

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
Факультет інформатики та обчислювальної техніки
(повна назва інституту/факультету)

Кафедра інформатики та програмної інженерії
(повна назва кафедри)

«До захисту допущено»

Завідувач кафедри

_____ Едуард ЖАРІКОВ
(підпис) (ім'я прізвище)

“ _____ ” _____ 2025 р.

Дипломний проєкт
на здобуття ступеня бакалавра
за освітньо-професійною програмою «Інженерія програмного забезпечення
інформаційних систем»
спеціальності «121 Інженерія програмного забезпечення»

на тему: Програмне забезпечення для збереження та валідації документів
на основі смарт-контрактів в блокчейнах

Виконала студентка IV курсу, групи ІІІ-311
(шифр групи)

Біленька Олеся Русланівна _____
(прізвище, ім'я, по батькові) (підпис)

Керівник доцент, к.т.н., доц., Лісовиченко Олег Іванович _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент доц., к.т.н., доц. каф. ІСТ Ткач Михайло _____
Мартинівич _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті
немає запозичень з праць інших авторів без
відповідних посилань.

Студент _____
(підпис)

Київ – 2025

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії
Рівень вищої освіти – перший (бакалаврський)
Спеціальність – 121 Інженерія програмного забезпечення
Освітньо-професійна програма – Інженерія програмного забезпечення
інформаційних систем

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Едуард ЖАРІКОВ
(підпис) (ім'я прізвище)

“ ” _____ 2025 р.

ЗАВДАННЯ
на дипломний проєкт студенту

Біленькій Олесі Русланівні

(прізвище, ім'я, по батькові)

1. Тема проєкту Програмне забезпечення для збереження та валідації документів на основі смарт-контрактів в блокчейнах

керівник проєкту Лісовиченко Олег Іванович, к.т.н., доцент

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

затверджені наказом по університету від «23» травня 2025 р. №1706-с

2. Термін подання студентом проєкту «16» червня 2025 року

3. Вихідні дані до проєкту: технічне завдання

4. Зміст пояснювальної записки

1) Передпроєктне обстеження предметної області: аналіз предметної області, аналіз існуючих рішень, опис бізнес-процесів, постановка задачі.

2) Розроблення вимог до програмного забезпечення: варіанти використання програмного забезпечення, аналіз системних вимог, розроблення функціональних вимог, розроблення нефункціональних вимог

3) Конструювання та розроблення програмного забезпечення: архітектура програмного забезпечення, обґрунтування вибору засобів розробки, конструювання програмного забезпечення, аналіз безпеки даних.

4) Аналіз якості та тестування програмного забезпечення: аналіз якості ПЗ, опис процесів тестування, опис контрольного прикладу.

5) Розгортання та супровід програмного забезпечення: розгортання програмного забезпечення, супровід програмного забезпечення.

5. Перелік графічного матеріалу

1) Схема структурна варіантів використань _____

2) Схема структурна класів програмного забезпечення _____

3) Креслення вигляду екранних форм _____

6. Консультанти розділів проєкту

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

7. Дата видачі завдання «15» березня 2025 року

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1	Вивчення рекомендованої літератури	15.03.2025	
2	Аналіз існуючих методів розв'язання задачі	28.03.2025	
3	Постановка та формалізація задачі	06.04.2025	
4	Розробка інформаційного забезпечення	10.04.2025	
5	Алгоритмізація задачі	22.04.2025	
6	Обґрунтування вибору використаних технічних засобів	28.04.2025	
7	Розробка програмного забезпечення	01.05.2025	
8	Налагодження програми	05.05.2025	
9	Виконання графічних документів	07.05.2025	
10	Оформлення пояснювальної записки	12.05.2025	
11	Подання ДП на попередній захист	02.06.2025	
12	Подання ДП рецензенту	10.06.2025	
13	Подання ДП на основний захист	16.06.2025	

Студентка

(підпис)

Олеся БІЛЕНЬКА

(ініціали, прізвище)

Керівник

(підпис)

Олег ЛІСОВИЧЕНКО

(ініціали, прізвище)

АНОТАЦІЯ

Пояснювальна записка дипломного проєкту складається з п'яти розділів, містить 37 таблиць, 24 рисунків та 28 джерел – загалом 86 сторінок.

Дипломний проєкт присвячений розробці веб-застосунку для зберігання та верифікації автентичності цифрових документів з використанням смарт-контрактів в мережі Ethereum.

У роботі реалізовано безпечне збереження криптографічних хешів документів у блокчейні, що забезпечує незмінність даних та прозору перевірку без залучення довіреної сторони. Для безпечного оновлення контрактної логіки впроваджено проксі-патерн UUPS з контролем доступу через мультипідписний гаманець (Gnosis Safe). Комунікація між мікросервісами організована через Apache Kafka.

Мета проєкту — створення децентралізованої системи перевірки достовірності документів, яка поєднує переваги блокчейн-технологій із зручністю класичного веб-інтерфейсу.

У першому розділі розглянуто аналіз предметної області, існуючих рішень, моделей зберігання та перевірки цифрових документів.

Другий розділ присвячений визначенню вимог до програмного забезпечення, моделюванню бізнес-процесів, побудові діаграм варіантів використання та трасування вимог.

У третьому розділі викладено архітектуру системи, структуру смарт-контрактів, класів та бази даних, описано використаний технологічний стек.

Четвертий розділ присвячений опису реалізації основних алгоритмів, перевірки безпеки системи, структури контрактів та механізму верифікації.

П'ятий розділ включає оцінку результатів, висновки та рекомендації щодо подальшого розвитку проєкту.

Програмне забезпечення впроваджено в мережі Ethereum Sepolia, з використанням Infura, PostgreSQL та Spring Boot.

КЛЮЧОВІ СЛОВА: БЛОКЧЕЙН, СМАРТ-КОНТРАКТ, ETHEREUM, JWT, KAFKA, UUPS PROXY, ВЕРИФІКАЦІЯ ДОКУМЕНТІВ, ВЕБ-ДОДАТОК, SPRING BOOT, ANGULAR, WEB3J, POSTGRESQL.

ABSTRACT

The explanatory note of the diploma project consists of five sections, contains 37 tables, 24 figures and 28 sources – in total 86 pages.

The diploma project is dedicated to the development of a web application for storing and verifying the authenticity of digital documents using smart contracts on the Ethereum blockchain.

The system ensures secure storage of cryptographic document hashes in the blockchain, guaranteeing data immutability and transparent verification without the involvement of a trusted third party. To enable safe logic upgrades, the project implements the UUPS proxy pattern with access control managed through a multisignature wallet (Gnosis Safe). Communication between microservices is handled asynchronously using Apache Kafka.

The goal of the project is to create a decentralized document authenticity verification system that combines the advantages of blockchain technologies with the convenience of a traditional web interface.

The **first chapter** covers the analysis of the subject area, existing solutions, and models for document storage and verification.

The **second chapter** is dedicated to defining software requirements, modeling business processes, and developing use case diagrams and traceability matrices.

The **third chapter** outlines the system architecture, the structure of smart contracts, classes, and the database, as well as the chosen technology stack.

The **fourth chapter** describes the implementation of the main algorithms, system security checks, contract structure, and the verification mechanism.

The **fifth chapter** contains the evaluation of results, conclusions, and recommendations for further development.

The software is deployed on Ethereum Sepolia, using Infura, PostgreSQL, and Spring Boot.

KEYWORDS: BLOCKCHAIN, SMART CONTRACT, ETHEREUM, JWT, KAFKA, UUPS PROXY, DOCUMENT VERIFICATION, WEB APPLICATION, SPRING BOOT, ANGULAR, WEB3J, POSTGRESQL.

Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

“ЗАТВЕРДЖЕНО”

Завідувач кафедри

_____ Едуард ЖАРІКОВ

“ ” _____ 2025 р.

**ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ ЗБЕРЕЖЕННЯ ТА ВАЛІДАЦІЇ
ДОКУМЕНТІВ НА ОСНОВІ СМАРТ-КОНТРАКТІВ В БЛОКЧЕЙНАХ**

Технічне завдання

КП.ІП-з1101.045470.01.91

“ПОГОДЖЕНО”

Керівник проєкту:

_____ Олег ЛІСОВИЧЕНКО

Нормоконтроль:

_____ Олександр
СТЕЛЬМАХ

Виконавець:

_____ Олеся БІЛЕНЬКА

Київ – 2025

Зміст

1 НАЙМЕНУВАННЯ ТА ГАЛУЗЬ ЗАСТОСУВАННЯ	3
2 ПІДСТАВА ДЛЯ РОЗРОБКИ	4
3 ПРИЗНАЧЕННЯ РОЗРОБКИ	5
4 ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	6
4.1 Вимоги до функціональних характеристик	6
4.1.1 Користувацького інтерфейсу	6
4.1.2 Облік користувачів:	9
4.2 Вимоги до надійності	9
4.3 Умови експлуатації	10
4.3.1 Вид обслуговування	10
4.3.2 Обслуговуючий персонал	11
4.4 Вимоги до складу і параметрів технічних засобів	11
4.5 Вимоги до інформаційної та програмної сумісності	12
4.5.1 Вимоги до вхідних даних	13
4.5.2 Вимоги до вихідних даних	13
4.5.3 Вимоги до мови розробки	13
4.5.4 Вимоги до середовища розробки	14
4.5.5 Вимоги до представлення вихідних кодів	14
4.6 Вимоги до маркування та пакування	14
4.7 Вимоги до транспортування та зберігання	15
4.8 Спеціальні вимоги	15

1 НАЙМЕНУВАННЯ ТА ГАЛУЗЬ ЗАСТОСУВАННЯ

Назва розробки: ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ ЗБЕРЕЖЕННЯ ТА ВАЛІДАЦІЇ ДОКУМЕНТІВ НА ОСНОВІ СМАРТ-КОНТРАКТІВ В БЛОКЧЕЙНАХ.

Галузь застосування:

Наведене технічне завдання поширюється на розробку програмного забезпечення «Програмні засоби захисту інформації» (шифр 045470), яке використовується для збереження, перевірки достовірності та валідації електронних документів з використанням смарт-контрактів у блокчейн-мережах.

Програмне забезпечення призначене для застосування у галузях, де важливими є безпека даних, автентичність електронних документів та неможливість їх несанкціонованої модифікації. Потенційними користувачами є державні установи, фінансові організації, освітні заклади, юридичні особи, а також інші суб'єкти, які працюють з чутливою або юридично значущою інформацією в електронній формі.

2 ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки «Програмне забезпечення для збереження та валідації документів на основі смарт-контрактів в блокчейнах» є завдання на дипломне проєктування, затверджене кафедрою інформатики та програмної інженерії Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського».

3 ПРИЗНАЧЕННЯ РОЗРОБКИ

Розробка призначена для реалізації функціонального та експлуатаційного середовища зберігання, перевірки достовірності та валідації електронних документів за допомогою смарт-контрактів у блокчейн-мережах. Система автоматизує процес фіксації хешів документів у реєстрі, їх подальшої перевірки та забезпечує інтеграцію з вебінтерфейсом для модераторів і користувачів.

Метою розробки є забезпечення високої надійності, незмінності та доступності даних, а також уніфікований процес перевірки автентичності документів. Це досягається шляхом використання децентралізованих технологій (EVM-сумісний блокчейн), стандартизованих алгоритмів хешування та захищеного обміну даними між мікросервісами системи.

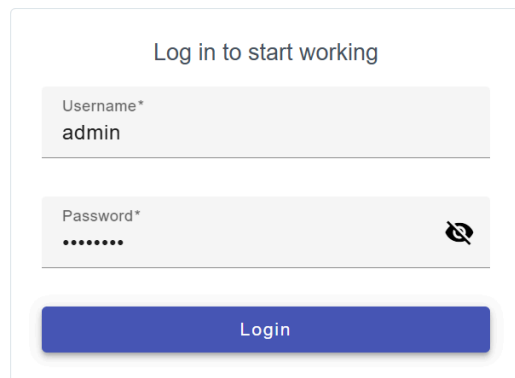
4 ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Вимоги до функціональних характеристик

Програмне забезпечення повинно забезпечувати виконання наступних основних функцій:

4.1.1 Користувацького інтерфейсу

- Авторизація користувача за email та паролем.



Log in to start working

Username*
admin

Password*
.....

Login

Рисунок 4.1 – Авторизація користувача

- Перегляд списку завантажених документів.



Рисунок 4.2 – Сторінка користувача

- Завантаження нового документа у форматі PDF.

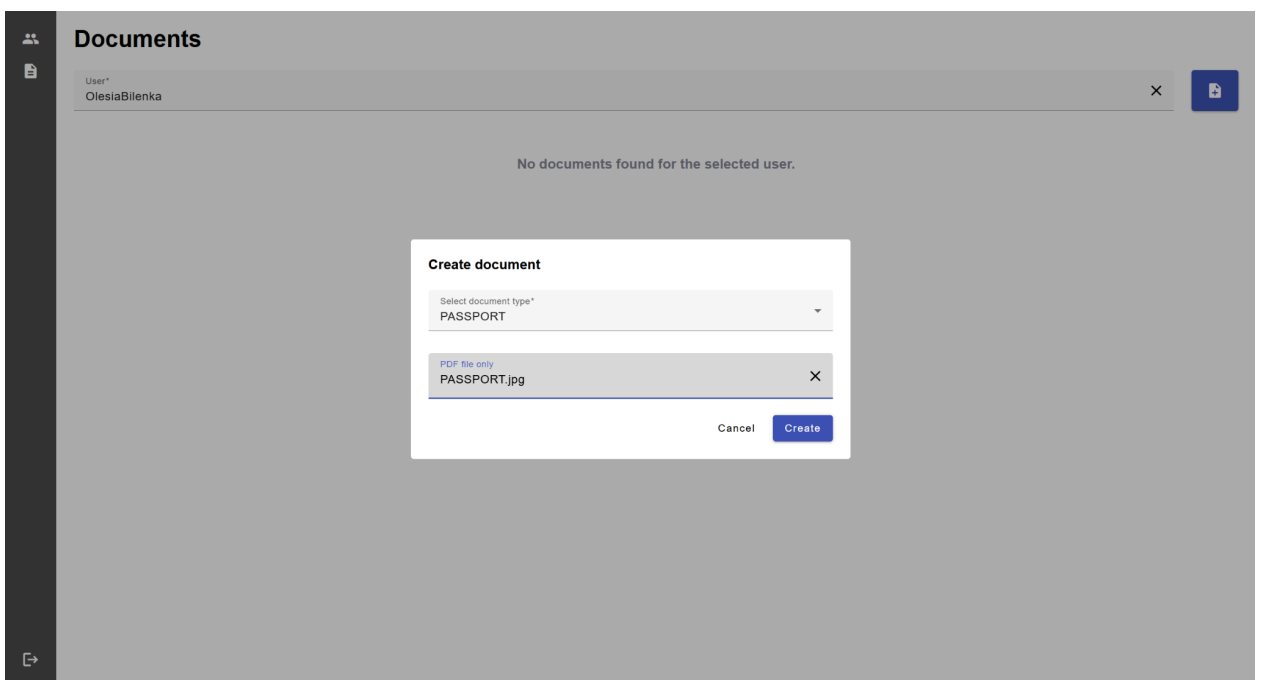


Рисунок 4.3 – Форма завантаження документа

- Перевірка статусу документа (верифіковано / не верифіковано).

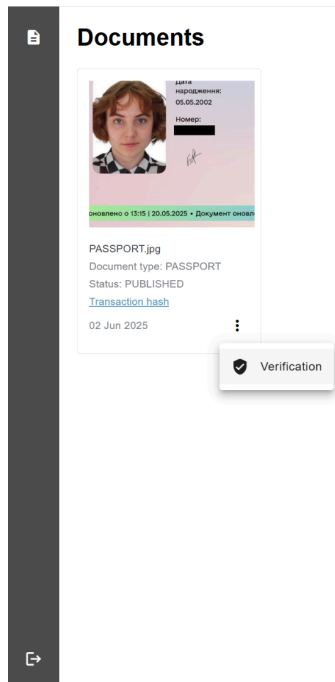


Рисунок 4.4 – Кнопка “Верифікувати документ”

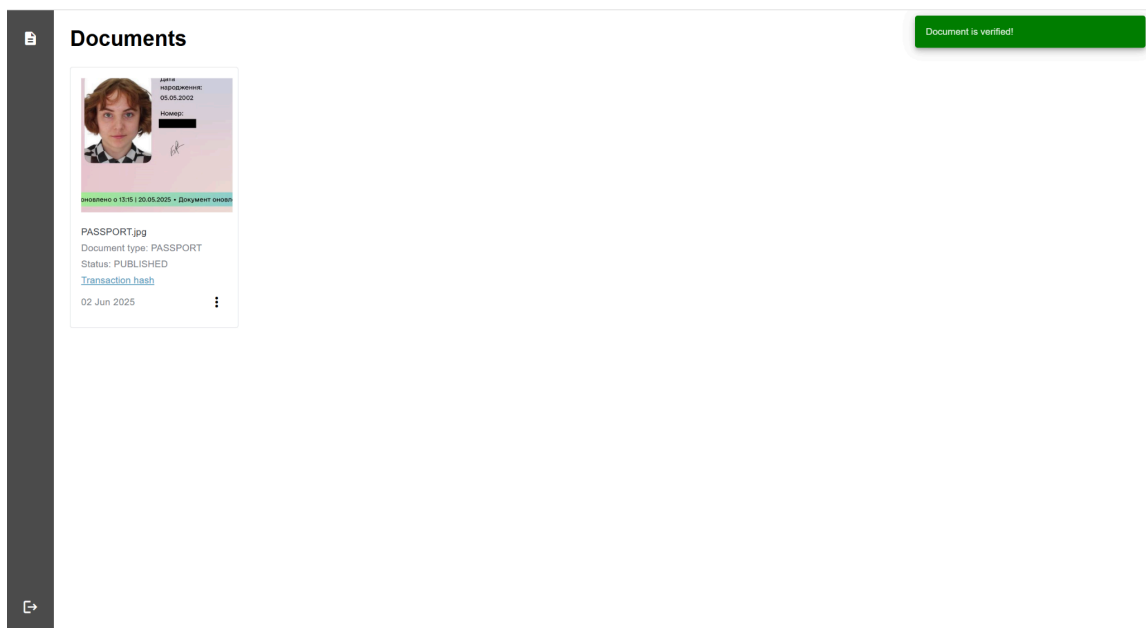


Рисунок 4.5 – Виведення повідомлення про успішну верифікацію документа

- Перехід до транзакції у блокчейн-експлорері.

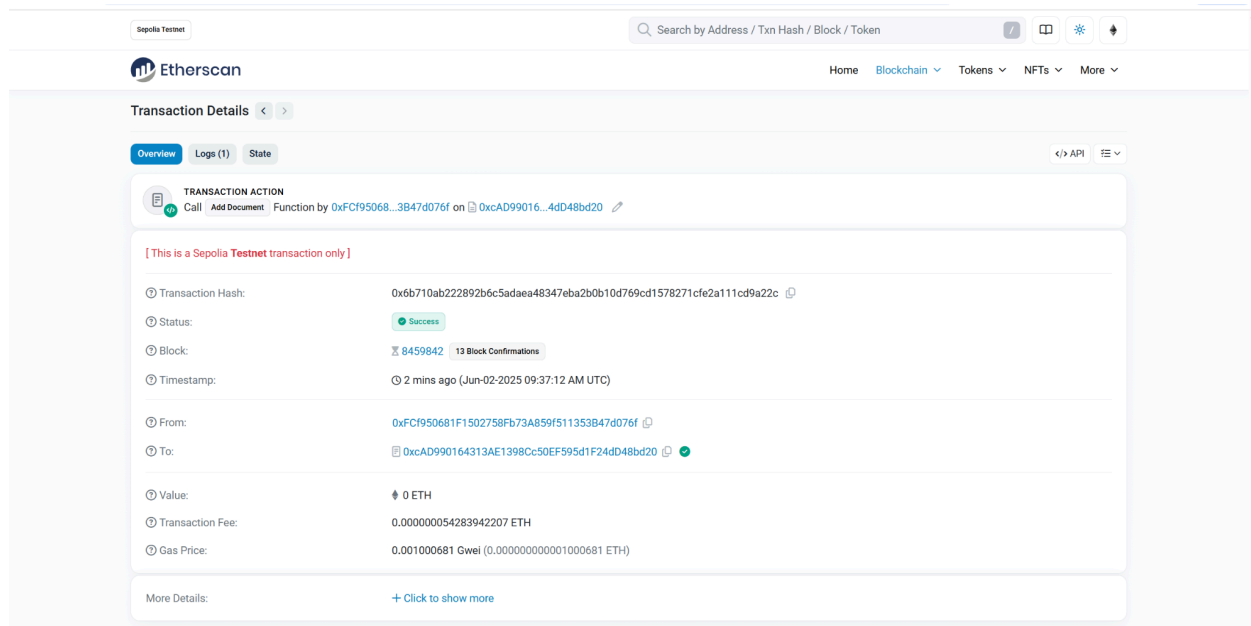


Рисунок 4.6 – Перегляд транзакції запису документу в блокчейн

4.1.2 Облік користувачів:

4.1.2.1 Для користувача:

- Авторизація у системі.
- Перегляд власного профілю.

4.1.2.2 Для адміністратора системи:

- Перегляд та керування всіма користувачами системи.

4.1.2.3 Для модератора системи:

- Перегляд та керування всіма документами системи.

4.2 Вимоги до надійності

Програмне забезпечення повинно передбачати контроль введення інформації на всіх етапах взаємодії користувача з системою. Зокрема:

- Перевірка формату електронної пошти під час реєстрації та входу в систему.
- Перевірка відповідності завантажених файлів (допускаються лише документи у форматі PDF).

- Валідація заповнення обов'язкових полів перед надсиланням форм.
- Повідомлення користувачів про помилки з поясненням причин (наприклад, «Файл не відповідає вимогам» або «Невірний формат email»).
- Заборона дій, які не дозволені відповідно до ролі (наприклад, користувач не може завантажити документ іншому користувачу)

Забезпечення цілісності інформації в базі даних.

Для забезпечення цілісності інформації в базі даних реалізовані такі механізми:

- Використання транзакцій для критичних операцій (наприклад, створення документа з прив'язкою до користувача та записом транзакції з блокчейну).
- Перевірка унікальності збереженого хешу документа, щоб уникнути дублювання.
- Обмеження на каскадне видалення важливих сутностей (наприклад, документи не можуть бути видалені без явного підтвердження та перевірки прав доступу).
- Використання зовнішніх ключів (foreign keys) у схемі БД для зв'язку між користувачами та документами.

4.3 Умови експлуатації

Умови експлуатації згідно СанПін 2.2.2.542 – 96.

4.3.1 Вид обслуговування

Вимоги до виду обслуговування не висуваються.

4.3.2 Обслуговуючий персонал

Оновлення програмного забезпечення, у разі потреби, може здійснюватися адміністратором системи шляхом повторного розгортання відповідних сервісів за допомогою Docker Compose.

Вимоги до обслуговуючого персоналу не висуваються.

У разі розширення функціональності або зміни конфігурації системи (наприклад, перемикання на інший блокчейн-провайдер або міграція бази даних), необхідна наявність фахівця з базовими знаннями у сфері DevOps, Docker та основ Web3-інтеграції.

4.4 Вимоги до складу і параметрів технічних засобів

Програмне забезпечення повинно функціонувати на IBM-сумісних персональних комп'ютерах під управлінням операційної системи Windows, Linux або macOS з підтримкою Docker та сучасних веббраузерів (Google Chrome, Mozilla Firefox, Edge).

Мінімальна конфігурація технічних засобів:

- Тип процесора: Intel Core i5 (4 ядра, 2.4 ГГц або аналогічний AMD);
- Об'єм оперативної пам'яті (ОЗП): 4 ГБ;
- Об'єм вільного місця на жорсткому диску: не менше 10 ГБ;
- Підключення до мережі Інтернет зі швидкістю не менше 20 Мбіт/с;
- Встановлене середовище виконання Docker та Docker Compose;
- Встановлений сучасний веббраузер з підтримкою JavaScript та WebAssembly.

Рекомендована конфігурація технічних засобів:

- Тип процесора: Intel Core i7 (6 або більше ядер, 3.0 ГГц і вище);
- Об'єм оперативної пам'яті (ОЗП): 8 ГБ або більше;
- Об'єм вільного місця на диску: 20 ГБ;
- Підключення до мережі Інтернет зі швидкістю від 100 Мбіт/с;
- ОС: Ubuntu 22.04 LTS / Windows 10 / macOS 13 Ventura;
- Docker Engine v24+, Docker Compose v2.20+;
- Браузер: Google Chrome (v120+);
- Роздільна здатність екрану: не менше 1366×768 пікселів.

4.5 Вимоги до інформаційної та програмної сумісності

Програмне забезпечення повинно бути сумісним з сучасними операційними системами та стандартними інструментами розробки й розгортання.

Система розроблена з урахуванням кросплатформенності та повинна коректно функціонувати під управлінням наступних операційних систем:

- Операційні системи сімейства Windows (Windows 10, Windows 11);
- Сучасні дистрибутиви Linux (Ubuntu 20.04+, Debian, Arch, Fedora тощо);
- Операційні системи macOS (версії 12 Monterey і вище).

Для роботи програмного забезпечення необхідна наявність таких платформ та середовищ:

- Docker та Docker Compose для розгортання мікросервісної архітектури;
- Node.js (для фронтенду);
- Java 17+ (для бекенд-сервісів);
- PostgreSQL 13+ (як основна СУБД);
- Kafka (Apache Kafka 3.0+) — як транспортний рівень взаємодії між сервісами;
- Web3-провайдер, сумісний з EVM-блокчейнами (наприклад, Infura або Alchemy).

Програмне забезпечення використовує відкриті формати для обміну інформацією (JSON, HTTPS, Web3), що забезпечує інформаційну сумісність з іншими системами при необхідності інтеграції.

4.5.1 Вимоги до вхідних даних

Вхідні дані повинні бути представлені у наступному форматі:

- Документи — файли у форматі PDF, розміром не більше 10 МБ, що завантажуються модератором у систему.
- Користувацькі дані — текстові поля (email, ім'я), що вводяться через вебінтерфейс.
- Хеш документа — 32-байтове значення у форматі SHA-256, що генерується з PDF-файлу.
- Kafka-повідомлення — у форматі JSON, містять ідентифікатор користувача, хеш документа, тип операції та мітку часу.

4.5.2 Вимоги до вихідних даних

Результати повинні бути представлені у наступному форматі:

- Відповіді API — у форматі JSON, із полями status, data, error, що описують результат виконання запиту.
- Хеші транзакцій — у форматі hex-рядка (наприклад, 0x...), які повертаються після успішного запису до блокчейну.
- Статуси документів — як рядкові значення: pending, verified, failed.
- Вебінтерфейс — відображає результат перевірки документа, його статус, час реєстрації та посилання на відповідну транзакцію у блокчейні.

4.5.3 Вимоги до мови розробки

Розробку виконано з використанням наступних мов програмування:

- Java 17 — для реалізації бекенд-сервісу (core-service, blockchain-service).
- TypeScript — для реалізації фронтенд-частини на Angular.
- Solidity — для реалізації смарт-контрактів на платформі Ethereum.

4.5.4 Вимоги до середовища розробки

Розробку виконано з використанням наступних інструментів та середовищ:

- Spring Boot 3.x — для створення REST API та логіки бекенду.
- Angular 16+ — як фронтенд-фреймворк.
- Docker / Docker Compose — для контейнеризації всіх сервісів.
- Visual Studio Code — як основна IDE.
- Apache Kafka — як брокер повідомлень.
- Hardhat — для розробки, тестування та деплою смарт-контрактів.

4.5.5 Вимоги до представлення вихідних кодів

Вихідний код програми має бути представлений у вигляді: структурованого проекту (Git), з окремими репозиторіями:

- backend-service – основна логіка бекенду
- blockchain-service – взаємодія з блокчейном
- frontend – Angular застосунок
- contracts – Solidity-контракти

Забезпечено наявність README-файлів, скриптів запуску, .env.example для налаштувань.

4.6 Вимоги до маркування та пакування

Вимоги до маркування та пакування не висуваються.

4.7 Вимоги до транспортування та зберігання

Вимоги до транспортування та зберігання не висуваються.

4.8 Спеціальні вимоги

Відсутні.

