Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра обчислювальної техніки

Бакалаврська кваліфікаційна робота

на тему: «Програмний засіб збору інформації про користувачів у мережі web3»
08-54.БКР.028.00.000 ТЗ

Виконав: студент 4 курсу, групи 2СП-21б
спеціальності 123 — «Комп'ютерна інженерія»
освітня програма – «Системне програмування»
_____ Іваніщенко В.М

Керівник к.т.н., доц. каф. ОТ
_____ Дудник О. В

Рецензент к.т.н., доц. каф. ПЗ
_____ Романюк О.В.
(прізвище та ініціали)

Допущено до захисту
Зав. Кафедри
д.т.н., проф.Азаров О.Д.
11. 06 2025 р.

ВНТУ – 2025

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення
Рівень вищої освіти перший бакалаврський
Галузь знань 12 – Інформаційні технології
Спеціальність 123 – Комп'ютерна інженерія
Освітньо-професійна програма – Системне програмування

ЗАТВЕРДЖУЮ
Завідувач кафедри ОТ
д.т.н., проф.Азаров О.Д.

«_21_» __березня__ 2025 р.

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ
Іваніщенку Владиславу Миколайовичу

1 Тема роботи – «Програмний засіб збору інформації про користувачів у мережі web3», керівник роботи Дудник Олександр Вікторович, к.т.н., доцент кафедри ОТ, затверджені наказом вищого навчального закладу від 20 березня 2025 року №97.

2 Термін подання студентом роботи: 13.05.2025 року.

3 Вихідні дані до роботи: середовище розробки – Visual Studio Code, Jetbrain Rider, Remix, система управління базами даних –PostgreSql; мова запитів –Sql; мови розробки – HTML, CSS, JavaScript/TypeScript, фреймворк React, C# фреймворк .Net Core, Solidity, операційна система – Windows 10/11.

4 Зміст розрахунково-пояснювальної записки: вступ; аналіз стану розвитку навчальних систем та постановка задач дослідження; розробка методу, моделі та алгоритмів роботи веб-системи; розробка модулів програми; тестування програми; висновки; список використаних джерел; додатки; ілюстративна частина.

5 Перелік ілюстративного матеріалу: титульний слайд – автор, науковий керівник бакалаврської роботи, тема; мета, об'єкт та предмет дослідження; задачі дослідження; наукова новизна; практична цінність отриманих результатів; блок-схема загального алгоритму роботи веб-системи; блок-схема алгоритму

керування обліковими записами; блок-схема алгоритму тестуванн
діаграми; тестування веб-системи; апробація та публікації результатів
впровадження; фінальний слайд.

6 Консультанти розділів роботи наведено в таблиці 1

Таблиця 1 — Консультанти роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дат	
		завдання видав	зав... пр...
1 – 3	Дудник О.В. , к.т.н., доцент кафедри ОТ	21.03.2025	13.0
Нормоконтроль	ас. Каф. ОТ Швець С.І.	14.05.2025	30.0

7 Дата видачі завдання 21 березня 2025 року.

8 Календарний план виконання БКР приведений в таблиці 2

Таблиця 2 — Календарний план

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Строк виконання етапів роботи	Прим...
1	Огляд web3 технологій та формування задачі	21.03.2025 – 14.03.2025	Викон...
2	Архітектура та логіка функціонування системи	15.04.2025 – 23.04.2025	Викон...
3	Розробка модулів програмного продукту	24.04.2025 – 13.05.2025	Викон...

Студент Іваніщенко І...

(підпис) (прізвище та ...

Керівник б.к.р Дудник О...

(підпис) (прізвище та інші...

АНОТАЦІЯ

Іваніщенко В.М Програмний засіб збору інформації про користувачів у мережі web3 . Бакалаврська кваліфікаційна робота зі спеціальності 123 «Комп'ютерна інженерія», освітня програма «Комп'ютерна інженерія». Вінниця: ВНТУ, 2025. 78 с.

На укр. мові. Бібліогр.: назв: 20, рис.: 5; табл. 3.

У роботі досліджено сучасний стан розвитку web3-проєктів та DeFi-застосунків. Запропоновано рішення для ідентифікації користувачів шляхом запису унікального ідентифікатора до блокчейн-мережі Sepolia. Зібрані параметри пристрою зберігаються у JSON-форматі в базі даних PostgreSQL, розгорнутій у середовищі Docker. Клієнт створено на React, серверну частину — на C# (.NET Core), смарт-контракти написано мовою Solidity. Розроблено вебзастосунок який використаний як основа для децентралізованої верифікації та аналітики користувачів.

Ключові слова: web3, ідентифікація, блокчейн, децентралізація, смарт контракт.

ABSTRACT

Ivanishchenko V.M. Software tool for collecting information about users on the web3 network. Bachelor's thesis (project) in the speciality 123 ‘Computer Engineering’, educational programme ‘Computer Engineering’. Vinnytsia: VNTU, 2025. 78 p.

In Ukrainian. Bibliography: 20 names, 5 figures, 3 tables.

The paper examines the current state of development of web3 projects and DeFi applications. A solution is proposed for user identification by recording a unique identifier to the Sepolia blockchain network. The collected device parameters are stored in JSON format in a PostgreSQL database deployed in a Docker environment. The client is created on React, the server part — on C# (.NET Core), smart contracts are written in Solidity. The developed web application can be used as a basis for decentralised user verification and analytics.

Keywords: web3, identification, blockchain,decentralisation, smart contract.

ЗМІСТ

ВСТУП

Із розвитком web3-технологій зростає потреба в надійних інструментах верифікації користувачів, що дозволяють забезпечити безпеку, конфіденційність і стабільність децентралізованих сервісів. Особливо **актуальним** є захист фінансової складової криптопроектів від ботів, мультиакаунтів та зловживань, які загрожують чесності airdrop-програм та інших винагород. Формування унікального цифрового ідентифікатора користувача на основі технічного профілю пристрою відкриває можливості для створення системи, здатної виявляти аномальну активність та підвищувати цілісність web3-екосистем без порушення приватності.

Проводячи **аналіз** серед досліджень на тематику важливих напрямів застосування сучасних технологій є створення веб-ресурсів, що орієнтовані на інтерактивну взаємодію з користувачем. Зрозуміли, що їх застосування охоплює широкий спектр — від фінансових сервісів до децентралізованих платформ, де критично важливою є перевірка унікальності користувачів. У web3-середовищі такі сервіси повинні забезпечувати постійну доступність, масштабованість і зручність — незалежно від геолокації користувача. Саме це вимагає впровадження механізмів ідентифікації, що працюють автоматично, без необхідності ручної реєстрації, але з високим рівнем точності.

Попри активний розвиток web3-середовища, більшість існуючих сервісів залишаються технічно складними для масової інтеграції проектами. Часто вони потребують підключення цілих сервісів, та не мають підтвердження дій в блокчейні.Через це простіше обходити подібні захисти.

Новизна роботи полягає в зростанні уваги до питань безпеки та ідентифікації в децентралізованих системах, зокрема у web3-проєктах. З розвитком криптоспільнот і збільшенням кількості учасників різноманітних програм стимулювання активності (airdrops, bounty, тестнети), дедалі актуальнішим стає захист інфраструктури від автоматизованих атак, зловживань та фальсифікації ідентичностей. У той час як блокчейн забезпечує прозорість

транзакцій, саме відсутність перевірки унікальності користувача є одним з найвразливіших елементів у таких системах, що призводить до зниження довіри та фінансових втрат з боку проєктів.

З огляду на широке впровадження web3-рішень у різні сфери, створення інструменту для збору технічних даних про користувачів має значний потенціал для посилення безпеки децентралізованих платформ. Такий підхід дозволяє зменшити ризики використання ботів, дублювання особистостей та інших форм зловживань, які напряму впливають на цілісність і надійність криптопроєктів. Системи, здатні верифікувати користувачів на основі пристрою, не порушуючи приватності, стають ключовими для збереження чесності та стабільності web3-екосистем.

Тому розробка програмного засобу для збору користувацьких даних у web3-мережі є актуальним завданням на сучасному етапі розвитку децентралізованих технологій, адже вона спрямована на посилення безпеки, верифікації та протидії зловживанням у криптосередовищі.

Метою дослідження є розробка програмного засобу для підвищення надійності та безпеки взаємодії користувачів із web3-застосунками шляхом впровадження механізму збору технічного фінгерпринта пристрою, створення унікального ідентифікатора (visitor ID) та його закріплення за EVM-гаманцем у блокчейн-мережі. Запропонований підхід дозволяє знизити ризики зловживань, автоматизованих атак і дублювання облікових записів, що є критично важливим для проектів з фінансовими складовими у децентралізованих системах.

Об’єктом дослідження є процес розробки програмного засобу для збору технічної інформації про користувачів у web3-мережі з метою покращення безпеки та верифікації в децентралізованих системах.

Предмет дослідження є засоби реалізації програмного забезпечення для збору користувацьких технічних даних, ідентифікації пристроїв та взаємодії з блокчейн-мережею у web3-середовищі.

Методи дослідження включають:

— технології модульної розробки програмних засобів для реалізації компонентів системи збору та обробки користувацьких даних у web3-середовищі;

— організації зберігання та структурування технічних даних користувача у реляційній базі даних PostgreSQL у середовищі Docker;

– підходи до розробки простого та функціонального інтерфейсу для взаємодії з web3-гаманцем;

– тестування для перевірки працездатності основних компонентів системи, зокрема взаємодії з блокчейн-мережею та збереження даних у базі.

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

API — Application Programming Interface — інтерфейс прикладного програмування.

EVM —Ethereum Virtual Machine — віртуальна машина Ethereum.

ID — Identifier — ідентифікатор.

REST — Representational State Transfer — стиль архітектури систем.

UI — User Interface — користувацький інтерфейс.

1 ОГЛЯД WEB3 ТЕХНОЛОГІЙ ТА ФОРМУВАННЯ ЗАДАЧІ

1.1 Тенденції розвитку систем ідентифікації в web3-середовищі

У зв’язку зі швидким поширенням децентралізованих застосунків, NFT-платформ, DeFi-протоколів та інших сервісів у межах web3-екосистеми, зростає потреба у механізмах, що дозволяють розпізнавати та відрізняти унікальних користувачів без використання традиційної авторизації. Стандартні засоби автентифікації (логін, пароль, електронна пошта) не відповідають принципам децентралізації, оскільки передбачають централізовану обробку персональних даних.

У такому контексті актуальності набувають підходи, які дозволяють визначати унікальність взаємодії користувача без збирання персональних даних та створення централізованих облікових записів. У запропонованій системі авторизація здійснюється через пряме підключення користувача до застосунку за допомогою EVM-сумісного гаманця, що є загальноприйнятою практикою у web3-середовищі. Паралельно з цим реалізується механізм збору технічних параметрів середовища користувача (браузер, тип пристрою, часовий пояс тощо), який слугує антифрод-рішенням.

Загальна тенденція розвитку web3-платформ полягає у прагненні забезпечити баланс між анонімністю користувача та захистом від зловживань. Усе частіше розробники інтегрують механізми, які дозволяють виявляти підозрілу активність або повторне використання ресурсів без необхідності ідентифікації особи. Зокрема, використовуються одноразові верифікаційні транзакції, тимчасові токени, обмеження на кількість взаємодій з однієї адреси або пристрою. Це формує новий підхід до ідентифікації у web3 — функціональний, статистичний і незалежний від персональних даних.

1.2 Підходи до збору даних у web3- середовищі

У середовищі web3 поняття «збір даних» суттєво відрізняється від традиційного веб-простору. Через децентралізовану природу інфраструктури та

відсутність централізованих баз користувачів, система не має доступу до класичних ідентифікаторів, таких як логін, електронна пошта чи номер телефону. Відповідно, розробники web3-застосунків змушені формувати альтернативні підходи до збору й обробки інформації про користувачів, які відповідають принципам приватності, анонімності та самостійного контролю над особистими даними.

Один із ключових способів взаємодії з користувачем у web3 — це підключення криптогаманця. Адреса гаманця є основною точкою входу у більшість децентралізованих застосунків. Водночас вона не гарантує унікальності користувача, оскільки одна особа може мати десятки або навіть сотні гаманців. Це створює серйозні виклики в контексті стимулюючих програм (airdrop, whitelist, testnet-reward тощо), де важливо розмежовувати реальних учасників від ботів або мультиакаунтів:

У зв’язку з цим у web3-середовищі почали активно застосовуватись підходи, спрямовані на збір мета-даних або технічних характеристик взаємодії. Серед них можна виділити фіксацію параметрів браузера, операційної системи, мови, часового поясу, роздільної здатності екрану, типу пристрою, доступних плагінів тощо. Такі параметри дозволяють сформувати умовно унікальний цифровий слід користувача без прямої ідентифікації особистості. У деяких реалізаціях ця інформація агрегується та зберігається на стороні бекенду або в розподіленому сховищі, що дозволяє відслідковувати повторні взаємодії з одного пристрою.

З метою підвищення надійності системи часто використовується комбінація кількох механізмів: підключення гаманця, фіксація технічної інформації та контроль транзакційної активності. У сукупності це дозволяє будувати моделі поведінки, виявляти аномальну активність, та — в разі потреби — обмежувати доступ до критичних функцій або фінансово значущих дій. Такий підхід є більш гнучким, ніж класична авторизація, та не суперечить децентралізованій філософії взаємодії у web3.

1.3 Огляд аналогів

У сфері web3 наявні численні рішення, спрямовані на автентифікацію користувачів, проте більшість із них зосереджені на безпечному підключенні гаманця, а не на контролі унікальності чи боротьбі зі зловживаннями. Найбільш поширеними є інтеграції з криптогаманцями через бібліотеки на кшталт Web3Modal, Wagmi, Ethers.js, Viem, які надають базовий інтерфейс для підключення користувача до децентралізованого застосунку. Однак сам факт підключення не виключає можливості повторного використання або підміни гаманця.

У відповідь на проблему мультиакаунтів і бот-активності деякі проєкти реалізують механізми Soulbound Token (SBT) — незмінних токенів, які прикріплюються до адреси після проходження верифікації. Наприклад, такі токени використовуються у проєктах Gitcoin Passport, Proof of Humanity, або BrightID, які намагаються підтверджувати унікальність учасника шляхом соціальних графів або репутаційних схем. Незважаючи на інноваційність підходу, такі рішення зазвичай є надмірно складними для інтеграції в невеликі проєкти та часто вимагають участі користувача у зовнішній верифікаційній процедурі.

Ще одним прикладом є Worldcoin, який пропонує біометричну верифікацію через сканування райдужної оболонки ока. Незважаючи на високий рівень унікальності, цей підхід суперечить ключовим принципам приватності web3 і не прийнятний для широкого використання через централізацію збору біометричних даних.

На відміну від наведених рішень, запропонований у цій роботі підхід є легким, технічно автономним і не потребує залучення сторонніх протоколів чи KYC. Він базується на збиранні технічних характеристик середовища користувача, які формують унікальний ідентифікатор (visitor ID). Після підключення криптогаманця цей ідентифікатор одноразово записується у блокчейн-мережу (Sepolia), що створює стійкий зв’язок між пристроєм

користувача та його гаманцем. Такий підхід дозволяє ефективно обмежити повторну участь в активностях без складних верифікаційних процесів та без компрометації конфіденційності.

1.4 Постановка цілей, задач і вимог до системи

Зважаючи на виявлені проблеми та тенденції розвитку технологій у web3-екосистемі, основною метою цієї роботи є розробка програмного засобу, що дозволяє реалізувати механізм фіксації технічних параметрів пристрою користувача, формування унікального ідентифікатора (visitor ID), прив’язку його до криптогаманця та запис у блокчейн-мережу у вигляді одноразової транзакції.

