓	✓	✓	⚠ Налаштування
Mobile Safari	17	iOS	✓	⚠	✓	⚠ MetaMask app
Chrome Mobile	120+	Android	✓	✓	✓	✓ Повна

Рисунок В.14 – Слайд № 14

ВИСНОВКИ

- **Гібридна архітектура зберігання:** поєднання Blockchain для незмінних хешів, IPFS для документів та MongoDB для кешу забезпечує безпеку та ефективність.
- **Повний функціонал для трьох ролей:** система охоплює всі процеси - від видання документів емітентами до їх перевірки верифікаторами за <5 секунд.
- **Висока продуктивність:** верифікація документів відбувається за 3-5 секунд (перший раз) та прискорюється до 23-50 мс завдяки кешуванню.
- **Міжнародна стандартизація:** реалізовано підтримку W3C DID (did:ethr) та Verifiable Credentials для забезпечення сумісності.
- **Наукове визнання:** робота апробована на міжнародних конференціях 2025 року та Всеукраїнському конкурсі студентських робіт.

Рисунок В.15 – Слайд № 15