Пояснювальна записка до дипломного проєкту

на тему: **Програмне забезпечення для збереження та валідації документів
на основі смарт-контрактів в блокчейнах**

КП.ІП-31101.045470.02.81

Київ – 2025

Зміст

1	ПЕРЕДПРОЄКТНЕ ОБСТЕЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ	6
1.1	Постановка завдання дипломного проєктування	6
1.2	Аналіз галузі та технологічного контексту зберігання електронних документів із використанням блокчейну	7
1.3	Аналіз існуючих рішень	8
1.3.1	Огляд існуючих централізованих і децентралізованих рішень для зберігання та перевірки електронних документів	8
1.3.2	Аналіз криптографічних, автентифікаційних та архітектурних рішень для побудови захищеної системи верифікації документів	13
1.4	Аналіз функціональних сценаріїв та моделювання процесів верифікації електронних документів	17
2	РОЗРОБЛЕННЯ ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	30
2.1	Варіанти використання програмного забезпечення для верифікації документів	30
2.2	Розроблення функціональних вимог	37
2.3	Розроблення нефункціональних вимог	39
2.4	Аналіз економічних показників програмного забезпечення	41
2.5	Постановка завдання на розробку програмного забезпечення	42
3	КОНСТРУЮВАННЯ ТА РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	45
3.1	Архітектура програмного забезпечення	45
3.2	Архітектурні рішення та обґрунтування вибору засобів розробки	48
3.3	Конструювання програмного забезпечення	51
3.3.1	Опис структури смарт контрактів	55
3.4	Аналіз безпеки даних оф-чейн та он-чейн модулів	63
4	АНАЛІЗ ЯКОСТІ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	68
4.1	Аналіз якості ПЗ	68
4.2	Опис процесів тестування	69
4.3	Опис контрольного прикладу	73
	Висновки до розділу	80
5	РОЗГОРТАННЯ ТА СУПРОВІД ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	82

	3
5.1 Розгортання програмного забезпечення	82
5.2 Супровід програмного забезпечення	83
Висновки до розділу	83
ВИСНОВКИ	84
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	86
ДОДАТКИ	89

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

IDE	- Integrated Development Environment – інтегроване середовище розробки
API	- Application Programming Interface – прикладний програмний інтерфейс
SDK	- Software Development Kit – набір інструментів розробника
IT	- Information Technology – інформаційні технології
ER	- Entity-Relationship Diagram – діаграма зв'язків між сутностями
OC	- Операційна система
БД	- База даних
JWT	- JSON Web Token – формат токена для аутентифікації
UUPS	- Universal Upgradeable Proxy Standard – проксі-патерн для оновлюваних контрактів
EVM	- Ethereum Virtual Machine – віртуальна машина Ethereum
VS Code	- Visual Studio Code – текстовий редактор та IDE від Microsoft
DoS	- Denial of Service – атака на відмову в обслуговуванні
MITM	- Man-in-the-Middle – атака "людина посередині"
Kafka	- Apache Kafka – система обміну повідомленнями
HTTP	- Hypertext Transfer Protocol – протокол передачі гіпертексту
ICMP	- Internet Control Message Protocol – протокол перевірки доступності
Multisig	- Multi-signature wallet – гаманець з мультипідписом
DAO	- Decentralized Autonomous Organization – децентралізована автономна організація
UUID	- Universally Unique Identifier – універсальний унікальний ідентифікатор

Вступ

У сучасних умовах цифрової трансформації державних та комерційних сервісів особливу актуальність набувають системи, що забезпечують надійне зберігання та перевірку достовірності цифрових документів [1]. Втрата контролю над автентичністю файлів, а також відсутність публічної прозорості процесів перевірки, призводить до зниження довіри до цифрових сервісів як з боку користувачів, так і з боку регуляторів.

Одним з найперспективніших напрямків вирішення цієї проблеми є використання технологій блокчейн [2]. Провідні інституції, як-от MIT, IBM Research, Ethereum Foundation та численні стартапи в галузі Web3, активно розробляють децентралізовані системи для зберігання доказів, записів транзакцій та реєстрації авторства, що гарантує незмінність інформації та прозорість перевірки [3]. У сфері документообігу з'являються рішення, як-от OpenCerts [4], Dock [5], Kilt Protocol [6], які реалізують ідеї децентралізованої ідентичності та перевірки походження документів. Однак більшість таких рішень або є вузькоспеціалізованими, або потребують значних ресурсів для інтеграції та підтримки.

1 ПЕРЕДПРОЄКТНЕ ОБСТЕЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Постановка завдання дипломного проєктування

У рамках даного дипломного проєкту повинно бути розроблено веб-застосунок, який дозволяє завантажувати, зберігати, верифікувати та перевіряти автентичність документів користувачів за допомогою Ethereum-смарт-контрактів [7]. Для забезпечення гнучкості та безпечного оновлення логіки контракту реалізовано UUPS-проксі [8] з керуванням через мультипідписний гаманець [9]. Передача подій між компонентами системи відбувається через Kafka [10], а бізнес-логіка реалізована на базі Spring Boot [11]. Клієнтська частина створена за допомогою Angular [12] з підтримкою JWT-аутентифікації.

Система розрахована на організації або сервіси, які мають потребу:

- у прозорому зберіганні цифрових слідів документів;
- у верифікації даних з відкритого блокчейну;
- у простому веб-інтерфейсі для адміністрування документів користувачів.

Мета даного проєкту: Підвищення безпеки, прозорості та ефективності процесу зберігання та валідації цифрових документів шляхом автоматизації управління їхньою автентичністю за допомогою смарт-контрактів у блокчейні EVM, асинхронної комунікації через Kafka та мікросервісної архітектури на Spring Boot.

Система має забезпечити:

Автоматизацію перевірки автентичності документів шляхом використання хешування та збереження у блокчейні.

Прозорий контроль доступу до документів через чітку рольову модель (адміністратор, модератор, користувач).

Гнучкість і масштабованість за рахунок мікросервісної архітектури.

Надійний обмін даними між сервісами через подієву модель на основі Apache Kafka.

Оптимізацію розгортання та адміністрування завдяки контейнеризації (Docker, Docker Compose).

Система спрямована на мінімізацію ризиків фальсифікації документів та створення захищеного, масштабованого і зручного рішення для роботи з цифровими документами.

1.2 Аналіз галузі та технологічного контексту зберігання електронних документів із використанням блокчейну

Зберігання та перевірка достовірності електронних документів стає все більш актуальним завданням в умовах цифрової трансформації бізнесу, державних структур і освіти. З переходом процесів в онлайн зростає потреба в системах, які гарантують автентичність, цілісність і незмінність документів без необхідності довіряти централізованим операторам.

У традиційних рішеннях (Google Drive, Dropbox тощо) існує ризик втрати контролю: оператор системи або зловмисник може змінити чи видалити документ. Це створює критичні загрози, особливо для юридичних чи освітніх документів.

Альтернативою є децентралізовані рішення на базі блокчейн-технологій, які дозволяють зберігати незмінні записи — зокрема хеші документів, що виступають унікальними цифровими відбитками. Зміна хоча б одного байта документа призведе до зміни хешу, тому перевірка достовірності проста та надійна.

Найчастіше для цього використовують смарт-контракти, які приймають хеш, ID користувача, тип документа тощо. Це дозволяє не лише фіксувати факт існування документа, а й реалізовувати контроль доступу, перевірку та оновлення.

Однак використання лише смарт-контрактів обмежене — взаємодія з ними без зручного бекенду та інтерфейсу потребує технічних знань. Тому важливим є створення повноцінного веб-застосунку з підтримкою REST API, зручного frontend'у та інтеграції з blockchain-сервісом.

Також необхідно враховувати юридичний контекст: блокчейн гарантує незмінність, але без юридичної прив'язки (електронний підпис, ідентифікація) запис у мережі сам по собі не має правової сили.

Висновок: існує потреба в простому, безпечному веб-рішенні, яке:

- хешує документи на клієнтській стороні;
- записує хеші в блокчейн;
- надає REST API для взаємодії;
- дозволяє зручно перевіряти достовірність документів.

Таке рішення поєднує надійність блокчейну з доступністю класичного веб-застосунку і є ефективним інструментом для організацій, що цінують прозорість і контроль над цифровими документами.

1.3 Аналіз існуючих рішень

Проаналізуємо відоме на сьогодні алгоритмічне забезпечення у даній області та технічні рішення, що допоможуть у реалізації Програмного забезпечення для збереження та валідації документів на основі смарт-контрактів в блокчейнах. Далі будуть розглянуті готові програмні рішення, допоміжні програмні засоби та засоби розробки.

1.3.1 Огляд існуючих централізованих і децентралізованих рішень для зберігання та перевірки електронних документів

У сучасному цифровому середовищі існує багато рішень, які частково реалізують функціонал, пов'язаний зі зберіганням документів, їхньою перевіркою та захистом від підробки. Проте більшість із них мають суттєві обмеження у контексті забезпечення прозорості, незмінності даних або

доступності інтерфейсів для звичайних користувачів. Нижче наведено приклади двох таких систем: одна реалізує зберігання документів без блокчейну, а інша — на основі блокчейну, але без зручного інтерфейсу.

1.3.1.1 Google Drive

Google Drive є одним із найпоширеніших сервісів для зберігання та обміну документами в хмарі [13]. Сервіс надає зручний інтерфейс для завантаження, спільного доступу, редагування та керування файлами, що робить його популярним серед як індивідуальних користувачів, так і організацій.

Однак Google Drive є централізованим рішенням. Це означає, що повний контроль над даними має провайдер сервісу. Попри логування змін та систему відновлення версій, відсутній механізм, який би гарантував незмінність документа з моменту його створення. Користувач не має можливості перевірити, чи файл не було змінено кимось іншим без його відома [14], а також не може довести, що певний файл існував у певний момент часу без залучення самого Google як третьої сторони.

Це обмеження є критичним для сценаріїв, де потрібна достовірність, незмінність та самостійна перевірка без посередника — наприклад, у юридичних або державних процесах, що вимагають прозорості.

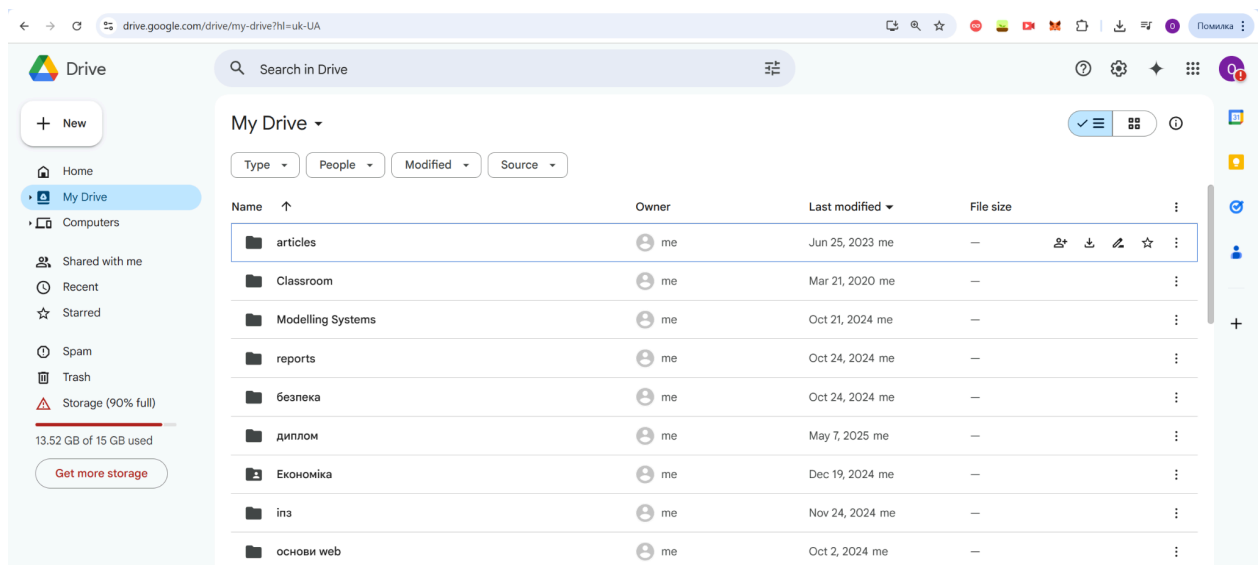


Рисунок 1.1 – Приклад користувацької панелі Google Drive

1.3.1.2 Smart Contract Storage (Ethereum Remix) — збереження хешу в блокчейні без UI

З іншого боку, існують блокчейн-рішення, які забезпечують високу довіру до збережених даних за рахунок децентралізації та незмінності записів. Наприклад, можна створити простий смарт-контракт в Ethereum, який записує хеш документа у блокчейн [15]. Для цього часто використовують онлайн-інструмент Remix IDE [16], який дозволяє розгорнути контракт і взаємодіяти з ним напряму через консоль.

Таке рішення повністю вирішує проблему прозорості та достовірності даних: будь-хто може перевірити, що певний хеш збережено в мережі, і підтвердити автентичність документа незалежно від централізованих сервісів. Однак воно має суттєвий недолік - відсутність зручного користувацького інтерфейсу. Для того, щоб додати або перевірити запис, потрібно вручну вводити параметри в інтерфейс Remix IDE, розуміти формат Solidity-структур, мати підключений гаманець MetaMask і сплачувати комісію за транзакцію.

Таке рішення фактично недоступне для користувачів без технічної підготовки та не може бути інтегроване в реальні бізнес-процеси без додаткового backend-/frontend-шару.

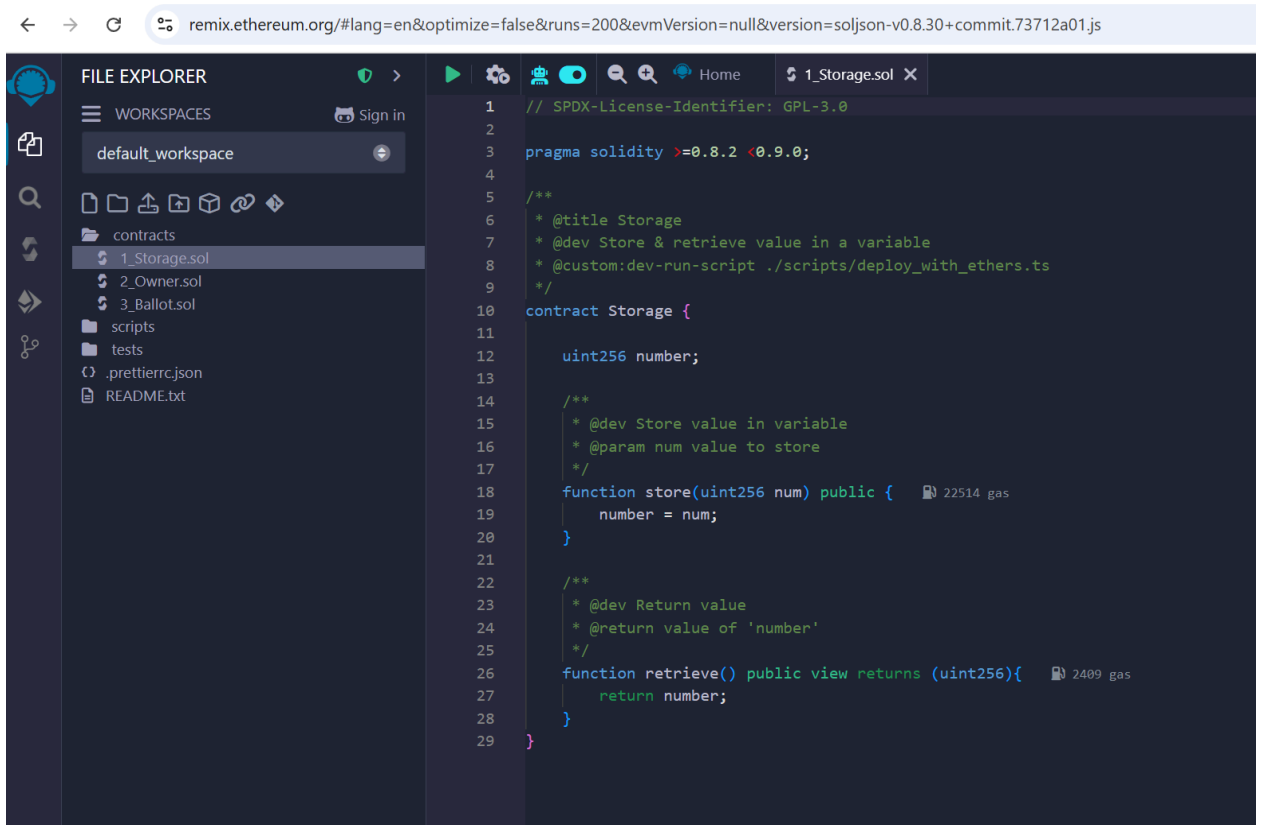


Рисунок 1.2 – Приклад користування Remix IDE

Для порівняння проєкту з аналогом можна скористатись таблицею 1.3.

Таблиця 1.1 – Порівняння з аналогами

Функціонал	IsItReal app	Google Drive	SC Storage	Пояснення
Збереження документа / хешу	+	+	+	Можливість зберігати файл або його хеш для подальшої перевірки
Гарантія незмінності даних	+	-	+	Забезпечення того, що дані не можуть бути змінені після збереження

Продовження таблиці 1.1

Функціонал	IsItReal app	Google Drive	SC Storage	Пояснення
Прозорість (зовнішня перевірка)	+	-	+	Можливість перевірити дані (наприклад, хеш документа) незалежними особами
Наявність фронтенду	+	+	-	Наявність зручного веб-інтерфейсу для взаємодії з системою
Наявність бекенду	+	+	-	Серверна логіка обробки запитів, перевірки прав, взаємодії з БД або блокчейном
Простота для кінцевого користувача	+	+	-	Мінімальні технічні знання для роботи з системою

Продовження таблиці 1.1

REST API для інтеграції	+	+	-	Можливість інтегрувати систему з іншими сервісами через HTTP API
Підтримка MetaMask / Web3	+	-	+	Підключення гаманця користувача для підпису транзакцій у блокчейні
Перевірка автентичнос ті документа	+	-	+	Зіставлення хешу документа з тим, що збережений у реєстрі
Можливість видалення документа	+	+	-	Функціонал для видалення документа або його запису із системи
Можливість оновлення запису	+	+	-	Підтримка зміни вже збереженої інформації при повторному завантаженні

1.3.2 Аналіз криптографічних, автентифікаційних та архітектурних рішень для побудови захищеної системи верифікації документів

Розробка системи зберігання та перевірки автентичності документів потребує як ефективних алгоритмічних підходів, так і стабільних технічних рішень. У даному підрозділі розглянуто існуючі алгоритми хешування, моделі автентифікації, архітектурні шаблони для веб-застосунків та технології, що можуть бути використані в реалізації системи.

У контексті зберігання цифрових відбитків документів важливим є вибір хеш-функції [17]. Хеш-функція повинна бути односторонньою (невідновлюваною), стійкою до колізій та швидкою в обчисленні.

Таблиця 1.2 - Найбільш поширені алгоритми:

Алгоритм	Довжина хешу	Швидкість	Захист від колізій	Коментар
MD5	128 біт	дуже висока	слабка	не рекомендується для безпечних застосувань
SHA-1	160 біт	висока	слабка	застарілий, зламаний у 2017 році
SHA-256	256 біт	помірна	висока	стандарт у блокчейні Ethereum
SHA-3	256/512 біт	помірна	висока	новіший стандарт з покращеною стійкістю

У проєкті буде використано SHA-256, оскільки він є офіційним стандартом у більшості блокчейн-систем, зокрема в Ethereum [18], і має хороше співвідношення між продуктивністю та безпекою.

У системі реалізовано автентифікацію на основі токена (JWT – JSON Web Token) [19]. Даний підхід є зручним для побудови REST API і підтримує розмежування доступу до документів.

Таблиця 1.3 - Варіанти автентифікації:

Підхід	Переваги	Недоліки
Сесійна автентифікація	Простий у реалізації	Погано масштабується
OAuth2	Підтримка авторизації через сторонні сервіси	Ускладнена реалізація
JWT (вибраний)	Простий, масштабований	Потребує захисту токенів

JWT було обрано як найкраще рішення для захисту API-запитів у сучасній веб-архітектурі.

У рамках проєкту було проаналізовано декілька популярних блокчейн-платформ [20], які потенційно можуть використовуватись для зберігання хешів документів та реалізації логіки смарт-контрактів. Основними критеріями вибору були: надійність, наявність інструментів розробки, доступність тестової мережі, підтримка контрактів із ролями та оновленнями, а також рівень підтримки в екосистемі.

Таблиця 1.4 - Варіанти блокчейнів:

Критерій	EVM (Ethereum)	Solana	NEAR Protocol	Пояснення вибору
Мова смарт-контрактів	Solidity	Rust / C	Rust / AssemblyScript	Solidity поширеніша для смарт-контрактів та краще документована
Доступність документації	дуже висока	обмежена	менш поширена	EVM має велику базу знань

Продовження таблиці 1.4

Критерій	EVM (Ethereum)	Solana	NEAR Protocol	Пояснення вибору
Інструменти розробки (Remix, Truffle, Hardhat)	повна підтримка	часткова підтримка	слабка підтримка	EVM має зрілу екосистему
Підтримка проксі-контрактів (UUPS)	повна підтримка	не підтримує	не підтримує	UUPS можливий тільки в EVM
Web3 бібліотеки (web3.js, ethers.js, Web3j)	стандарт де-факто	часткова	обмежена	Підключення до EVM простіше
Готові рішення (OpenZeppelin, Gnosis Safe)	доступні	відсутні	відсутні	Підвищує безпеку та швидкість розробки

В результаті аналізу було обрано Ethereum Virtual Machine (EVM)-сумісну платформу - а саме Ethereum Sepolia [21]. Цей вибір обумовлений низкою технічних та екосистемних переваг, які роблять EVM оптимальним варіантом для дипломного проєкту.

Для того щоб смарт-контракт можна було оновлювати без втрати даних, у потрібно використати проксі патерн [22].

Таблиця 1.5 - Порівняння проксі-патернів:

Критерій	UUPS proxy	Transparent proxy	Пояснення вибору
Оновлення логіки	реалізує контракт	через проксі-адмін	UUPS гнучкіший

Продовження таблиці 1.5

Критерій	UUPS proxy	Transparent proxy	Пояснення вибору
Ефективність по gas	нижча	вища	UUPS не виконує перевірки ролі щоразу
Простота для майбутнього розширення	висока	обмежена	Нові функції додаються легко в обох
Безпека (у поєднанні з Multisig)	висока	висока	В обох схемах безпека залежить від контролю оновлення
Контроль upgrade-функцій	в логіці	в проксі	Менше довіри до зовнішнього проксі-контракту

UUPS було обрано, оскільки він краще масштабується, споживає менше gas [23], забезпечує чіткий контроль над оновленням логіки.

Система реалізується у вигляді мікросервісної архітектури [24], де основні компоненти поділені на:

- а) Frontend – Angular застосунок для взаємодії з користувачем;
- б) Backend – Spring Boot REST API;
- в) Blockchain Service – окремий сервіс на Java, що взаємодіє зі смарт-контрактом через Web3j [25];
- г) Смарт-контракти – Solidity-контракти, що зберігають хеші документів у мережі Sepolia (Ethereum testnet):

- 1) Основний смарт контракт, який є центральним компонентом системи. Його основна задача - надійне та незмінне зберігання криптографічних хешів документів, прив'язаних до користувача та типу документа.

- 2) Multisig-гаманець - 3 метою безпеки, контроль над чутливими операціями (наприклад, оновлення логіки контракту або зміна ролей доступу) передано на мультисиг-гаманець [26]. Це дозволяє уникнути централізації прав та мінімізувати ризики у разі втрати/компрометації одного приватного ключа.
- 3) UUPS проху: оновлюваність смарт-контракту: для того щоб смарт-контракт можна було оновлювати без втрати даних, у проєкті використано UUPS (Universal Upgradeable Proxy Standard) проху pattern. На відміну від класичного TransparentProxy, UUPS має простішу реалізацію і делегує логіку оновлення саме реалізаційному контракту, а не проксі.

1.4 Аналіз функціональних сценаріїв та моделювання процесів верифікації електронних документів

Для опису бізнес процесу використовується BPMN модель основні процеси зображені на рисунках 1.3 – 1.8.

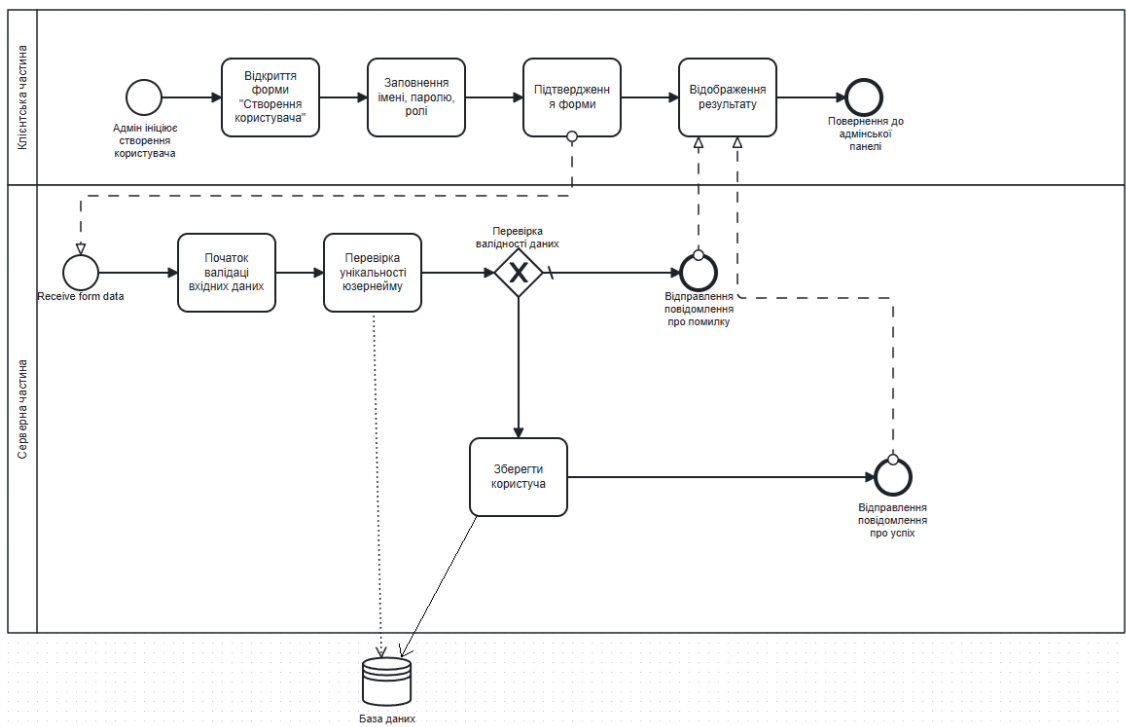


Рисунок 1.3 – Схема створення користувача

Опис послідовності процесу створення користувача:

Клієнтська частина (Frontend):

- Адміністратор ініціює створення користувача - відкриває відповідну секцію в інтерфейсі.
- Відкриття форми “Створення користувача” - завантажується форма для введення даних.
- Заповнення імені, паролю, ролі - адміністратор вводить необхідні поля.
- Підтвердження форми - дані надсилаються на сервер.
- Відображення результату - користувач отримує повідомлення про успішне створення або помилку.
- Повернення до адміністративної панелі — після перегляду результату.

Серверна частина (Backend):

- Receive form data — сервер отримує POST-запит з даними форми.
- Початок валідації вхідних даних — перевіряється коректність полів (наприклад, непусті значення, формат паролю тощо).
- Перевірка унікальності юзернейму — запит до бази даних для перевірки, чи вже існує користувач із таким ім'ям.
-

Перевірка валідності даних (gateway) — точка розгалуження:

- Невалідні дані: сервер формує та відправляє повідомлення про помилку. Повідомлення надходить на клієнт для відображення.
- Дані валідні: збереження користувача в базу даних. Відправлення повідомлення про успіх клієнту.

База даних:

Приймає два запити:

- Читання: перевірка наявності імені користувача.
- Запис: збереження нового користувача після проходження валідації.

Результат:

- Якщо все виконано правильно - новий користувач створюється.
- У разі помилки - користувач отримує відповідне повідомлення.

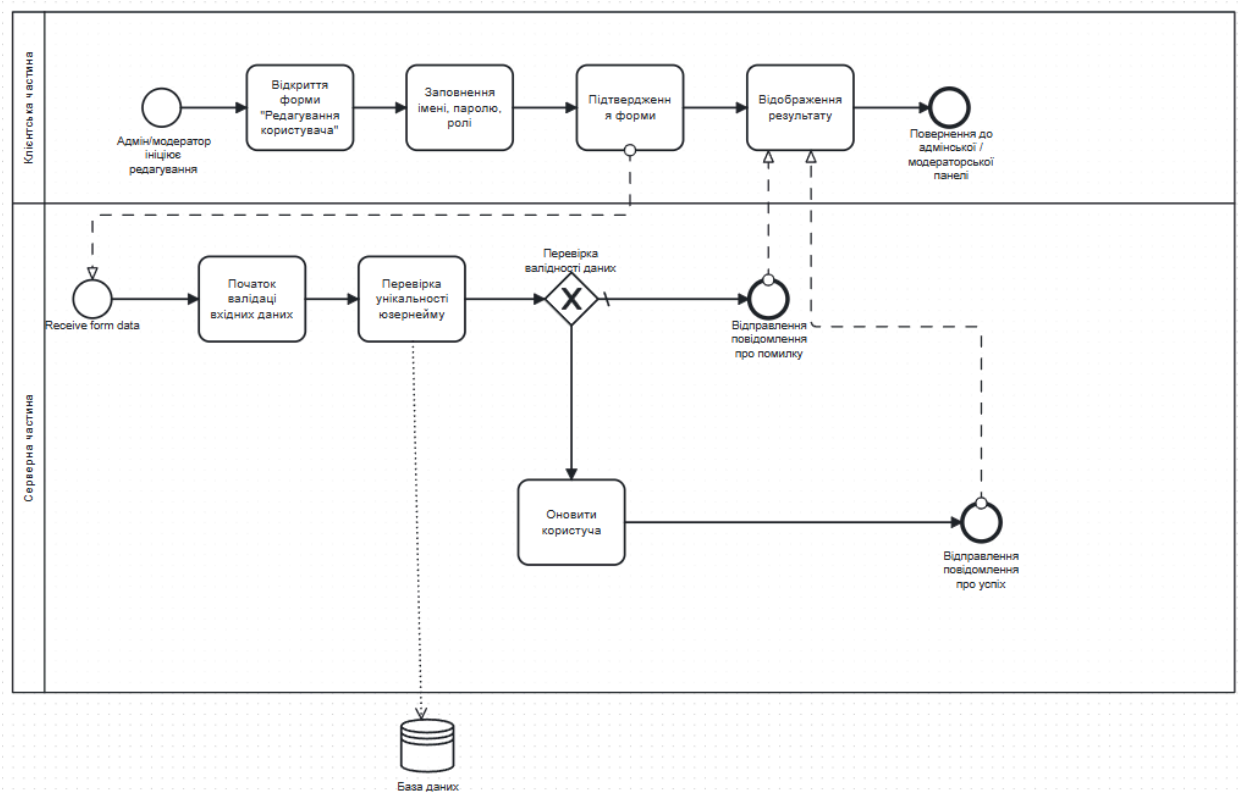


Рисунок 1.4 – Схема редагування користувача

Опис послідовності процесу "Редагування користувача"

Клієнтська частина (Frontend):

- Адміністратор або модератор ініціює редагування користувача - переходить до відповідного розділу.
- Відкривається форма "Редагування користувача".
- Введення нових значень полів: імені, паролю, ролі.
- Підтвердження форми - відправлення даних на сервер.
- Відображення результату: успішного редагування або помилки.
- Повернення до адміністративної/модераторської панелі.