Для досягнення поставленої мети необхідно реалізувати такі задачі:

— створити архітектуру системи, яка включає фронтенд, бекенд, базу даних та компонент взаємодії з блокчейном;

—реалізувати підключення EVM-сумісного гаманця (наприклад, MetaMask) для авторизації користувача;

— забезпечити одноразовий запис зв’язку між visitor ID та адресою гаманця у тестову блокчейн-мережу Sepolia;

— організувати передачу зібраного JSON-файла з технічними параметрами на серверну частину та зберігання його у реляційній базі даних;;

2 АРХІТЕКТУРА ТА ЛОГІКА ФУНКЦІОНУВАННЯ СИСТЕМИ

2.1 Визначення функціональної структури рішення

Розроблений Розроблюваний програмний засіб представляє собою комплексну систему, основною метою якої є забезпечення фіксації користувацької активності в межах web3-середовища. Ключовою особливістю даного рішення є прагнення реалізувати цей функціонал без прямого порушення фундаментальних принципів анонімності, що є наріжним каменем для багатьох користувачів децентралізованих систем. Система побудована на основі взаємодії кількох ключових компонентів, кожен з яких виконує чітко визначений набір функціональних завдань. Ці завдання охоплюють весь цикл, починаючи від збору первинної технічної інформації про сесію користувача, продовжуючи генерацією та фіксацією унікального ідентифікатора в блокчейн-мережі, і завершуючи безпечним зберіганням зібраних даних на серверній інфраструктурі для подальшого аналізу чи верифікації. Така архітектура дозволяє досягти високого рівня модульності та гнучкості, що є важливим для подальшого розвитку та адаптації системи до змінних вимог.

В основі програмного засобу лежить концепція декомпозиції складного завдання на низку менших, керованих функціональних блоків. Цей підхід не тільки спрощує розробку та тестування окремих частин системи, але й забезпечує можливість їх незалежного масштабування та модифікації. Загальна архітектура системи передбачає чіткий поділ відповідальності між клієнтською частиною, що виконується безпосередньо у браузері користувача, модулем взаємодії з криптографічним гаманцем, смарт-контрактом, розгорнутим у блокчейн-мережі, серверною частиною, що обробляє та зберігає дані, та системою зберігання даних.

Принцип функціонування системи полягає у послідовному виконанні операцій. Спочатку клієнтський модуль збирає набір неідентифікуючих технічних параметрів оточення користувача. На основі цих параметрів генерується унікальний ідентифікатор сесії, так званий visitor ID. Далі, для

підтвердження унікальності дії та прив'язки її до певного, хоча й потенційно анонімного, джерела контролю (EVM-гаманця), користувачу пропонується підписати транзакцію через свій гаманець. Ця транзакція ініціює виклик функції смарт-контракту, який перевіряє унікальність visitor ID та, у разі успіху, зберігає його в блокчейні. Одночасно, зібрані технічні параметри разом з адресою гаманця та visitor ID надсилаються на серверну частину для збереження в базі даних.

Ключовим аспектом є спроба зберегти анонімність. Хоча система фіксує адресу EVM-гаманця, передбачається, що сам гаманець може не бути прямо пов'язаний з реальною особою користувача. Запис у блокчейн слугує доказом унікальної взаємодії, не розкриваючи при цьому більше інформації, ніж адреса гаманця та згенерований visitor ID. Зібрані технічні дані, хоч і можуть слугувати для створення "цифрового відбитку", зберігаються окремо і використовуються для аналізу патернів активності та потенційної верифікації повторних візитів, але не для прямої ідентифікації особи.

Клієнтський модуль є першою точкою взаємодії користувача з системою та відіграє критичну роль у процесі збору початкових даних. Його функціонування починається автоматично при завантаженні відповідної веб-сторінки або веб-додатку, де інтегровано даний програмний засіб. Основна відповідальність цього модуля полягає в ініціалізації процесу збору технічних параметрів браузера та операційної системи користувача. Ці параметри підбираються таким чином, щоб мінімізувати ризик прямої ідентифікації особи, але водночас забезпечити достатню унікальність для формування visitor ID.

До переліку таких параметрів можуть входити: версія браузера та його рушія, список встановлених плагінів та їх версій, характеристики екрану (роздільна здатність, глибина кольору), налаштування часового поясу та мови системи, інформація про операційну систему, доступні шрифти та інші конфігураційні дані, які можна отримати за допомогою стандартних JavaScript API браузера. Важливо підкреслити, що збір таких даних відбувається прозоро

для користувача, хоча й може бути обумовлений політикою конфіденційності ресурсу.

Після успішного збору необхідного набору технічних параметрів, клієнтський модуль приступає до генерації visitor ID. Цей ідентифікатор формується шляхом застосування криптографічної хеш-функції (наприклад, SHA-256) до конкатенації зібраних параметрів або їх комбінації. Використання хешування дозволяє перетворити потенційно великий обсяг різнорідної інформації в рядок фіксованої довжини, який є унікальним для конкретного набору вхідних даних з високою ймовірністю. Саме цей visitor ID надалі буде використовуватися для фіксації у блокчейні та зв'язування з даними на сервері.

Окрім збору даних та генерації ID, клієнтський модуль також відповідає за відображення користувацького інтерфейсу. Цей інтерфейс інформує користувача про необхідність підключення його EVM-сумісного гаманця для подальшої авторизації та підтвердження дії. На цьому етапі модуль ініціює запит на підключення до гаманця, використовуючи відповідні бібліотеки, та готує дані для наступного етапу – взаємодії з гаманцем для підпису транзакції. Інтерфейс розробляється з урахуванням принципів мінімалізму та зрозумілості, щоб не перевантажувати користувача зайвою інформацією та чітко пояснювати необхідність наступних кроків.

Блок взаємодії з гаманцем є критично важливим компонентом, що забезпечує зв'язок між клієнтською частиною системи та блокчейн-мережею через криптографічний гаманець користувача. Реалізація цього блоку зазвичай спирається на використання спеціалізованих JavaScript бібліотек, таких як ethers.js або web3.js, які надають стандартизований інтерфейс для роботи з EVM-сумісними гаманцями, найпопулярнішим з яких є MetaMask.

Процес взаємодії починається після того, як клієнтський модуль успішно зібрав технічні параметри та згенерував visitor ID. На цьому етапі користувачу через інтерфейс пропонується підключити свій гаманець. Бібліотека ethers.js дозволяє ініціювати запит на підключення, який обробляється розширенням гаманця у браузері. Користувач повинен буде явно надати дозвіл веб-додатку на

доступ до його адреси гаманця. У разі успішного підключення, система отримує публічну адресу гаманця користувача. Ця адреса буде використана для ідентифікації ініціатора транзакції в блокчейні та для зв'язування з технічними даними на сервері.

Наступним кроком є підпис транзакції. Важливо зазначити, що в даному контексті "транзакція" не обов'язково означає переказ криптовалюти. Це може бути виклик функції смарт-контракту, яка записує visitor ID. Блок взаємодії з гаманцем формує необхідні дані для такої транзакції. Ці дані включають адресу смарт-контракту, назву функції, яку потрібно викликати (наприклад, функцію для запису visitor ID), та сам visitor ID як параметр.

Користувачу буде запропоновано підписати цю транзакцію своїм приватним ключем через інтерфейс гаманця. Підпис транзакції є криптографічним доказом того, що власник гаманця свідомо ініціював цю дію. Це забезпечує автентичність та цілісність запиту до смарт-контракту. Бібліотека ethers.js обробляє процес надсилання підписаної транзакції до вузла блокчейн-мережі Sepolia. Система повинна також передбачати обробку можливих помилок на цьому етапі, таких як відмова користувача від підпису, недостатня кількість коштів для оплати газу (навіть у тестовій мережі) або проблеми зі з'єднанням з мережею. Успішне відправлення транзакції та її подальше включення до блоку в мережі Sepolia завершує цей етап взаємодії.

Смарт-контракт, розгорнутий у тестовій мережі Sepolia, виконує одну з найважливіших функцій у системі – забезпечення унікальності запису visitor ID та створення незмінного доказу цієї дії. Вибір тестової мережі Sepolia обумовлений її доступністю, стабільністю та відсутністю реальних фінансових витрат на транзакції, що ідеально підходить для розробки, тестування та демонстрації функціоналу.

Сам смарт-контракт розробляється мовою Solidity, яка є стандартом для EVM-сумісних блокчейнів. Його логіка є відносно простою, але критично важливою. Основна функція контракту полягає в прийомі visitor ID від користувача (через підписану транзакцію) та перевірці, чи не був цей

ідентифікатор зареєстрований раніше. Для зберігання вже записаних visitor ID та ефективної перевірки їх унікальності, смарт-контракт використовує структуру даних типу mapping. Наприклад, це може бути mapping(bytes32 => bool) private recordedVisitorIDs;, де ключем є хеш visitor ID, а значенням – булевий прапорець, що вказує на факт запису.

Коли користувач через свій гаманець викликає відповідну функцію смарт-контракту (наприклад, recordVisitorID(bytes32 _visitorID)), контракт спочатку перевіряє, чи існує вже запис для переданого _visitorID у recordedVisitorIDs. Якщо такий запис вже існує (recordedVisitorIDs[_visitorID] == true), транзакція може бути відхилена або контракт може повернути відповідний статус, що вказує на неможливість повторного запису. Це гарантує, що кожен унікальний visitor ID може бути зафіксований у блокчейні лише один раз.

Якщо ж visitor ID є унікальним, смарт-контракт встановлює відповідний прапорець у mapping (recordedVisitorIDs[_visitorID] = true;) і, можливо, генерує подію (event). Події в Solidity є важливим механізмом для сповіщення зовнішніх додатків про зміни стану контракту. Наприклад, може бути згенерована подія VisitorIDRecorded(address indexed user, bytes32 visitorID), яка містить адресу гаманця, що ініціював запис, та сам записаний visitor ID. Клієнтська або серверна частина системи може підписатися на ці події для отримання підтвердження успішного запису.

Розгортання смарт-контракту в мережі Sepolia та його взаємодія з клієнтською частиною через web3 API (за допомогою ethers.js) вимагає наявності ABI (Application Binary Interface) контракту та його адреси в мережі. ABI описує функції та структури даних контракту, дозволяючи зовнішнім додаткам правильно формувати запити до нього. Безпека смарт-контракту, навіть такого простого, є важливою. Необхідно уникати відомих вразливостей Solidity, хоча для даного функціоналу ризики є мінімальними.

Серверна частина системи, реалізована на платформі .NET Core, відіграє роль центрального вузла для збору, обробки та зберігання детальної інформації, пов'язаної з активністю користувачів. Вибір .NET Core зумовлений його високою

продуктивністю, кросплатформеністю та потужними інструментами для розробки веб-API та роботи з базами даних.

Основним завданням серверної частини є прийом даних від клієнтського модуля. Ці дані зазвичай надсилаються у вигляді JSON-структури через захищене HTTP-з'єднання (HTTPS) на спеціально визначений API ендпоінт, наприклад, POST /api/activity_log. Ця JSON-структура містить зібрані технічні параметри користувача (такі як інформація про браузер, операційну систему, налаштування екрану тощо), адресу EVM-гаманця, яка була отримана під час взаємодії з гаманцем, та згенерований visitor ID, який також був записаний у смарт-контракт.

Після отримання JSON-структури, серверна частина виконує низку операцій. По-перше, відбувається валідація вхідних даних: перевірка коректності формату, наявності всіх необхідних полів та, можливо, відповідності певним очікуваним значенням чи діапазонам. Це допомагає забезпечити цілісність даних перед їх збереженням. По-друге, серверна логіка асоціює отримані технічні параметри з конкретною адресою гаманця та visitor ID. Цей зв'язок є ключовим для подальшого аналізу.

Далі, оброблені та валідовані дані готуються для збереження в базі даних. Серверна частина взаємодіє із системою зберігання, якою в даному випадку є PostgreSQL. Ця взаємодія може бути реалізована за допомогою ORM (Object-Relational Mapper), наприклад, Entity Framework Core, що спрощує роботу з базою даних, абстрагуючи SQL-запити, або через прямі запити до бази даних. Інформація зберігається структуровано, що дозволяє ефективно витягувати та аналізувати її в майбутньому.

Окрім основного функціоналу прийому та збереження даних, серверна частина також може відповідати за додаткові завдання, такі як ведення логів роботи системи, моніторинг помилок, а також, потенційно, надання API для внутрішніх аналітичних інструментів або для верифікації повторного доступу на основі зібраних параметрів. Важливими аспектами є забезпечення безпеки

серверної частини (захист від несанкціонованого доступу, атак типу XSS, SQL-ін'єкцій) та її масштабованості для обробки зростаючого навантаження.

Система зберігання даних є фундаментальним компонентом, що забезпечує довготривале та надійне збереження інформації, зібраної серверною частиною. У якості сховища для даного програмного засобу обрано реляційну базу даних PostgreSQL. Цей вибір обґрунтований високою надійністю PostgreSQL, її відповідністю стандартам ACID, широкими можливостями для роботи зі складними запитами та вбудованою підтримкою для зберігання та індексування даних у форматі JSON, що є особливо корисним для збереження технічних параметрів користувача.

Розгортання PostgreSQL здійснюється у середовищі Docker. Використання Docker-контейнерів надає низку переваг, зокрема, ізоляцію середовища бази даних, легкість розгортання та перенесення між різними системами, а також спрощене управління версіями та конфігураціями. Docker дозволяє швидко підняти екземпляр PostgreSQL з необхідними налаштуваннями, забезпечуючи консистентність середовища розробки, тестування та продуктивної експлуатації.

Структура бази даних (схема) ретельно проектується для ефективного зберігання та доступу до даних. Основна таблиця, умовно названа user_activity_records, може містити такі поля: унікальний ідентифікатор запису (первинний ключ), адреса EVM-гаманця користувача (наприклад, тип VARCHAR(42)), записаний у блокчейн visitor ID (наприклад, тип VARCHAR(66) або BYTEA), поле для зберігання зібраних технічних параметрів у форматі JSON (тип JSONB в PostgreSQL є оптимальним, оскільки він зберігає дані у бінарному форматі та підтримує індексацію), а також мітка часу створення запису (тип TIMESTAMP).

Використання типу JSONB для технічних параметрів дозволяє зберігати гнучку структуру даних, яка може змінюватися або доповнюватися без необхідності зміни схеми таблиці. PostgreSQL надає потужні оператори та функції для запитів до даних всередині JSONB-полів, що дозволяє проводити фільтрацію та агрегацію за окремими атрибутами технічного профілю

користувача. Для прискорення запитів, особливо за адресою гаманця або visitor ID, створюються відповідні індекси.

Важливим аспектом є забезпечення персистентності даних при використанні Docker. Дані PostgreSQL повинні зберігатися у Docker-томах (volumes), що гарантує їх збереження навіть при зупинці, видаленні або оновленні контейнера з базою даних. Також необхідно продумати стратегії резервного копіювання та відновлення даних для запобігання їх втрати. Система зберігання, таким чином, є не просто сховищем, а організованою структурою, що підтримує аналітичні можливості та забезпечує цілісність і доступність інформації.