Серверна частина (Backend):

- Сервер приймає оновлені дані з форми.

- Початок валідації вхідних даних - перевіряється правильність формату, непорожні значення тощо.
- Перевірка унікальності юзернейму - щоб уникнути конфліктів.

Перевірка валідності даних (gateway):

- Якщо валідація не пройдена: сервер надсилає повідомлення про помилку назад на клієнт.
- Якщо все коректно: оновлення даних користувача в базі. Надсилання повідомлення про успіх.

База даних:

- Читається: для перевірки унікальності імені.
- Оновлюється: при успішній валідації для збереження змін користувача.

Результат:

- Якщо дані валідні й ім'я не дублюється - користувач оновлюється.
- У випадку проблем - користувач бачить повідомлення про помилку.

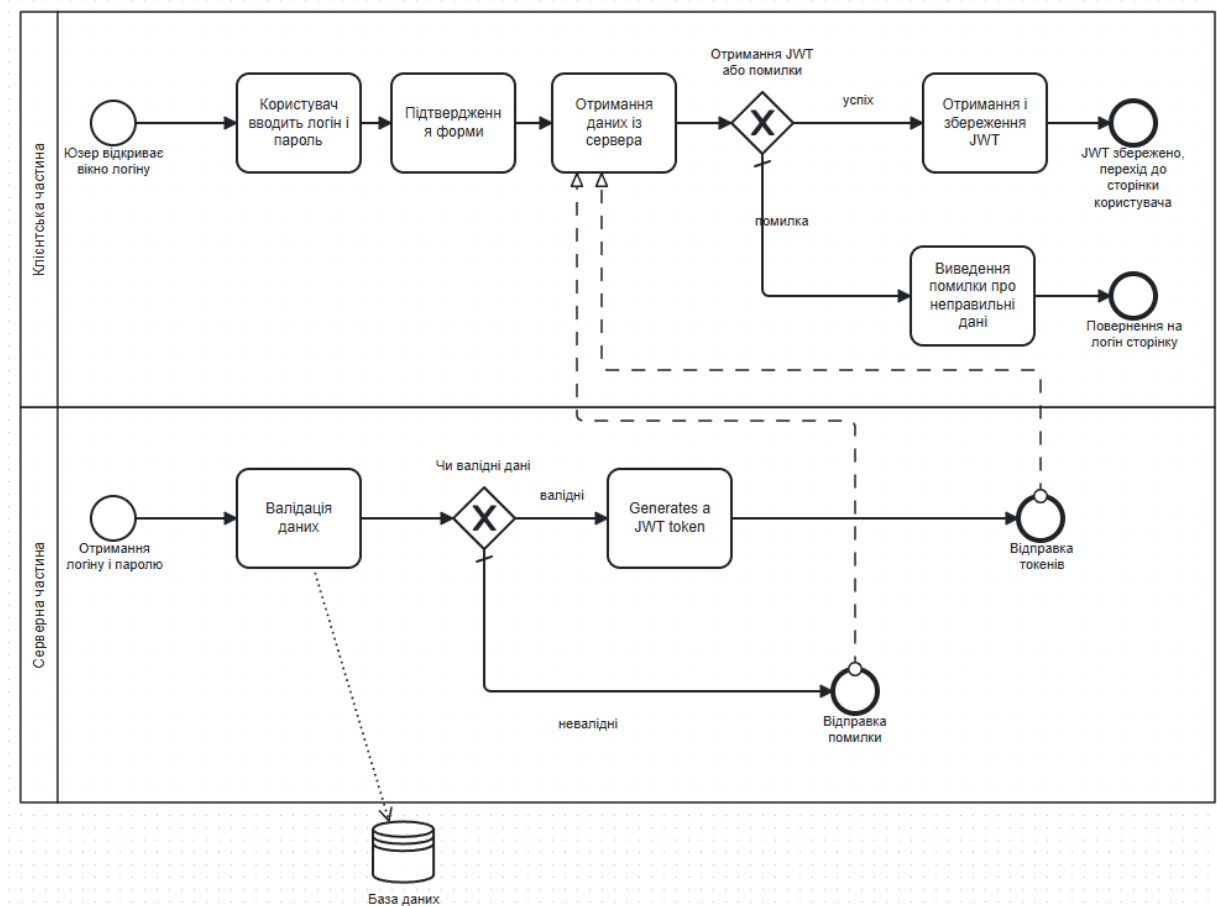


Рисунок 1.5 – Схема авторизації

Опис послідовності процесу "Авторизація користувача з JWT"

Клієнтська частина (Frontend):

- Користувач відкриває вікно логіну.
- Вводить логін і пароль.
- Підтверджує форму входу (натискає "Увійти").
- Очікує відповідь від сервера - ця відповідь може містити JWT або помилку.

Результат перевірки:

- Якщо успіх: JWT збережено на клієнті. Користувача перенаправлено на сторінку профілю/кабінету.
- Якщо помилка: Виводиться повідомлення про неправильні дані. Залишається/повертається на сторінку логіну.

Серверна частина (Backend):

- Отримує логін і пароль від клієнта.
- Виконує валідацію даних:
- Перевіряє, чи існує такий користувач у базі даних.
- Зіставляє пароль.

Умовне розгалуження:

- Якщо дані невалідні: надсилається повідомлення про помилку назад клієнту.
- Якщо дані валідні: генерується JWT токен. Токен надсилається клієнту у відповіді.

База даних:

- Перевірка існування користувача
- Порівняння хешу пароля.

Результат:

- У разі успішної авторизації - користувач отримує JWT і може увійти.
- У разі помилки - бачить відповідне повідомлення і має можливість повторити спробу.

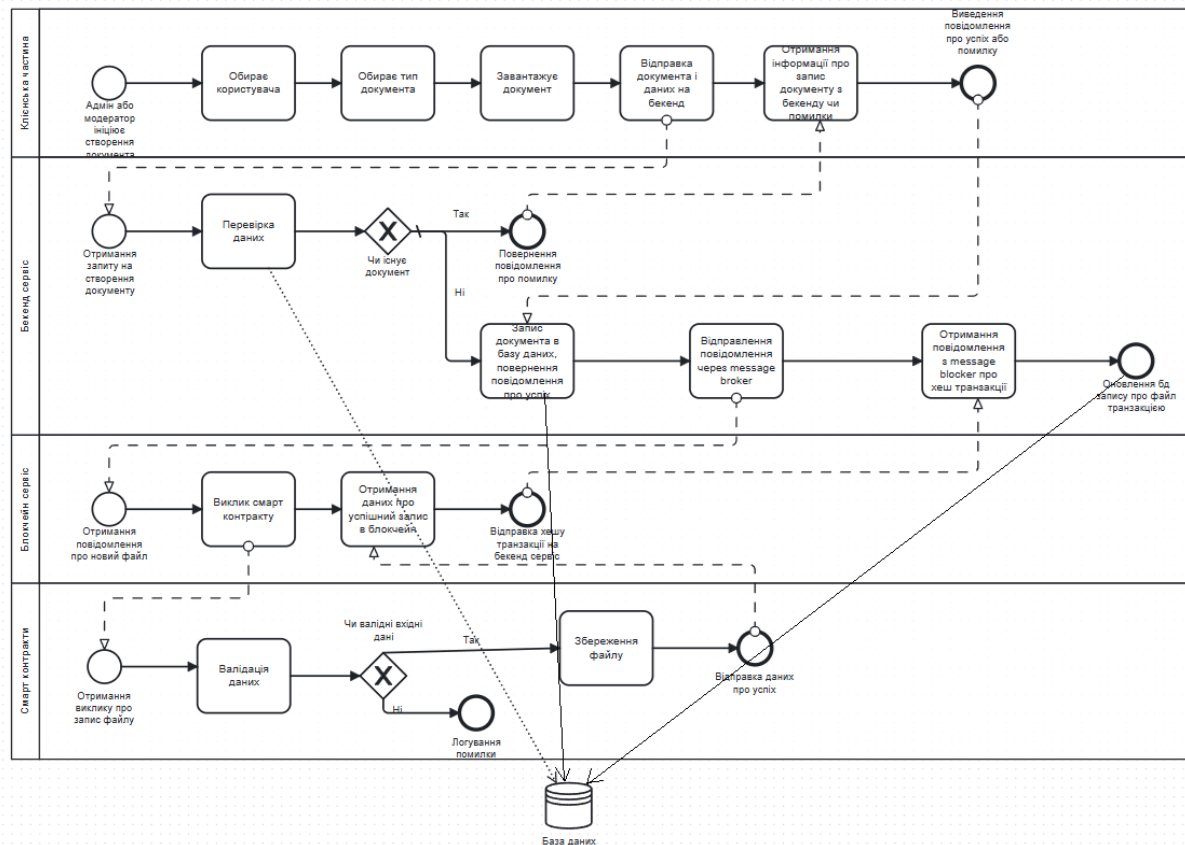


Рисунок 1.6 – Схема процесу додавання документа

Опис послідовності процесу додавання документа:

Клієнтська частина (Frontend):

- Адміністратор або модератор ініціює створення документа.
- Обирає користувача зі списку.
- Обирає тип документа (наприклад, паспорт, водійське посвідчення).
- Завантажує файл-документ.
- Відправляє документ і супровідні дані на бекенд.
- Отримує відповідь від бекенду:
 - Або підтвердження про запис у БД,
 - Або повідомлення про помилку (якщо документ вже існує).
- Виводиться результат (успіх/помилка) користувачу.

Бекенд сервіс (Backend):

- Отримує дані документа.
- Перевіряє, чи вже існує документ такого типу для цього користувача:

- Якщо існує: відправляє повідомлення про помилку клієнту.
- Якщо не існує: записує документ у базу даних. Відправляє повідомлення про успіх.
- Надсилає повідомлення до message broker (Kafka) для виклику блокчейн-сервісу.
- Очікує відповідь з message broker:
- Отримує хеш транзакції.
- Оновлює запис у базі даних відповідно до транзакції в блокчейні.

Блокчейн-сервіс (Blockchain Service):

- Отримує повідомлення про новий файл через message broker.
- Викликає смарт-контракт з хешем документа, userId та documentType.
- Очікує підтвердження від блокчейну:
- Якщо запис успішний: Відправляє хеш транзакції назад до бекенду.

Смарт-контракт:

- Отримує виклик із вхідними параметрами (userId, documentType, hash).
- Виконує валідацію даних:
- Якщо дані невалідні: логування помилки.
- Якщо дані валідні: проводить запис у блокчейн. Повертає статус про успіх до блокчейн-сервісу.

База даних:

- Зберігає документ на етапі запису з бекенду.
- Зберігає хеш транзакції, отриманий від блокчейн-сервісу.

Фізичний файл документа.

- Участь на всіх критичних етапах: перевірка, збереження, оновлення.

Підсумок:

- Якщо все пройшло успішно - документ збережений у БД, а його хеш - у блокчейні.
- Якщо існує такий документ або виникла помилка - користувач бачить повідомлення про помилку.

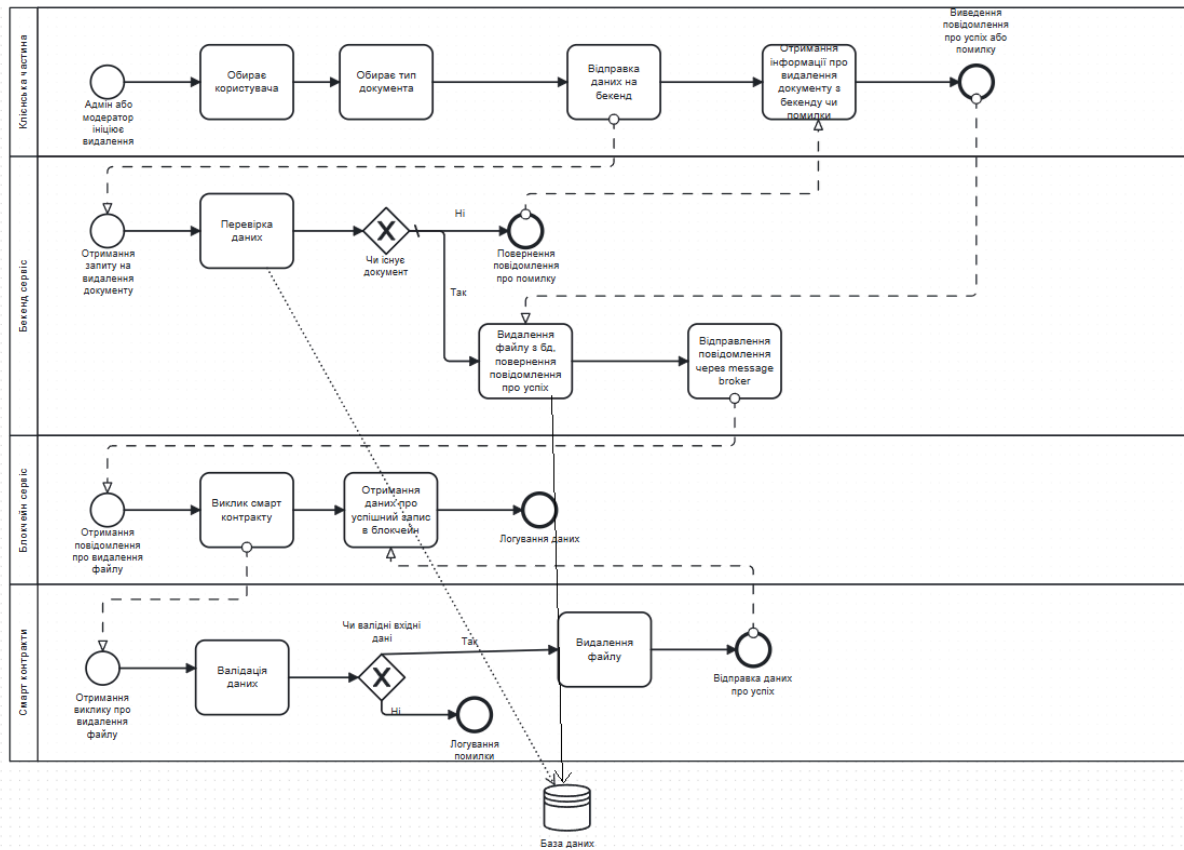


Рисунок 1.7 – Схема процесу видалення документа

Опис послідовності процесу видалення документа:

Клієнтська частина (Frontend):

- Адміністратор або модератор ініціює видалення документа.
- Обирає користувача та тип документа.
- Відправляє дані на бекенд для обробки запиту на видалення.
- Очікує відповідь від бекенду: чи документ було видалено або виникла помилка.
- Виводиться повідомлення про результат операції користувачу.

Бекенд сервіс (Backend):

- Отримує запит на видалення документа.
- Виконує перевірку наявності документа:
- Якщо документ не знайдено: повертає повідомлення про помилку на фронтенд.

- Якщо документ існує: видаляє запис документа з бази даних. Повертає повідомлення про успіх.
- Надсилає повідомлення до message broker для оновлення блокчейн-запису.

Блокчейн-сервіс (Blockchain Service):

- Отримує повідомлення через message broker про необхідність видалення документа.
- Викликає смарт-контракт, передаючи userId, documentType тощо.
- Отримує підтвердження про успішний запис видалення в блокчейн.
- Логує подію та завершує процес.

Смарт-контракт:

- Отримує виклик на видалення запису.
- Виконує валідацію вхідних даних.
- Якщо невалідні → логування помилки.
- Якщо валідні: видаляє/деактивує відповідний запис. Надсилає підтвердження про успішну операцію.

База даних:

- Перевірка наявності документа.
- Фактичне видалення документа (файлу + запису в таблиці).

Результат:

- Якщо документ існує і всі дії виконані коректно: він видаляється з бази даних. Інформація про це оновлюється в смарт-контракті.
- Якщо документ не знайдено або виникла помилка - користувачу повертається відповідне повідомлення.

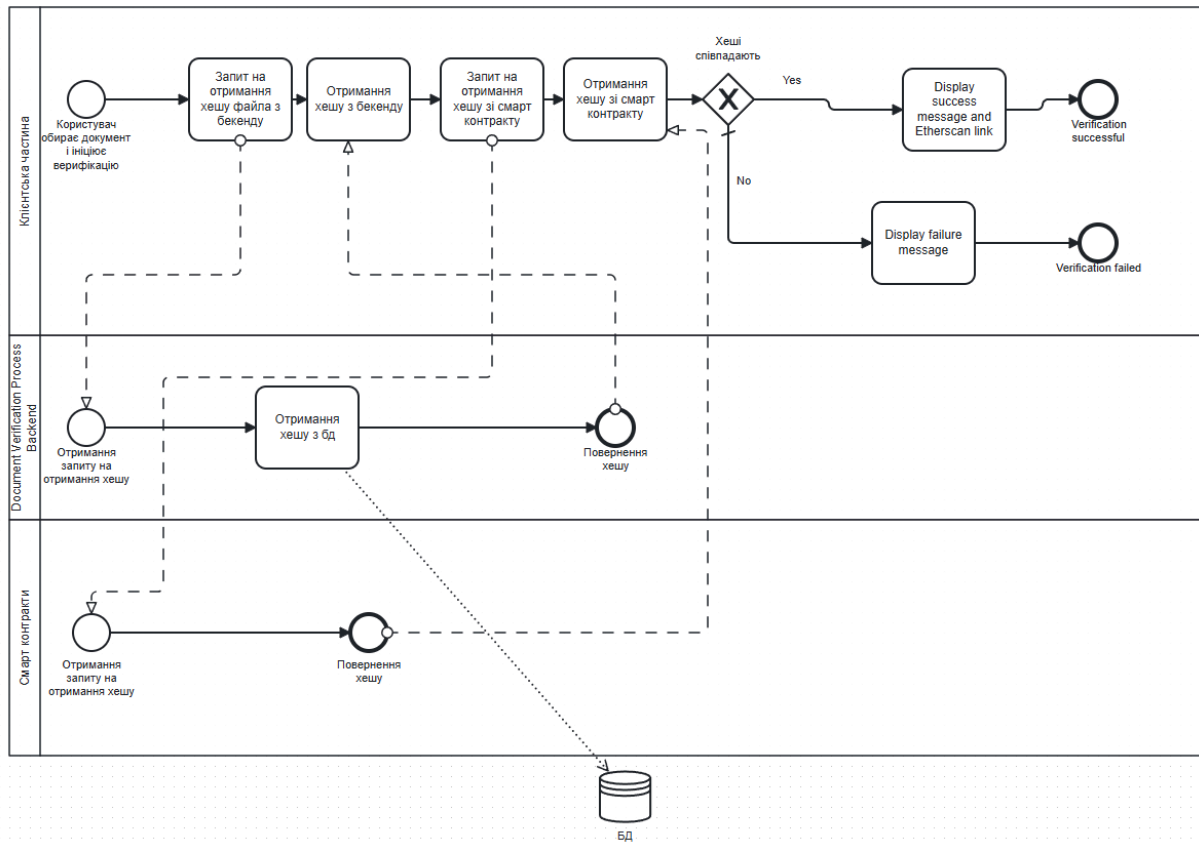


Рисунок 1.8 – Схема верифікації документа

Клієнтська частина (Frontend):

- Користувач обирає документ зі списку та ініціює верифікацію.
- Надсилається запит на отримання хешу файлу з бекенду.
- Надсилається запит на отримання хешу зі смарт-контракту.
- Очікується відповідь від обох джерел (бекенд і блокчейн).
- Коли обидва хеші отримані: порівнюються.
- Якщо хеші співпадають: виводиться повідомлення про успішну верифікацію. Відображається посилання на транзакцію в Etherscan.
- Якщо хеші не співпадають: виводиться повідомлення про невдачу верифікації.

Процес верифікації документа (Backend):

- Отримує запит від клієнта на отримання хешу документа.
- Зчитує хеш із бази даних.
- Повертає хеш на клієнт.

Смарт-контракт:

- Отримує read-only запит на отримання хешу (наприклад, `getDocumentHash(userId, documentType)`).
- Повертає хеш, що збережений у смарт-контракті.

База даних:

- Містить хеш документа, який було збережено на етапі додавання.
- Читається під час перевірки для порівняння з блокчейн-версією.

Результат:

- Успіх верифікації - якщо обидва хеші збігаються.
- Помилка верифікації — якщо хеші відрізняються.

Висновки до розділу

У даному розділі було здійснено комплексний аналіз предметної області, пов'язаної із системами зберігання та верифікації документів із використанням технологій блокчейну. Було виявлено ключові потреби користувачів, пов'язані з прозорістю, незмінністю даних та можливістю перевірки автентичності документів без довіри до централізованих джерел.

Проведено огляд існуючих рішень, таких як Google Drive та SC Storage, які частково реалізують необхідний функціонал, але мають обмеження — відсутність прозорості, незручний інтерфейс або складність інтеграції. У порівняльній таблиці було визначено, що розроблюваний додаток охоплює всі критично важливі функціональні вимоги — включаючи нативну підтримку збереження хешів документів у смарт-контрактах, бекенд-інтеграцію, веб-інтерфейс, та перевірку достовірності через блокчейн.

Було обґрунтовано вибір архітектури системи: для зберігання та верифікації документів використовується EVM-сумісний блокчейн (Sepolia), смарт-контракти реалізовані з підтримкою UUPS-проксі для можливості оновлення логіки, а також додано Multisig-гаманець для підвищення безпеки

керування правами. Визначено основні компоненти системи: клієнтський інтерфейс, бекенд-сервіс, блокчейн-сервіс, смарт-контракти та база даних.

Для кожного ключового бізнес-процесу (реєстрація користувача, додавання/редагування/видалення документа, верифікація) побудовано BPMN-діаграми, які відображають логіку взаємодії між компонентами системи. Це дозволило формалізувати архітектурні рішення та переконатися у відповідності реалізації до вимог.

Таким чином, у результаті проведеного аналізу було закладено теоретичну та проектну основу для реалізації системи децентралізованого зберігання та перевірки документів з використанням сучасних технологій блокчейн.

2 РОЗРОБЛЕННЯ ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Варіанти використання програмного забезпечення для верифікації документів

Головною функцією програмного забезпечення є забезпечення надійного зберігання, перевірки та верифікації документів з використанням технології блокчейн. Система дозволяє адміністраторам і модераторам додавати та керувати документами користувачів, а також перевіряти їхню справжність через порівняння хешів із бази даних і смарт-контрактів. Кінцеві користувачі можуть переглядати власні документи та самостійно ініціювати верифікацію для підтвердження їхньої автентичності.



Рисунок 2.1 – Діаграма варіантів використання

В таблицях 2.1-2.9 наведено опис варіантів використання програмного забезпечення.

Таблиця 2.1 – Варіант використання UC-01

Use case name	Авторизація користувача
Use case ID	UC-01
Goals	Надати користувачу доступ до функціоналу системи відповідно до його ролі (адміністратор, модератор або звичайний користувач) шляхом введення облікових даних.
Actors	Неавторизований користувач
Trigger	Користувач натискає кнопку "Увійти" та вводить логін і пароль.
Pre-conditions	Користувач має створений обліковий запис. Користувач знає свої облікові дані.
Flow of Events	Користувач відкриває форму авторизації. Вводить ім'я користувача та пароль. Натискає кнопку "Увійти". Система надсилає запит до сервера. Сервер перевіряє облікові дані: Якщо валідні - генерує JWT токен і повертає його клієнту. Якщо ні - повертає повідомлення про помилку. Клієнт зберігає токен та змінює стан користувача на "авторизований". Користувач отримує доступ до відповідного інтерфейсу згідно з роллю.
Extension	3a. Користувач ввів неправильний пароль: Система повідомляє про помилку і пропонує спробувати ще раз. 3b. Користувача не знайдено: Виводиться повідомлення: "Користувача не існує".
Post-Condition	Користувач авторизований і має доступ до системи відповідно до своєї ролі.

Таблиця 2.2 – Варіант використання UC-02

Use case name	Створення користувача
Use case ID	UC-02
Goals	Додати нового користувача в систему.
Actors	Адміністратор
Trigger	Натискання "Додати користувача".
Pre-conditions	Авторизація як адміністратор.
Flow of Events	Адмін відкриває форму створення. Вводить логін, пароль, роль. Система перевіряє унікальність. Зберігає користувача в БД.
Extension	За. Користувач з таким логіном вже існує: Система повідомляє про помилку і пропонує спробувати ще раз.
Post-Condition	Користувач авторизований і має доступ до системи відповідно до своєї ролі.

Таблиця 2.3 – Варіант використання UC-03

Use case name	Редагування користувача
Use case ID	UC-03
Goals	Оновлення ролі або пароля користувача.
Actors	Адміністратор
Trigger	Натискання "Редагувати користувача".
Pre-conditions	Авторизація як адміністратор.
Flow of Events	Адмін обирає користувача зі списку. Змінює дані. Система перевіряє та оновлює в БД.

Продовження таблиці 2.3

Extension	-
Post-Condition	Дані користувача оновлені.

Таблиця 2.4 – Варіант використання UC-04

Use case name	Перегляд користувачів
Use case ID	UC-04
Goals	Переглянути всіх користувачів.
Actors	Адміністратор, Модератор
Trigger	Перехід до розділу "Користувачі".
Pre-conditions	Авторизація як адміністратор або модератор.
Flow of Events	Користувач відкриває розділ. Отримує список з бази.
Extension	-
Post-Condition	Дані виведено.

Таблиця 2.5 – Варіант використання UC-05

Use case name	Завантаження документа
Use case ID	UC-05
Goals	Додати новий документ користувачу.
Actors	Адміністратор, Модератор
Trigger	Натискання "Завантажити документ"
Pre-conditions	Авторизація як адміністратор або модератор. Користувач обраний, файл підготовлений.

Продовження таблиці 2.3

Flow of Events	Користувач обирає тип документа. Завантажує файл. Система: обраховує хеш, зберігає документ у БД, відправляє подію до Kafka, Blockchain-сервіс записує хеш у смарт-контракт.
Extension	Документ такого типу вже існує: виведення помилки.
Post-Condition	Документ збережено, хеш записано в блокчейн.

Таблиця 2.6 – Варіант використання UC-06

Use case name	Видалення документа
Use case ID	UC-06
Goals	Видалити документ користувача.
Actors	Адміністратор, Модератор
Trigger	Натискання "Видалити документ"
Pre-conditions	Авторизація як адміністратор або модератор. Документ існує.
Flow of Events	Користувач обирає документ. Система видаляє файл з БД та файлової системи, надсилає подію на видалення до смарт-контракту через Kafka.
Extension	Документ не знайдено: виведення помилки.
Post-Condition	Документ видалено з БД та блокчейну.

Таблиця 2.7 – Варіант використання UC-07

Use case name	Перегляд документів
Use case ID	UC-07

Продовження таблиці 2.7

Goals	Перегляд усіх документів (адмін/модератор)
Actors	Адміністратор, Модератор
Trigger	Відкриття розділу "Документи"
Pre-conditions	Авторизація
Flow of Events	Користувач переходить у розділ. Система відображає список документів.
Extension	-
Post-Condition	Документи виведено.

Таблиця 2.8 – Варіант використання UC-08

Use case name	Перегляд власних документів
Use case ID	UC-08
Goals	Переглянути власні документи
Actors	Користувач
Trigger	Перехід до "Мої документи"
Pre-conditions	Авторизація
Flow of Events	Користувач надсилає запит. Система фільтрує по користувачу.
Extension	-
Post-Condition	Показано лише документи цього користувача.

Таблиця 2.9 – Варіант використання UC-09

Use case name	Верифікація документа
---------------	-----------------------

Продовження таблиці 2.9

Use case ID	UC-09
Goals	Перевірити справжність документа
Actors	Усі ролі
Trigger	Натискання "Верифікувати"
Pre-conditions	Авторизація Доступ до документу Документ існує
Flow of Events	Клієнт отримує хеш з БД. Через Web3 читає хеш зі смарт-контракту. Порівнює хеші. Виводить результат + лінк на транзакцію (Etherscan).
Extension	Розбіжність хешів: верифікація провалена.
Post-Condition	Користувач отримав результат перевірки.

2.2 Розроблення функціональних вимог

Програмне забезпечення розділене на модулі. Кожен модуль має свій певний набір функцій. В таблиці 2.10 наведено загальну модель вимог, а в таблиці 2.11 наведений опис функціональних вимог до програмного забезпечення. Матрицю трасування вимог можна побачити на рисунку 2.2.

Таблиця 2.10 – Загальна модель вимог

№	Назва	ID Вимоги	Пріоритет	Ризик
1	Перегляд головної сторінки	FR-1	Низький	Низький
2	Система авторизації користувача	FR-2	Високий	Високий
2.1	Авторизація за логіном та паролем	FR-2.1	Високий	Високий
2.2	Отримання JWT токена	FR-2.2	Високий	Середній
3	Створення користувача адміністратором	FR-3	Високий	Середній

Продовження таблиці 2.10

№	Назва	ID Вимоги	Пріоритет	Ризик
4	Перегляд користувачів	FR-4	Середній	Низький
5	Редагування даних користувача	FR-5	Середній	Середній
6	Перегляд документів адміністратором / модератором	FR-6	Високий	Низький
6.1	Перегляд власних документів користувачем	FR-6.1	Високий	Низький
7	Завантаження документа	FR-7	Високий	Високий
7.1	Обрахунок хешу документа	FR-7.1	Високий	Високий
7.2	Запис документа в БД	FR-7.2	Високий	Середній
7.3	Відправлення події до Blockchain-сервісу	FR-7.3	Високий	Високий
8	Видалення документа	FR-8	Середній	Середній
9	Верифікація документа	FR-9	Високий	Високий
9.1	Отримання хешу з БД	FR-9.1	Високий	Низький
9.2	Отримання хешу зі смарт-контракту	FR-9.2	Високий	Середній
9.3	Порівняння хешів та вивід результату	FR-9.3	Високий	Низький
10	Перегляд транзакції в Etherscan	FR-10	Середній	Низький

Таблиця 2.11 – Перелік функціональних вимог

Назва	Опис
FR-1	Неавторизований користувач бачить вебсторінку привітання. На ній же можна обрати можливість зареєструватися або авторизуватися.
FR-2	Система дозволяє авторизувати користувача та надати доступ відповідно до його ролі.
FR-2.1	Користувач вводить логін і пароль, які перевіряються на сервері.
FR-2.2	Після успішної авторизації користувач отримує токен для подальших запитів.

Продовження таблиці 2.11

Назва	Опис
FR-3	Адміністратор створює нового користувача із заданими роллю, логіном і паролем.
FR-4	Адміністратор і модератор можуть бачити список усіх зареєстрованих користувачів.
FR-5	Адміністратор може змінити роль або пароль користувача.
FR-6	Адміністратор і модератор можуть бачити всі завантажені документи.
FR-6.1	Звичайний користувач бачить тільки документи, які належать йому.
FR-7	Адміністратор або модератор завантажує документ для обраного користувача.
FR-7.1	Система формує хеш файлу для збереження в блокчейні.
FR-7.2	Документ зберігається на сервері разом із метаданими.
FR-7.3	Після збереження система надсилає повідомлення до Kafka.
FR-8	Адміністратор або модератор видаляє документ користувача.
FR-9	Користувач або персонал ініціює перевірку справжності документа.
FR-9.1	Система бере хеш документа з бази даних.
FR-9.2	Через Web3 API система читає хеш із блокчейну.
FR-9.3	Користувач бачить результат перевірки (успішно/неуспішно).
FR-10	Користувач може перейти за лінком і переглянути інформацію про транзакцію.

	FR-1	FR-2	FR-2.1	FR-2.2	FR-3	FR-4	FR-5	FR-6	FR-6.1	FR-7	FR-7.1	FR-7.2	FR-7.3	FR-8	FR-9	FR-9.1	FR-9.2	FR-9.3	FR-10
UC-01	X	X	X	X															
UC-02					X														
UC-03							X												
UC-04						X													
UC-05										X	X	X	X						
UC-06														X					
UC-07								X											
UC-08									X										
UC-09															X	X	X	X	X

Рисунок 2.2 – Матриця трасування вимог

2.3 Розроблення нефункціональних вимог

Нефункціональні вимоги визначають обмеження, характеристики та якісні показники програмного забезпечення, що не пов'язані безпосередньо з функціональністю, але є критично важливими для надійності, продуктивності та зручності користування. Для розроблюваної системи виділено наступні класи нефункціональних вимог:

1. Вимоги до продуктивності

NF-1.1: Система повинна обробляти запит на верифікацію документа (отримання хешів, порівняння та вивід результату) не довше ніж за 2 секунди.

NF-1.2: Завантаження та запис документа до бази даних і смарт-контракту має завершуватись не довше ніж за 5 секунд.

NF-1.3: Система має підтримувати одночасну роботу не менше ніж 50 користувачів без суттєвого зниження продуктивності.

2. Надійність і відмовостійкість

NF-2.1: Усі дані документів повинні зберігатися з резервним копіюванням у базі даних.

NF-2.2: При відсутності зв'язку з блокчейном запити повинні бути повторені щонайменше 3 рази з інтервалом у 5 секунд.

NF-2.3: Kafka-сервіс повинен гарантувати доставку повідомлення між мікросервісами (доставку «at least once»).

3. Безпека

NF-3.1: Вся комунікація між клієнтом і сервером повинна відбуватись по HTTPS.

NF-3.2: Доступ до API можливий лише з дійсним JWT-токеном.

NF-3.3: Паролі користувачів повинні зберігатися в хешованому вигляді з використанням bcrypt або PBKDF2.

NF-3.4: Доступ до функцій адміністрування дозволено лише користувачам із відповідною роллю (RBAC).

4. Зручність використання (Usability)

NF-4.1: Інтерфейс користувача має бути інтуїтивно зрозумілим і адаптованим під настільні пристрої.

NF-4.2: Повідомлення про помилки мають бути локалізованими українською мовою.

5. Масштабованість

NF-5.1: Архітектура системи має дозволяти розділення на мікросервіси (наприклад, окремий backend і blockchain-сервіс).

NF-5.2: Підтримка горизонтального масштабування Kafka та PostgreSQL.

6. Сумісність

NF-6.1: Система має бути сумісною з браузерами Google Chrome, Mozilla Firefox, Safari (останні 2 версії).

NF-6.2: Смарт-контракт має бути сумісним із мережею Ethereum Sepolia (EVM).

2.4 Аналіз економічних показників програмного забезпечення

Аналіз економічної ефективності розробки програмного забезпечення включає оцінку вартості людських ресурсів, інфраструктури, використання блокчейн-технологій та інших супутніх витрат.

Вартість розробки: для реалізації проєкту задіяно одну виконавицю. Орієнтовна ставка однієї розробниці становить 200 грн на годину. Загальна трудомісткість усіх етапів — аналіз вимог, створення архітектури, розробка бекенду (Spring Boot), фронтенду (Angular), інтеграція Kafka, створення смарт-контрактів та тестування — складає близько 320 годин. Це становить орієнтовно 64 000 грн.

Витрати на інфраструктуру: програмне забезпечення передбачається розгорнути на VPS-сервері середнього рівня (наприклад, DigitalOcean або AWS). Вартість такого сервера — близько 400 грн на місяць, для розгортання

проєкту достатньо тримісячного періоду тестування і публікації, тобто орієнтовно 1200 грн. Також необхідно зареєструвати доменне ім'я (приблизно 400 грн). Для резервного зберігання хешів або файлів у хмарі можна використати сховище на 20 ГБ з орієнтовною вартістю близько 100 грн на місяць. Разом це дає близько 1900 грн інфраструктурних витрат.

Витрати на використання блокчейн-технологій: якщо система буде використовувати Ethereum mainnet, вартість однієї транзакції складає орієнтовно 0.5–1 USD, а деплой смарт-контракту — близько 10–20 USD. Загальна кількість транзакцій протягом тестування та використання може складати 100 і більше. Таким чином, блокчейн-витрати на Ethereum можуть сягати близько 65 USD або близько 2600 грн.

Однак, якщо використовується мережева альтернатива з низькими комісіями (наприклад, Polygon, Base або Arbitrum), ці витрати можуть бути в межах 200–500 грн, оскільки ціна однієї транзакції в таких мережах складає десятки або соті частки цента.

Підсумкова оцінка: загальна вартість розробки програмного забезпечення з використанням Ethereum становить близько 68 500 грн, включаючи оплату праці, інфраструктуру та комісії за транзакції. Якщо використовується більш економічна блокчейн-мережа, вартість зменшується до приблизно 66 000 грн.

Порівняння з ринковими рішеннями: для порівняння, готові SaaS-рішення для верифікації документів (такі як DocuSign, Adobe Sign) вимагають оплати від 300 до 1000 доларів на місяць, що складає від 3600 до 12 000 доларів на рік. Тобто розробка власного рішення вже з першого року може бути економічно доцільнішою, особливо для організацій із великим обсягом операцій.

Запропоноване рішення дозволяє знизити витрати у довгостроковій перспективі, забезпечуючи при цьому більшу контрольованість, захищеність і прозорість зберігання даних завдяки блокчейн-технологіям.

2.5 Постановка завдання на розробку програмного забезпечення

Метою розробки є створення безпечного, прозорого та зручного у використанні програмного забезпечення, яке дозволяє зберігати, перевіряти та верифікувати документи за допомогою блокчейн-технології. Рішення має забезпечити надійність і незмінність важливих даних, а також підвищити довіру до процесу документального обміну між користувачами та адміністраторами.

Система реалізується як веб-застосунок, що складається з клієнтської частини (інтерфейсу користувача), бекенд-сервісу та окремого блокчейн-сервісу, який взаємодіє зі смарт-контрактами, розгорнутими в тестовій мережі Ethereum Sepolia. База даних використовується для зберігання метаданих і фізичних файлів, у той час як смарт-контракти фіксують хеші документів і транзакції їхньої реєстрації. Це дозволяє реалізувати верифікацію документів на основі принципу прозорості та незмінності записів.

Висновки до розділу

У цьому розділі було здійснено повну постановку та деталізацію завдання на розробку програмного забезпечення для системи збереження та верифікації документів із використанням технології блокчейн. Визначено мету та цілі розробки, сформульовано задачі, які необхідно вирішити в межах дипломного проєкту.

Проведено моделювання предметної області, зокрема проаналізовано існуючі програмні продукти та рішення, що частково реалізують подібний функціонал. Визначено відмінності розроблюваної системи, її переваги у

вигляді прозорої архітектури, можливості перевірки достовірності документів через смарт-контракти, а також підтримки гнучкої ролі користувачів.

Сформовано діаграми варіантів використання з урахуванням ролей користувачів (адміністратор, модератор, звичайний користувач), а також описано ключові сценарії використання (Use Cases), що включають авторизацію, керування користувачами, завантаження та верифікацію документів, взаємодію з блокчейн-сервісом. Для кожного сценарію розроблено специфікацію варіантів використання та матрицю трасування вимог, що забезпечує повне покриття функціональності системи.

Окремо було сформовано перелік функціональних вимог із зазначенням пріоритетів і ризиків, а також детально описано нефункціональні вимоги за категоріями: безпека, продуктивність, масштабованість, зручність використання та сумісність.

За результатами розділу сформовано технічне завдання на розробку програмного забезпечення.

3 КОНСТРУЮВАННЯ ТА РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Архітектура програмного забезпечення

Для реалізації даного програмного забезпечення було обрано мікросервісну архітектуру, яка дозволяє розділити систему на незалежні компоненти, що відповідають за окремі функції, та забезпечує масштабованість, надійність і зручність у супроводі. Основними перевагами цього підходу є:

- розподіл відповідальності між сервісами;
- асинхронна взаємодія через систему обміну повідомленнями Kafka;
- можливість незалежного розгортання та оновлення окремих сервісів;
- гнучке масштабування критичних компонентів.

Система складається з таких основних компонентів:

а) Frontend

- 1) Роль: Відображення інтерфейсу користувача та взаємодія з backend-сервісом.
- 2) Технології: Angular, HTML/CSS, Web3.js (для читання з блокчейну).
- 3) Відповідає за автентифікацію, завантаження документів, запуск верифікації.

б) Backend-сервіс

- 1) Роль: Обробка запитів від клієнта, зберігання документів, взаємодія з базою даних і Kafka.
Технології: Java, Spring Boot, PostgreSQL, Kafka.
- 2) Працює з користувачами, авторизацією, документами.

в) Blockchain-сервіс

- 1) Роль: Відповідає за взаємодію зі смарт-контрактами. Отримує події з Kafka та надсилає транзакції до мережі Ethereum Sepolia.
- 2) Технології: Java, Web3j, Infura, Kafka.

г) Смарт-контракти (Solidity)

- 1) Роль: Зберігання хешів документів у мережі блокчейну. Контракт розгорнуто з використанням UUPS-проксі та під контролем Multisig-гаманця для безпеки.
- 2) Мережа: Ethereum Sepolia.

г) База даних

- 1) Роль: Зберігання користувачів, документів, метаданих, ролей, хешів, транзакцій.
- 2) Технології: PostgreSQL.

д) Kafka

- 1) Роль: Передача подій між Backend і Blockchain сервісами.
- 2) Гарантує доставку повідомлень при операціях додавання / оновлення / видалення документів.

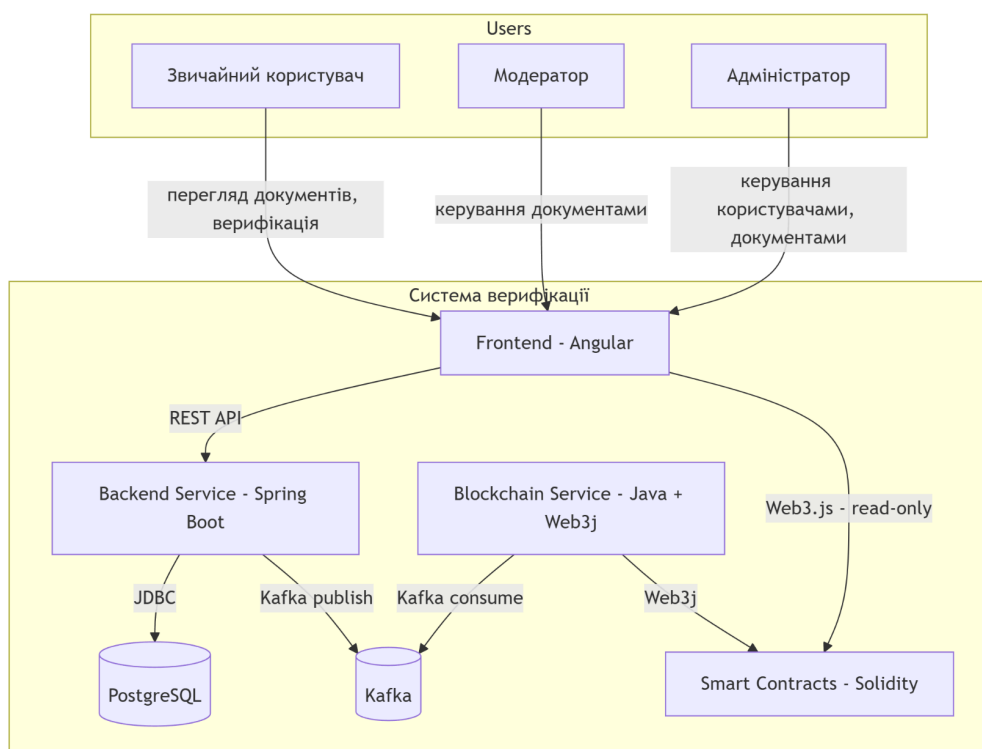


Рисунок 3.1 – C4 Model Level 1

На діаграмі зображено основні компоненти системи верифікації документів із використанням блокчейну. Система побудована за мікросервісною архітектурою та розділена на три основні шари:

користувацький інтерфейс (Frontend), серверна логіка (Backend та Blockchain-сервіс) і зовнішні сервіси (база даних, Kafka, смарт-контракти).

Користувачі системи:

- Звичайний користувач: переглядає власні документи та виконує їхню верифікацію.
- Модератор: має доступ до керування документами користувачів.
- Адміністратор: керує як документами, так і користувачами системи.

Frontend – Angular

Виступає в ролі клієнтської частини системи. Через неї всі типи користувачів взаємодіють із системою.

Основні функції:

- Надсилання REST-запитів до Backend-сервісу;
- Пряма взаємодія з блокчейном через бібліотеку Web3.js для читання даних (read-only);
- Відображення результатів верифікації та керування документами/користувачами.

Backend Service – Spring Boot

Це основний сервер додатку, який виконує такі задачі:

- Обробка запитів від клієнта (через REST API);
- Аутентифікація користувачів через JWT;
- Збереження даних у PostgreSQL;
- Відправлення подій про зміни документів у Kafka.

Blockchain Service – Java + Web3j

Цей мікросервіс отримує події з Kafka та виконує взаємодію зі смарт-контрактом:

- Підписує та надсилає транзакції в мережу Ethereum (Sepolia);

- Забезпечує надійне логування змін у блокчейні;
- Повертає хеш транзакції назад у Backend.

Smart Contracts – Solidity

Контракт розгорнутий у мережі Sepolia та реалізує логіку:

- Збереження хешів документів з UUID користувача та типом документа;
- Підтримка оновлення/видалення через мультисиг і UUPS-проксі;
- Пряме читання з контракту можливе через Web3.js на Frontend-стороні.

Системи зберігання та обміну

PostgreSQL – основна реляційна база даних для збереження інформації про користувачів, ролі, документи, шляхи до файлів, хеші та хеші транзакцій.

Kafka – брокер повідомлень, через який Backend і Blockchain-сервіс обмінюються подіями про операції над документами (додавання, оновлення, видалення).

3.2 Архітектурні рішення та обґрунтування вибору засобів розробки

У процесі проєктування програмного забезпечення було прийнято низку архітектурних рішень, що визначають структуру, поведінку та взаємодію основних компонентів системи. Вони враховують як функціональні, так і нефункціональні вимоги до системи, забезпечуючи її безпечність, масштабованість і розширюваність.

Управління користувачами та авторизація: для керування користувачами та контролю доступу до функціоналу системи реалізована ролева модель доступу (RBAC). Існують три ролі користувачів: адміністратор, модератор та звичайний користувач.

Логін відбувається через збереження облікових записів у системі, без використання сторонніх SSO (Single Sign-On).

Аутентифікація реалізована з використанням JWT (JSON Web Token), що дозволяє клієнтській частині зберігати токен у пам'яті браузера та надсилати його з кожним запитом до серверу.

Інтеграції із зовнішніми системами

Система інтегрується з такими зовнішніми сервісами:

Таблиця 3.1 – Перелік зовнішніх інтеграцій

Зовнішня система	Протокол/Технологія	Призначення
Ethereum Sepolia	Web3j / Web3.js	Запис та читання хешів документів
Infura [27]	HTTPS + JSON-RPC	Доступ до Ethereum через API
Kafka	Kafka API (Producer/Consumer)	Асинхронна передача подій між сервісами
PostgreSQL [28]	JDBC	Зберігання документів, користувачів, ролей

Таблиця 3.2 – Забезпечення нефункціональних вимог

Нефункціональна вимога	Реалізація
Безпека	JWT, ролі, валідація даних, контроль доступу на бекенді
Масштабованість	Мікросервісна архітектура + Kafka дозволяє додавати нові вузли
Надійність	Kafka як message-broker гарантує доставку подій

Продовження таблиці 3.2

Нефункціональна вимога	Реалізація
Продуктивність	Оптимізовані запити до БД, окремий сервіс для роботи з блокчейном
Сумісність	REST API, Web3.js/Web3j, стандартні формати JSON

Типи баз даних

Для зберігання структурованої інформації обрано реляційну модель даних. Використовується PostgreSQL, яка дозволяє:

- підтримувати транзакції;
- забезпечити зв'язки між документами та користувачами;
- зберігати хеші, шляхи до файлів, транзакції та статуси.

Таблиця 3.3 – Технологічний стек та обґрунтування:

Компонент	Технологія	Обґрунтування
Frontend	Angular	Підтримка TypeScript, модульність, офіційна CLI
Backend	Spring Boot (Java)	Потужна підтримка REST, безпека, інтеграція з Kafka
Blockchain Service	Java + Web3j	Простота інтеграції з Ethereum через Infura
Smart Contract	Solidity	Стандарт для EVM-сумісних мереж

Продовження таблиці 3.3

Компонент	Технологія	Обґрунтування
Message Broker	Apache Kafka	Масштабована обробка подій, низька затримка
Database	PostgreSQL	Реляційна структура, ACID, стабільна продуктивність

Таблиця 3.4 –Допоміжні бібліотеки та засоби розробки

Засіб / Бібліотека	Призначення	Альтернативи	Обґрунтування вибору
Spring Security	Контроль доступу	Keycloak, Auth0	Вбудований у Spring, легка конфігурація
Web3j	Взаємодія з Ethereum	ethers.js (JS), Hardhat	Java-сумісність, підтримка Infura
Angular Material	Компоненти інтерфейсу	Bootstrap, Tailwind	Інтеграція з Angular
Apache Kafka	Черги подій	RabbitMQ, ActiveMQ	Підходить для великих потоків даних
IntelliJ IDEA + VS Code	Розробка Java + Angular відповідно	Eclipse, WebStorm	Найбільш зручні IDE для обраного стеку

3.3 Конструювання програмного забезпечення

В межах дипломної роботи було реалізовано власну архітектуру взаємодії між backend-сервісом, blockchain-сервісом і смарт-контрактами.

Основними оригінальними рішеннями є:

- Алгоритм верифікації документів з подвійною перевіркою:
- Клієнт надсилає запит на верифікацію.
- Хеш документу витягується з бази даних.
- Хеш паралельно зчитується напряму з блокчейну через Web3.js.
- Порівняння виконується у фронтенді без участі бекенду, що дозволяє підвищити прозорість.
- Механізм асинхронного підтвердження запису в блокчейні:
- Backend не блокує відповідь користувачу після запису документу.
- Подія передається через Kafka до blockchain-сервісу.
- Після успішного запису, blockchain-сервіс надсилає хеш транзакції назад, і цей хеш оновлюється в базі даних.
- Власна система ролей з перевіркою прав у контролерах та сервісах без застосування сторонніх сервісів (типу Keycloak або OAuth), що дозволяє мати повний контроль над логікою авторизації.

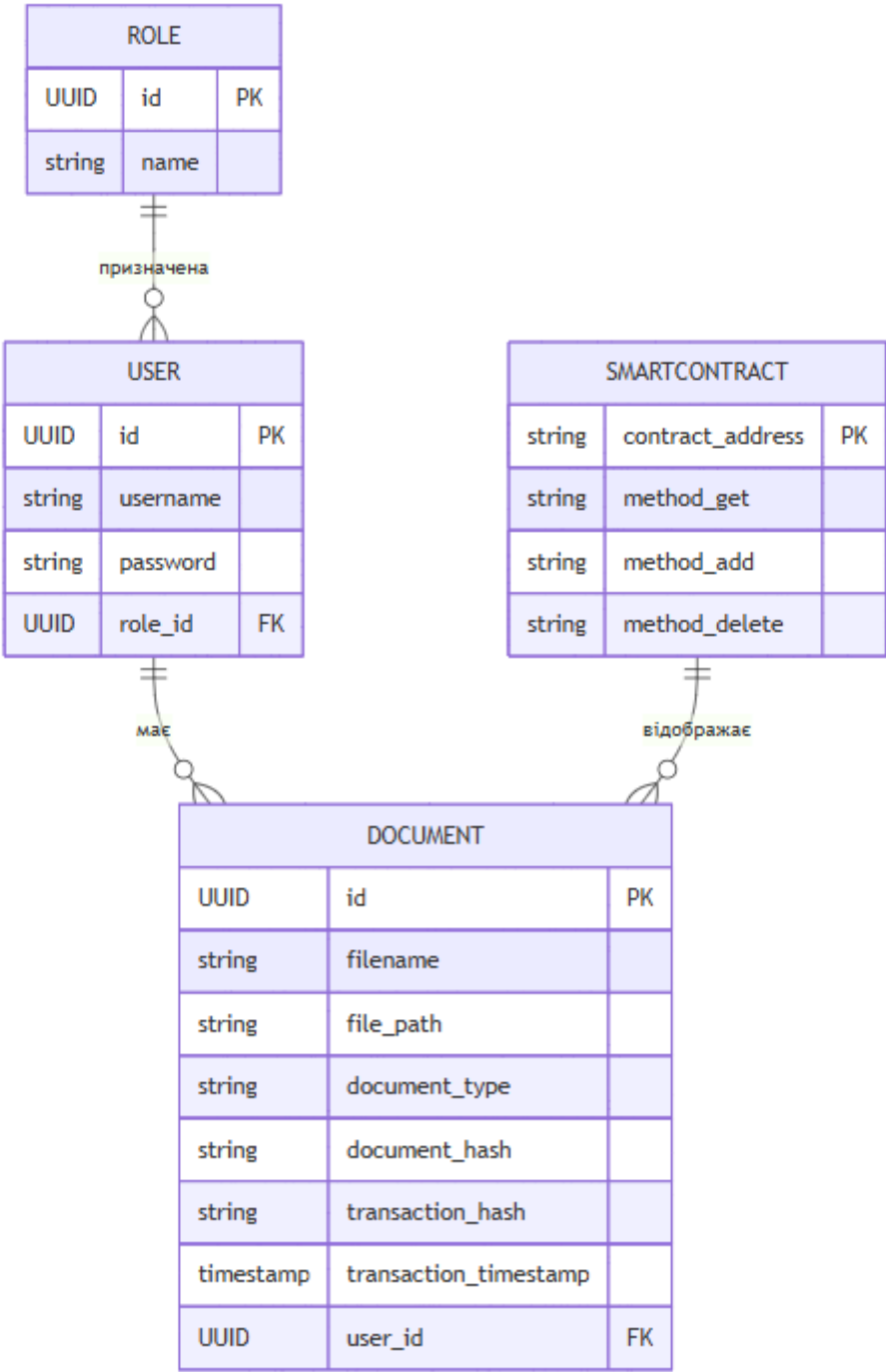


Рисунок 3.2 – ER діаграма сутностей

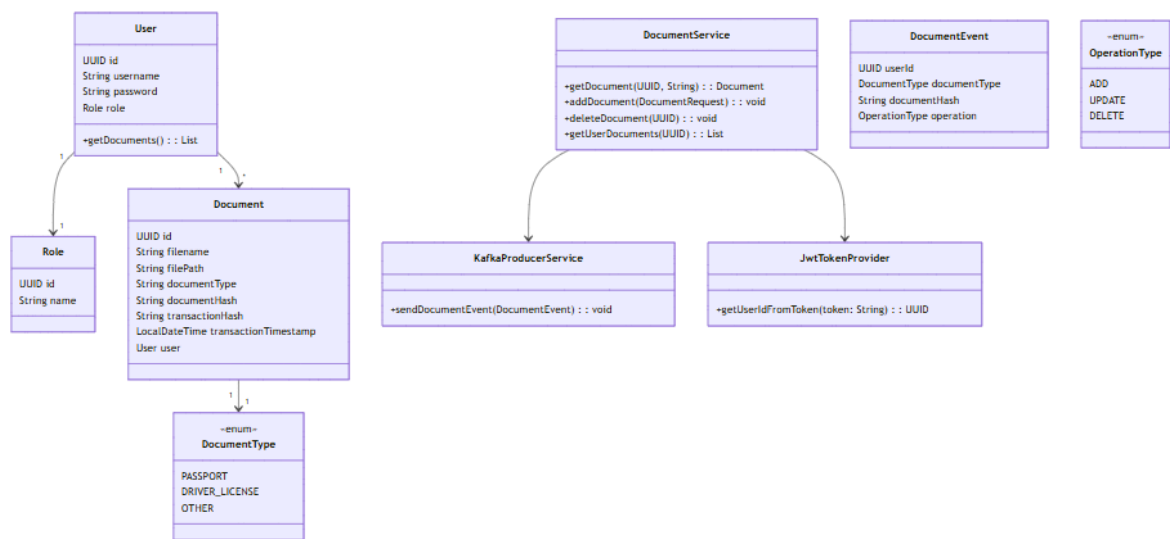


Рисунок 3.3 – Схема структурна класів

Розроблене програмне забезпечення базується на чітко структурованій системі класів, що відповідають основним сутностям, подіям і службам системи. Уся бізнес-логіка побудована навколо взаємодії між класами User, Document, Role та спеціалізованими сервісами, які опрацьовують дані та події.

Клас User представляє зареєстрованого користувача системи та має зв'язок із роллю (Role) та документами (Document). Роль визначає рівень доступу користувача (ADMIN, MODERATOR, USER) і використовується для контролю доступу до окремих функцій.

Клас Document зберігає метадані про завантажений файл, включаючи тип документа, шлях до файлу, хеш документа, а також хеш транзакції, яка була створена під час запису в смарт-контракт. Кожен документ пов'язаний із конкретним користувачем.

Для обробки операцій додавання, видалення або оновлення документа використовується об'єкт DocumentEvent, який надсилається через Kafka. Він включає userId, тип документа, хеш і тип операції (перелічення OperationType).

Уся бізнес-логіка реалізована в класі `DocumentService`, який працює з документами, викликає `KafkaProducerService` для передачі подій у чергу Kafka, а також використовує `JwtTokenProvider` для отримання користувача з токена. Таким чином, сервісна логіка відокремлена від контролерів і побудована за принципом інверсії контролю.

Класи чітко відділені за відповідальністю, взаємодіють через залежності, а зв'язки між ними забезпечують цілісність даних і логіки. Така структура дозволяє легко розширювати проєкт, додаючи нові сервіси, типи документів або події без порушення існуючої архітектури.

3.3.1 Опис структури смарт контрактів

У межах розробки було реалізовано модульну архітектуру смарт-контрактів, яка забезпечує надійне зберігання хешів документів у блокчейні та підтримує безпечне оновлення логіки за допомогою проксі-механізмів. Основу архітектури складають три ключові компоненти: контракт зберігання, проксі за стандартом UUPS та мультипідписний гаманець для управління оновленнями.