Функціональна структура програмного засобу побудована таким чином, щоб забезпечити чітке відокремлення критичних етапів обробки: збір початкової інформації, взаємодія з криптографічним гаманцем користувача, фіксація унікального ідентифікатора в блокчейні, та подальше збереження зібраних даних на сервері з наступною валідацією унікальності. Ця модульність є ключовою перевагою, оскільки вона не тільки спрощує розробку та супровід окремих компонентів, але й надає значну гнучкість у плані розширюваності системи. Кожен компонент може бути масштабований або навіть замінений незалежно від інших, відповідно до специфічних потреб конкретного проекту або зростаючого навантаження на систему.

Розглянемо загальний потік даних. Користувач заходить на веб-ресурс. Клієнтський модуль у його браузері збирає технічні параметри та генерує visitor ID. Далі, користувач підключає свій EVM-гаманець. Блок взаємодії з гаманцем, використовуючи ethers.js, отримує адресу гаманця та ініціює підпис транзакції для виклику смарт-контракту в мережі Sepolia, передаючи йому visitor ID. Смарт-контракт перевіряє унікальність visitor ID і, якщо він унікальний, зберігає його. Паралельно, клієнтський модуль надсилає JSON-структуру, що містить зібрані технічні параметри, адресу гаманця та visitor ID, на серверну частину, реалізовану на .NET Core. Серверна частина валідує ці дані та зберігає їх у базі даних PostgreSQL, розгорнутій у Docker.

Щодо аспектів анонімності, важливо розуміти, що система прагне до псевдоанонімності, а не абсолютної анонімності. Адреса EVM-гаманця є публічною інформацією в блокчейні. Якщо ця адреса може бути пов'язана з реальною особою через інші сервіси (наприклад, KYC на біржах), то анонімність порушується. Однак, система не збирає прямих ідентифікаторів особи (ім'я, email тощо). Visitor ID, заснований на технічних параметрах, також може бути досить унікальним, створюючи "цифровий відбиток". Запис у блокчейн забезпечує прозорий та незмінний доказ факту взаємодії, пов'язаний з певним visitor ID та адресою гаманця. Зібрані на сервері дані можуть використовуватися для аналізу поведінки та виявлення повторних візитів з високим ступенем ймовірності, навіть якщо користувач намагається змінити деякі параметри.

Перспективи розвитку даного програмного засобу можуть включати перехід на основну мережу (mainnet) для реальних застосувань, хоча це пов'язано з реальними витратами на газ. Можливе вдосконалення алгоритмів генерації visitor ID для підвищення їх стабільності та унікальності, а також розробка більш складних аналітичних інструментів для обробки зібраних даних. Також можна розглянути інтеграцію з іншими web3-сервісами або розширення функціоналу для підтримки різних блокчейн-мереж. Важливим напрямком є постійне дослідження та адаптація до нових методів забезпечення приватності та стандартів безпеки у web3-просторі. Простота супроводу, забезпечена модульною архітектурою, дозволить відносно легко впроваджувати такі зміни та оновлення в майбутньому.

2.2 Проектування архітектури взаємодії між компонентами

Архітектура розробленої системи побудована за клієнт-серверною моделлю з інтеграцією в блокчейн-мережу. Вона охоплює три основні рівні: фронтенд (React), бекенд (ASP.NET Core) та інфраструктуру зберігання/блокчейну. Кожен рівень виконує свою роль, взаємодіючи з іншими компонентами через чітко визначені API. Для розуміння подано діаграму на рисунку 2.1

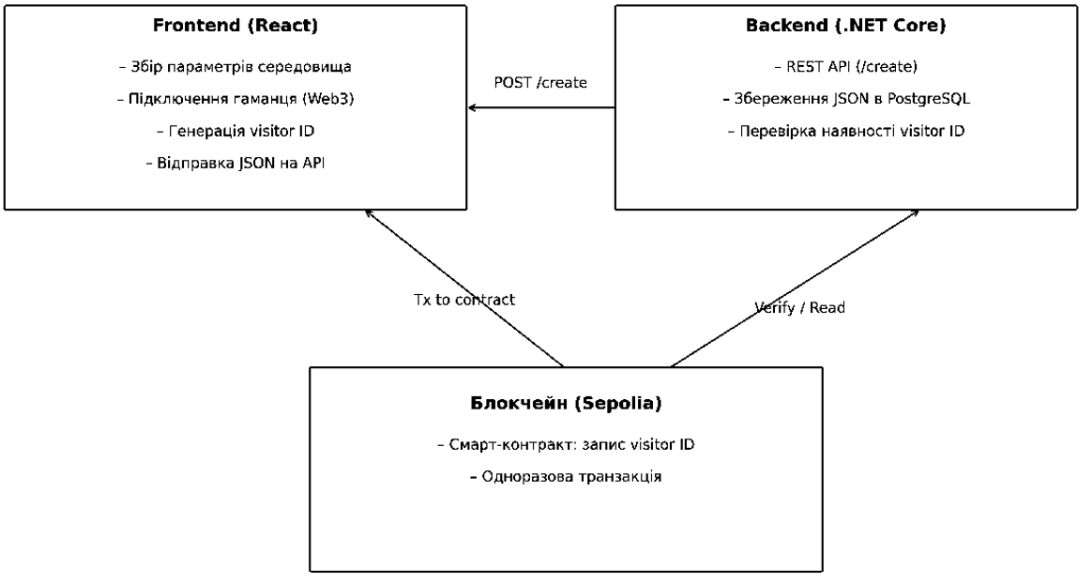

Рисунок 2.1 — Структура взаємодії

Клієнтська частина реалізована на основі бібліотеки React та використовує модифіковану версію бібліотеки FingerprintJS, що дозволяє збирати технічні параметри середовища користувача. На основі цієї інформації формується унікальний ідентифікатор користувача (visitor ID), який виступає в системі як умовний цифровий підпис пристрою. У межах тієї ж клієнтської частини реалізовано механізм підключення криптогаманця через Web3-провайдер (наприклад, MetaMask), що дозволяє користувачу ініціювати транзакцію для запису visitor ID у блокчейн. Зібрані параметри формуються в структурований об'єкт, який надсилається на сервер за допомогою HTTP-запиту через REST API.

Серверна частина, реалізована на платформі ASP.NET Core, виконує роль логічного посередника між фронтендом і базою даних. Вона приймає об'єкти типу DTO, що містять visitor ID та супутні технічні параметри. Після серіалізації отриманої інформації, вона зберігається у базі даних PostgreSQL у форматі JSON. Бекенд також реалізує функцію перевірки наявності запису по visitor ID, що дозволяє швидко визначити, чи було вже зафіксовано взаємодію з тим самим пристроєм. API-сервіси, що обслуговують запити, описані у відповідному класі маршрутів, а бізнес-логіка винесена у сервісний рівень програми. Уся система контейнеризована за допомогою Docker, що забезпечує її портативність і

простоту розгортання.

Блокчейн-рівень функціонує як незалежна мережа верифікації, де безпосередньо фіксується відповідність між адресою гаманця користувача та згенерованим ідентифікатором. Усі взаємодії з контрактом реалізуються на стороні фронтенда, а сам контракт розгорнуто в тестовій мережі Sepolia. Його призначення — унеможливити повторний запис одного й того ж visitor ID, забезпечивши тим самим одноразову верифікацію. Смарт-контракт, описаний через ABI-файл hashStorage.abi.json, отримує visitor ID та зберігає його у ланцюжку блоків як доказ того, що користувач із цим пристроєм вже був зареєстрований.

Загалом архітектура реалізованого рішення забезпечує логічне розділення обов'язків між компонентами, прозору маршрутизацію даних і надійну фіксацію унікальних ідентифікаторів у розподіленому середовищі без потреби у класичній автентифікації чи залученні особистої інформації користувача.

2.3 Логіка взаємодії з блокчейн мережою Sepolia

У розробленій системі блокчейн-мережа Sepolia використовується для децентралізованого зберігання унікальних ідентифікаторів користувачів (visitor ID), що дозволяє забезпечити одноразову верифікацію без необхідності централізованої автентифікації.

Смарт-контракт, розгорнутий у мережі Sepolia, реалізує два основні методи:

— storeHash(bytes16 _hash): записує переданий хеш (visitor ID) у блокчейн, якщо такий хеш вже існує, транзакція буде відхилена, що гарантує унікальність запису;

— isHashExists(bytes16 _hash), перевіряє наявність хешу у сховищі контракту, повертаючи булеве значення.

Ці функції забезпечують механізм, при якому кожен користувач може бути верифікований лише один раз, що є ефективним засобом протидії мультиакаунтингу та зловживанням у системах з фінансовою мотивацією.

На стороні клієнта, після збору технічних параметрів та генерації visitor ID, користувач підключає свій EVM-сумісний гаманець (наприклад, MetaMask) та ініціює транзакцію до смарт-контракту для запису свого ідентифікатора. Цей процес забезпечує зв'язок між пристроєм користувача та його криптографічною адресою без розкриття особистих даних.

Архітектурно смарт-контракт реалізує просту структуру типу mapping(bytes16 => bool), що дозволяє ефективно перевіряти існування запису у постійному сховищі без складної логіки. Такий формат зберігання є дешевим з точки зору gas-витрат і підходить для високонавантажених систем, де потрібна лише одноразова верифікація факту взаємодії.

Використання тестової мережі Sepolia дозволяє ефективно тестувати та демонструвати функціональність системи без витрат на реальні транзакції в основній мережі Ethereum.

У перспективі реалізовану модель можливо масштабувати до основної мережі Ethereum або до сумісних EVM-мереж (Polygon, Arbitrum, Optimism), де подібний підхід може застосовуватись як базовий рівень верифікації пристрою без порушення приватності та без централізованого реєстру.

2.4 Системні виклики та обґрунтування технологічного вибору

У процесі розробки системи ідентифікації для Web3-середовища постала низка викликів, пов’язаних із досягненням балансу між децентралізацією, ефективністю та захистом від зловживань. Цей підрозділ зосереджений на аналізі ключових технологічних рішень, які були прийняті на етапі архітектурного проєктування, а також обґрунтовує доцільність обраних інструментів і підходів.

Однією з основних проблем стало уникнення централізованого зберігання даних про користувачів, зокрема персональної інформації. В контексті Web3 будь-яке порушення конфіденційності суперечить базовим принципам екосистеми. Саме тому модель ідентифікації, реалізована в даній системі, базується виключно на технічних характеристиках користувацького середовища

без залучення cookie, IP-адрес, електронної пошти чи будь-яких ідентифікаторів, які можуть бути прив’язані до особистості.

Другим викликом було забезпечення одноразовості запису, що є критичним для фільтрації ботів та запобігання мультиакаунтингу. На рівні смарт-контракту для цього реалізовано перевірку на унікальність visitor ID перед його записом у блокчейн. Такий підхід дозволяє гарантувати, що користувач не зможе повторно здійснити реєстрацію з того самого пристрою, навіть за умов зміни гаманця або мережевого середовища.

Також окремо слід згадати виклики, пов’язані з вибором середовища для реалізації серверної частини. Було розглянуто декілька варіантів, серед яких Node.js, Python (FastAPI) і .NET Core. Остаточний вибір на користь ASP.NET Core було зроблено через поєднання високої продуктивності, типізованої структури, підтримки асинхронності та стабільної інтеграції з PostgreSQL через Entity Framework. Завдяки цьому вдалося зменшити кількість помилок на етапі валідації запитів і забезпечити більш суворий контроль даних, що надходять із фронтенду.

Docker було обрано як основне середовище для контейнеризації, оскільки він дає змогу створити ізольовану інфраструктуру, незалежну від середовища розгортання. Це дозволяє гарантувати стабільну роботу системи як у локальному середовищі розробки, так і на staging/production-серверах. Крім того, використання docker-compose дозволяє масштабувати окремі сервіси без впливу на загальну структуру проєкту.

Особливу увагу в проєкті було приділено UX/UI-компоненту. Зважаючи на те, що взаємодія користувача з Web3-застосунком зазвичай відбувається через розширення типу MetaMask, було необхідно забезпечити максимально зрозумілий і простий процес підтвердження транзакції. Реалізовані повідомлення про статус, відображення прогресу та візуальна індикація результатів дій користувача підвищують загальний рівень довіри та зменшують відсоток відмов.

Крім цього, важливим компонентом стало логування подій. На стороні сервера реалізовано механізм журналювання запитів та відповідей, що дозволяє проводити аудит транзакцій та виявляти потенційно шкідливу активність. Це рішення не лише покращує захищеність системи, а й відкриває можливості для інтеграції аналітики, побудови графів взаємодії та виявлення шаблонів поведінки.

Таким чином, технологічні вибори, реалізовані в системі, не були випадковими — вони стали результатом зваженого аналізу цілей, обмежень і можливостей, які надає сучасна інфраструктура Web3. Кожен компонент — від клієнтської бібліотеки до бази даних і смарт-контракту — є частиною цілісного рішення, орієнтованого на стійкість до атак, відповідність принципам конфіденційності та масштабованість. Усі ці фактори в сукупності формують технологічний фундамент, здатний підтримувати високі вимоги до ідентифікації в децентралізованих системах нового покоління.

2.5 Підсумки архітектурного проектування

У цьому розділі було здійснено формалізацію та проєктування архітектури програмного засобу, що реалізує збір технічної інформації про користувача з подальшою фіксацією в блокчейн-середовищі. Основною метою розробки є створення безпечного, масштабованого та ефективного механізму для однозначної верифікації користувача у Web3-системах без порушення принципів децентралізації та збереження конфіденційності.

Розроблена архітектура базується на принципах модульності, ізольованості компонентів, захисту від зловживань і повторного доступу до функціоналу системи. Вона поєднує сучасні фреймворки клієнтської частини, технології серверної обробки запитів і засоби децентралізованого зберігання інформації, забезпечуючи інтеграцію в екосистему Ethereum через тестову мережу Sepolia. Кожен компонент системи спроєктовано з урахуванням принципу розділення обов’язків і незалежності розгортання, що спрощує супровід, тестування і масштабування.

Клієнтська частина реалізована з використанням фреймворку React, який забезпечує гнучкість у побудові інтерфейсів, модульність компонентів і підтримку сучасних підходів до розробки SPA-застосунків. У проєкті використано кастомізовану бібліотеку FingerprintJS, яка дозволяє зібрати широкий спектр технічних параметрів пристрою користувача (включаючи User-Agent, screen resolution, timezone, мовні налаштування та інші), на основі яких формується унікальний visitor ID. Цей ідентифікатор є головним засобом розрізнення унікальних пристроїв і виконує роль цифрового підпису клієнта без залучення персональних даних.

Сучасний збірник Vite, застосований у проєкті, значно підвищує продуктивність розробки: забезпечує миттєве оновлення коду, гарячі перезавантаження та інтеграцію з інструментами Web3, такими як ethers.js. Це дало змогу скоротити час на компіляцію та налагодження інтерфейсу. Реалізація зв’язку між фронтендом і блокчейном передбачає інтеграцію з MetaMask, що дозволяє користувачу здійснити транзакцію, не покидаючи вебінтерфейсу. Підписана транзакція фіксує visitor ID у блокчейн-мережі Sepolia, підтверджуючи унікальність пристрою.

Серверна частина проекту реалізована з використанням ASP.NET Core. Ця технологія була обрана завдяки високій продуктивності, підтримці REST API, зручності маршрутизації та наявності ORM-рішення Entity Framework Core, що значно спрощує роботу з базою даних PostgreSQL. Структура сервісу охоплює ендпоїнти для прийому запитів з фронтенду, перевірки наявності visitor ID у базі та обробки транзакційного статусу. Отримані з клієнта параметри зберігаються у форматі JSON, що дозволяє легко розширювати структуру та здійснювати аналітику з мінімальними витратами на переформатування даних. Програмна логіка реалізує мікросервісний підхід з чітко визначеними контрактами взаємодії між компонентами.