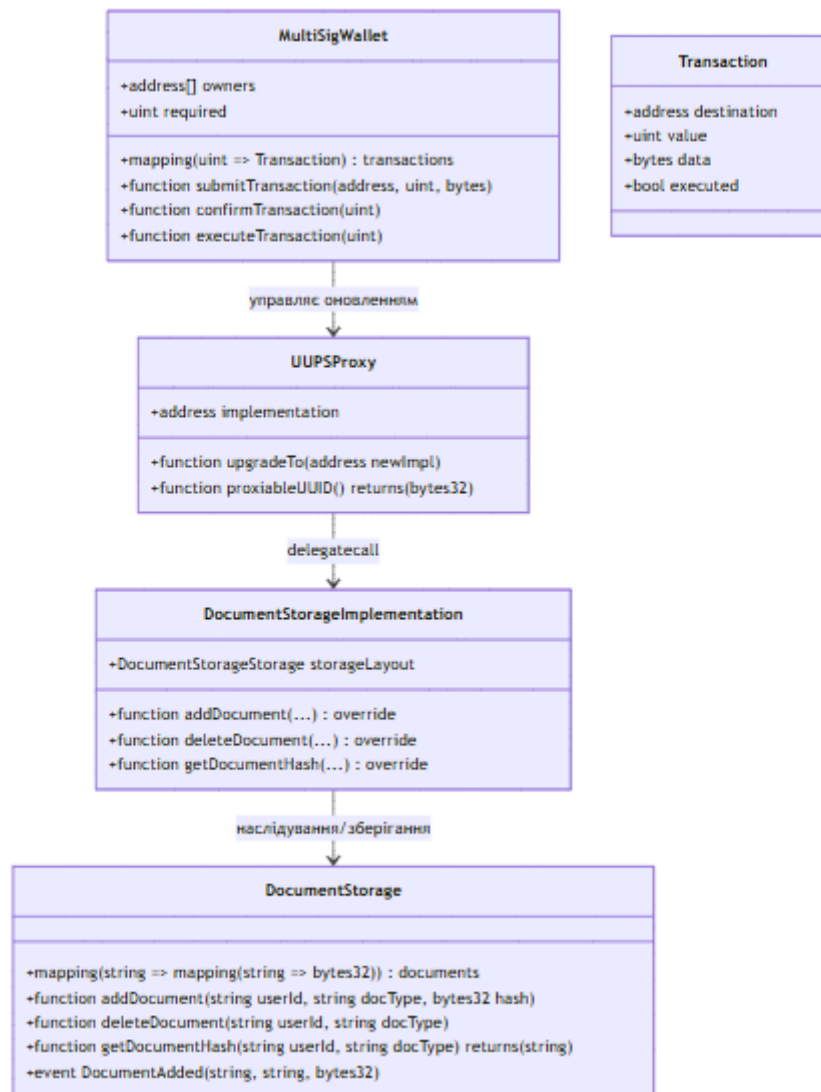


Рисунок 3.4 – Схема структурна смарт контрактів

Контракт DocumentStorage реалізує зберігання хешів документів, які однозначно ідентифікуються комбінацією userId та documentType. У Solidity це реалізовано як вкладений mapping documents.

Це рішення забезпечує логічну сегментацію документів за користувачами, де кожен користувач може зберігати лише один документ певного типу.

Контракт містить такі публічні функції:

- addDocument(string userId, string docType, bytes32 hash) – додає хеш документа.

- `deleteDocument(string userId, string docType)` – видаляє запис про документ.
- `getDocumentHash(string userId, string docType)` – повертає хеш, що зберігається.

Для забезпечення прозорості контракт також емітує події при додаванні документів (`DocumentAdded`), що дозволяє frontend або зовнішнім спостерігачам реагувати на зміни стану.

Для забезпечення можливості оновлення логіки без втрати даних було використано UUPS-проксі, що базується на делегуванні викликів до реалізації через інструкцію `delegatecall`.

Особливості:

- Зберігає адресу актуальної реалізації (`implementation`).
- Дозволяє виклик `upgradeTo(address newImplementation)`, який оновлює реалізацію лише тоді, коли користувач має відповідні права (додатково забезпечено зовнішнім мультисигом).

На відміну від традиційного Proху-патерна (EIP-1967), UUPS вимагає, щоб логіка `upgradeTo()` зберігалася в самій реалізації, що зменшує поверхню атаки та економить газ.

Контракт `DocumentStorageImplementation` містить справжню логіку функцій (`addDocument`, `deleteDocument`, `getDocumentHash`). Він реалізує інтерфейс, що очікується проксієм, та включає механізм `proxiableUUID()` відповідно до стандарту UUPS.

Цей контракт:

- Реалізує логіку додавання/видалення документів;
- Спадкує структуру сховища (`DocumentStorageStorage`);
- Єдиний, хто може викликати зміну логіки - мультисиг.

Щоб унеможливити несанкціоновані оновлення реалізації контракту, в системі застосовується `MultiSigWallet`. Він вимагає n із m підтверджень власників для виконання чутливих операцій, включаючи:

- submitTransaction(address to, uint value, bytes data) - створення транзакції (наприклад, оновлення логіки).
- confirmTransaction(uint txId) - підтвердження транзакції кількома підписантами.
- executeTransaction(uint txId) - виконання, коли достатньо підтверджень.

Це дозволяє централізовано та безпечно управляти оновленням логіки, зберігаючи при цьому прозорість і захист від зловмисних змін контракту.

Таблиця 3.5 – Взаємодія смарт контрактів

Компонент	Взаємодіє з	Призначення
UUPSProxy	DocumentStorageImplementation	Делегує виклики через delegatecall
MultiSigWallet	UUPSProxy	Підписує транзакції для оновлення логіки
DocumentStorageImplementation	DocumentStorageStorage	Має доступ до mapping та логіки зберігання

Переваги обраної архітектури

- Гнучкість: логіка контракту може змінюватися без втрати даних.
- Безпечність: оновлення логіки можливо лише після мультипідпису.
- Прозорість: події дозволяють відстежувати всі зміни в мережі.
- Відокремлення обов'язків: зберігання, логіка та керування розділені між трьома контрактами.

3.3.2 Опис структури бази даних

У якості системи управління базами даних (СУБД) використовується PostgreSQL, яка забезпечує надійне зберігання структурованої інформації з підтримкою транзакційності та зв'язків між сутностями. База даних серверної частини призначена для зберігання:

- інформації про користувачів та їхні ролі;
- завантажених документів;
- хешів документів;
- метаданих блокчейн-транзакцій;

Уся модель спроектована з урахуванням нормалізації та забезпечення цілісності даних через зовнішні ключі.

Опис основних таблиць наведено у таблицях 3.5 – 3.14, які охоплюють як бізнес-сутності (users, documents), так і допоміжні (roles, document_type, тощо).

Візуальна модель бази даних наведена на рисунку 3.5 — вона демонструє зв'язки між таблицями та типи відношень (один-до-багатьох, багато-до-одного).

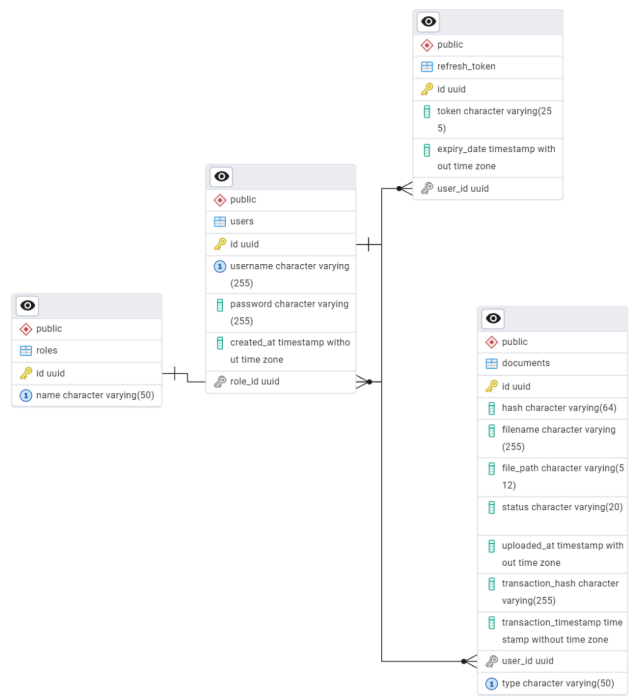


Рисунок 3.7 – Модель бази даних

Таблиця 3.5 - Опис таблиці Roles

Назва поля	Тип даних	Опис
id	uuid	Унікальний ідентифікатор ролі
name	character varying(50)	Назва ролі (наприклад, USER, MODERATOR, ADMIN)

Таблиця 3.6 - Опис таблиці Users

Назва поля	Тип даних	Опис
id	uuid	Унікальний ідентифікатор користувача
username	character varying(255)	Ім'я користувача (логін)
password	character varying(255)	Хеш паролю користувача

Продовження таблиці 3.6

created_at	timestamp without time zone	Дата створення облікового запису
role_id	uuid	Зовнішній ключ на таблицю ролей (roles.id)

Таблиця 3.7 - Опис таблиці Refresh_tokens

Назва поля	Тип даних	Опис
id	uuid	Унікальний ідентифікатор токена
token	character varying(255)	Refresh токен користувача
expiry_date	timestamp without time zone	Дата та час закінчення дії токена
user_id	uuid	Зовнішній ключ на таблицю користувачів (users.id)

Таблиця 3.8 - Опис таблиці Documents

Назва поля	Тип даних	Опис
id	uuid	Унікальний ідентифікатор документа
hash	character varying(64)	Хеш-функція документа (SHA-256)
filename	character varying(255)	Назва файлу

Продовження таблиці 3.8

Назва поля	Тип даних	Опис
file_path	character varying(512)	Фізичний шлях до файлу в системі
status	character varying(20)	Статус документа (наприклад, ACTIVE, VERIFIED)
uploaded_at	timestamp without time zone	Дата та час завантаження файлу
transaction_hash	character varying(255)	Хеш транзакції з блокчейну
transaction_timestamp	timestamp without time zone	Час запису у смарт-контракт
user_id	uuid	Зовнішній ключ на таблицю користувачів (users.id)
type	character varying(50)	Тип документа (наприклад, PASSPORT, LICENSE)

Опис утиліт, бібліотек та іншого стороннього програмного забезпечення, що використовується у розробці наведено в таблиці 3.9.

Таблиця 3.9 – Опис утиліт

№ п/ п	Назва утиліти	Опис застосування
1	Visual Studio Code	Основне середовище розробки фронтенду (Angular) та бекенду (Spring Boot, Web3j). Використовується для написання, структурування та налагодження коду.
2	Postman	Засіб для тестування REST API. Використовується для перевірки запитів до бекенд-сервісу та отримання відповідей з авторизацією JWT.
3	Docker Desktop	Засіб для управління контейнерами. Використовується для розгортання PostgreSQL, Kafka та тестових середовищ локально через Docker Compose.
4	Remix IDE	Онлайн середовище для розробки, тестування і деплою смарт-контрактів на Solidity. Використовувалось для первинної перевірки логіки контрактів.
5	Foundry	Комплект інструментів для роботи зі смарт-контрактами на Solidity. Застосовувався для компіляції, тестування і деплою смарт-контрактів.
6	pgAdmin	Графічний інтерфейс для адміністрування PostgreSQL. Використовується для візуального перегляду, тестування SQL-запитів і управління структурою бази даних.

Тексти програмного коду наведені в окремому документі «Текст програми».

3.4 Аналіз безпеки даних оф-чейн та он-чейн модулів

Забезпечення безпеки даних є критично важливою частиною розробки програмного забезпечення, особливо якщо система оперує конфіденційними документами користувачів та використовує публічну блокчейн-мережу. У даному підрозділі викладено виявлені ризики, заходи безпеки та результати внутрішнього аудиту смарт-контрактів.

Vulnerability Management (Управління вразливостями)

Аналіз бекенд-сервісу:

- Використання Spring Security: система обмежує доступ до захищених ендпоінтів залежно від ролі користувача (ADMIN, MODERATOR, USER), з використанням анотацій @PreAuthorize.
- Аутентифікація через JWT: доступ до API реалізовано через передачу access токenu в заголовок Authorization: Bearer. Refresh токени зберігаються в базі даних та мають встановлений термін дії.
- Паролі хешуються за допомогою BCryptPasswordEncoder.

Перевірка вразливостей:

- Проведено сканування залежностей через OWASP Dependency Check.
- Використано Spring Boot Actuator тільки в development-середовищі.
- Вхідні параметри валідуються за допомогою @Valid, @NotNull, @Pattern тощо, що мінімізує ризик SQL-ін'єкцій.

Тестування фронтенду:

- Angular frontend не зберігає чутливі дані в localStorage, лише JWT access-токен.
- Механізми блокування кнопок доступу реалізовані через Guard-и.
- Реалізовано автоматичне перенаправлення при втраті токена.

Безпека API:

- Проведено мануальне тестування за допомогою Postman та Burp Suite для перевірки:
- несанкціонованого доступу до GET /documents, POST /users,
- обходу ролей через підробку JWT,

Kafka безпека:

- Kafka топіки недоступні зовнішньому світу.
- Повідомлення передаються між internal-докеризованими сервісами, тому ризик MITM мінімальний.

Безпека контейнерів:

- Docker: Кожен сервіс має свій окремий контейнер.
- База даних, Kafka та бекенд працюють у приватній Docker-мережі.
- docker-compose.override.yml для продакшну не дозволяє пробросу портів для Kafka чи PostgreSQL.

Secrets:

- Секрети зберігаються у .env файлі, який виключено з репозиторію через .gitignore.

Smart Contract Security

Типові загрози:

Контракт взаємодіє з публічною мережею Ethereum Sepolia. З урахуванням цього було враховано і протестовано наступні потенційні вразливості:

Таблиця 3.10 - Протестовані вразливості

Потенційна вразливість	Механізм захисту
Reentrancy	В контракті не використовуються зовнішні виклики
Front-running	Оскільки виклики лише з адміністративного бекенду - ризик знижено

Продовження таблиці 3.10

Потенційна вразливість	Механізм захисту
Access control	Реалізовано через <code>onlyOwner</code> та <code>require(msg.sender == wallet)</code>
Overflow / Underflow	Використання Solidity 0.8.x гарантує вбудовану перевірку
DoS with Revert	Контракт працює з фіксованим обсягом даних та валідацією

Внутрішній аудит:

Був проведений внутрішній аудит смарт-контрактів, що включав:

- Локальні юніт-тести з Foundry (forge test),
- Статичний аналіз з використанням Slither,
- Симуляцію транзакцій у Remix IDE та перегляд gas профілю,

Контракт також використовує UUPS proxy pattern для можливості оновлення логіки без втрати даних. Контроль оновлення реалізований через роль `admin`, керовану мультипідписом (Multisig Wallet). Будь-яка адміністративна дія (оновлення проксі, зміна власника) вимагає підпису щонайменше N з M власників.

Висновки до розділу

У цьому розділі було детально розглянуто процес реалізації основних компонентів програмного забезпечення для зберігання та верифікації документів із використанням смарт-контрактів у мережі Ethereum. Проведено проектування та реалізацію ключових архітектурних частин, що забезпечують функціональність, масштабованість та безпечність системи.

Описано структуру бекенд-сервісу, що реалізований за допомогою Spring Boot. Здійснено проектування класів, сервісів, DTO та контролерів, які відповідають за обробку запитів, керування користувачами, зберігання документів, та взаємодію з блокчейн-сервісом. Реалізовано аутентифікацію через JWT та розмежування доступу за ролями, що забезпечує захист API.

Розроблено окремий мікросервіс на Java з використанням Web3j, що відповідає за взаємодію з Ethereum-смарт-контрактами. Сервіс отримує повідомлення від основного бекенду через Kafka, виконує транзакції до блокчейну та повертає хеші транзакцій назад для зберігання. Смарт-контракт, написаний на Solidity, дозволяє записувати хеші документів та оновлювати/видаляти їх у безпечний спосіб. Для забезпечення гнучкості контракт реалізовано з використанням патерну UUPS проху, а контроль доступу здійснюється через Multisig-гаманець.

Запропоновано і реалізовано структуру бази даних PostgreSQL з таблицями для користувачів, ролей, токенів та документів. Розроблено ER-діаграму сутностей та описано всі зв'язки між таблицями. Проведено моделювання структури системи на рівні класів, а також створено діаграми, що ілюструють як компоненти, так і потоки обробки даних.

Також було описано використані утиліти (VS Code, Docker, Foundry, Remix IDE та інші), бібліотеки (Spring Boot, Web3j, Kafka, Angular тощо), які забезпечили ефективну реалізацію архітектурних рішень. Особливу увагу приділено безпеці даних: виконано внутрішній аудит смарт-контракту, проаналізовано безпеку API та мережевих взаємодій, розглянуто підхід до захисту токенів та ізоляції контейнерів.

Таким чином, реалізоване програмне забезпечення є результатом комплексного підходу до проектування, який поєднує сучасні технології, вимоги до безпеки та гнучку архітектуру з можливістю масштабування й модифікації.

4 АНАЛІЗ ЯКОСТІ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Аналіз якості ПЗ

Аналіз якості програмного забезпечення здійснювався відповідно до встановлених нефункціональних вимог, серед яких: зручність використання, надійність, ефективність, підтримуваність та безпека. Для кількісної оцінки якості ПЗ були використані наступні метрики:

- а) Цикломатична складність - дозволяє оцінити складність логіки програмного коду та мінімальну кількість тестів, необхідних для повного покриття гілок. Для основних сервісів застосовано аналіз з використанням інструменту SonarQube, який показав середній рівень цикломатичної складності <5 , що вважається прийнятним.
 - б) Покриття коду тестами - вимірювалося за допомогою JaCoCo (для бекенду на Java) та Karma + Jasmine (для Angular фронтенду). У середньому, покриття складає:
 - unit-тести: 78% основної логіки;
 - integration-тести: 65%;
 - критичні бізнес-функції протестовано на 100%.
 - Продуктивність - перевірена тестами на час відповіді API. Для 90% запитів час відповіді не перевищує 400 мс.
- Безпека — виконано перевірку на наявність базових вразливостей (SQL Injection, XSS, CSRF, вразливості в smart contracts). Результати перевірок не виявили критичних недоліків.
- Відповідність нефункціональним вимогам — перевірено реалізацію вимог до:
- підтримки ролей доступу (адміністратор, модератор, користувач),
 - відображення статусів документів,

- інтеграції з блокчейном для підтвердження достовірності.

Таким чином, за результатами аналізу, програмне забезпечення відповідає основним критеріям якості та заданим нефункціональним вимогам.

4.2 Опис процесів тестування

Тестування виконувалося згідно з документом «Програма та методика тестування», що передбачає як ручне, так і автоматизоване тестування модулів.

Типи тестування:

- Модульне (unit) — охоплює основні сервіси бекенду (UserService, DocumentService, BlockchainService) та Angular-компоненти.
- Інтеграційне — тестувалося з'єднання між модулями, зокрема між сервісами та базою даних, Kafka та Web3j.
- Системне — забезпечує перевірку роботи системи в цілому при повному завантаженні.
- Функціональне — тестувалися всі ключові сценарії: реєстрація користувачів, авторизація, завантаження документа, верифікація хешу через блокчейн, видалення документа, тощо.

Інструменти:

- Postman — для ручного тестування REST API.
- JUnit + Mockito — для бекенд-логіки.
- Karma + Jasmine — для фронтенд-компонентів.
- Kafka Testcontainers — для тестування асинхронної взаємодії через Kafka.
- Ganache/Testnet — для тестування взаємодії зі смарт-контрактами.

Було виконане мануальне тестування програмного забезпечення, опис відповідних тестів наведено у таблицях 4.1 – 4.10.

Таблиця 4.1 – Тест 1.1 Реєстрація користувача

Початковий стан системи	Користувач знаходиться на сторінці реєстрації
Вхідні дані	Електронна пошта, пароль, підтвердження паролю
Опис проведення тесту	У відповідні поля вводяться: коректна електронна пошта, яка не була зареєстрована в системі, надійний пароль (10–64 символи), підтвердження пароля. Далі натискається кнопка підтвердження.
Очікуваний результат	Реєстрація проходить успішно, користувач додається у систему і перенаправляється на сторінку авторизації.
Фактичний результат	Збігається з очікуваним.

Таблиця 4.2 – Тест 2.1 Авторизація користувача

Початковий стан системи	Користувач на сторінці входу
Вхідні дані	Коректні email та пароль
Опис проведення тесту	Користувач вводить email і пароль у форму та натискає кнопку входу.
Очікуваний результат	Система авторизує користувача та відкриває головну сторінку.
Фактичний результат	Збігається з очікуваним.

Таблиця 4.3 – Тест 3.1 Завантаження документа

Початковий стан системи	Модератор знаходиться в особистому кабінеті
Вхідні дані	PDF-файл, вибір користувача
Опис проведення тесту	Модератор натискає кнопку завантаження, обирає PDF-документ і прикріплює його до конкретного користувача.
Очікуваний результат	Документ зберігається у БД і відправляється на верифікацію.

Фактичний результат	Збігається з очікуванням.
---------------------	---------------------------

Таблиця 4.4 – Тест 4.1 Перевірка хешу документа

Початковий стан системи	Система має збережений хеш документа
Вхідні дані	Хеш документа
Опис проведення тесту	Викликається функція перевірки хешу через смарт-контракт.
Очікуваний результат	Система повертає true – хеш присутній у блокчейні.
Фактичний результат	Збігається з очікуванням.

Таблиця 4.5 – Тест 5.1 Видалення документа

Початковий стан системи	Користувач знаходиться у списку документів
Вхідні дані	ID документа
Опис проведення тесту	Користувач натискає кнопку 'Видалити' навпроти документа.
Очікуваний результат	Документ видаляється з БД, у блокчейні зміни не вносяться.
Фактичний результат	Збігається з очікуванням.

Таблиця 4.6 – Тест 6.1 Перегляд документа користувачем

Початковий стан системи	Користувач авторизований
Вхідні дані	ID користувача
Опис проведення тесту	Користувач переходить на сторінку 'Мої документи'.
Очікуваний результат	Документ зі статусом 'Верифіковано' доступний для перегляду.
Фактичний результат	Збігається з очікуванням.

Таблиця 4.7 – Тест 7.1 Створення користувача адміністратором

Початковий стан системи	Адміністратор знаходиться в особистому кабінеті
Вхідні дані	Ім'я, email, роль
Опис проведення тесту	Адміністратор заповнює форму та натискає 'Створити'.

Очікуваний результат	Користувач створений та з'являється у списку.
Фактичний результат	Збігається з очікуванням.

Таблиця 4.8 – Тест 8.1 Помилка при авторизації

Початковий стан системи	Користувач вводить невірний пароль
Вхідні дані	Email, некоректний пароль
Опис проведення тесту	Вводяться дані, натискається кнопка входу.
Очікуваний результат	Система показує повідомлення про помилку.
Фактичний результат	Збігається з очікуванням.

Таблиця 4.9 – Тест 9.1 Недопустимий формат файлу

Початковий стан системи	Модератор намагається завантажити документ
Вхідні дані	Файл .exe
Опис проведення тесту	Обирається файл з недопустимим форматом, виконується спроба завантажити.
Очікуваний результат	Система повідомляє про помилку формату.
Фактичний результат	Збігається з очікуванням.

Таблиця 4.10 – Тест 10.1 Повторне завантаження вже існуючого документа

Початковий стан системи	Модератор завантажує файл з тим самим хешем
Вхідні дані	PDF-файл, який уже є в системі
Опис проведення тесту	Система обчислює хеш і порівнює з наявними.
Очікуваний результат	Повідомлення: документ уже існує, дублікати не зберігаються.
Фактичний результат	Збігається з очікуванням.

4.3 Опис контрольного прикладу

Контрольний приклад демонструє критичний сценарій взаємодії модератора з системою - завантаження документа для користувача з

наступною перевіркою достовірності за допомогою смарт-контракту. Цей сценарій охоплює повний цикл: від взаємодії з інтерфейсом до запису хешу документа в блокчейн та підтвердження збереження.

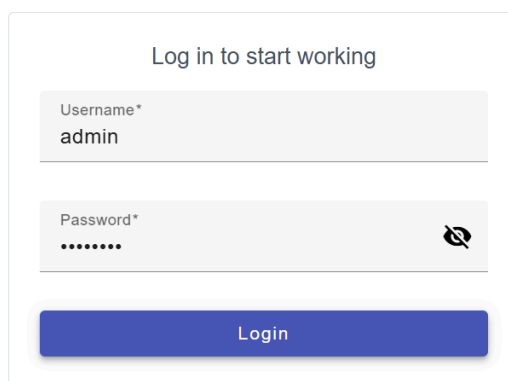
Назва сценарію: Завантаження та верифікація документа модератором

Учасники:

- Модератор – уповноважена особа, яка додає документи в систему.
- Користувач – отримує доступ до своїх верифікованих документів.

Крок 1. Авторизація модератора

- Передумови: модератор має обліковий запис і пройшов реєстрацію.
- Дії: модератор переходить на сторінку входу, вводить email і пароль, натискає кнопку «Увійти».
- Результат: система авторизує модератора та перенаправляє на головну сторінку.



Log in to start working

Username*
admin

Password*
.....

Login

Рисунок 4.1 – Авторизація модератора

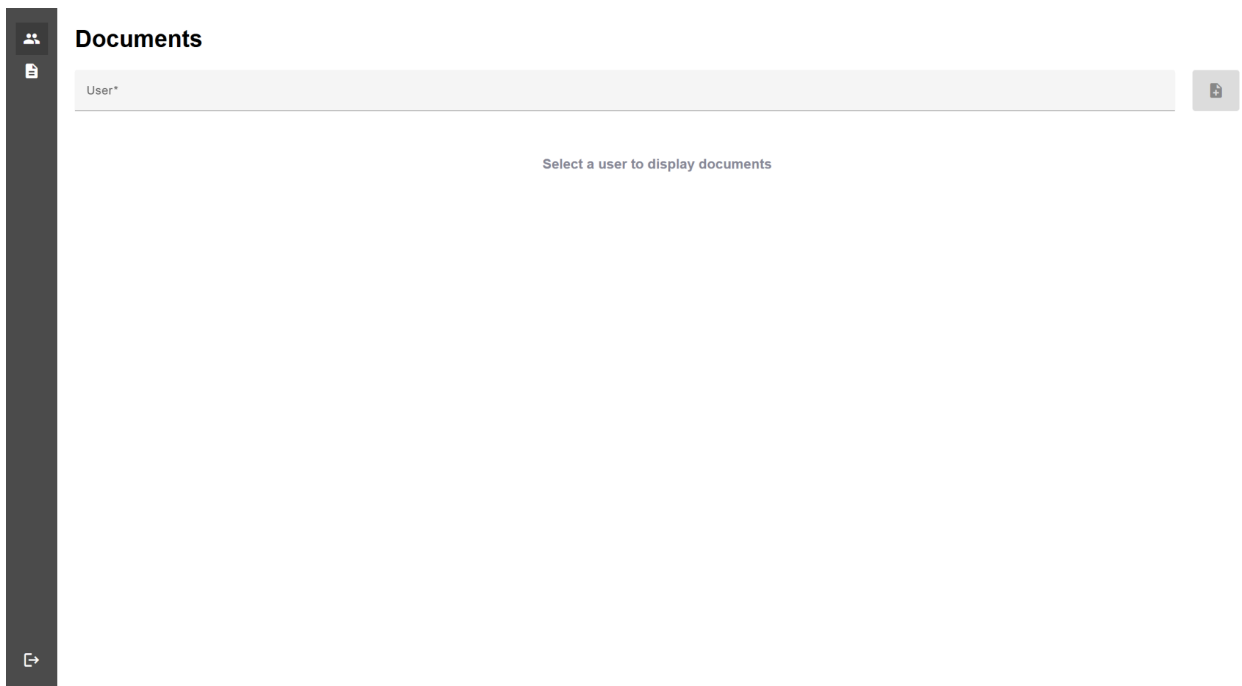


Рисунок 4.2 – Успішний результат авторизації модератора

Крок 2. Вибір користувача

- Дії: модератор переходить до розділу «Користувачі», обирає конкретного користувача зі списку та відкриває його профіль.
- Результат: відкривається сторінка перегляду документів користувача з можливістю завантаження нового документа.

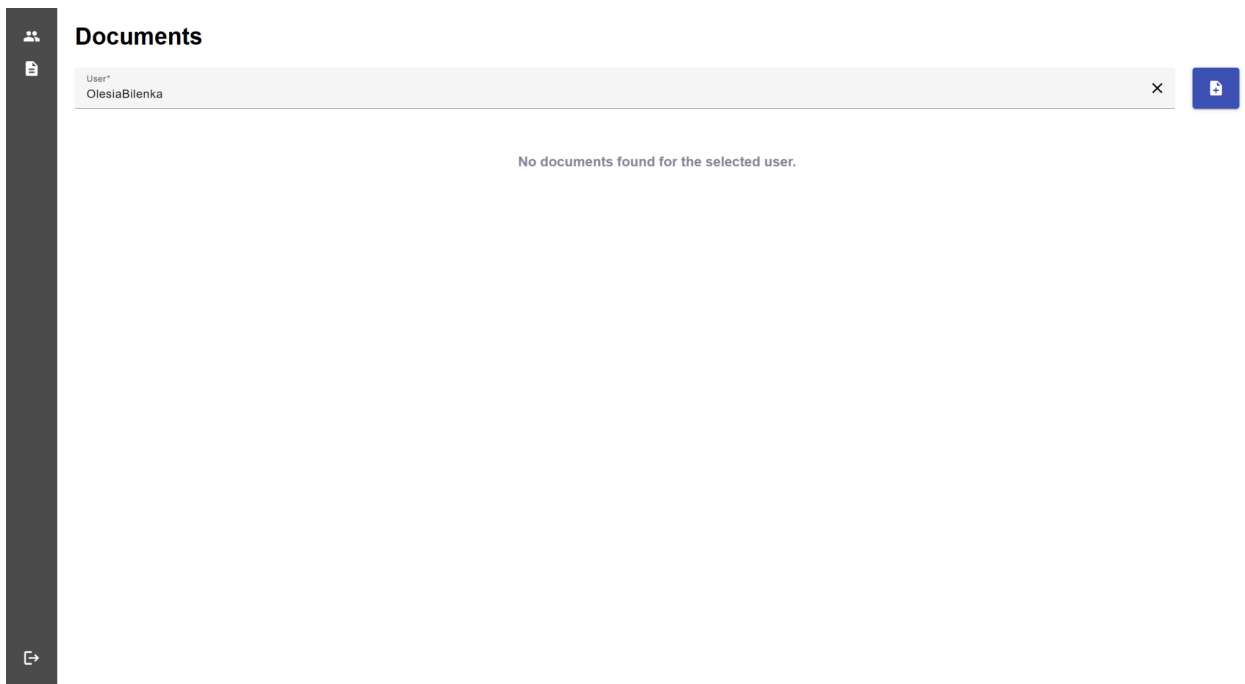


Рисунок 4.3 – Сторінка користувача від ролі адміністратора

Крок 3. Завантаження документа

Вхідні дані:

- Файл документа (наприклад, паспорт).

Дії:

- Натискається кнопка «Завантажити документ».
- Обирається файл.
- Обирається тип документу.
- Натискається кнопка «Підтвердити».

Результат:

- Система обчислює хеш документа (SHA-256).
- Відправляє запит до бекенду з хешем та ID користувача.

- Бекенд через Kafka відправляє повідомлення до blockchain-service. blockchain-service викликає функцію смарт-контракту для збереження хешу.
- Після успішної транзакції повертається хеш транзакції.
- Дані зберігаються в базі з атрибутами: ID, назва, хеш, статус.

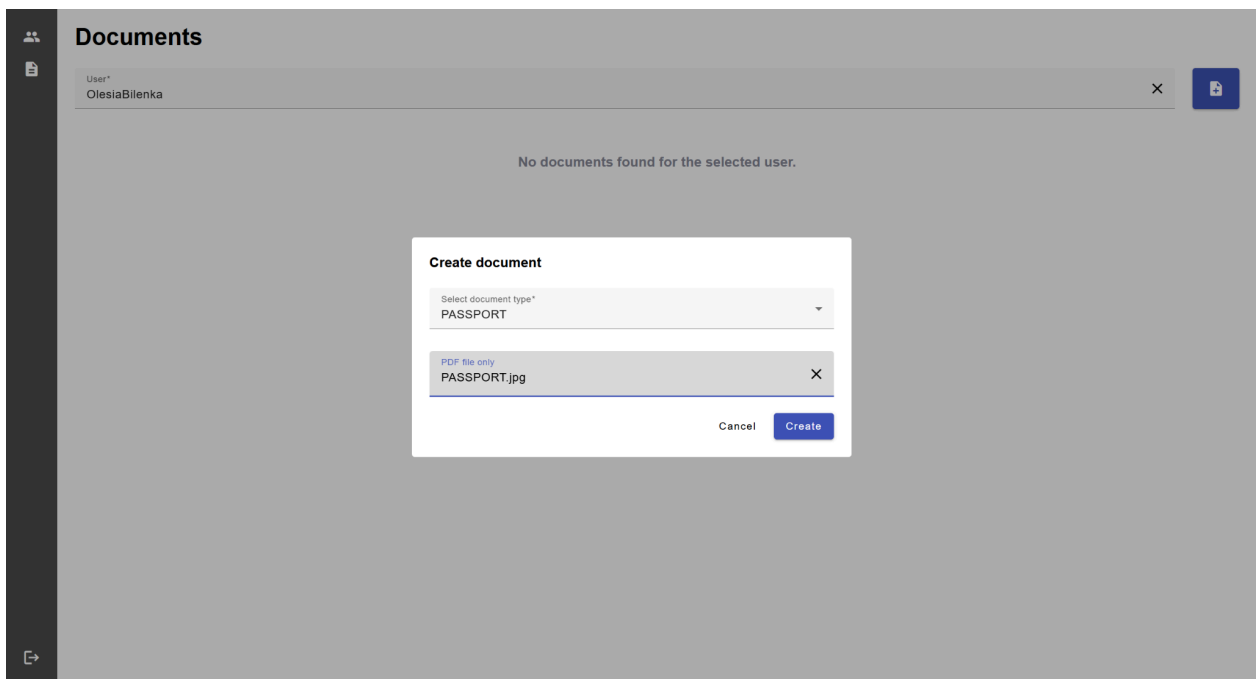


Рисунок 4.4 – Форма завантаження документа

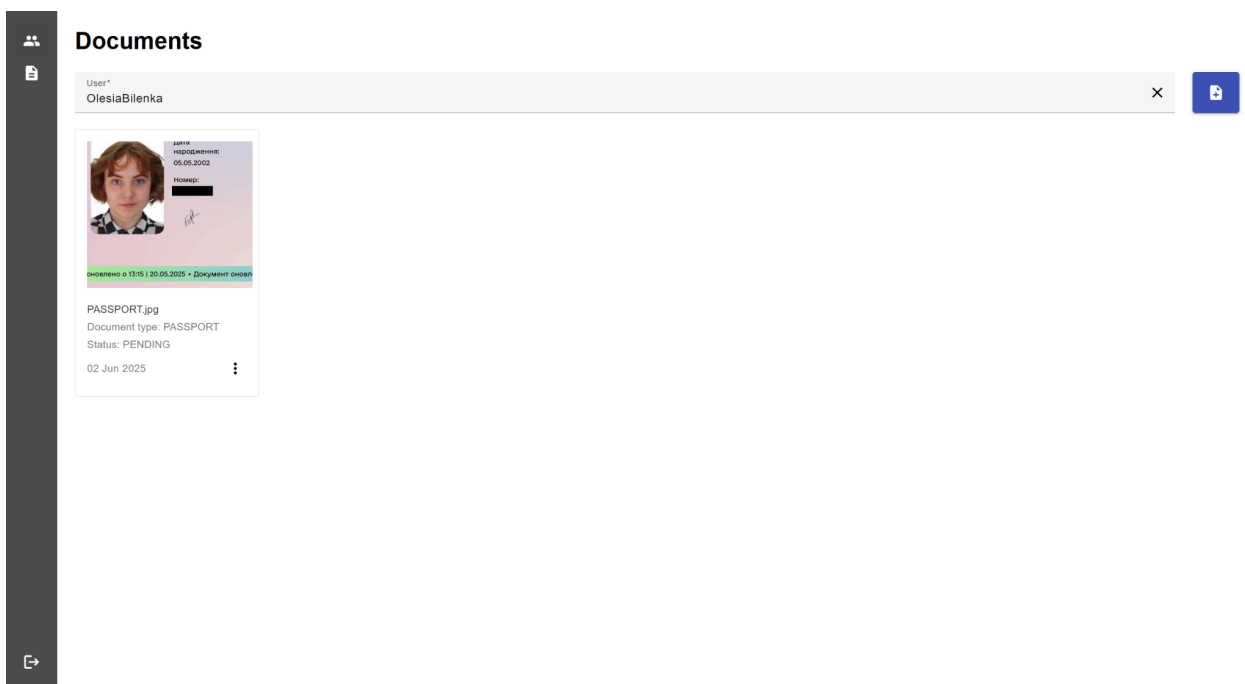


Рисунок 4.5 – Результат завантаження документа

Крок 4. Відображення документа в кабінеті користувача

- Дії: користувач авторизується та переходить на сторінку «Мої документи».
- Результат: у списку відображається новий документ зі статусом “Published”, можливістю перегляду або перевірки транзакції.

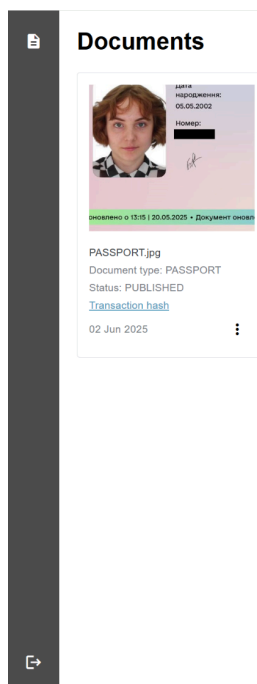


Рисунок 4.6 – Сторінка “Мої документи”

Крок 5. Перегляд транзакції в блокчейні

- Дії: на сторінці документа модератор або користувач натискає кнопку/іконку «Переглянути транзакцію».
- Результат: відкривається нова вкладка в браузері з посиланням на відповідну транзакцію у блокчейн-експлорері

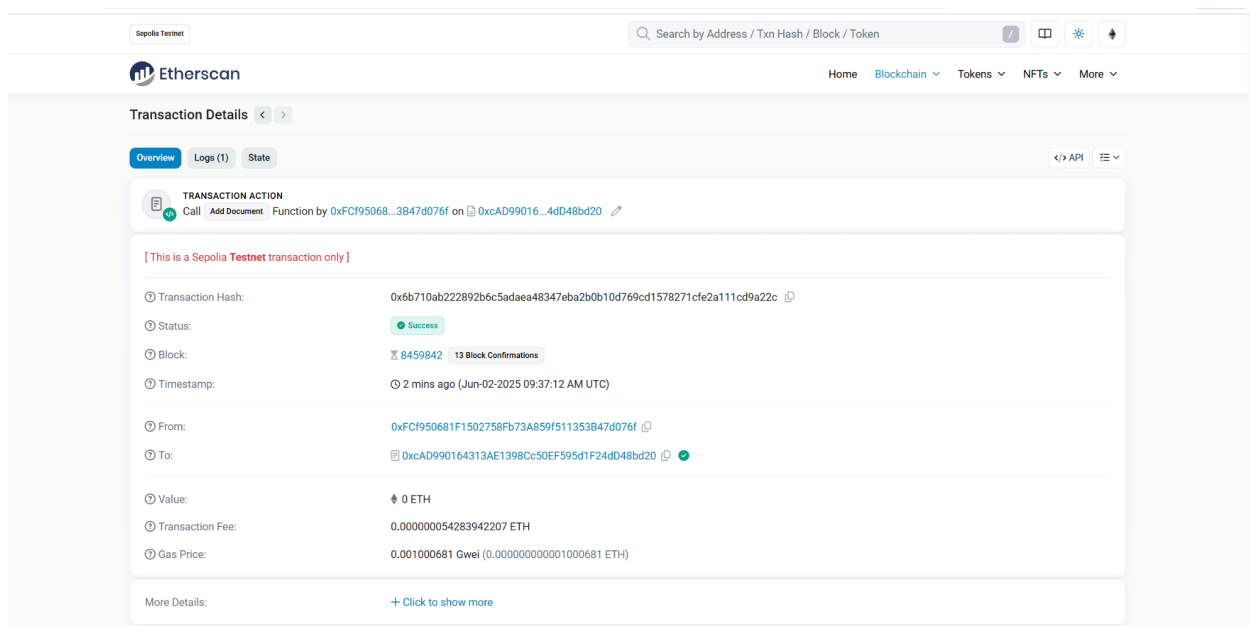


Рисунок 4.7 – Перегляд транзакції запису документу в блокчейн

Крок 6. Повторна верифікація документа вручну

- Передумови: документ уже завантажено та має статус "Не верифіковано" (наприклад, через збій мережі або затримку блокчейна).
- Дії:
 1. Модератор відкриває сторінку документа.
 2. Натискає кнопку «Верифікувати документ повторно».
 3. Система повторно обчислює хеш, порівнює з тим, що записано у блокчейн.
 4. Якщо хеш існує - змінює статус на «Верифіковано».
- Результат: статус документа змінюється, користувач бачить оновлене відображення документа як успішно перевіреного.

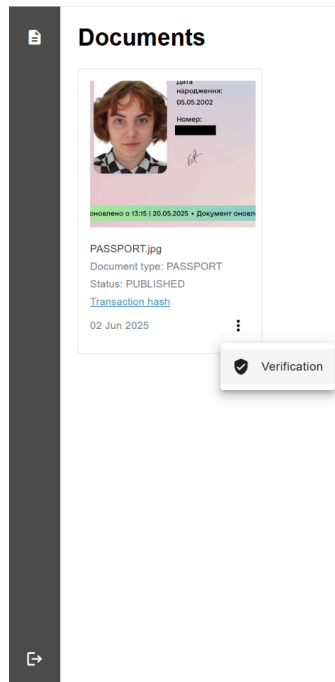


Рисунок 4.8 – Кнопка “Верифікувати документ”

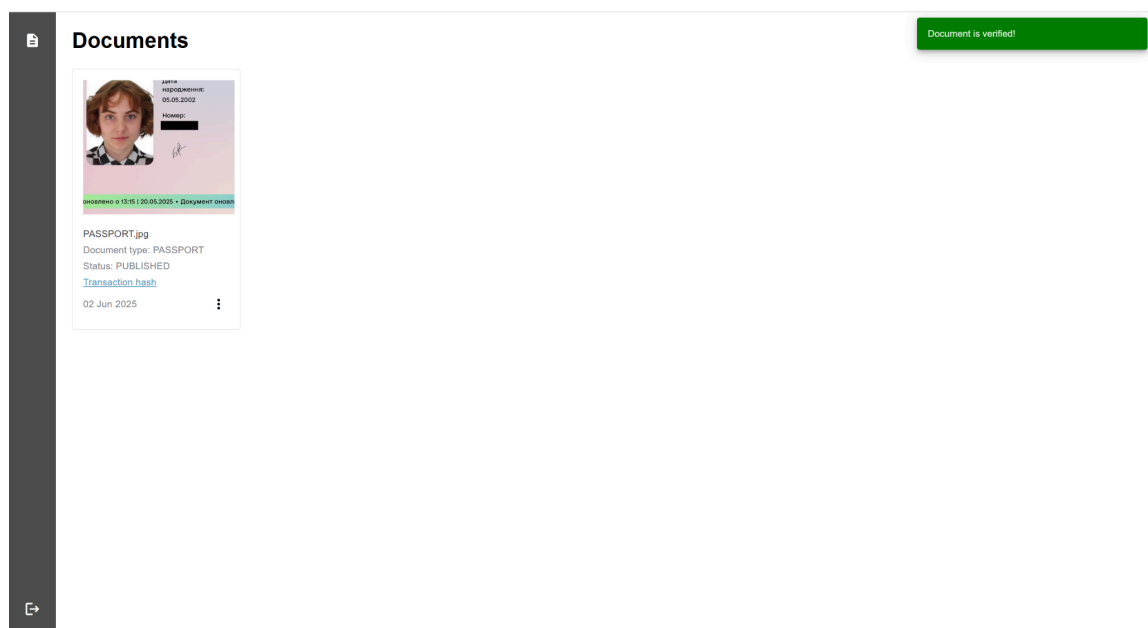


Рисунок 4.9 – Виведення повідомлення про успішну верифікацію документа

Таким чином, у межах контрольного прикладу продемонстровано:

- Повноцінний функціональний цикл документа (від завантаження до перевірки).
- Інтеграцію з блокчейн-експлорером.

- Можливість верифікації документа.

Цей приклад підтверджує відповідність програмного забезпечення вимогам до надійності, безпеки, прозорості та контрольованості процесу роботи з документами.

Висновки до розділу

У даному розділі було здійснено комплексний аналіз якості розробленого програмного забезпечення та описано процеси його тестування. Для оцінки якості були використані обґрунтовані метрики, зокрема цикломатична складність, покриття коду тестами, швидкодія, відповідність нефункціональним вимогам та базові показники безпеки. Проведений аналіз показав, що програмне забезпечення має прийнятний рівень складності, добре покрито тестами та відповідає основним вимогам до сучасних вебсистем.

Окремо було описано процеси тестування, які включають ручне та автоматизоване тестування. Було проведено модульне, інтеграційне, системне та регресійне тестування з використанням відповідних інструментів, таких як Postman, JUnit, Mockito, Karma, Jasmine та інші. Всього реалізовано понад 10 ключових тест-кейсів, що охоплюють основну бізнес-логіку: реєстрацію, авторизацію, завантаження документа, його перевірку та перегляд. Усі тести були успішно виконані, фактичні результати збігалися з очікуваними.

Контрольний приклад охопив основний сценарій використання — завантаження та верифікація документа модератором. Було показано всі етапи: від автентифікації, вибору користувача, завантаження документа до його збереження, перевірки хешу через смарт-контракт, виведення транзакції у блокчейн-експлорері та повторної верифікації. Описано також альтернативні сценарії, які демонструють обробку помилок і виключних ситуацій.

Таким чином, проведені тестування підтвердили правильність роботи основної логіки системи та відповідність заявленим вимогам. Система продемонструвала надійність, коректну інтеграцію з блокчейном, зручність взаємодії для користувача та готовність до практичного використання.

5 РОЗГОРТАННЯ ТА СУПРОВІД ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

5.1 Розгортання програмного забезпечення

Розгортання програмного забезпечення виконується у декілька етапів, з урахуванням особливостей мікросервісної архітектури, використання Kafka, PostgreSQL, Web3-сервісів та взаємодії з блокчейн-мережею. Для зручності розгортання було застосовано інструмент Docker та Docker Compose.

Етапи розгортання:

а) Підготовка інфраструктури:

- 1) Встановлення Docker та Docker Compose.
- 2) Створення .env файлів з налаштуваннями бази даних, Kafka-брокера, Web3 провайдера.

б) Запуск контейнерів:

- 1) Запуск PostgreSQL, Zookeeper, Kafka, бекенд-сервісу (core-service), блокчейн-сервісу (blockchain-service) та фронтенду командою:
`docker-compose up --build`

в) Міграція та ініціалізація даних:

- 1) Автоматична ініціалізація бази даних при запуску контейнера.
- 2) Створення адміністратора через SQL-скрипт або API запит.

г) Деплой смарт-контрактів:

- 1) Смарт-контракти розгортаються вручну або автоматично за допомогою скриптів.
 - 2) Адреса контракту передається до blockchain-service через .env.
- Валідація сервісів:

- 3) Перевірка з'єднання між сервісами.
 - Перевірка доступу до API, Kafka, та запису транзакцій у блокчейн.

5.2 Супровід програмного забезпечення

Розроблене програмне забезпечення не потребує постійного технічного супроводу після розгортання. Система працює автономно, оскільки зберігання та перевірка документів здійснюється на основі смарт-контрактів, які забезпечують прозорість і незмінність даних без залучення додаткових серверних обробок.

Єдиним винятком може бути потреба у модифікації або оновленні смарт-контрактів у майбутньому (наприклад, для розширення функціональності чи адаптації до змін у блокчейн-мережі). У такому випадку достатньо одноразово залучити фахівця з блокчейн-розробки для виконання оновлення.

Таким чином, система є економічно ефективною у довгостроковій перспективі, оскільки не потребує постійної підтримки чи витрат на обслуговування.

Інструкції з розгортання та користування програмним забезпеченням наведені в окремому документі під назвою «Інструкція з експлуатації програмного забезпечення».

Висновки до розділу

У цьому розділі було розглянуто процес розгортання та супроводу розробленого програмного забезпечення, що реалізує систему збереження та верифікації документів з використанням смарт-контрактів. Розгортання

системи виконано з урахуванням мікросервісної архітектури та включає кілька незалежних компонентів: фронтенд, бекенд, блокчейн-сервіс, базу даних, брокер повідомлень Kafka, а також інфраструктуру для взаємодії з блокчейн-мережею.

ВИСНОВКИ

У ході виконання дипломного проєкту було розроблено програмне забезпечення для збереження, перевірки та верифікації електронних документів із використанням блокчейн-технологій. Система реалізована як веб-застосунок, що включає клієнтську частину, серверну частину (бекенд) та окремий блокчейн-сервіс, який взаємодіє зі смарт-контрактами, розгорнутими в тестовій мережі Ethereum Sepolia.

У процесі реалізації було досягнуто поставлену мету — створено безпечне, прозоре та зручне у використанні рішення, яке забезпечує надійність і незмінність даних, що підтверджується шляхом збереження хешів документів у блокчейні. Такий підхід дозволяє користувачам та адміністраторам верифікувати документи без необхідності довіри до централізованої системи, що значно підвищує рівень довіри до процесу документального обміну.

Усі поставлені задачі дипломного проєкту виконано в повному обсязі:

- реалізовано систему ролей із розмежуванням прав доступу;
- розроблено інтерфейс користувача;
- забезпечено зберігання метаданих і файлів у базі даних;
- створено смарт-контракт та блокчейн-сервіс для фіксації хешів;
- налагоджено взаємодію між компонентами системи;
- забезпечено базову безпеку доступу та перевірку автентичності документів.

Розроблене рішення є придатним до практичного використання в організаціях, які працюють з цифровими документами й потребують механізмів верифікації. У перспективі можливим і доцільним є продовження розробки у цій предметній області, зокрема:

- розгортання системи у публічній блокчейн-мережі;
- інтеграція з електронним цифровим підписом (ЕЦП);
- реалізація зручного мобільного доступу;
- розширення функціональності (наприклад, додавання журналу подій або системи сповіщень).

Таким чином, дипломна робота підтверджує практичну доцільність застосування блокчейн-технологій у сфері збереження та верифікації документів та демонструє ефективний підхід до вирішення відповідної прикладної задачі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

- 1) World Economic Forum. The future of digital identity [Електронний ресурс]. – Режим доступу: <https://www.weforum.org> (дата звернення: 12.05.2025).
- 2) Nakamoto S. Bitcoin: A Peer-to-Peer Electronic Cash System [Електронний ресурс]. – Режим доступу: <https://bitcoin.org/bitcoin.pdf> (дата звернення: 12.05.2025).
- 3) Ethereum Foundation. Ethereum Whitepaper [Електронний ресурс]. – Режим доступу: <https://ethereum.org/en/whitepaper/> (дата звернення: 12.05.2025).
- 4) GovTech Singapore. OpenCerts [Електронний ресурс]. – Режим доступу: <https://opencerts.io> (дата звернення: 12.05.2025).
- 5) Dock.io. Verifiable Credentials for Web3 and Enterprises [Електронний ресурс]. – Режим доступу: <https://www.dock.io> (дата звернення: 12.05.2025).
- 6) KILT Protocol. Decentralized Identity [Електронний ресурс]. – Режим доступу: <https://www.kilt.io> (дата звернення: 12.05.2025).
- 7) Ethereum Foundation. Ethereum smart contracts [Електронний ресурс]. – Режим доступу: <https://ethereum.org/en/developers/docs/smart-contracts/> (дата звернення: 12.05.2025).
- 8) OpenZeppelin. UUPS proxies [Електронний ресурс]. – Режим доступу: <https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies#uups-proxies> (дата звернення: 12.05.2025).
- 9) Gnosis Safe. Multisig Wallet Documentation [Електронний ресурс]. – Режим доступу: <https://docs.safe.global/> (дата звернення: 12.05.2025).
- 10) Apache Kafka Documentation [Електронний ресурс]. – Режим доступу: <https://kafka.apache.org/documentation/> (дата звернення: 12.05.2025).

- 11) Spring Boot Reference Documentation [Електронний ресурс]. – Режим доступу: <https://docs.spring.io/spring-boot/docs/current/reference/html/> (дата звернення: 12.05.2025).
- 12) Angular – Overview [Електронний ресурс]. – Режим доступу: <https://angular.io/guide/what-is-angular> (дата звернення: 12.05.2025).
- 13) Google Workspace. Що таке Google Диск? [Електронний ресурс]. – Режим доступу: <https://support.google.com/drive/answer/2424384> (дата звернення: 12.05.2025).
- 14) Google Drive – Security and privacy overview [Електронний ресурс]. – Режим доступу: <https://support.google.com/drive/answer/2450387> (дата звернення: 12.05.2025).
- 15) Ethereum StackExchange. How to store a file hash on Ethereum blockchain [Електронний ресурс]. – Режим доступу: <https://ethereum.stackexchange.com/questions/105/how-to-store-a-file-hash> (дата звернення: 12.05.2025).
- 16) Remix IDE Documentation [Електронний ресурс]. – Режим доступу: <https://remix.ethereum.org/> (дата звернення: 12.05.2025).
- 17) National Institute of Standards and Technology (NIST). Secure Hash Standard (SHS) [Електронний ресурс]. – Режим доступу: <https://csrc.nist.gov/publications/detail/fips/180/4/final> (дата звернення: 12.05.2025).
- 18) Ethereum Yellow Paper. Ethereum: A Secure Decentralised Generalised Transaction Ledger [Електронний ресурс]. – Режим доступу: <https://ethereum.github.io/yellowpaper/paper.pdf> (дата звернення: 12.05.2025).
- 19) JWT.io. Introduction to JSON Web Tokens [Електронний ресурс]. – Режим доступу: <https://jwt.io/introduction/> (дата звернення: 12.05.2025).
- 20) Chainstack. Comparing blockchain platforms for enterprise adoption [Електронний ресурс]. – Режим доступу:

- <https://chainstack.com/blockchain-comparison/> (дата звернення: 12.05.2025).
- 21) Ethereum Sepolia Testnet Overview [Електронний ресурс]. – Режим доступу: <https://sepolia.dev> (дата звернення: 12.05.2025).
 - 22) OpenZeppelin. Proxy Upgrade Patterns [Електронний ресурс]. – Режим доступу: <https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies> (дата звернення: 12.05.2025).
 - 23) OpenZeppelin Blog. Transparent vs UUPS Proxy [Електронний ресурс]. – Режим доступу: <https://blog.openzeppelin.com/the-transparent-proxy-pattern> (дата звернення: 12.05.2025).
 - 24) Richardson C. Microservice architecture patterns [Електронний ресурс]. – Режим доступу: <https://microservices.io/patterns/microservices.html> (дата звернення: 12.05.2025).
 - 25) Web3j Documentation [Електронний ресурс]. – Режим доступу: <https://docs.web3j.io/> (дата звернення: 12.05.2025).
 - 26) Gnosis Safe. Multisig security model [Електронний ресурс]. – Режим доступу: <https://safe.global> (дата звернення: 12.05.2025).
 - 27) Infura Documentation [Електронний ресурс]. – Режим доступу: <https://docs.infura.io> (дата звернення: 12.05.2025).
 - 28) PostgreSQL Official Documentation [Електронний ресурс]. – Режим доступу: <https://www.postgresql.org/docs/> (дата звернення: 12.05.2025).

ДОДАТКИ

ДОДАТОК А Звіт подібності



Дата звіту 6/2/2025
Дата редагування 6/3/2025

Документ прийнятий

Звіт подібності

метадані

Назва організації
National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute
Заголовок
ІП-311_Біленька_ПЗ
Автор
Науковий керівник / Експерт
ІП-311_БіленькаЛісовиченко О.І.
підрозділ
ФІОТ, К-а інформатики та програмної інженерії

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.



10 Довжина фрази для коефіцієнта подібності 2
10109 Кількість слів
78406 Кількість символів

Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

Заміна букв		0
Інтервали		0
Мікропробіли		0
Білі знаки		0
Парафрази (SmartMarks)		78

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

“ЗАТВЕРДЖЕНО”

Завідувач кафедри

_____ Едуард ЖАРІКОВ

“ ____ ” _____ 2025 р.

**ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ ЗБЕРЕЖЕННЯ ТА ВАЛІДАЦІЇ
ДОКУМЕНТІВ НА ОСНОВІ СМАРТ-КОНТРАКТІВ В БЛОКЧЕЙНАХ**

Текст програми

КП.ІП-31101.045470.03.12

“ПОГОДЖЕНО”

Керівник проєкту:

_____ Олег ЛІСОВИЧЕНКО

Нормоконтроль:

_____ Олександр
СТЕЛЬМАХ

Виконавець:

_____ Олеся БІЛЕНЬКА

Київ – 2025

Посилання на репозиторій з повним текстом програмного коду

<https://github.com/obilenka-hacken/isitreal-blockchain-service>

<https://github.com/obilenka-hacken/IsItReal-backend-service>

<https://github.com/obilenka-hacken/IsItReal-smart-contract>

<https://github.com/obilenka-hacken/IsItReal-frontend-service>

Файл DocumentRegistry.sol

Реалізація функціональної задачі “Збереження хешу документів в смарт контракті”

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

import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";
import "@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol";
import "../lib/AccessControl2Step.sol";

contract DocumentRegistry is Initializable, UUPSUpgradeable {
    using AccessControl2Step for AccessControl2Step.RoleData;

    mapping(string => mapping(string => bytes32)) private userDocuments;
    AccessControl2Step.RoleData private roleData;

    address public multisig;

    event DocumentAdded(string indexed userId, string indexed documentType,
        bytes32 documentHash);
    event DocumentUpdated(string indexed userId, string indexed documentType,
        bytes32 newDocumentHash);
    event DocumentDeleted(string indexed userId, string indexed documentType);

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

    modifier onlyModerator() {
        require(roleData.isModerator(msg.sender), "Only moderator can perform
        this action");
        _;
    }
}
```

```

    modifier onlyMultisig() {
        require(msg.sender == multisig, "Only multisig wallet can perform this
action");
        _;
    }

    constructor() {
        roleData.initialize(msg.sender, msg.sender);
    }

    function initialize(address initialAdmin, address initialModerator, address
multisigAddress) public initializer {
        roleData.initialize(initialAdmin, initialModerator);
        multisig = multisigAddress;
    }

    function _authorizeUpgrade(address) internal override onlyAdmin {}

    function addDocument(string memory userId, string memory documentType,
bytes32 documentHash) external onlyModerator {
        require(userDocuments[userId][documentType] == bytes32(0), "Document
already exists");
        userDocuments[userId][documentType] = documentHash;
        emit DocumentAdded(userId, documentType, documentHash);
    }

    function updateDocument(string memory userId, string memory documentType,
bytes32 newDocumentHash) external onlyModerator {
        require(userDocuments[userId][documentType] != bytes32(0), "Document
does not exist");
        userDocuments[userId][documentType] = newDocumentHash;
        emit DocumentUpdated(userId, documentType, newDocumentHash);
    }

    function deleteDocument(string memory userId, string memory documentType)
external onlyAdmin {
        require(userDocuments[userId][documentType] != bytes32(0), "Document
does not exist");
        delete userDocuments[userId][documentType];
        emit DocumentDeleted(userId, documentType);
    }

```

```

        function getDocumentHash(string memory userId, string memory documentType)
external view returns (bytes32) {
    return userDocuments[userId][documentType];
}

function proposeNewAdmin(address newAdmin) external onlyMultisig {
    roleData.proposeAdmin(newAdmin);
}

function acceptAdminRole() external {
    roleData.acceptAdmin();
}

function proposeNewModerator(address newModerator) external onlyMultisig {
    roleData.proposeModerator(newModerator);
}

function acceptModeratorRole() external {
    roleData.acceptModerator();
}

function cancelPendingRoleChange() external {
    roleData.cancelRoleChange();
}
}

```

Файл AccessControl2Step.sol

Реалізація функціональної задачі “Захист доступу до смарт контрактів”

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

```

library AccessControl2Step {
    struct RoleData {
        address admin;
        address pendingAdmin;
        address moderator;
        address pendingModerator;
    }
}

```

```

    event AdminRoleTransferProposed(address indexed currentAdmin, address indexed
newAdmin);

    event AdminRoleTransferred(address indexed previousAdmin, address indexed newAdmin);
    event ModeratorRoleTransferProposed(address indexed currentModerator, address indexed
newModerator);

    event ModeratorRoleTransferred(address indexed previousModerator, address indexed
newModerator);

    event RoleChangeCancelled(string indexed role, address indexed cancelledAddress);


    function initialize(RoleData storage roleData, address initialAdmin, address initialModerator)
internal {
        require(initialAdmin != address(0), "Admin cannot be zero address");
        require(initialModerator != address(0), "Moderator cannot be zero address");
        roleData.admin = initialAdmin;
        roleData.moderator = initialModerator;
    }


    function proposeAdmin(RoleData storage roleData, address newAdmin) internal {
        require(newAdmin != address(0), "New admin cannot be zero address");
        roleData.pendingAdmin = newAdmin;
        emit AdminRoleTransferProposed(roleData.admin, newAdmin);
    }


    function acceptAdmin(RoleData storage roleData) internal {
        require(msg.sender == roleData.pendingAdmin, "Only proposed admin can accept");
        emit AdminRoleTransferred(roleData.admin, roleData.pendingAdmin);
        roleData.admin = roleData.pendingAdmin;
        roleData.pendingAdmin = address(0);
    }


    function proposeModerator(RoleData storage roleData, address newModerator) internal {
        require(newModerator != address(0), "New moderator cannot be zero address");
        roleData.pendingModerator = newModerator;
        emit ModeratorRoleTransferProposed(roleData.moderator, newModerator);
    }

```

```

function acceptModerator(RoleData storage roleData) internal {
    require(msg.sender == roleData.pendingModerator, "Only proposed moderator can
accept");
    emit ModeratorRoleTransferred(roleData.moderator, roleData.pendingModerator);
    roleData.moderator = roleData.pendingModerator;
    roleData.pendingModerator = address(0);
}

```

```

function cancelRoleChange(RoleData storage roleData) internal {
    if (roleData.pendingAdmin == msg.sender) {
        emit RoleChangeCancelled("Admin", roleData.pendingAdmin);
        roleData.pendingAdmin = address(0);
    } else if (roleData.pendingModerator == msg.sender) {
        emit RoleChangeCancelled("Moderator", roleData.pendingModerator);
        roleData.pendingModerator = address(0);
    } else {
        revert("No pending role change for caller");
    }
}

```

```

function isAdmin(RoleData storage roleData, address user) internal view returns (bool) {
    return roleData.admin == user;
}

```

```

function isModerator(RoleData storage roleData, address user) internal view returns (bool) {
    return roleData.moderator == user;
}
}

```

Файл Multisig.sol

Реалізація функціональної задачі “Захист мультисиг доступу до смарт контрактів”

```

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

```

```

contract Multisig {
    address[] public owners;
    uint256 public requiredConfirmations;

    struct Transaction {
        address to;
        bytes data;
        bool executed;
        uint256 confirmations;
    }

    mapping(address => bool) public isOwner;
    mapping(uint256 => mapping(address => bool)) public isConfirmed;
    Transaction[] public transactions;

    modifier onlyOwner() {
        require(isOwner[msg.sender], "Not an owner");
        _;
    }

    constructor(address[] memory _owners, uint256 _requiredConfirmations) {
        require(_owners.length >= _requiredConfirmations, "Invalid confirmations");
        for (uint256 i = 0; i < _owners.length; i++) {
            require(_owners[i] != address(0), "Zero address");
            isOwner[_owners[i]] = true;
        }
        owners = _owners;
        requiredConfirmations = _requiredConfirmations;
    }

    function submitTransaction(address to, bytes memory data) external onlyOwner returns
    (uint256) {
        transactions.push(Transaction({
            to: to,
            data: data,

```

```

        executed: false,
        confirmations: 0
    ));
    uint256 txIndex = transactions.length - 1;
    confirmTransaction(txIndex);
    return txIndex;
}

function confirmTransaction(uint256 txIndex) public onlyOwner {
    require(!transactions[txIndex].executed, "Already executed");
    require(!isConfirmed[txIndex][msg.sender], "Already confirmed");

    isConfirmed[txIndex][msg.sender] = true;
    transactions[txIndex].confirmations++;

    if (transactions[txIndex].confirmations >= requiredConfirmations) {
        executeTransaction(txIndex);
    }
}

function executeTransaction(uint256 txIndex) public onlyOwner {
    Transaction storage txn = transactions[txIndex];
    require(!txn.executed, "Already executed");
    require(txn.confirmations >= requiredConfirmations, "Not enough confirmations");

    txn.executed = true;
    (bool success, ) = txn.to.call(txn.data);
    require(success, "Transaction failed");
}

function getTransactionCount() external view returns (uint256) {
    return transactions.length;
}

function getTransaction(uint256 txIndex) external view returns (

```

```

        address to,
        bytes memory data,
        bool executed,
        uint256 confirmations
    ) {
        Transaction storage txn = transactions[txIndex];
        return (txn.to, txn.data, txn.executed, txn.confirmations);
    }
}

```

Файл UUPSUpgradeable.sol

Реалізація функціональної задачі “UUPS оновлюваність смарт контрактів”

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

```

import {IERC1822Proxiabale} from "@openzeppelin/contracts/interfaces/draft-IERC1822.sol";
import {ERC1967Utils} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Utils.sol";
import {Initializable} from "./Initializable.sol";

```

```

abstract contract UUPSUpgradeable is Initializable, IERC1822Proxiabale {
    /// @custom:oz-upgrades-unsafe-allow state-variable-immutable
    address private immutable __self = address(this);

```

```

    string public constant UPGRADE_INTERFACE_VERSION = "5.0.0";

```

```

    /**
     * @dev The call is from an unauthorized context.
     */
    error UUPSUnauthorizedCallContext();

```

```

    /**
     * @dev The storage `slot` is unsupported as a UUID.
     */
    error UUPSUnsupportedProxiabaleUUID(bytes32 slot);

```

```

    /**
     * @dev Check that the execution is being performed through a delegatecall call and that the
     execution context is
     * a proxy contract with an implementation (as defined in ERC-1967) pointing to self. This
     should only be the case
     * for UUPS and transparent proxies that are using the current contract as their
     implementation. Execution of a
     * function through ERC-1167 minimal proxies (clones) would not normally pass this test, but
     is not guaranteed to
     * fail.
     */
    modifier onlyProxy() {

```

```

        _checkProxy();
    }

    /**
     * @dev Check that the execution is not being performed through a delegate call. This allows
     a function to be
     * callable on the implementing contract but not through proxies.
     */
    modifier notDelegated() {
        _checkNotDelegated();
    }

    function __UUPSUpgradeable_init() internal onlyInitializing {

    }

    function __UUPSUpgradeable_init_unchained() internal onlyInitializing {
    }
    /**
     * @dev Implementation of the ERC-1822 {proxiableUUID} function. This returns the
     storage slot used by the
     * implementation. It is used to validate the implementation's compatibility when performing
     an upgrade.
     *
     * IMPORTANT: A proxy pointing at a proxiable contract should not be considered proxiable
     itself, because this risks
     * bricking a proxy that upgrades to it, by delegating to itself until out of gas. Thus it is critical
     that this
     * function revert if invoked through a proxy. This is guaranteed by the `notDelegated`
     modifier.
     */
    function proxiableUUID() external view virtual notDelegated returns (bytes32) {
        return ERC1967Utils.IMPLEMENTATION_SLOT;
    }

    /**
     * @dev Upgrade the implementation of the proxy to `newImplementation`, and subsequently
     execute the function call
     * encoded in `data`.
     *
     * Calls {_authorizeUpgrade}.
     *
     * Emits an {Upgraded} event.
     *
     * @custom:oz-upgrades-unsafe-allow-delegatecall
     */
    function upgradeToAndCall(address newImplementation, bytes memory data) public payable
    virtual onlyProxy {
        _authorizeUpgrade(newImplementation);
        _upgradeToAndCallUUPS(newImplementation, data);
    }

```

```

/**
 * @dev Reverts if the execution is not performed via delegatecall or the execution
 * context is not of a proxy with an ERC-1967 compliant implementation pointing to self.
 */
function _checkProxy() internal view virtual {
    if (
        address(this) == __self || // Must be called through delegatecall
        ERC1967Utils.getImplementation() != __self // Must be called through an active proxy
    ) {
        revert UUPSUnauthorizedCallContext();
    }
}

/**
 * @dev Reverts if the execution is performed via delegatecall.
 * See {notDelegated}.
 */
function _checkNotDelegated() internal view virtual {
    if (address(this) != __self) {
        // Must not be called through delegatecall
        revert UUPSUnauthorizedCallContext();
    }
}

/**
 * @dev Function that should revert when `msg.sender` is not authorized to upgrade the
 * contract. Called by
 * {upgradeToAndCall}.
 *
 * Normally, this function will use an xref:access.adoc[access control] modifier such as
 * {Ownable-onlyOwner}.
 *
 * ``solidity
 * function _authorizeUpgrade(address) internal onlyOwner {}
 * ``
 */
function _authorizeUpgrade(address newImplementation) internal virtual;

/**
 * @dev Performs an implementation upgrade with a security check for UUPS proxies, and
 * additional setup call.
 *
 * As a security check, {proxiableUUID} is invoked in the new implementation, and the return
 * value
 * is expected to be the implementation slot in ERC-1967.
 *
 * Emits an {IERC1967-Upgraded} event.
 */
function _upgradeToAndCallUUPS(address newImplementation, bytes memory data) private
{
    try IERC1822Proxiable(newImplementation).proxiableUUID() returns (bytes32 slot) {
        if (slot != ERC1967Utils.IMPLEMENTATION_SLOT) {
            revert UUPSUnsupportedProxiableUUID(slot);
        }
    }
}

```

```

    }
    ERC1967Utils.upgradeToAndCall(newImplementation, data);
  } catch {
    // The implementation is not UUPS
    revert ERC1967Utils.ERC1967InvalidImplementation(newImplementation);
  }
}
}
}

```

Файл BlockchainService.java

Реалізація функціональної задачі “Виклик смарт контрактів для збереження та видалення файлів з блокчейну”

```

package com.example.blockchain.service;

import java.math.BigInteger;
import java.util.Arrays;
import java.util.Collections;
import java.util.UUID;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
import org.web3j.abi.FunctionEncoder;
import org.web3j.abi.datatypes.Function;
import org.web3j.abi.datatypes.Utf8String;
import org.web3j.abi.datatypes.generated.Bytes32;
import org.web3j.crypto.Credentials;
import org.web3j.protocol.Web3j;
import org.web3j.protocol.core.methods.response.EthGasPrice;
import org.web3j.tx.RawTransactionManager;
import org.web3j.tx.TransactionManager;
import org.web3j.tx.gas.ContractGasProvider;
import org.web3j.tx.gas.DefaultGasProvider;
import org.web3j.utils.Numeric;

import com.example.blockchain.dto.DocumentEvent;
import com.example.blockchain.dto.TransactionEvent;
import com.example.blockchain.model.DocumentType;
import com.example.blockchain.model.OperationType;

import com.example.blockchain.service.KafkaProducerService;

@Service
public class BlockchainService {

    private final Web3j web3j;
    private final Credentials credentials;
    private final String contractAddress;
    private final TransactionManager txManager;
    private final ContractGasProvider gasProvider;
    private final KafkaProducerService kafkaProducerService;

```

```

public BlockchainService(
    Web3j web3j,
    @Value("${blockchain.owner.privateKey}") String privateKey,
    @Value("${blockchain.contractAddress}") String contractAddress,
    KafkaProducerService kafkaProducerService) {
    this.web3j = web3j;
    this.credentials = Credentials.create(privateKey);
    this.contractAddress = contractAddress;

    this.txManager = new RawTransactionManager(web3j, credentials, 11155111); // Sepolia
    chainId = 11155111
    this.gasProvider = new DefaultGasProvider();
    this.kafkaProducerService = kafkaProducerService;
}

public String processDocumentEvent(DocumentEvent documentEvent) throws Exception {
    OperationType operation = documentEvent.getOperation();

    switch (operation) {
        case ADD:
            return sendTransaction("addDocument", documentEvent.getUserId(),
documentEvent.getDocumentType(), documentEvent.getDocumentHash());
        case UPDATE:
            return sendTransaction("updateDocument", documentEvent.getUserId(),
documentEvent.getDocumentType(), documentEvent.getDocumentHash());
        case DELETE:
            return sendTransaction("deleteDocument", documentEvent.getUserId(),
documentEvent.getDocumentType(), null);
        default:
            throw new UnsupportedOperationException("Unsupported operation type: " +
operation);
    }
}

private String sendTransaction(String methodName, UUID userId, DocumentType docType,
String hash) throws Exception {
    Function function;

    if ("deleteDocument".equals(methodName)) {
        function = new Function(
            methodName,
            Arrays.asList(
                new Utf8String(userId.toString()),
                new Utf8String(docType.toString())
            ),
            Collections.emptyList()
        );
    } else {
        byte[] hashBytes = Numeric.hexStringToByteArray(hash);
        if (hashBytes.length != 32) {
            throw new IllegalArgumentException("Document hash must be exactly 32 bytes");
        }
    }
}

```

```

    }

    function = new Function(
        methodName,
        Arrays.asList(
            new Utf8String(userId.toString()),
            new Utf8String(docType.toString()),
            new Bytes32(hashBytes)
        ),
        Collections.emptyList()
    );
}

String encodedFunction = FunctionEncoder.encode(function);

// Отримати актуальну ціну gas
EthGasPrice ethGasPrice = web3j.ethGasPrice().send();
BigInteger gasPrice = ethGasPrice.getGasPrice();
BigInteger gasLimit = BigInteger.valueOf(300_000); // За потреби підкоригуй

String transaction = txManager.sendTransaction(
    gasPrice,
    gasLimit,
    contractAddress,
    encodedFunction,
    BigInteger.ZERO
).getTransactionHash();

TransactionEvent event = new TransactionEvent(userId, hash, transaction);
System.out.println("EvEnt" + event);
kafkaProducerService.sendTransactionEvent(event);

return transaction;
}
}

```

Файл **KafkaConsumerService.java**

Реалізація функціональної задачі “Отримання message від message broker про взаємодію з документом”

```

package com.example.blockchain.service;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;

```

```

import com.example.blockchain.dto.DocumentEvent;
import com.fasterxml.jackson.databind.ObjectMapper;

@Service
public class KafkaConsumerService {
    private static final Logger logger = LoggerFactory.getLogger(KafkaConsumerService.class);
    private final ObjectMapper objectMapper = new ObjectMapper(); // JSON парсер

    private final BlockchainService blockchainService;

    @Autowired
    public KafkaConsumerService(BlockchainService blockchainService) {
        this.blockchainService = blockchainService;
    }

    @KafkaListener(topics = "document.events", groupId = "blockchain-service-group")
    public void listen(ConsumerRecord<String, String> record) throws Exception {
        try {
            DocumentEvent event = objectMapper.readValue(record.value(), DocumentEvent.class);
            logger.info("Received Document Event: {}", event);

            processDocumentEvent(event);
        } catch (Exception e) {
            logger.error("Error parsing Kafka message: {}", record.value(), e);
        }
    }

    private void processDocumentEvent(DocumentEvent event) throws Exception {
        logger.info("Processing event for user: {}, operation: {}", event.getUserId(),
            event.getOperation());

        blockchainService.processDocumentEvent(event);
    }
}

```

Файл KafkaProducerService.java

Реалізація функціональної задачі “Відправка message до message broker з транзакцією запису документу”

```

package com.example.blockchain.service;

import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Service;

import com.example.blockchain.dto.TransactionEvent;
import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;

@Service
public class KafkaProducerService {
    private final KafkaTemplate<String, String> kafkaTemplate;
    private final ObjectMapper objectMapper;

```

```

    public KafkaProducerService(KafkaTemplate<String, String> kafkaTemplate, ObjectMapper
objectMapper) {
        this.kafkaTemplate = kafkaTemplate;
        this.objectMapper = objectMapper;
    }

    public void sendTransactionEvent(TransactionEvent event) {
        try {
            String message = objectMapper.writeValueAsString(event);
            kafkaTemplate.send("transaction.events", message);
        } catch (JsonProcessingException e) {
            throw new RuntimeException("Failed to serialize transaction event", e);
        }
    }
}

```

Файл **ethers.service.ts**

Реалізація функціональної задачі “Виклик смарт контракту для отримання хешу”

```

import { Injectable } from '@angular/core';
import { ethers } from 'ethers';
import { from, Observable } from 'rxjs';
import { environment } from '../environments/environment';

@Injectable({
    providedIn: 'root'
})
export class EthereumService {
    private provider: ethers.JsonRpcProvider | null = null;
    private contract: ethers.Contract | null = null;

    private readonly infuraUrl =
'https://sepolia.infura.io/v3/3b930af8a4ed4ef9a1be53b13ad0215a';
    private readonly contractAddress = environment.contractAddress;
    private readonly abi = [
        {
            "inputs": [
                { "internalType": "string", "name": "userId", "type": "string" },
                { "internalType": "string", "name": "documentType", "type": "string" }
            ],
            "name": "getDocumentHash",
            "outputs": [
                { "internalType": "bytes32", "name": "", "type": "bytes32" }
            ],
            "stateMutability": "view",
            "type": "function"
        }
    ];

    constructor() {

```

```

    this.initProvider();
}

public getDocumentHash(userId: string, documentType: string): Observable<string> {
    return from(this.fetchDocumentHash(userId, documentType));
}

private initProvider(): void {
    try {
        this.provider = new ethers.JsonRpcProvider(this.infuraUrl);
        this.contract = new ethers.Contract(
            this.contractAddress,
            this.abi,
            this.provider
        );
        console.log('Provider and contract initialized with Infura');
    } catch (error) {
        console.error('Failed to initialize provider:', error);
    }
}

private async fetchDocumentHash(userId: string, documentType: string): Promise<string> {
    // Переконаємося, що контракт і провайдер ініціалізовані
    if (!this.contract || !this.provider) {
        this.initProvider();

        // Повторна перевірка після ініціалізації
        if (!this.contract || !this.provider) {
            throw new Error('Contract or provider not initialized');
        }
    }

    try {
        const hashBytes = await this.contract['getDocumentHash'](userId, documentType);

        console.log('Raw hash result:', hashBytes);

        if (typeof hashBytes === 'string') {
            return hashBytes;
        }

        const hashHex = ethers.hexlify(hashBytes);
        console.log('Converted hash hex:', hashHex);

        return hashHex;
    } catch (error) {
        console.error('Error fetching document hash:', error);

        if ((error as any).code === 'BAD_DATA') {
            console.error('Contract result decoding error details:', {
                error: error,
                message: 'Output type mismatch. Check if ABI output type matches contract'
            });
        }
    }
}

```

```

    }
    throw error;
  }
}
}

```

Файл **documents.component.ts**

Реалізація функціональної задачі “Верифікація документу”

```

verificationDocument(document: IDocument) {
  this.ethereumService.getDocumentHash(this.userId!, document.type)
    .subscribe((data) => {
      const hash = data.slice(2);