Контейнеризація середовища здійснена за допомогою Docker, що спрощує розгортання та забезпечує ізольованість компонентів. За допомогою docker-compose реалізовано окремі сервіси для веб-сервера, бази даних і допоміжних

утиліт, що гарантує відтворюваність конфігурації при розгортанні в різних середовищах (розробка, staging, production). Крім того, застосування Docker дозволяє швидко масштабувати систему в залежності від навантаження, змінюючи кількість інстансів окремих сервісів без шкоди загальній структурі.

Смарт-контракт, написаний на мові Solidity, містить мінімальний, але достатній функціонал: запис нового visitor ID, а також перевірку наявності запису за хешем. Така архітектура дозволяє мінімізувати вартість транзакцій і зменшити складність аудиту без шкоди для функціональності. Особливість полягає в тому, що повторна спроба запису ідентичного хешу буде відхилена мережею, що фактично унеможливлює повторну реєстрацію одного і того ж пристрою та забезпечує гарантію одноразовості.

Інтеграція клієнтської та серверної частини із блокчейн-компонентом реалізована через Web3.js/ethers.js, залежно від сценарію застосування. Система взаємодії побудована таким чином, щоб виклик методу смарт-контракту з фронтенду був максимально простим та захищеним. Верифікація успішного запису здійснюється шляхом опитування стану або отримання подій через RPC.

Проектоване рішення має низку суттєвих переваг. По-перше, воно дозволяє уникнути централізованих механізмів KYC/AML або CAPTCHA, які або надто обтяжливі для користувача, або легко обходяться ботами. По-друге, мінімальний обсяг переданих даних і відсутність персональної інформації відповідає найкращим практикам конфіденційності та інформаційної гігієни. По-третє, гнучкість архітектури дозволяє інтегрувати цей модуль у більші системи Web3, що працюють з винагородами, доступами до закритих сервісів або просто як фільтр у вхідному трафіку.

Розроблена система також передбачає можливість подальшого розширення функціональності: зокрема, введення таймштампів, агрегування статистики звернень, зв’язування кількох visitor ID з різними гаманцями, що відкриває перспективи для формування довготривалих поведінкових профілів без використання cookie або сторонніх ідентифікаторів. Потенційно така система може слугувати основою для формування більш комплексної системи

авторизації в децентралізованих застосунках.

Загалом архітектура системи є завершеним технічним рішенням, що поєднує в собі переваги Web2 та Web3: швидкість і масштабованість класичних вебсервісів з довірою і незмінністю децентралізованих мереж. Це забезпечує їй потенціал для подальшого розвитку, впровадження в реальні продукти та розширення в бік багаторівневої аналітики, агрегування поведінкових метрик та формування цифрових профілів нового покоління.

3 РОЗРОБКА МОДУЛІВ ПРОГРАМНОГО ПРОДУКТУ

3.1 Реалізація фронтенд-модуля та системи фінгерпринтингу

Фронтенд-модуль розробленого програмного забезпечення реалізовано із застосуванням сучасного інструментарію для створення високопродуктивних веб-додатків. Основу клієнтської частини становить JavaScript-фреймворк React, який є одним із найпоширеніших інструментів для побудови інтерфейсів користувача на основі компонентної архітектури. Використання React дає змогу забезпечити гнучку, масштабовану структуру проєкту, що легко адаптується до змін у бізнес-логіці або дизайні системи. Для забезпечення максимальної швидкодії, зменшення часу розробки та ефективного керування залежностями обрано інструмент збірки Vite, що позиціонується як сучасна альтернатива Webpack та інших традиційних збирачів.

Vite використовує переваги нативної підтримки модулів ES6 у браузерах та забезпечує швидкий старт проєкту, миттєве оновлення інтерфейсу за допомогою функції hot module replacement (HMR), а також мінімальні затримки під час компіляції змін у коді. Це особливо критично під час активної фази розробки, коли швидкий зворотний зв'язок між написаним кодом і візуальним результатом дозволяє суттєво підвищити продуктивність команди. Архітектура клієнтської частини проєкту організована за принципами модульності та інкапсуляції, що дає змогу легко повторно використовувати компоненти, підтримувати масштабування системи, а також ефективно тестувати окремі її елементи.

Ключовою функцією фронтенд-модуля є збір технічних характеристик пристрою користувача з метою генерації унікального цифрового ідентифікатора (visitor ID), який виступає як криптографічний підпис середовища користувача. Для реалізації цього функціоналу застосовується форк бібліотеки FingerprintJS — інструменту для створення браузерних "відбитків" на основі сукупності множини параметрів, доступних через API браузера. Форк дозволяє не лише розширити перелік параметрів, які враховуються при побудові ідентифікатора, а

й адаптувати логіку до специфічних вимог проєкту (наприклад, виключити параметри, що порушують конфіденційність, або додати нові, специфічні для певної платформи).

Процедура збору включає зчитування низки апаратних та програмних характеристик середовища користувача: версії та тип браузера, операційної системи, роздільної здатності екрана, поточної часової зони, мови інтерфейсу, інформації про наявність та функціональність Canvas, WebGL, AudioContext та інші параметри, які сумарно формують детальний цифровий профіль пристрою. Зібрані параметри передаються на вхід криптографічної хеш-функції (у поточній реалізації використовується SHA-256), яка повертає детермінований хеш-код. Для сумісності з архітектурою смарт-контракту, отриманий хеш урізається або конвертується до формату bytes16. Це дозволяє скоротити обсяг збережених даних та уникнути розкриття персональної інформації, зберігаючи при цьому унікальність ідентифікатора.

Наступним важливим етапом є верифікація пристрою за допомогою інтеграції з EVM-сумісним гаманцем. Система підтримує авторизацію через криптогаманці, такі як MetaMask, які дозволяють користувачам ідентифікувати себе через підписування транзакцій. Для цього використовується бібліотека ethers.js, яка забезпечує зручний інтерфейс для підключення до Ethereum-базованих мереж, зчитування адреси користувача, ініціалізації підпису та виклику функцій смарт-контракту. Після генерації visitor ID, користувач підтверджує транзакцію, яка містить хеш його пристрою, і ця транзакція зберігається у блокчейн-мережі Sepolia. Такий підхід забезпечує одноразову верифікацію пристрою без необхідності централізованої перевірки чи зберігання даних на сервері, що значно підвищує безпеку та прозорість процесу.

Крім того, у фронтенд-модулі реалізовано асинхронну передачу зібраних даних на сервер через REST API. Після отримання параметрів від користувача, вони формуються у структурований JSON-об'єкт і передаються на серверну частину, реалізовану на базі .NET Core. На сервері дані обробляються та

зберігаються у реляційній базі даних PostgreSQL для подальшого аналізу, аудиту чи побудови аналітики.

Інтерфейс користувача побудований на базі React-компонентів, зокрема TransactionForm.jsx та Information.jsx. Перший компонент відповідає за логіку підключення гаманця, ініціалізацію транзакції, обробку статусу (наприклад, очікування підпису, підтвердження, помилки), тоді як другий — за виведення інформації про пристрій, статус верифікації та інші допоміжні повідомлення. Компоненти написані з дотриманням принципів гнучкої верстки (responsive design) та оптимізовані для коректного відображення у всіх сучасних браузерах.

Особлива увага приділена юзабіліті — інтерфейс мінімалістичний, інтуїтивно зрозумілий і не потребує від користувача додаткових знань чи реєстрації для взаємодії з системою. Усі основні дії зведені до кількох кроків, що забезпечує швидке проходження верифікації без втручання оператора чи адміністратора. Завдяки використанню сучасних підходів до стилізації та рендерингу, інтерфейс виглядає сучасно, реагує на події в реальному часі, а також адаптований під різні типи пристроїв (десктопи, планшети, смартфони).

У підсумку, фронтенд-модуль виконує роль ключового вузла комунікації між користувачем, блокчейн-мережею та серверною логікою системи. Він забезпечує збір унікальної технічної інформації, формування криптографічного ідентифікатора, інтеграцію з децентралізованими гаманцями, збереження даних у блокчейні та передачу інформації на серверну частину, а також візуалізацію всіх цих процесів через зручний інтерфейс. Така багаторівнева архітектура дозволяє побудувати безпечну, масштабовану та автономну систему верифікації пристроїв, що може бути використана у багатьох сферах — від систем онлайн-голосування до захищених корпоративних порталів.

3.2 Реалізація бекенд модуля

У контексті створення інформаційної системи збору технічних ідентифікаторів користувачів у web3-середовищі важливу роль відіграє серверна частина, що відповідає за зберігання, обробку та перевірку достовірності

отриманих даних. Реалізація бекенд модуля стала ключовим етапом у розробці всієї системи, адже саме на його основі забезпечується надійність і стабільність функціонування програмного комплексу в цілому.

Розробка бекенду здійснювалась із застосуванням сучасного програмного стеку на базі .NET-технологій, зокрема ASP.NET Core. Цей вебфреймворк з відкритим кодом став оптимальним вибором завдяки високій продуктивності, підтримці багатопотокового виконання, а також широким можливостям для інтеграції з базами даних, хмарними рішеннями та контейнерною інфраструктурою. Важливо підкреслити, що в рамках реалізації було прийнято рішення використовувати підхід Minimal API — сучасну концепцію побудови вебсервісів, яка значно спрощує структуру коду, зменшує накладні витрати при обробці HTTP-запитів і дозволяє швидше досягти цільової продуктивності системи.

Використання Minimal API дало змогу уникнути традиційної багатошарової побудови із контролерами, інтерфейсами та розширеною маршрутизацією. Замість цього застосунок містить компактну конфігурацію маршрутів, яка безпосередньо визначає логіку обробки запитів у центральному файлі конфігурації. Такий підхід особливо виправданий у випадках, коли система повинна обробляти великі обсяги вхідної інформації, переданої у форматі JSON, з мінімальними затримками та максимальною пропускною здатністю.

Бекенд модуль реалізує функціонал збереження технічних параметрів користувача, що були зібрані на стороні фронтенду. До таких параметрів, зокрема, належать унікальний ідентифікатор користувача (visitor ID), мова системи, часовий пояс, тип та версія браузера, інформація про аудіоконтекст, WebGL, параметри графічного адаптера, роздільна здатність екрана та інші специфічні ознаки середовища виконання. Ці дані є критично важливими для побудови стійких цифрових відбитків користувачів, що, своєю чергою, дозволяє розв’язувати завдання автентифікації, захисту від шахрайства або ведення аналітики в межах децентралізованих платформ.

Процес обробки запитів побудований відповідно до принципів REST-архітектури: клієнт надсилає запит методом POST для збереження нової інформації, або GET — для перевірки наявності раніше збереженого ідентифікатора. Отримані дані проходять попередню перевірку, валідацію та серіалізацію. Усі параметри зберігаються в структурованому вигляді, що дозволяє зберігати гнучку схему з можливістю її розширення без необхідності створення нових міграцій при кожному додаванні поля.

Для збереження даних було обрано реляційну систему управління базами даних PostgreSQL. Це рішення є виправданим з огляду на її стабільність, відкриту ліцензію, широкі можливості щодо масштабування, а також підтримку спеціального типу даних JSONB. Саме цей тип дозволяє зберігати вкладені JSON-структури без втрати продуктивності, що забезпечує баланс між гнучкістю NoSQL-сховищ та надійністю реляційного підходу. Кожен запис містить основну інформацію про користувача, унікальний ідентифікатор, дату створення та набір параметрів у вигляді JSONB-документу.

Обробка взаємодії з базою даних реалізована за допомогою ORM-технології Entity Framework Core, яка забезпечує автоматичне відображення об'єктної моделі на структуру реляційної бази. Таке рішення дозволяє легко управляти змінами структури БД, створювати міграції, виконувати запити високого рівня абстракції та підтримувати типову архітектуру з використанням патерну Repository.

Особливу увагу було приділено створенню стабільного та ізольованого середовища виконання. Для цього було застосовано інструменти контейнеризації, зокрема Docker. Усі компоненти бекенду, включно з API-сервісом та базою даних, були обгорнуті в окремі контейнери, що дозволило забезпечити відтворюваність середовища, ізоляцію залежностей та зручне масштабування. Конфігурація здійснювалася через файл docker-compose, який описує взаємозв’язки між сервісами, обмеження ресурсів, параметри середовища (логін, пароль, назва БД) та правила збереження даних через томи. У такий спосіб

створено повноцінне середовище, яке може бути розгорнуте як локально, так і у хмарній інфраструктурі.

Важливим аспектом реалізації стало впровадження базових механізмів безпеки. Зокрема, усі вхідні запити перевіряються на коректність формату, наявність обов'язкових полів, допустиму довжину та тип значень. Це дозволяє мінімізувати ризики атак типу SQL-ін'єкцій або масових запитів із некоректними даними. Крім того, система обробки помилок реалізована централізовано — виняткові ситуації логуються та обробляються через відповідні middleware-компоненти. Додатково впроваджено політику CORS, яка обмежує доступ до API лише з певних доменів, що запобігає небажаним міжсайтовим запитам. Також реалізовано обмеження на розмір тіла запиту, що дозволяє уникати перевантаження сервера.

Побудова сервісу від самого початку враховувала можливості масштабування. Усі компоненти спроєктовані таким чином, щоб їх можна було замінювати, оновлювати або розгортати незалежно один від одного. Бекенд реалізовано у вигляді набірної структури з сервісів, що дозволяє безболісно впроваджувати нові функції — зокрема, додавання токенної автентифікації (JWT), механізмів логування активності, створення адміністративного інтерфейсу або інтеграцію з системами моніторингу та алертингу.

Не менш важливою є інтеграція з фронтендом і середовищем web3. Сервіс функціонує як точка обміну даними між клієнтським інтерфейсом та системою зберігання. Користувач, який взаємодіє з децентралізованою платформою, проходить автентифікацію через свій криптогаманець. Після цього на фронтенді генерується унікальний ідентифікатор, який надсилається на сервер разом з технічними параметрами. Таким чином, бекенд зберігає слід взаємодії користувача з платформою, не порушуючи при цьому принципів анонімності, властивих web3-архітектурі.

У підсумку реалізований серверний модуль є повнофункціональним компонентом інформаційної системи, який забезпечує високий рівень продуктивності, гнучкості, безпеки та масштабованості. Його побудова на базі

Minimal API дозволила скоротити час на розробку, зменшити обсяг супровідного коду та зосередитися на вирішенні прикладних завдань. Рішення щодо використання PostgreSQL як основного сховища, а також інструментів контейнеризації, зробило систему придатною до використання в реальних умовах — як у локальному середовищі, так і в масштабованій хмарній інфраструктурі.

Таким чином, бекенд модуль повністю виконує свою функцію у контексті збору технічних ідентифікаторів користувачів, обробки їх унікальності та подальшої інтеграції з іншими компонентами платформи. Його реалізація засвідчує ефективне використання сучасних технологій у розробці інформаційних систем нового покоління, орієнтованих на потреби web3.

3.3 Реалізація взаємодії з блокчейном

Важливою складовою реалізованої системи є механізм взаємодії з публічним блокчейн-середовищем. У сучасному цифровому ландшафті питання автентифікації та верифікації користувачів або пристроїв стоїть надзвичайно гостро. Традиційні централізовані підходи, що покладаються на сервери для зберігання облікових даних, паролів, або ідентифікаторів, мають низку суттєвих недоліків. Вони є вразливими до єдиної точки відмови, цілеспрямованих атак, витоків даних та цензури. Більше того, вони часто вимагають від користувачів надання значної кількості персональної інформації, що створює ризики для приватності. Особливо складною є задача забезпечення унікальності учасника в системах, де це критично важливо, наприклад, при розподілі цифрових активів (airdrop), проведенні голосувань або створенні систем із захистом від Sybil-атак, коли один зловмисник намагається отримати непропорційний вплив шляхом створення великої кількості псевдонімних ідентичностей. Саме для вирішення цих проблем, зокрема для забезпечення одноразової верифікації унікального ідентифікатора пристрою без застосування централізованих методів автентифікації або реєстрації, і була розроблена представлена система. Основна мета — створити децентралізований, надійний та прозорий механізм, який би

дозволив один раз підтвердити унікальність пристрою, прив'язавши його до блокчейн-транзакції, і унеможливити повторну реєстрацію цього ж пристрою.

У системі реалізовано підхід, коли після ретельного збору технічних параметрів пристрою користувача формується унікальний хеш — visitor ID. Цей процес є ключовим для забезпечення унікальності. Технічні параметри можуть включати широкий спектр характеристик, таких як інформація про операційну систему, версію браузера, встановлені шрифти, розширення екрану, часовий пояс, конфігурацію апаратного забезпечення (наскільки це доступно через браузерні API) та інші цифрові "відбитки". Збір цих даних відбувається на стороні клієнта, і важливо, щоб він був достатньо детальним для створення високого ступеня унікальності, але водночас не надто інвазивним, щоб не порушувати приватність користувача. Після збору цих параметрів вони об'єднуються і проходять через криптографічну хеш-функцію, наприклад, SHA-256. Результатом є унікальний рядок фіксованої довжини, який ідентифікує конкретний пристрій (або, точніше, його програмно-апаратну конфігурацію на момент перевірки). Цей visitor ID потім зберігається в блокчейн-мережі Ethereum (Sepolia) через спеціально створений смарт-контракт. Вибір блокчейну як основи для такої системи не є випадковим. Блокчейн забезпечує незмінність (immutability) – одного разу записані дані неможливо змінити або видалити, що гарантує постійність реєстрації. Він забезпечує прозорість – будь-хто може перевірити наявність запису, але не може розшифрувати, що саме являє собою хеш. І найголовніше – він забезпечує децентралізацію, усуваючи потребу в довірі до центрального органу.

Технологічно смарт-контракт було написано мовою Solidity версії ^0.8.x. Вибір Solidity є логічним, оскільки це домінуюча мова для розробки смарт-контрактів на Ethereum та EVM-сумісних блокчейнах. Версія ^0.8.x була обрана через її вбудовані механізми захисту від поширених вразливостей, таких як арифметичні переповнення та недоповнення, що підвищує загальну безпеку контракту. Сам контракт розроблено як легковаговий і зосереджений на виконанні двох основних функцій: фіксація ідентифікатора у постійному

сховищі та перевірка його наявності. Мінімалістичний підхід дозволяє зменшити витрати на розгортання контракту та, що важливіше, знизити вартість кожної транзакції для користувачів, оскільки складність операцій безпосередньо впливає на кількість споживаного газу. Структура контракту передбачає використання mapping(bytes16 => bool), де ключем є 16-байтне значення visitor ID. mapping – це структура даних у Solidity, що працює як хеш-таблиця, дозволяючи ефективно зберігати та отримувати значення за ключем. У цьому випадку, ми зберігаємо логічне значення (bool), яке вказує, чи був даний visitor ID вже зареєстрований (true) чи ні (false, що є значенням за замовчуванням для mapping).

Для збереження газу та зменшення розміру транзакції було свідомо обрано обмежений обсяг ключа — 16 байт, хоча початково він створюється як SHA-256 хеш (32 байти). Це рішення є компромісом між надійністю і ефективністю. Зберігання 32 байт замість 16 байт у сховищі смарт-контракту є значно дорожчим з точки зору газу. Оскільки SHA-256 генерує 32 байти, для використання 16-байтного ключа необхідно було здійснити його обрізання, наприклад, взявши перші або останні 16 байт. Це, безумовно, підвищує теоретичну ймовірність колізій – ситуації, коли два різні набори вхідних даних (два різні пристрої) генерують однаковий 16-байтний хеш. Однак, простір значень для 16 байт є надзвичайно великим, і ймовірність випадкової колізії залишається астрономічно низькою для більшості практичних застосувань, особливо враховуючи, що йдеться про унікальні конфігурації пристроїв, а не про цілеспрямовані атаки на пошук колізій. Для систем типу airdrop або базових whitelist, цей рівень надійності вважається прийнятним, враховуючи значну економію на транзакційних витратах, що особливо важливо при очікуваній великій кількості користувачів. Якщо б система вимагала абсолютного унеможливлення колізій, наприклад, у фінансових системах високого ризику, довелося б використовувати повний 32-байтний хеш, незважаючи на вищі витрати.

Основна функція запису, storeHash(bytes16 _hash), є центральним елементом механізму верифікації. Вона приймає 16-байтний хеш як вхідний параметр. Перш за все, вона реалізує перевірку того, чи існує вже переданий ідентифікатор. Це робиться шляхом звернення до mapping: hashes[_hash]. Якщо такий запис уже є (тобто hashes[_hash] повертає true), транзакція скасовується через require(hashes[_hash] == false, "Hash already exists!");. Оператор require перевіряє умову, і якщо вона не виконується, він зупиняє виконання транзакції, повертає невикористаний газ відправнику та надає повідомлення про помилку. Тим самим виключаючи повторну фіксацію. Це є фундаментальною властивістю системи, що забезпечує одноразовість верифікації. Це критично важливо для забезпечення анти-фрод логіки, коли кожен пристрій (умовно — кожен користувач) може пройти процедуру підтвердження лише один раз. Якщо ж хеш ще не існує, функція встановлює hashes[_hash] = true;, записуючи новий ідентифікатор у сховище блокчейну. Ця операція запису вимагає газу і є транзакцією, що змінює стан блокчейну.

Інша функція — isHashExists(bytes16 _hash) — є view-функцією. У Solidity view-функції (раніше constant) гарантують, що вони не змінюють стан блокчейну. Вони лише читають дані. Важливою перевагою view-функцій є те, що їх можна викликати локально на вузлі Ethereum без створення транзакції і, відповідно, безкоштовно (без gas). Це дозволяє перевірити наявність ID у смарт-контракті, що особливо зручно для попередньої перевірки перед виконанням транзакції storeHash. Фронтенд може викликати цю функцію перед тим, як запропонувати користувачеві підписати транзакцію, і якщо хеш вже існує, просто повідомити користувача про це, уникаючи зайвих дій та потенційних витрат на невдалу транзакцію. Це значно покращує досвід користувача та ефективність системи.

На фронтенді реалізація інтеграції здійснена через бібліотеку ethers.js. Ця бібліотека є однією з найпопулярніших та найпотужніших інструментів для взаємодії з блокчейном Ethereum з JavaScript-додатків. Вона надає інтерфейс для ініціалізації Web3-провайдера. Web3-провайдер – це міст між веб-додатком та

блокчейн-мережею. Найпоширенішим прикладом є MetaMask – браузерне розширення, яке діє як криптогаманець та дозволяє користувачам керувати своїми активами та підписувати транзакції. ethers.js дозволяє легко підключитися до MetaMask або іншого провайдера (наприклад, Infura або Alchemy для доступу до вузлів Ethereum). Після встановлення з'єднання, бібліотека дозволяє формування контракту через ABI-файл. ABI (Application Binary Interface) – це JSON-файл, який описує інтерфейс смарт-контракту: його функції, їхні параметри та типи даних, що повертаються. Маючи ABI та адресу розгорнутого контракту, ethers.js створює об'єкт контракту в JavaScript, який дозволяє викликати функції смарт-контракту так, ніби це звичайні JavaScript-функції. Далі відбувається передавання параметрів та підписання транзакцій користувачем. Після генерації visitor ID на стороні клієнта, скрипт створює нову транзакцію до функції storeHash, передаючи обрізаний хеш як параметр. ethers.js формує запит на транзакцію, і MetaMask (або інший гаманець) показує користувачеві спливаюче вікно з деталями транзакції (куди вона відправляється, яку функцію викликає, скільки газу може коштувати) і просить підтвердження. Користувач повинен власноруч підписати транзакцію своїм приватним ключем, що зберігається в гаманці. Це є доказом його згоди. Після успішного включення транзакції в блок, система отримує підтвердження та змінює локальний статус взаємодії, наприклад, показуючи користувачеві повідомлення про успішну верифікацію.

Для забезпечення надійності взаємодії передбачено кілька запобіжників. Як вже згадувалося, перед ініціалізацією транзакції фронтенд може попередньо викликати функцію isHashExists і у разі її позитивної відповіді — не дозволяти повторне відправлення транзакції. Це перший рівень захисту від непотрібних дій. Також передбачена логіка обробки статусу транзакції (pending, confirmed, failed) з відображенням відповідних повідомлень користувачеві. Взаємодія з блокчейном є асинхронною. Після підписання транзакція потрапляє в мемпул, де чекає на включення в блок майнерами (або валідаторами в PoS). Цей процес може зайняти від кількох секунд до кількох хвилин (або навіть довше у періоди

високого навантаження на мережу). Фронтенд повинен відстежувати цей процес. Він показує статус pending (очікування), а потім, коли транзакція підтверджена (включена в блок і має достатню кількість підтверджень), показує confirmed. Якщо ж транзакція з якоїсь причини не вдалася (наприклад, через помилку require в контракті або недостатню кількість газу), вона отримує статус failed. Коректне інформування користувача про ці стани є надзвичайно важливим для уникнення плутанини та розчарування. Це зменшує кількість некоректних транзакцій і підвищує UX (User Experience).

Використання тестової мережі Sepolia зумовлено її відповідністю специфікаціям головної мережі Ethereum, що дозволяє тестувати смарт-контракти та взаємодію з ними в умовах, максимально наближених до реальних, без ризику втрати справжніх коштів. Sepolia є однією з рекомендованих тестових мереж, яка має активну підтримку та стабільну роботу. Важливою перевагою є підтримка Etherscan (або Sepolia Etherscan) – популярного блокчейн-експлорера, який дозволяє легко моніторити транзакції, перевіряти стан контрактів та аналізувати дані в мережі. Це незамінний інструмент як для розробників, так і для користувачів, які хочуть перевірити статус своєї транзакції. Крім того, Sepolia дозволяє безкоштовного отримання токенів через faucet (кран). Faucet – це сервіси, які роздають тестові ETH, необхідні для оплати газу під час розгортання контрактів та виконання транзакцій у тестовій мережі. Адреса смарт-контракту фіксована, що означає, що всі користувачі взаємодіють з одним і тим же екземпляром контракту, забезпечуючи єдиний реєстр ідентифікаторів. А його ABI доступний у фронтенд-коді, що дозволяє JavaScript-додатку "розуміти", як спілкуватися з контрактом. Кожна транзакція після підтвердження зберігає зв’язок між унікальним пристроєм (через його visitor ID) та криптографічною адресою користувача, оскільки кожна транзакція підписана гаманцем користувача і містить адресу відправника (msg.sender у смарт-контракті, хоча в даному контракті вона явно не зберігається, але вона завжди доступна в даних транзакції). Це створює умови для незаперечної верифікації, де

можна довести, що певна адреса гаманця асоційована з унікальним пристроєм, який пройшов верифікацію.

Важливо зазначити, що жодні персональні дані або вразливі атрибути не потрапляють у блокчейн — лише заздалегідь хешоване значення технічного профілю пристрою. Хешування є одностороннім процесом: легко отримати хеш з вхідних даних, але практично неможливо відновити вхідні дані з хешу. Це означає, що навіть якщо хтось проаналізує дані в блокчейні, він побачить лише набір 16-байтних хешів, які не несуть жодної прямої інформації про користувачів або їхні пристрої. Це дозволяє дотримуватись вимог щодо конфіденційності, таких як GDPR, та уникнути ризиків, пов’язаних із порушенням приватності. Вся чутлива інформація залишається на стороні клієнта і ніколи не передається. Запропонована логіка забезпечує децентралізовану ідентифікацію з одноразовою участю, що є особливо корисним для застосунків типу airdrop, де потрібно справедливо розподілити токени між унікальними учасниками, унеможливлюючи створення ферм акаунтів для отримання більшої частки. Також це корисно для внутрішніх whitelist-систем, де доступ до певних функцій або продажів надається лише попередньо верифікованим учасникам. У захищених голосуваннях це може слугувати механізмом "один пристрій – один голос", хоча для повноцінних систем голосування можуть знадобитися додаткові механізми. На p2p-платформах це може підвищити рівень довіри, підтверджуючи, що учасник є унікальним, а не черговим ботом.

Інтеграція взаємодії з блокчейн-середовищем дозволяє створити систему, в якій не потрібно створювати акаунт, проходити класичну реєстрацію чи верифікацію через e-mail. Цей підхід кардинально відрізняється від традиційних веб-систем. Користувачеві достатньо лише пристрою, з якого буде знято цифровий відбиток, гаманця, який слугує одночасно ідентифікатором та інструментом для підпису транзакцій, та згоди на участь у транзакції, що є свідомим актом підтвердження. Така модель спрощує вхід для користувачів, особливо тих, хто вже знайомий із Web3-технологіями, і усуває необхідність довіряти свої дані ще одному централізованому сервісу. Така модель має

потенціал бути використаною як частина ширших рішень у сфері web3-доступу, де доступ до децентралізованих додатків (dApps) може надаватися на основі такої верифікації. Вона може стати основою для цифрового голосування, де кожен унікальний пристрій отримує право голосу. Або ж може розвинутися у нативну авторизацію без паролів, де комбінація гаманця та верифікованого пристрою слугує надійним методом входу, що є більш безпечним та зручним, ніж традиційні паролі. Цей підхід відкриває двері до нових парадигм взаємодії користувачів із цифровими системами, ставлячи на перше місце децентралізацію, безпеку та контроль користувача над своїми даними та своєю ідентичністю в мережі. Розглядаючи майбутнє, можна уявити собі екосистеми, де така одноразова верифікація пристрою стає базовим рівнем довіри, на якому будуються складніші системи репутації, доступу та взаємодії, повністю реалізовані на принципах децентралізації та криптографічної довіри.

3.4 Підсумки реалізації

У межах реалізації даного дипломного проєкту було успішно створено функціональний прототип програмного засобу, що є свідченням практичної життєздатності та значущості досліджуваної теми. Цей прототип не є лише теоретичною конструкцією, а являє собою працюючу систему, яка втілює ключові принципи децентралізованої ідентифікації пристроїв. Основна функціональність, закладена в основу цього засобу, дозволяє здійснювати комплексний збір технічної інформації про пристрій користувача, що є першим і критично важливим кроком у формуванні унікального цифрового відбитка. Цей збір відбувається з урахуванням сучасних можливостей браузерних API, але водночас із прагненням мінімізувати інвазивність та поважати приватність користувача. На основі зібраних даних система формує унікальний ідентифікатор (visitor ID), який діє як неповторна мітка для конкретної програмно-апаратної конфігурації. Наступним кроком є встановлення нерозривного зв’язку між цим ідентифікатором та Web3-ідентичністю користувача, що реалізується через прив’язування його до криптографічної

адреси гаманця. Це досягається в момент, коли користувач підписує транзакцію, підтверджуючи свою згоду та володіння гаманцем. Кульмінацією цього процесу є одноразова фіксація цієї інформації в блокчейн-мережі. Ця фіксація є незворотною та публічно верифікованою, що забезпечує головну мету – створення доказу унікальності та первинної взаємодії.