      if (document.hash === hash) {
        this._snackBar.open('Document is verified!', "",
          {
            duration: 2000,
            panelClass: ['snackbar-success']
          }
        );
      } else {
        this._snackBar.open('Document is not verified!', "",
          {
            duration: 2000,
            panelClass: ['snackbar-error']
          }
        );
      }
    })
}

```

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

“ЗАТВЕРДЖЕНО”

Завідувач кафедри

_____ Едуард ЖАРІКОВ

“ ____ ” _____ 2025 р.

**ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ ЗБЕРЕЖЕННЯ ТА ВАЛІДАЦІЇ
ДОКУМЕНТІВ НА ОСНОВІ СМАРТ-КОНТРАКТІВ В БЛОКЧЕЙНАХ**

Програма та методика тестування

КП.ІП-з1101.045470.04.51

“ПОГОДЖЕНО”

Керівник проєкту:

_____ Олег ЛІСОВИЧЕНКО

Нормоконтроль:

_____ Михайло ТКАЧ

Виконавець:

_____ Олеся БІЛЕНЬКА

Київ – 2025

ЗМІСТ

1	ОБ'ЄКТ ВИПРОБУВАНЬ	3
2	МЕТА ТЕСТУВАННЯ	4
3	МЕТОДИ ТЕСТУВАННЯ	5
4	ЗАСОБИ ТА ПОРЯДОК ТЕСТУВАННЯ	7

1 ОБ'ЄКТ ВИПРОБУВАНЬ

Об'єктом випробування є розроблене програмне забезпечення для збереження та валідації електронних документів із використанням смарт-контрактів у блокчейн-мережі.

Програмне забезпечення складається з таких компонентів:

- вебклієнт, реалізований на Angular, призначений для взаємодії користувача із системою;
- бекенд-сервіс, реалізований на платформі Spring Boot;
- смарт-контракт, розгорнутий у тестовій мережі Ethereum (EVM-сумісній);
- інфраструктурні компоненти, зокрема PostgreSQL для зберігання метаданих та Apache Kafka для обміну повідомленнями між сервісами.

Програмне забезпечення тестувалося в середовищі Docker, з використанням операційної системи Ubuntu 22.04 та веббраузерів Google Chrome, Mozilla Firefox. Також було перевірено коректність роботи у середовищі Windows 10.

2 МЕТА ТЕСТУВАННЯ

Метою тестування є наступне:

- перевірка правильності роботи програмного забезпечення відповідно до функціональних вимог;
- перевірка коректного збереження та валідації документів через смарт-контракти;
- перевірка взаємодії між компонентами системи (фронтенд, бекенд, блокчейн, база даних, Kafka);
- перевірка сумісності вебзастосунку з останніми версіями сучасних браузерів (Google Chrome, Mozilla Firefox, Opera);
- перевірка сумісності системи з різними операційними системами (Windows, Linux);
- виявлення помилок, збоїв або недоліків у логіці роботи, зовнішньому вигляді або UX з метою подальшого їх усунення;
- перевірка зручності та інтуїтивності графічного інтерфейсу користувача;
- перевірка стабільності роботи при паралельних діях різних ролей (користувач, модератор, адміністратор);
- тестування обробки помилкових ситуацій (некоректне завантаження документа, відсутність відповіді з блокчейну тощо);
- перевірка безпеки (зокрема доступу до функціоналу згідно з роллю користувача).

3 МЕТОДИ ТЕСТУВАННЯ

Відповідно до піраміди тестування, для програмного забезпечення були обрані наступні типи тестів:

- Юніт-тестування (Unit testing) — основа піраміди, тестуються окремі модулі або функції, зокрема логіка хешування документа, перевірка ролей, перевірка даних на бекенді;
- Інтеграційне тестування (Integration testing) — тестування взаємодії між компонентами: бекендом, базою даних, Kafka, смарт-контрактом;
- Системне тестування (System testing) — тестування повної системи в Docker-середовищі з реальними сценаріями роботи користувача;
- UI/мануальне тестування (Manual/UI testing) — перевірка користувацького інтерфейсу та зручності взаємодії;
- E2E тестування (End-to-End) — симуляція повного користувацького флоу: завантаження документа, перевірка через блокчейн, перевірка результату.

Для тестування програмного забезпечення використовувалися такі методи:

- Статичне тестування – проводився огляд коду, документації та конфігураційних файлів на відповідність вимогам, стандартам кодування та архітектурним принципам;
- Динамічне тестування – застосовувалось на етапі виконання програми: запускалися різні сценарії користувацької взаємодії з метою виявлення помилок і збоїв;
- Функціональне тестування – здійснювалася перевірка основних функцій системи: автентифікація, завантаження документа, збереження хешу, перевірка достовірності через смарт-контракт, управління ролями;

- Системне тестування – проводилася перевірка всієї системи в інтегрованому середовищі Docker Compose із повним набором сервісів;
 - Мануальне тестування – проводилося без автоматизованих інструментів, із використанням вручну підготовлених тест-кейсів для перевірки бізнес-логіки;
 - Тестування «чорної скриньки» – тестування функціональності системи без аналізу внутрішньої реалізації, наприклад, перевірка відповіді при завантаженні файлу;
 - Тестування «білої скриньки» – перевірка внутрішньої логіки, контролю прав доступу, роботи Kafka-обробників, функцій валідації;
- Тестування «сірої скриньки» – комбінований підхід: враховувався загальний бізнес-флоу і часткове знання про реалізацію логіки, наприклад, як саме хеш записується у смарт-контракт і потім перевіряється.

4 ЗАСОБИ ТА ПОРЯДОК ТЕСТУВАННЯ

Під час проведення тестування використовувалися наступні допоміжні засоби:

- Стороннє програмне забезпечення:
 - Postman — для перевірки REST API запитів до бекенд-сервісу;
 - Metamask — для взаємодії зі смарт-контрактами в мережі Ethereum;
 - Ganache / Hardhat — для розгортання та тестування смарт-контрактів у локальному середовищі;
 - Docker / Docker Compose — для розгортання всіх сервісів у тестовому середовищі;
 - Swagger UI — для перегляду структури та тестування API.
- Утиліти:
 - JUnit — для написання та виконання юніт-тестів бекенд-логіки;
 - Browser Developer Tools (Chrome, Firefox) — для аналізу поведінки фронтенду та мережевих запитів;
 - PgAdmin / DBeaver — для контролю та перевірки стану бази даних PostgreSQL;
 - Kafka CLI tools — для моніторингу обміну повідомленнями між сервісами.

Порядок проведення тестування

Тестування проводилося відповідно до наступного плану:

1. Статичний аналіз коду на предмет помилок, невикористаних фрагментів, дотримання архітектурних вимог;
2. Динамічне тестування на відповідність функціональним вимогам:
 - автентифікація та розмежування прав доступу;
 - завантаження, збереження та перевірка документів;
 - запис і зчитування хешу зі смарт-контракту;

- взаємодія через Kafka;
3. Функціональне тестування для перевірки роботи кожного окремого модуля (користувацький інтерфейс, API, обробка транзакцій, база даних);
Системне тестування — перевірка повної роботи всіх сервісів у зв'язці (frontend + backend + blockchain + Kafka + PostgreSQL);
 4. Кросбраузерне тестування — перевірка роботи інтерфейсу у Google Chrome, Mozilla Firefox, Opera;
 5. Кросплатформне тестування — запуск та тестування застосунку на Windows 10, Ubuntu 22.04;
 6. Тестування зручності використання — оцінка інтерфейсу з погляду кінцевого користувача;
 7. Тестування на граничних і помилкових вхідних даних (наприклад, великі файли, некоректні формати, спроба подвійного завантаження тощо);
 8. Тестування відновлення після збоїв — перевірка стабільності роботи після перезапуску контейнерів або втрати підключення до блокчейну;
 9. Тестування безпеки — перевірка відсутності несанкціонованого доступу до функцій, перевірка обмеження дій згідно з роллю користувача.

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

“ЗАТВЕРДЖЕНО”

Завідувач кафедри

_____ Едуард ЖАРІКОВ

“ ____ ” _____ 2025 р.

**ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ ЗБЕРЕЖЕННЯ ТА ВАЛІДАЦІЇ
ДОКУМЕНТІВ НА ОСНОВІ СМАРТ-КОНТРАКТІВ В БЛОКЧЕЙНАХ**

Керівництво користувача

КП.ІП-з1101.045470.05.34

“ПОГОДЖЕНО”

Керівник проєкту:

_____ Олег ЛІСОВИЧЕНКО

Нормоконтроль:

_____ Олександр
СТЕЛЬМАХ

Виконавець:

_____ Олеся БІЛЕНЬКА

Київ – 2025

ЗМІСТ

1 Призначення програми	3
2 ПІДГОТОВКА ДО РОБОТИ З ПРОГРАМНИМ ЗАБЕЗПЕЧЕННЯМ	4
2.1 Системні вимоги для коректної роботи	4
Завантаження застосунку	5
3 Виконання програми	7
Основні кроки роботи з програмою:	7
Завершення роботи із застосунком:	9

1 ПРИЗНАЧЕННЯ ПРОГРАМИ

Програмне забезпечення, розроблене в межах дипломного проєкту, призначене для зберігання, перевірки достовірності та відстеження електронних документів із використанням технології блокчейн та смарт-контрактів.

Система складається з вебінтерфейсу, бекенд-сервісу та смарт-контракту, що взаємодіють між собою для забезпечення цілісності та незмінності даних. Кожен документ, завантажений у систему, проходить процес хешування, після чого його хеш записується у смарт-контракт у мережі Ethereum. Це дозволяє перевірити справжність документів у будь-який момент без потреби довіряти центральному серверу.

Кінцева збірка програмного забезпечення включає:

- Вебзастосунок, реалізований за допомогою Angular;
- Бекенд-сервіс на базі Spring Boot;
- Смарт-контракт на Solidity, розгорнутий у тестовій EVM-сумісній мережі;
- Налаштовану взаємодію між сервісами через Kafka;
- Механізми автентифікації, управління ролями (адміністратор, модератор, користувач) та відображення статусу документів;
- Docker Compose конфігурацію для зручного розгортання всієї системи.

Вміст репозиторію містить:

- Вихідний код усіх компонентів системи;
- Інструкції зі збирання та запуску;
- Файли для розгортання інфраструктури (Dockerfile, docker-compose.yml);

2 ПІДГОТОВКА ДО РОБОТИ З ПРОГРАМНИМ ЗАБЕЗПЕЧЕННЯМ

2.1 Системні вимоги для коректної роботи

Для розгортання та коректного функціонування системи необхідне виконання таких мінімальних системних вимог:

Клієнтська частина (Angular):

- Браузер з підтримкою сучасних вебтехнологій (Google Chrome, Firefox, Microsoft Edge);
- Доступ до Інтернету;
- Роздільна здатність екрану від 1280×720.

Серверна частина (Spring Boot сервіс):

- Процесор: 2 ядра, 2.0 ГГц або більше;
- Оперативна пам'ять: від 4 ГБ;
- Дисковий простір: мінімум 500 МБ;
- Java 17 або новіша;
- Платформа: Linux/macOS/Windows (рекомендовано Linux);
- Docker (рекомендовано для зручного розгортання).

База даних (PostgreSQL):

- PostgreSQL 14 або новіша;
- Вільне місце для зберігання даних: від 1 ГБ (залежно від обсягу документів);
- Можливість створення користувачів та баз даних.

Блокчейн-інфраструктура:

- Доступ до Ethereum testnet або локального вузла (наприклад, Hardhat або Ganache);
- Web3-сумісне середовище;

- Інтеграція з Gnosis Safe або іншим multisig (якщо використовується).

Kafka:

- Apache Kafka 3.0 або вище;
- Потрібен ZooKeeper (або використання KRaft у нових версіях Kafka);
- Порт 9092 відкритий між сервісами.

Рекомендоване середовище розгортання (для всіх компонентів):

- Docker + Docker Compose;
- Вільний порт для кожного сервісу:
 - Angular: 4200
 - Spring Boot: 8080
 - PostgreSQL: 5432
 - Kafka: 9092
 - Smart contract API (якщо окремо): 8545 або 3000.

Завантаження застосунку

Система розгортається локально або на сервері за допомогою Docker Compose. Щоб розпочати роботу, необхідно виконати такі кроки:

1. Клонувати репозиторій з офіційного джерела (GitHub).
Встановити Docker та Docker Compose.
Перейти в кореневу директорію проєкту та виконати команду:
`docker-compose up --build`
2. Дочекатися завершення ініціалізації всіх сервісів (Angular, Spring Boot, PostgreSQL, Kafka).
3. Перейти у веббраузері за адресою `http://localhost:8080` для відкриття клієнтського інтерфейсу.
4. Авторизуватись або зареєструватися під обліковим записом користувача, модератора або адміністратора.

Усі налаштування (наприклад, URL-адреси API, ключі доступу до блокчейну) можуть бути змінені у відповідних конфігураційних файлах або змінних середовища (.env).

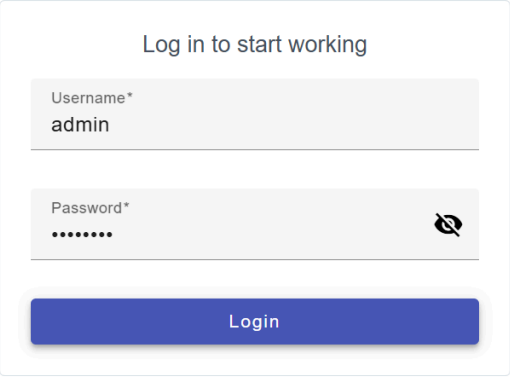
3 ВИКОНАННЯ ПРОГРАМИ

Після запуску програмного застосунку користувач потрапляє на екран авторизації, де необхідно ввести логін та пароль. У разі успішної автентифікації користувач перенаправляється до головної сторінки інтерфейсу, яка відповідає його ролі у системі (користувач, модератор або адміністратор).

Основні кроки роботи з програмою:

1. Авторизація:

- Введення облікових даних у форму входу.
- Успішна перевірка через бекенд-сервіс.



Log in to start working

Username*
admin

Password*
.....

Login

Рисунок 3.1 – Авторизація користувача

2. Для звичайного користувача:

- Перегляд списку документів, завантажених від імені користувача.
- Можливість перевірити достовірність кожного документа — здійснюється запит до смарт-контракту для підтвердження наявності хешу в блокчейні.

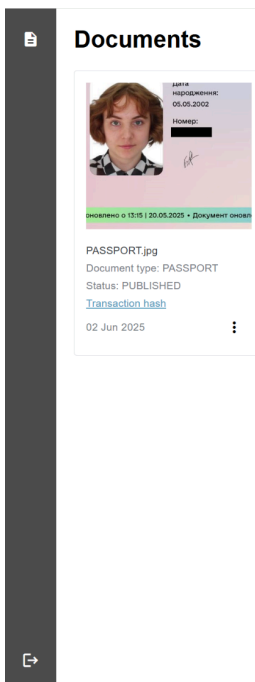


Рисунок 3.2 – Сторінка користувача

3. Для модератора:

- Завантаження нового документа для збереження та верифікації.

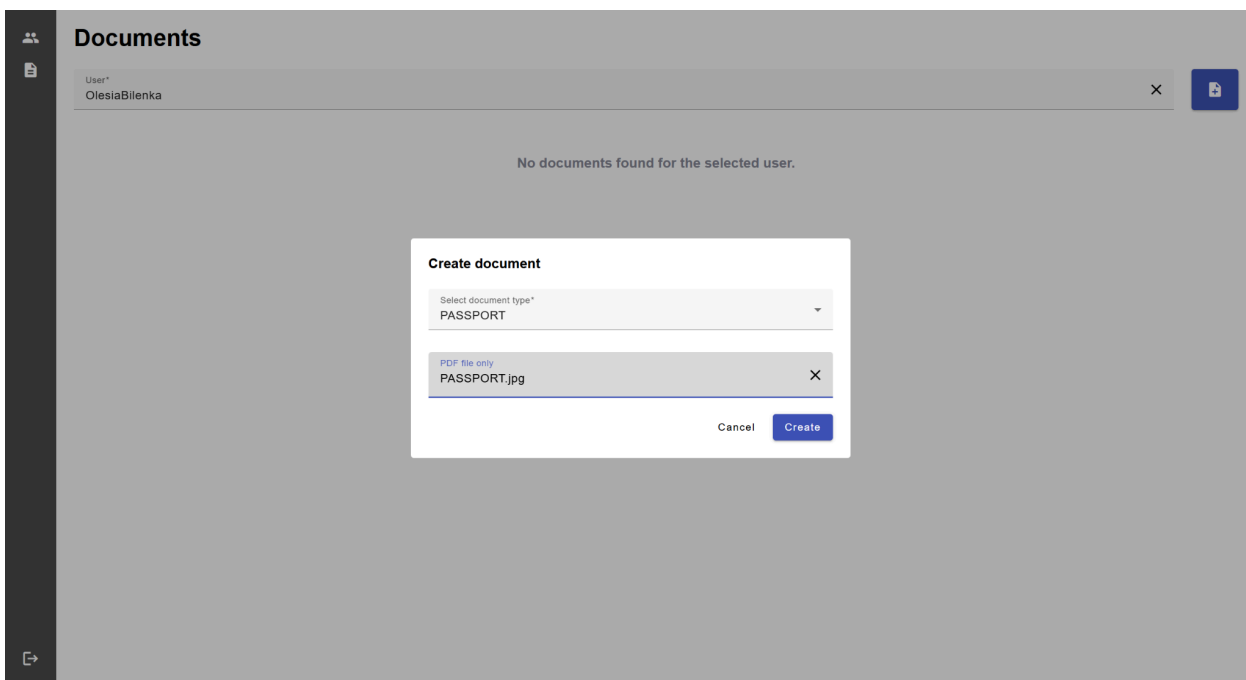


Рисунок 3.3 – Форма завантаження документа

4. Для адміністратора:

- Управління обліковими записами (створення, редагування, блокування користувачів).

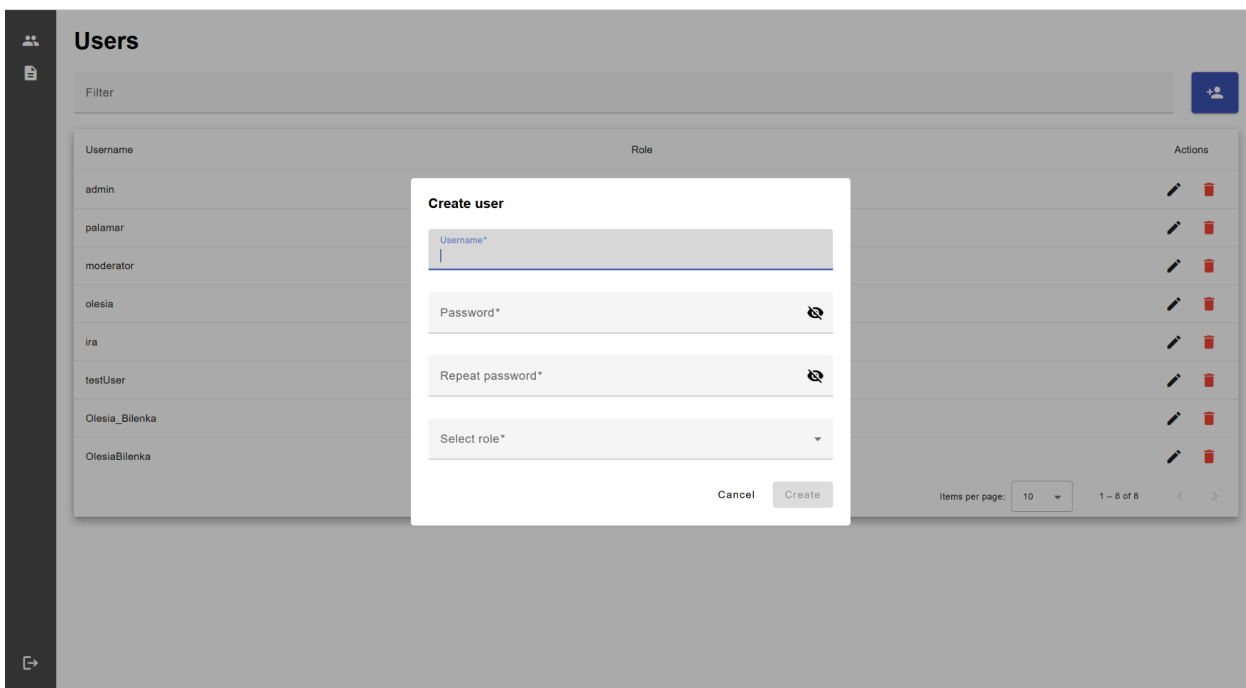


Рисунок 3.4 – Сторінка редагування користувачів

Завершення роботи із застосунком:

Для виходу із системи потрібно натиснути кнопку "Вийти", після чого користувач повертається на екран авторизації. Локальні дані сесії очищаються автоматично.

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

“ЗАТВЕРДЖЕНО”

Завідувач кафедри

_____ Едуард ЖАРІКОВ

“ ____ ” _____ 2025 р.

**ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ ЗБЕРЕЖЕННЯ ТА ВАЛІДАЦІЇ
ДОКУМЕНТІВ НА ОСНОВІ СМАРТ-КОНТРАКТІВ В БЛОКЧЕЙНАХ**

Графічний матеріал

КП.ІП-з1101.045470.06.99

“ПОГОДЖЕНО”

Керівник проекту:

_____ Олег ЛІСОВИЧЕНКО

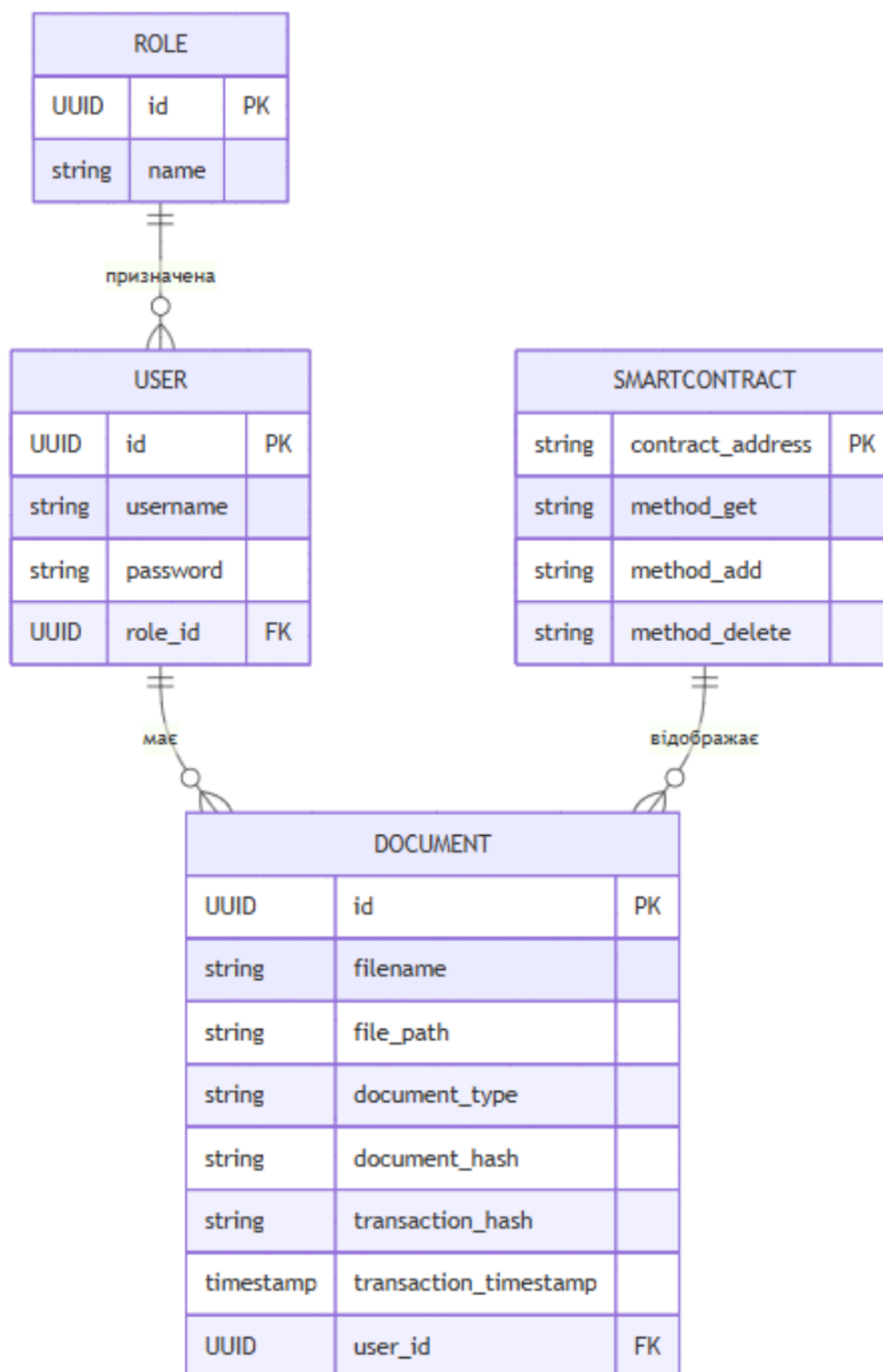
Нормоконтроль:

_____ Олександр
СТЕЛЬМАХ

Виконавець:

_____ Олеся БІЛЕНЬКА

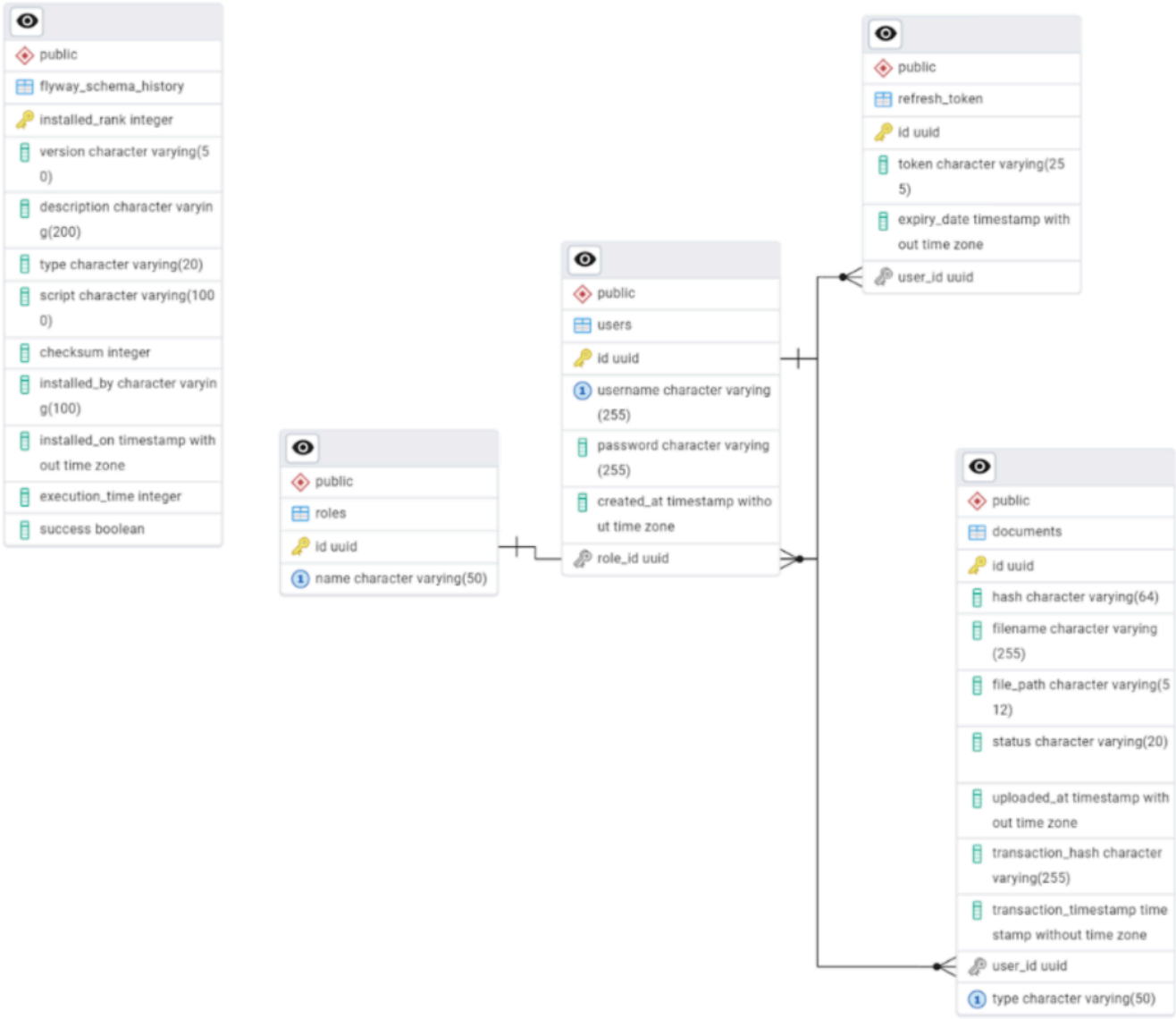
Київ – 2025



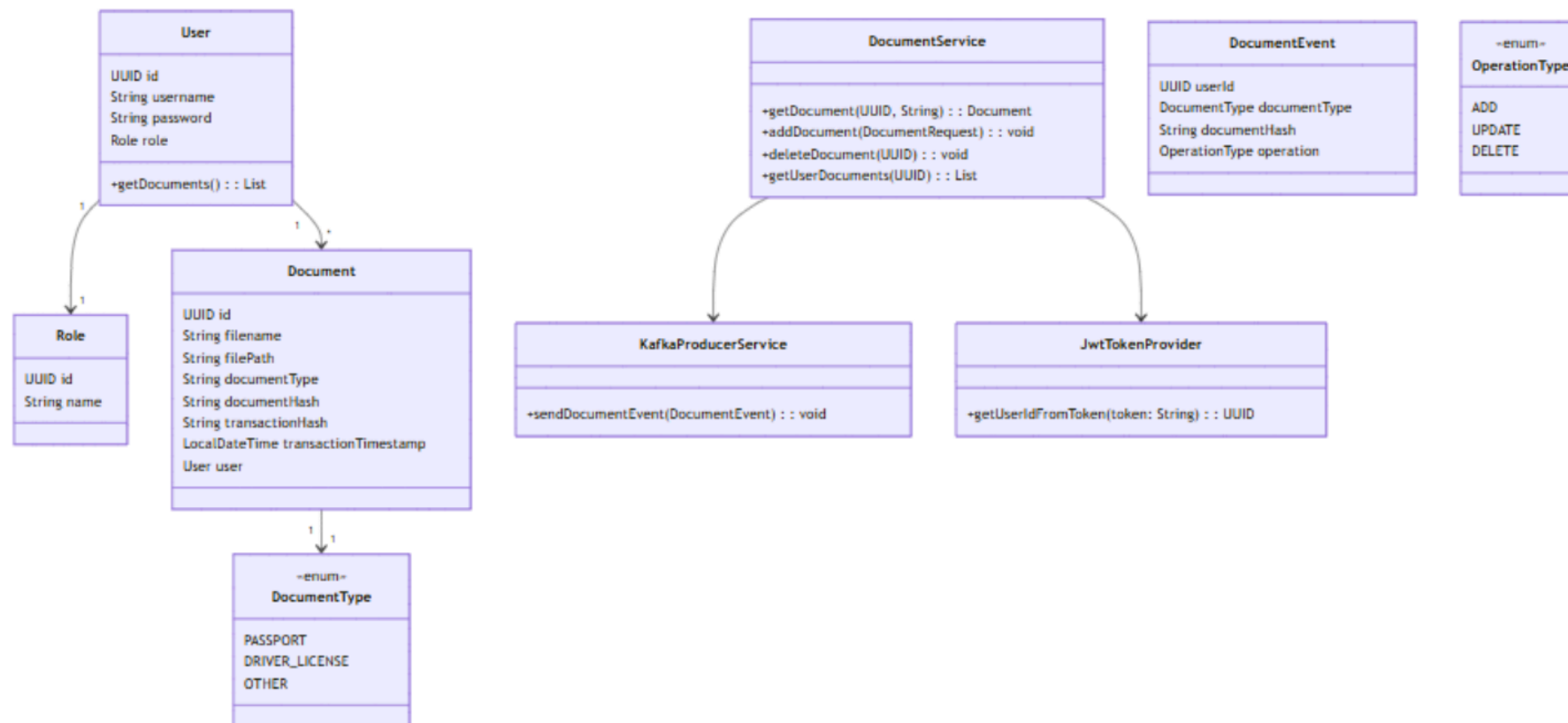
						КПІ.ІПІ-з1101.045470.06.99.ДС			
						ER діаграма сутностей	Лит.	Маса	Масштаб
Зм.	Арк.	№ докум.	Підп.	Дата					
Розробила	Біленька О.Р.								
Перевірів	Лісовиченко О.І.								
Т. контр.							Аркуш	Аркушів	
Н. контр.	Степняк О.П.				Програмне забезпечення для збереження та валідації документів на основі смарт-контрактів в блокчейнах	КПІ ім.Ігоря Сікорського Кафедра ІПІ гр. ІПІ-311			
Затвердив	Жаріков Е.В.								



					КПІ.ПІ-з1101.045470.06.99.CCB			
Зм.	Арк.	№ докум.	Підп.	Дата	Схема структурна варіантів використання	Лит.	Маса	Масштаб
Розробила		Біленька О.Р.						
Перевірила		Лісовиченко О.І.						
Т. контр.					Програмне забезпечення для збереження та валідації документів на основі смарт-контрактів в блокчейнах	Аркуш		Аркушів
Н. контр.		Стельмах О.П.				КПІ ім.Ігоря Сікорського Кафедра ІПІ гр. ІП-311		
Затвердив		Жаріков Е.В.						



					КПІ.ІІІ-з1101.045470.06.99.СБД			
					Схема бази даних	Лит.	Маса	Масштаб
Зм.	Арк.	№ докум.	Підп.	Дата				
Розробила		Біленька О.Р.						
Перевірила		Лісовиченко О.І.						
Т. контр.						Аркуш		Аркушів
Н. контр.		Степняк О.П.			Програмне забезпечення для збереження та валідації документів на основі смарт-контрактів в блокчейнах	КПІ ім.Ігоря Сікорського Кафедра ІПІ гр. ІП-311		
Затвердив		Жаріков Е.В.						



					КПІ.ІП-31101.045470.06.99.ССК			
Зм.	Арк.	№ документа	Підпис	Дата	Схема структурна класів програмного забезпечення	Літера	Маса	Масштаб
Розробила		Біленька О.Р.						
Перевірила		Лісовиченко О.І.						
Т. контр.						Аркуш		Аркушів
Н. контр.		Стельмах О.П.			Програмне забезпечення для збереження та валідації документів на основі смарт-контрактів в блокчейнах	КПІ ім.Ігоря Сікорського Кафедра ІПІ гр. ІП-311		
Затвердив		Жаріков Е.В.						

Log in to start working

Username*

admin

Password*

.....

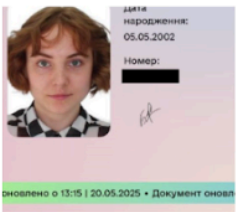
🗕

Login

					КПІ.ІП-31101.045470.06.99.ЕФ							
					Екранні форми	Літера		Маса		Масштаб		
Зм.	Арк.	№ документа	Підпис	Дата								
Розробила		Біленька О.Р.										
Перевірив		Лісовиченко О.І.										
Т. контр.						Аркуш		Аркушів				
Н. контр.		Стельмах О.П.			Програмне забезпечення для збереження та валідації документів на основі смарт-контрактів в блокчейнах					КПІ ім.Ігоря Сікорського Кафедра ІПІ гр. ІП-311		
Затвердив		Жаріков Е.В.										



Documents



PASSPORT.jpg
Document type: PASSPORT
Status: PUBLISHED
[Transaction hash](#)
02 Jun 2025



Verification



					КПІ.ІП-31101.045470.06.99.ЕФ							
Зм.	Арк.	№ документа	Підпис	Дата	Екранні форми	Літера		Маса		Масштаб		
Розробила		Біленька О.Р.										
Перевішив		Лісовиченко О.І.										
Т. контр.												
Н. контр.		Стельмах О.П.			Програмне забезпечення для збереження та валідації документів на основі смарт-контрактів в блокчейнах	Аркуш		Аркушів				
Затвердив		Жаріков Е.В.				КПІ ім.Ігоря Сікорського Кафедра ІПІ гр. ІП-311						

КПІ.ІП-31101.045470.06.99.ЕФ

Sepolia Testnet

Search by Address / Txn Hash / Block / Token

Etherscan

HomeBlockchainTokensNFTsMore

Transaction Details

OverviewLogs (1)State

TRANSACTION ACTION

CallAdd DocumentFunction by 0xFCf95068...3B47d076f on 0xcAD99016...4dD48bd20

[This is a Sepolia Testnet transaction only]

Transaction Hash:

0x6b710ab222892b6c5adaea48347eba2b0b10d769cd1578271cfe2a111cd9a22c

Status:

Success

Block:

845984213 Block Confirmations

Timestamp:

2 mins ago (Jun-02-2025 09:37:12 AM UTC)

From:

0xFCf950681F1502758Fb73A859f511353B47d076f

To:

0xcAD990164313AE1398Cc50EF595d1F24dD48bd20

Value:

0 ETH

Transaction Fee:

0.000000054283942207 ETH

Gas Price:

0.001000681 Gwei (0.000000000001000681 ETH)

More Details:

+ Click to show more

					КПІ.ІП-31101.045470.06.99.ЕФ						
					Екранні форми	Літера		Маса	Масштаб		
Зм.	Арк.	№ документа	Підпис	Дата							
Розробила		Біленька О.Р.									
Перевіряв		Лісовиченко О.І.									
Т. контр.						Аркуш		Аркушів			
Н. контр.		Степняк О.П.				Програмне забезпечення для збереження та валідації документів на основі смарт-контрактів в блокчейнах		КПІ ім.Ігоря Сікорського Кафедра ІПІ гр. ІП-311			
Затвердив		Жаріков Е.В.									