Запропонована архітектура охоплює три ключові рівні: фронтенд, бекенд і децентралізовану інфраструктуру, кожен з яких виконує свою специфічну роль, але водночас тісно інтегрований з іншими. Такий трирівневий підхід є класичним для сучасних веб-додатків, але в даному випадку він доповнений децентралізованим компонентом, що надає йому унікальних властивостей. Ця архітектура спроектована таким чином, що забезпечує цілісність даних на кожному етапі, починаючи від збору на клієнті, обробки на сервері та закінчуючи записом у незмінний блокчейн-реєстр. Вона також передбачає масштабованість, особливо завдяки використанню Docker для бекенду та можливості розгортання в хмарних середовищах, а також природній масштабованості блокчейн-мереж (хоча і з певними обмеженнями щодо пропускної здатності та вартості). І, що найважливіше, вона забезпечує відсутність залежності від централізованих сервісів у ключовому аспекті – верифікації унікальності. Хоча система має централізований бекенд для певних операцій, сама логіка підтвердження та зберігання доказу винесена в децентралізоване середовище, що усуває єдину точку відмови та контролю над процесом верифікації.

Фронтенд реалізовано на основі React із використанням Vite для високошвидкісної збірки. Вибір React як основної бібліотеки для побудови користувацького інтерфейсу зумовлений його широкою популярністю, потужною екосистемою та компонентним підходом. React дозволяє створювати складні інтерфейси з меншою кількістю коду та кращою організацією завдяки можливості розбивати UI на незалежні, перевикористовувані компоненти. Використання віртуального DOM (Virtual DOM) забезпечує високу продуктивність рендерингу, оновлюючи лише ті частини сторінки, які дійсно змінилися. Для управління станом могли б бути використані як вбудовані

механізми (Context API, Hooks like useState, useReducer), так і зовнішні бібліотеки (Zustand, Redux ToolKit), що дозволяє гнучко керувати потоком даних у додатку. Vite, у свою чергу, був обраний як сучасний інструмент для розробки та збірки проектів. На відміну від традиційних збирачів, таких як Webpack, Vite використовує нативні ES-модулі браузера під час розробки, що забезпечує майже миттєвий запуск сервера розробки та надзвичайно швидке гаряче оновлення модулів (Hot Module Replacement - HMR). Це значно прискорює ітераційний процес розробки. Для продакшн-збірки Vite використовує Rollup, що дозволяє створювати оптимізовані та ефективні бандли. У модулі інтегровано модифіковану бібліотеку FingerprintJS для збору параметрів браузера. FingerprintJS є потужним інструментом, здатним збирати десятки сигналів з браузера для створення стабільного та унікального ідентифікатора. Ці сигнали включають: версію браузера та операційної системи, встановлені шрифти (перевірка наявності певного списку шрифтів), timezone користувача, мови, налаштовані в браузері, роздільну здатність екрану та глибину кольору, підтримку та параметри WebGL (включаючи рендерер та вендора графічного адаптера), параметри Canvas API (рендеринг певного зображення чи тексту та хешування результату), параметри AudioContext API, список встановлених плагінів, налаштування cookies та localStorage, а також багато інших менш очевидних параметрів. "Модифікована" версія, ймовірно, означає, що або використовувалася кастомізована версія бібліотеки, або її вихідні дані додатково оброблялися – можливо, для виключення надто нестабільних параметрів або для адаптації до специфічних потреб проекту, а також для посилення приватності шляхом відмови від збору найбільш чутливих даних. Отримані дані перетворюються у криптографічний хеш, найімовірніше, SHA-256, що забезпечує їх анонімізацію та створення єдиного ідентифікатора. Цей хеш надалі використовується для створення транзакції до смарт-контракту через бібліотеку ethers.js та взаємодію з гаманцем користувача, як було описано раніше. Та одночасно зберігається у серверній базі даних через REST API. Цей крок є важливим для додаткової аналітики, моніторингу та можливості швидкого

пошуку або асоціації даних без необхідності постійного звернення до блокчейну, що може бути повільним та витратним. Фронтенд відправляє цей хеш разом із зібраними (можливо, також хешованими або анонімізованими) параметрами на бекенд.

Бекенд побудовано на ASP.NET Core із застосуванням Entity Framework Core для ORM-взаємодії з PostgreSQL. ASP.NET Core є сучасним, високопродуктивним та кросплатформним фреймворком від Microsoft для створення веб-додатків та API. Його вибір забезпечує надійність, масштабованість та можливість використання мови C# з її сильною типізацією та багатими можливостями. Архітектура бекенду, ймовірно, слідує патернам MVC або Web API, з чітким розділенням відповідальності між контролерами (обробка HTTP-запитів), сервісами (бізнес-логіка) та репозиторіями (взаємодія з даними). Entity Framework Core (EF Core) виступає як Object-Relational Mapper (ORM), що дозволяє розробникам працювати з базою даних, використовуючи об'єкти C#, замість написання SQL-запитів вручну. Це спрощує розробку, підвищує її швидкість та зменшує кількість помилок. EF Core підтримує PostgreSQL через відповідний провайдер, що дозволяє використовувати всі переваги цієї потужної, відкритої та надійної системи управління базами даних. PostgreSQL відома своєю стабільністю, підтримкою складних запитів та типів даних, а також хорошою продуктивністю. Бекенд виконує функції прийому вхідних JSON-структур від фронтенду, що містять visitor ID та, можливо, інші дані. Він проводить серіалізацію параметрів (перетворення JSON в об'єкти C# і навпаки), перевірки унікальності запису (ймовірно, перевіряючи, чи існує вже такий visitor ID у локальній базі даних) та їх збереження у сховище. Ця локальна база даних може слугувати кешем або додатковим джерелом інформації, доповнюючи дані з блокчейну. База даних у Docker-середовищі означає, що як сам бекенд-додаток, так і база даних PostgreSQL, розгортаються у вигляді контейнерів. Це забезпечує ізоляцію, відтворюваність середовища та легкість розгортання на будь-якій машині чи хмарній платформі. Docker також сприяє можливості масштабування (можна легко запускати більше екземплярів бекенду

або бази даних) та дозволяє автоматичне розгортання через CI/CD пайплайни, а також забезпечує гнучке конфігурування середовища виконання через змінні середовища або файли конфігурації.

Смарт-контракт, написаний на Solidity та розгорнутий у мережі Ethereum Sepolia, реалізує просту, але ефективну логіку контролю унікальності visitor ID. Як вже детально обговорювалося, його мінімалістичний дизайн є його сильною стороною. Використання Solidity 0.8.x, mapping(bytes16 => bool), функцій storeHash з перевіркою require та isHashExists (view) створює надійний, хоча й компромісний з точки зору ризику колізій, механізм. Контракт забезпечує публічний, незмінний і децентралізований спосіб зафіксувати факт первинної взаємодії користувача з системою. "Публічний" означає, що будь-хто може перевірити наявність хешу. "Незмінний" означає, що одного разу записаний хеш не може бути видалений або змінений. "Децентралізований" означає, що цей запис не контролюється жодною окремою організацією, а підтримується всією мережею Ethereum. Ця комбінація властивостей є ключовою, оскільки вона виключає повторну реєстрацію з одного пристрою (за умови, що visitor ID є стабільним та унікальним) та забезпечує антифрод-функціональність без порушення конфіденційності, оскільки зберігаються лише абстрактні хеші, а не персональні дані. Це є елегантним рішенням для багатьох Web3-задач.

Загальна реалізація демонструє, що створений прототип є не просто набором окремих компонентів, а цілісною системою, де кожен елемент виконує свою роль у загальному процесі. Вона наочно показує, що технології web3, зокрема блокчейн та децентралізовані смарт-контракти, можуть бути ефективно використані не лише для фінансових транзакцій, що є їхнім найвідомішим застосуванням, а й для побудови систем технічної ідентифікації, що не потребують персональних даних. Це розширює горизонти застосування блокчейну, показуючи його потенціал у вирішенні проблем ідентичності, довіри та безпеки в цифровому світі. Проект доводить, що можна створити систему, яка поєднує зручність веб-інтерфейсу (React), надійність серверної логіки (ASP.NET Core) та переваги децентралізації (Ethereum), досягаючи синергетичного ефекту.

Усі компоненти системи функціонально узгоджені, що було досягнуто через ретельне проектування архітектури та протоколів взаємодії. Вони взаємодіють через чітко визначені інтерфейси – REST API між фронтендом та бекендом, та ABI смарт-контракту для взаємодії з блокчейном. Ця чіткість інтерфейсів також означає, що компоненти можуть бути адаптовані для реального використання в різних сферах. Наприклад, фронтенд може бути інтегрований у будь-який існуючий веб-додаток, бекенд може бути розширений для підтримки додаткової логіки, а смарт-контракт може бути розгорнутий на інших EVM-сумісних мережах, включаючи L2-рішення для зменшення витрат. Потенційні сфери застосування є широкими — від whitelist-механік для NFT-дропів або токен-сейлів, до доступу до внутрішніх протоколів DAO (децентралізованих автономних організацій), де такий механізм може слугувати одним із способів підтвердження унікальності учасників при голосуванні або розподілі завдань. Цей проект слугує міцним фундаментом для подальших досліджень та розробок у галузі децентралізованої ідентифікації та побудови довірчих систем нового покоління. Він підкреслює перехід до більш орієнтованих на користувача, прозорих та безпечних цифрових взаємодій, де блокчейн відіграє роль не лише фінансової інфраструктури, а й універсального шару довіри.

4 РОЗГОРТАННЯ ТА ТЕСТУВАННЯ ПРОГРАМНИХ МОДУЛІВ

4.1 Розгортання логального середовища

Розгортання та налаштування повноцінного середовища виконання є критично важливою частиною дипломного проєкту, оскільки лише у функціональному, ізольованому оточенні можна коректно перевірити стабільність розробленої архітектури, взаємодію її компонентів, а також здійснити моделювання реальної роботи системи. У рамках даної дипломної роботи було реалізовано розгортання всієї системи локально, на операційній системі Windows, з використанням технології Docker для контейнеризації ключових компонентів, зокрема бекенд-сервісу на ASP.NET Core та реляційної бази даних PostgreSQL.

Docker — це платформа з відкритим вихідним кодом, що дозволяє автоматизувати розгортання, масштабування та керування застосунками в ізольованих контейнерах. На відміну від віртуальних машин, які вимагають окрему операційну систему, Docker використовує ядро хост-системи, що значно зменшує накладні витрати та підвищує швидкодію. Контейнери містять все необхідне для виконання застосунку: код, бібліотеки, залежності, налаштування — і саме це забезпечує їх портативність і повторюваність.

Таке рішення повністю відповідає технічним вимогам проєкту, який передбачає децентралізований збір технічних ідентифікаторів користувачів, їх подальше збереження у структурованому форматі та перевірку у блокчейн-мережі без централізованих акаунтів. Відмова від класичних серверних конфігурацій на базі VPS та доменів дозволила сконцентруватись саме на ефективності локального тестування, простоті масштабування та повторного запуску системи. Крім того, контейнеризоване середовище забезпечує повну повторюваність розгортання — незалежно від ОС або специфікації машини розробника.

На початковому етапі було зосереджено увагу на контейнеризації бекенд-модуля. Основною метою було ізолювати середовище виконання API-сервісу,

уникнути залежностей від глобальних налаштувань .NET SDK на машині та забезпечити контрольовану взаємодію з іншими модулями системи. У межах проєкту було розроблено багатоступеневий Dockerfile для ASP.NET Core-додатку, який передбачає окремі стадії для складання, публікації та запуску сервісу. Це дало змогу оптимізувати фінальний образ, видаливши непотрібні файли та зменшивши його розмір. На етапі запуску API-сервісу експортуються відповідні порти, що дозволяє локальним клієнтам (зокрема frontend-частині) звертатись до нього через HTTP.

Особливу увагу було приділено налаштуванню бази даних. У системі використовується PostgreSQL як основне сховище структурованої інформації, включно з visitor ID, user-agent, IP-адресою та іншими параметрами. Для забезпечення повної ізоляції цієї компоненти також було створено окремий контейнер з образом PostgreSQL, із заздалегідь заданими змінними середовища для ініціалізації користувача, пароля та імені бази даних. Дані зберігаються у зовнішньому томі, прив’язаному до шляху всередині контейнера, що дозволяє зберігати інформацію між перезапусками контейнера. Це рішення дозволяє повністю абстрагуватись від локальних установок PostgreSQL і уникнути конфліктів версій або залежностей.

З метою спрощення запуску обох сервісів — API та бази даних — було створено конфігураційний файл docker-compose. Він дозволяє запустити всю систему однією командою, автоматично створюючи необхідну внутрішню мережу, налагоджуючи залежності між контейнерами та визначаючи порядок їх ініціалізації. Завдяки цьому значно зменшується час на підняття інфраструктури, а також виключаються помилки, пов’язані з ручним налаштуванням. Docker Compose — це зручний інструмент для організації та керування кількома контейнерами в рамках одного середовища, що є особливо актуальним для мікросервісної архітектури, до якої тяжіє сучасний веб-розвиток.

Ще одним важливим аспектом локального деплою стало підключення бекенду до бази даних. У файлі конфігурації системи вказано connection string, який використовує ім’я docker-сервісу PostgreSQL як hostname. Це дозволяє

контейнерам спілкуватись між собою у межах docker-compose мережі без необхідності прив’язки до зовнішніх IP-адрес чи портів. Окрім цього, при запуску API відбувається автоматична перевірка наявності необхідних таблиць у базі та, у разі потреби, виконуються міграції з використанням Entity Framework Core. Цей ORM-фреймворк дозволяє працювати з базами даних як з об'єктами, спрощуючи створення, зчитування та оновлення даних, а також забезпечуючи типову перевірку правильності структури сховища.

Особисто я забезпечив реалізацію усього пайплайну контейнеризації — від написання Dockerfile до налагодження docker-compose.yml. Це дало мені змогу глибше зрозуміти процеси конфігурації середовища, налагодження міжконтейнерної взаємодії, а також основи безпеки при роботі з обмеженими мережами та змінними середовища. Я також самостійно проводив тести на коректність запуску, перевіряв підключення до бази, валідність HTTP-запитів та логування помилок.

Окремо слід зазначити, що фронтенд частина системи наразі не контейнеризована, але її можна легко винести в окремий Docker-контейнер. Це стане наступним етапом роботи над проектом і дозволить запускати всю систему у вигляді трьох взаємодіючих сервісів — API, бази даних і клієнтського інтерфейсу. На цьому етапі фронтенд працює як окремий локальний вебзастосунок, що звертається до контейнеризованого API, підключеного до бази. Такий підхід дозволяє перевірити функціональність ключових модулів і їхню взаємодію без необхідності оренди хостингових ресурсів.

Локальне розгортання, хоч і не включає реального блокчейн-середовища, повністю підтримує моделювання взаємодії з Ethereum через інтерфейс MetaMask та бібліотеку ethers.js. У рамках тестування використовувалась мережа Sepolia, яка дозволяє ініціювати транзакції через гаманці користувачів і здійснювати попередню перевірку visitor ID. Це забезпечує повну відповідність фронтенд-бекенд-блокчейн архітектурі, закладеній у концепції системи.

Розгортання без використання VPS чи доменів не лише не обмежило функціональність проекту, а навпаки, дозволило сфокусуватись на внутрішній

логіці, оптимізації ресурсів та досягненні високого рівня модульності. Я мав змогу самостійно налаштувати всі компоненти, зіткнутись із типовими проблемами (на кшталт конфліктів портів, неправильних залежностей чи помилок ініціалізації) і знайти для них практичні рішення.

Продовжуючи розгляд процесу розгортання локального середовища, доцільно поглибити аналіз додаткових аспектів, які стосуються як технічної реалізації, так і організаційної структури проєкту. Важливо розуміти, що навіть при локальному розгортанні система повинна відповідати принципам масштабованості, відмовостійкості та модульності.

Одним з ключових інструментів, що забезпечують модульність, є система управління версіями. У межах проєкту використовувалась Git як основний засіб відстеження змін коду, а також як платформа для створення окремих гілок розробки, тестування та інтеграції. Це дозволяє ізолювати нові функції, реалізовувати pull request та проводити код-рев'ю, навіть у локальному середовищі.

Крім того, проєкт супроводжується структурованою документацією у вигляді Markdown-файлів (README.md, INSTALL.md), що дає змогу швидко ознайомити нових учасників із процедурою розгортання, базовою структурою API, конфігураціями середовища та особливостями запуску. Документація генерується напівавтоматично за допомогою генераторів документації на базі шаблонів або IDE (наприклад, Visual Studio Code із плагінами).

Ще одним важливим блоком у структурі проєкту є моніторинг. У контейнеризованому середовищі Docker використовується вбудована система логування, яка дозволяє отримувати stdout/stderr з контейнерів. Ці логи агрегуються за допомогою docker logs, а також можуть бути переадресовані у зовнішні сервіси (наприклад, Grafana Loki або ELK Stack) для подальшого аналізу. Також було протестовано базове інтегроване рішення для метрик, реалізоване за допомогою Prometheus і node_exporter. Такий підхід дозволяє оцінити використання ресурсів окремими модулями, виявити вузькі місця та оптимізувати конфігурацію контейнерів.

Варто зазначити, що безпечність середовища відіграє ключову роль. Для зберігання секретів (ключів доступу, токенів) використовується змінні середовища у docker-compose файлі, які не зберігаються в репозиторії. У перспективі інтеграція з Vault або HashiCorp дозволить реалізувати централізоване управління секретами навіть у локальних середовищах.

Також було досліджено можливість масштабування локального розгортання до кластерного варіанту з використанням Docker Swarm або Kubernetes. Попри те, що реалізація цього підходу виходить за межі дипломного проєкту, було створено базову структуру yaml-файлів для можливого розгортання в Kubernetes-кластері. Ці конфігурації включають опис сервісів, ingress-контролерів, persistent volume claims для збереження даних та секретів, а також базову політику обмеження ресурсів.

У результаті виконаної роботи мною було повністю реалізовано локальне розгортання багатокомпонентної системи з підтримкою бази даних, API та взаємодії з блокчейном. Усі елементи інтегровані через єдиний пайплайн, що дозволяє миттєво запускати, тестувати та модифікувати окремі частини без потреби в зовнішньому хостингу чи складному CI/CD. Такий підхід особливо важливий у контексті розробки децентралізованих застосунків, які повинні бути готові до запуску в будь-якому середовищі — від локального ПК до розподіленої мережі.

4.2 Тестування застосунка

Розроблений застосунок для збору інформації про користувачів у середовищі web3 (з фронтендом на React, серверним .NET Core API, базою даних PostgreSQL та інтеграцією з блокчейном Ethereum (мережа Sepolia) через смартконтракт) пройшов комплексне тестування на кількох етапах. Задля забезпечення коректної роботи всіх компонентів системи було виконано ручне тестування API, модульне (unit) тестування серверної логіки та автоматизоване тестування інтерфейсу користувача. Кожен з цих етапів мав на меті виявити

можливі помилки на відповідному рівні та переконатися у правильності реалізації функціональності застосунка.

Ручне тестування API через Postman. На першому етапі здійснювалося ручне тестування серверних веб-сервісів за допомогою Postman. Postman – це спеціалізований HTTP-клієнт для тестування API, який дозволяє розробникам і тестувальникам формувати та надсилати HTTP-запити, групувати їх у колекції, змінювати параметри запитів, додавати контрольні точки (перевірки) та автоматизувати виконання наборів запитів.

Використовуючи цей інструмент, було перевірено ключові ендпоїнти .NET Core API сервера на коректність обробки запитів і відповідей. Тестувальник вручну відправляв HTTP-запити (GET, POST, PUT тощо) до відповідних URL-адрес API та аналізував відповіді від сервера, щоб виявити наявність помилок у роботі служб.

Зокрема, перевірялися ендпоїнти для отримання даних користувачів та додавання нових записів. Наприклад, для перевірки отримання інформації про користувача виконувався GET-запит на ендпоїнт (умовно) /api/users/{id} з існуючим id користувача; очікуваною була відповідь 200 OK із правильним JSON, що містить дані відповідного користувача. Для додавання нового користувача перевірявся POST-запит (наприклад, на /api/users) з передачею в тілі запиту JSON-даних нового користувача (ім’я, адреса, тощо) – у відповіді очікувався статус успіху (наприклад, 201 Created) та структура JSON із деталями щойно створеного користувача. В результаті таких запитів вручну перевірялося, що сервер дійсно зберіг отримані дані у базі PostgreSQL та повернув клієнту правильну інформацію (наприклад, ідентифікатор нового запису, внесені дані тощо). Окремо тестувалися ситуації з некоректними даними: надсилалися запити з неповною або помилковою інформацією, і Postman-фахівець перевіряв, що API повертає відповідні коди помилок (наприклад, 400 Bad Request для невірного формату даних) та змістовні повідомлення про помилку. Таким чином, ручне тестування через Postman підтвердило відповідність роботи серверних ендпоїнтів вимогам: кожен запит приводив до очікуваних змін (створення,

отримання чи оновлення даних) і отримання коректної відповіді від серверної частини, включно з перевіркою того, що інтеграція з блокчейном (через відповідні API-методи, які викликають смартконтракт) здійснюється без збоїв і дає очікуваний результат. Модульне тестування серверної логіки (NUnit). Наступним кроком було модульне (unit) тестування коду серверного застосунка. Модульне тестування – це метод тестування програмного забезпечення, за якого кожен окремий модуль або компонент програмного коду перевіряється незалежно від інших.

Модулем вважається найменша логічно завершена частина програми (функція, метод або клас), яку можна протестувати ізольовано. Для реалізації таких тестів у середовищі .NET було використано фреймворк NUnit.

NUnit — відкрите середовище модульного тестування застосунків для .NET, спочатку перенесене з Java-бібліотеки JUnit.

Цей фреймворк надає розробнику зручні засоби для написання тестів (атрибути позначення тестових методів, такі як [Test], і методи Assert для перевірки умов) та виконання наборів тестів, що перевіряють коректність роботи окремих частин коду.

У межах проєкту за допомогою NUnit було створено набір юніт-тестів для ключових функцій серверної частини API. Тести охопили основні методи бізнес-логіки: зокрема, методи, що відповідають за обробку даних користувачів (створення нового користувача, отримання інформації про користувача, оновлення даних) та функціональність, пов’язану зі взаємодією зі смартконтрактом (наприклад, методи, що відправляють транзакції або читають дані з блокчейну). Кожен тест перевіряв, що відповідний метод коректно обробляє передані в нього дані і повертає очікуваний результат. Для цього в тестах задавалися заздалегідь визначені вхідні дані, викликався метод API (або відповідний метод сервісного шару) і за допомогою тверджень (Assert) перевірялося, чи відповідає фактичний результат очікуваному. Наприклад, тест методу створення користувача міг передавати йому об’єкт з даними нового користувача та очікувати, що метод поверне об’єкт з присвоєним унікальним

ідентифікатором і статусом успішного збереження. Після виклику методу тест перевіряв, що новий користувач з’явився у сховищі (для юніт-тестів сховище могло бути замінене на імітоване або використано тестову базу даних) і що повернуті дані точно збігаються з введеними. Окрім перевірки правильності обробки даних, значна увага приділялася перевірці обробки помилок. Були написані тестові випадки, що моделюють нетипові або некоректні ситуації: наприклад, виклик методу зі некоректними або неповними даними, ситуація, коли база даних не доступна, або коли смартконтракт повертає помилку. У таких випадках очікувалося, що метод належним чином обробить ситуацію – кине виняток визначеного типу або поверне спеціальний код/повідомлення про помилку. За допомогою NUnit перевірялося, що при виникненні таких умов дійсно відбувається передбачена реакція (через Assert.Throws для винятків або через перевірку значення результату). Усі зовнішні залежності (база даних, блокчейн) в юніт-тестах за можливості ізолювалися – шляхом використання макетів (mock) або тестових дублікатів – щоб зосередитися саме на логіці методів. Результатом модульного тестування стала впевненість, що внутрішні компоненти серверної частини працюють правильно в різних сценаріях і стійко поводяться при некоректних даних чи помилках середовища.

Автоматизоване тестування інтерфейсу (Playwright). На завершення було проведено автоматизоване тестування фронтенд-частини застосунка з використанням фреймворку Playwright. Playwright – це інструмент автоматизації end-to-end тестування, який надає високорівневий API для керування веб-браузером, дозволяючи взаємодіяти з елементами сторінки та імітувати реальну поведінку користувача.

Даний інструмент підтримує різні браузери (Chromium, Firefox, WebKit) і може використовуватися з кількома мовами програмування (зокрема TypeScript/JavaScript, Python, C#), що робить його універсальним для тестування вебзастосунків на React. Автотестування через Playwright фактично виконувало повноцінні end-to-end сценарії: запускався браузер, завантажувався фронтенд-застосунок React та проводилась емуляція дій користувача, після чого

перевірявся стан інтерфейсу. У межах цього етапу були автоматизовані найважливіші функції і сценарії взаємодії користувача із системою. Зокрема, автотести перевіряли роботу таких сценаріїв, як реєстрація/додавання нового користувача через веб-форму, перегляд списку користувачів та інших даних, а також взаємодія користувача з елементами інтерфейсу, що запускають виконання смартконтракту (наприклад, натискання кнопки для підтвердження транзакції або оновлення даних, які зберігаються на блокчейні).

Для кожного такого сценарію тестовий скрипт Playwright автоматично виконував послідовність кроків, подібних до дій реального користувача: переходив на потрібну сторінку застосунка, заповнював необхідні поля вводу (імітуючи введення тексту в текстові поля, вибір опцій тощо), натискав на кнопки або посилання, які ініціюють певну функціональність, та відслідковував реакцію системи.Перевірки (асерції) в таких тестах виконувалися безпосередньо на рівні DOM сторінки: тест чекав появи певних елементів чи тексту. Якщо мав з’явитися сповіщувальний банер про успіх операції – скрипт перевіряв його наявність і правильний текст; якщо очікувалось, що після дії користувача зміняться дані на екрані – тест фіксував ці зміни (наприклад, новий рядок у таблиці з даними користувачів). Також автоматизовано перевірялися негативні сценарії на рівні інтерфейсу: наприклад, якщо користувач залишив обов’язкові поля порожніми і натиснув підтвердження, тест Playwright мав переконатися, що з’явилося відповідне повідомлення про помилку підсвічування полів вводу. Завдяки тому, що Playwright здійснює тестування у справжньому браузері, вдалося перевірити коректність роботи інтерфейсу у реальних умовах відображення та взаємодії. Це дало впевненість, що фронтенд на React правильно взаємодіє з серверним API та відображає дані, отримані як з бази даних, так і з блокчейна (через API-сервіси), відповідно до задуманої логіки. Автоматизовані UI-тести стали доповненням до ручних і модульних тестів, забезпечивши повний цикл перевірки застосунка від рівня окремих функцій до рівня кінцевого користувача.

Таким чином, трирівневий підхід до тестування – поєднання ручного тестування API (Postman), модульного тестування серверної частини (NUnit) та автоматизованого кінцевого тестування інтерфейсу (Playwright) – дозволив всебічно перевірити працездатність розробленого програмного засобу. Кожний етап тестування виявив та усунув потенційні проблеми на своєму рівні, що в результаті забезпечило високу якість і надійність застосунка у web3-середовищі.

4.3 Підсумки розгортання та тестування застосунка

Етап тестування та подальшого розгортання системи є важливою завершальною фазою життєвого циклу розробки програмного забезпечення. У межах цієї бакалаврської роботи було здійснено повний цикл валідації розробленої системи, починаючи від локального запуску модулів і завершуючи інтеграцією всіх компонентів у спільне ізольоване середовище. Основна увага була зосереджена на перевірці коректності взаємодії між підсистемами, дотриманні логіки обробки даних та загальній стабільності програмного комплексу в умовах стандартного та нештатного навантаження.

Підсумкова перевірка охоплювала як перевірку інтерфейсів прикладного програмування (API), так і тестування взаємодії користувача з клієнтською частиною. Програмний код було приведено у стан, придатний до роботи у виробничому середовищі. Значну увагу приділено ергономіці розгортання та універсальності запуску, що особливо важливо у випадках, коли середовище може варіюватися залежно від цільової платформи. Завдяки впровадженій контейнеризації усі модулі були запаковані у відповідні ізольовані сервіси, які взаємодіють через уніфіковану внутрішню мережу. Такий підхід дозволив мінімізувати зовнішні залежності й забезпечити повну повторюваність результатів у різних середовищах запуску.

Загальна інфраструктура розгортання, що базується на Docker та Docker Compose, забезпечила зручну ініціалізацію сервісів. Розроблені конфігурації дають змогу безперешкодно запускати систему навіть на нових машинах без необхідності налаштування середовища вручну. Цей підхід є особливо

актуальним для модульних систем, які можуть доповнюватись новими функціональними блоками або масштабуватись у хмарних середовищах.

У межах перевірки було досліджено взаємодію між фронтендом і бекендом, зокрема передачу унікальних технічних ідентифікаторів, верифікацію в блокчейн-мережі та збереження результатів до бази даних. Усі компоненти були перевірені на узгодженість та логічну цілісність. Також оцінено продуктивність системи при паралельній обробці кількох запитів, що імітує реальне навантаження. Система демонструє стабільність, передбачувану реакцію на нештатні ситуації, а також готовність до подальшого розширення функціональності.

Було також сформовано необхідну документацію, що включає інструкції з розгортання, опис налаштувань оточення, схеми компонентів системи та рекомендації щодо подальшого масштабування. Усі розгортання були зафіксовані в системі контролю версій, що гарантує збереження змін та можливість відновлення на будь-якому етапі розробки.

Для ілюстрації підсумків реалізації наведено візуальні фрагменти, які відображають ключові етапи розгортання та взаємодії з користувачем. Це включає демонстрацію мережевих запитів, відповідей від серверної частини, приклади взаємодії з гаманцем та збереження інформації до бази даних. Додатково представлено знімки тестового оточення та журналів, які підтверджують успішне виконання транзакцій та стабільність логіки роботи додатку.

Система реалізована таким чином, щоб забезпечити максимальну адаптивність до змінних умов середовища, а також можливість масштабування як у горизонтальному, так і у вертикальному напрямках. У перспективі передбачається розгортання системи на віддаленому хостингу з інтеграцією додаткових сервісів сповіщень та зовнішніх джерел автентифікації.

Таким чином, етап розгортання та тестування не лише підтвердив працездатність створеного рішення, а й відкрив потенціал для його подальшого вдосконалення. Система готова до адаптації під нові сценарії використання,

включаючи інтеграцію з реальними смарт-контрактами в публічному блокчейні. Усі дії, пов’язані з запуском, документовані й можуть бути повторені як у навчальних, так і у виробничих цілях. У результаті виконаної роботи створено надійну основу для подальшої експлуатації розробленого продукту у реальному середовищі.

Рисунки з моментів тестування (див. рис. 4.1, рис.4.2 та рис.4.3)

Рисунок 4.1 — Тестування ендпойнту create

Результатом успішного тестування є транзакція в мережі Sepolia яка має такий індитифікатор: 0xf3bed174fcca1ac3a78cd0e9adfd2b521973c265ecec2e3cb0eaf149c0ee293a.

Рисунок 4.2 — Тестування endpoint отримання фінгерпінта користувача

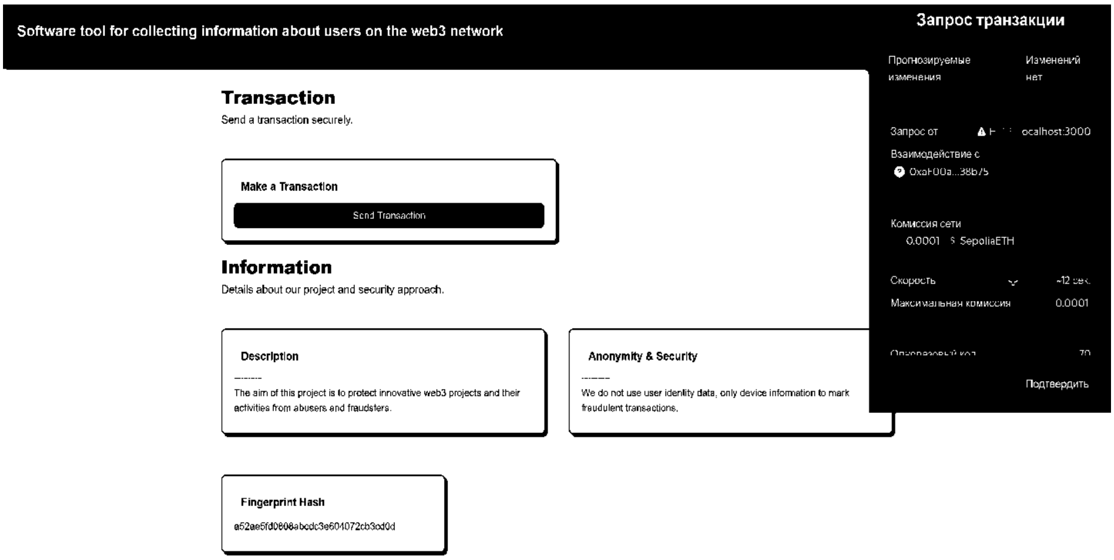

Рисунок 4.3 — Автоматичне тестування через Playwright (момент підтвердження транзакції)

ВИСНОВКИ

Розроблений інструмент у повному обсязі продемонстрував здатність вирішувати проблему розпізнавання унікальних користувачів без централізованої автентифікації та без використання персональних даних. Його функціональність охоплює всі ключові аспекти технічної ідентифікації у web3-середовищі — від збору характеристик пристрою до запису ідентифікатора в блокчейн. Застосування хеш-функцій та смарт-контрактів дозволило створити захищений, одноразовий, незмінний механізм перевірки унікальності, який не потребує додаткових перевірок на стороні сервера.

Системний підхід до архітектури проекту дозволив досягти високої модульності, що стало перевагою при розгортанні компонентів та налагодженні взаємодії між ними. Кожен рівень — фронтенд, сервер, база даних, контракт — функціонує як незалежна логічна одиниця, що спрощує масштабування, відлагодження та впровадження нових функцій. Завдяки цьому систему можна адаптувати до нових умов використання без необхідності повної перебудови.

На практиці запропоноване рішення може знайти застосування у різноманітних сценаріях: від захисту децентралізованих платформ від повторних взаємодій до підвищення достовірності статистичних даних у дослідницьких і комерційних проєктах. Його використання дозволяє зменшити навантаження на бекенд, оскільки перевірка унікальності відбувається на рівні блокчейну, де дані є незмінними та публічними. Це підвищує прозорість взаємодій і забезпечує відтворюваність результатів навіть через тривалий проміжок часу.

Система також є демонстрацією ефективного використання open-source інструментів і сучасного стеку технологій — від Vite та React до Docker, PostgreSQL і Solidity. Усі компоненти інтегруються через стандартизовані інтерфейси, що дозволяє замінювати або оновлювати їх без втрати сумісності. Завдяки цьому створено фундамент для подальших досліджень і практичного впровадження подібних архітектур в інші сфери — наприклад, у смарт-міста, цифрові паспорти пристроїв, доступ до децентралізованих API, або навіть у фільтрацію взаємодій в публічних DAO.

Під час реалізації були вирішені низка технічних викликів: забезпечено стабільну роботу API при передачі великих обсягів параметрів, узгоджено формат даних між фронтендом, сервером і контрактом, побудовано систему передачі даних у форматі JSON без втрати гнучкості. Окрема увага приділялася мінімізації gas-витрат у смарт-контракті за рахунок оптимізації формату ідентифікатора.

Проект також продемонстрував, що ідентифікація у web3 не обов'язково має базуватись на токенах, NFT чи wallet-балансі. Навпаки, технічні властивості пристрою, якщо їх обробити коректно, можуть служити доволі надійною ознакою унікальності користувача в умовах високої анонімності. Це відкриває шлях до створення нових типів безпарольної ідентифікації, децентралізованих авторизацій та інструментів взаємодії у web3-просторі.

Зокрема, виконано:

— проектування і реалізацію архітектури з чітко визначеним поділом на фронтенд, бекенд, базу даних та блокчейн-компонент;

— підключення EVM-сумісного гаманця MetaMask для автентифікації користувача через криптографічну підписку транзакцій;

— формування унікального ідентифікатора (visitor ID) на основі технічного фінгерпринту пристрою без збору персональних даних;

— передачу сформованого JSON-пакету на серверну частину та його збереження у реляційній базі даних PostgreSQL;

— реалізацію механізму одноразової верифікації у блокчейн-мережі Sepolia з використанням смарт-контракту на мові Solidity.

Отже, виконана робота не лише досягла поставленої мети, а й продемонструвала практичне застосування гібридного підходу до ідентифікації: поєднання локального технічного фінгерпринтингу з публічною фіксацією результату у блокчейні. Отриманий результат має як інженерну, так і дослідницьку цінність, підтверджуючи перспективність даного напряму для подальших розробок у сфері web3-ідентифікації, протидії абузам і підвищення рівня довіри у цифрових системах нового покоління.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1 Arcana Network. The Complete Web3 Authentication Guide in 2023 [Електронний ресурс] – Режим доступу: https://arcana-network.medium.com/the-complete-web3-authentication-guide-in-2023-6b9bb2407ee4

2 Fingerprint. How to Integrate Fingerprinting Into Your React Application [Електронний ресурс] – Режим доступу: https://fingerprint.com/blog/browser-fingerprinting-react/

3 QuickNode. How to Use Keccak256 Hash Function with Solidity [Електронний ресурс] – Режим доступу: https://www.quicknode.com/guides/ethereum-development/smart-contracts/how-to-use-keccak256-with-solidity

4 QuickNode. How to Write an Ethereum Smart Contract Using Solidity [Електронний ресурс] – Режим доступу: https://www.quicknode.com/guides/ethereum-development/smart-contracts/how-to-write-an-ethereum-smart-contract-using-solidity

5 Kashif Soofi. REST API with ASP.NET Core 7 and Postgres [Електронний ресурс] – Режим доступу: https://kashifsoofi.github.io/aspnetcore/rest/postgres/restapi-with-asp.net-core-7-and-postgres/

6 C# Corner. Building a Robust ASP.NET Core Web API with PostgreSQL [Електронний ресурс] – Режим доступу: https://www.c-sharpcorner.com/article/building-a-powerful-asp-net-core-web-api-with-postgresql/

7 Neon. Building a RESTful API with ASP.NET Core, Swagger, and Neon [Електронний ресурс] – Режим доступу: https://neon.tech/guides/aspnet-core-api-neon

8 Ian Hafkenschiel. Building a Web3 Wallet Example App with React and React Native [Електронний ресурс] – Режим доступу: https://www.ianhafkenschiel.com/blog/building-a-web3-wallet-example-app-with-react-and-react-native/

9 Nadcab Labs. The Future of Identity Verification in Web3 Development [Електронний ресурс] – Режим доступу: https://www.nadcab.com/blog/identity-verification-techniques-for-web3

10 Solidity Documentation. Solidity by Example — Solidity 0.8.31 documentation [Електронний ресурс] – Режим доступу: https://docs.soliditylang.org/en/latest/solidity-by-example.html

11 Solidity Documentation. Contract Metadata — Solidity 0.8.31 documentation [Електронний ресурс] – Режим доступу: https://docs.soliditylang.org/en/latest/metadata.html

12 Stack Overflow. Connect containerized .NET Core MVC/WebAPI app to locally installed Postgres [Електронний ресурс] – Режим доступу: https://stackoverflow.com/questions/45316662/connect-containerized-net-core-mvc-webapi-app-to-locally-installed-postgres

13 Medium. Integrating PostgreSQL with .NET 9 Using EF Core [Електронний ресурс] – Режим доступу: https://medium.com/@vosarat1995/integrating-postgresql-with-net-9-using-ef-core-a-step-by-step-guide-a773768777f2

14 Medium. Why You Should Use Blockchain for Digital Fingerprints [Електронний ресурс] – Режим доступу: https://medium.com/web3labs/why-you-should-use-blockchain-for-digital-fingerprints-107d3d403c03

15 MDPI. Securing Fingerprint Template Using Blockchain and Distributed Storage [Електронний ресурс] – Режим доступу: https://www.mdpi.com/2073-8994/12/6/951

16 MoralisHow to Connect MetaMask to a Website [Електронний ресурс] – Режим доступу: https://moralis.io/how-to-connect-metamask-to-a-website/

17 Alchemy . Introduction to Ethereum Smart Contracts [Електронний ресурс] – Режим доступу https://www.alchemy.com/overviews/ethereum-smart-contracts

18 OpenZeppelin Contracts Wizard: Smart Contract Generator [Електронний ресурс] – Режим доступу: https://wizard.openzeppelin.com/

19 Dev.to. How to Deploy Solidity Smart Contracts with Hardhat [Електронний ресурс] – Режим доступу: https://dev.to/jacobedawson/how-to-deploy-solidity-smart-contracts-with-hardhat-5f6f

20 GitHub Docs. About Webhooks [Електронний ресурс] – Режим доступу: https://docs.github.com/en/webhooks-and-events/webhooks/about-webhooks

ДОДАТОК А

Технічне завдання

Міністерство освіти і науки України

Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра обчислювальної техніки

ЗАТВЕРДЖУЮ
Завідувач кафедри ОТ
д.т.н. проф. Азаров О. Д.
«___» _____ 2025 р.

ТЕХНІЧНЕ ЗАВДАННЯ

на бакалаврську кваліфікаційну роботу

«Програмний засіб збору інформації про користувачів у мережі web3»

08-54.БКР.028.00.000 ТЗ

Керівник бакалаврської кваліфікаційної роботи:

_____ к.т.н., доц. Дудник О.В

Виконав:

_____студент гр. 2СП-21б Іваніщенко В.М.

Вінниця – 2025 року

1 Найменування та галузь застосування

Бакалаврська квалифікаційна робота: «Програмний засіб збору інформації про користувачів у мережі web3».

Галузь застосування – децентралізовані веб додатки.

2 Підстава для розробки

Завдання на роботу, яке затверджене на засіданні кафедри обчислювальної техніки.

3 Мета та призначення розробки

Мета дослідження — підвищення надійності, прозорості та безпеки процесу ідентифікації користувачів у web3-середовищі за допомогою розробленої системи, що інтегрує механізми збору технічних параметрів пристрою, формування унікального visitor ID, його верифікації за допомогою EVM-сумісного гаманця та фіксації у блокчейн-мережі. Реалізована архітектура дозволяє забезпечити децентралізовану автентифікацію без залучення персональних даних, що значно підвищує рівень приватності та довіри до цифрових сервісів.

Призначення роботи — розробка програмного засобу для децентралізованої ідентифікації користувачів у web3-середовищі з використанням технічних параметрів пристрою, авторизації через EVM-гаманець і збереження ідентифікатора в блокчейні.

5 Технічні вимоги

Серверна частина:

— операційна система —Windows 10/11 x64 або сумісна Linux-дистрибуція;

— платформа — .NET 6 SDK та ASP.NET Core Runtime;

— база даних — PostgreSQL 14 або новіша версія;

— середовище контейнеризації (опційно)— Docker 24.0+ та Docker Compose;

— процесор — мінімум 2 ядра з тактовою частотою 2.0 ГГц;

— оперативна пам’ять: не менше 4 ГБ;

— місце на диску: від 10 ГБ для зберігання журналів та даних користувачів;

Клієнтська частина:

— Браузер: Chrome 100+, Firefox 100+, Edge Chromium

— Операційна система — Windows 10+, macOS, Linux, Android 8+, iOS 13+ Web3-гаманець: встановлений MetaMask або інший EVM- гаманець;

— Підтримка: JavaScript, WebGL.

6 Конструктивні вимоги

Веб-додаток має відповідати ергономічним та технічним вимогам, текстова та графічна документація повинна відповідати стандартам України.

7 Перелік технічної документації, що пред’являється по закінченню робіт

По закінченню робіт пред’являється: пояснювальна записка ,технічне завдання, лістинги програми.

8 Вимоги до рівня уніфікації та стандартизації

При розробці програмних засобів слід дотримуватися уніфікації і ДСТУ 3008 та ДСТУ 8302.

9 Стадії та етапи розробки

Таблиця 3 — Етапи розробки

№ з/п	Назва етапів бакалаврської дипломної роботи	Строк виконання етапів роботи	Примітка
1	Огляд Web3 технологій та формування задачі	22.03.2025 – 31.03.2025	Виконано
2	Архітектура та логіка функціонування системи	01.04.2025 – 14.04.2025	Виконано
3	Розробка модулів програмного продукту	14.04.2025 – 26.04.2025	Виконано

4	Тестування застосунка	13.05.2025 – 24.05.2025	Виконано
5	Оформлення матеріалів до захисту БДР	25.05.2025 – 13.05.2025	Виконано

10 Порядок контролю та прийняття.

Всі етапи бакалаврської роботи контролюються науковим керівником згідно плану по виконанню роботи. Прийняття бакалаврської кваліфікаційної роботи здійснюється ДЕК, затвердженою зав. кафедрою згідно з графіком захисту. Дозволяється корегування бакалаврської дипломної роботи.

ДОДАТОК Б

Протокол перевірки навчальної роботи

Назва роботи:
Програмний засіб збору інформації про користувачів у мережі web3

Тип роботи: Бакалаврська кваліфікаційна робота
(БДР, МКР)

Підрозділ кафедра обчислювальної техніки
(кафедра, факультет)

Показники звіту подібності StrikePlagiarism

Оригінальність 99.5% Схожість 0.5%

Аналіз звіту подібності (відмітити потрібне):

✓ Запозичення, виявлення у роботі, оформлені коректно і не містять ознак плагіату.

□ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її виконання автором. Роботу направити на розгляд експертної комісії кафедри.

□ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Особа, відповідальна за перевірку _____ Захарченко С.М.
(підпис) (прізвище, ініціали)

Ознайомлені з повним звітом подібності, який був згенерований системою StrikePlagiarism щодо роботи.

Автор роботи _____ Іваніщенко В.М.
(підпис) (прізвище, ініціали)

Керівник роботи _____ _____
(підпис) (прізвище, ініціали)

ДОДАТОК В

ГРАФІЧНА ЧАСТИНА

ПРОГРАМНИЙ ЗАСІБ ЗБОРУ ІНФОРМАЦІЇ ПРО КОРИСТУВАЧІВ У МЕРЕЖІ WEB3

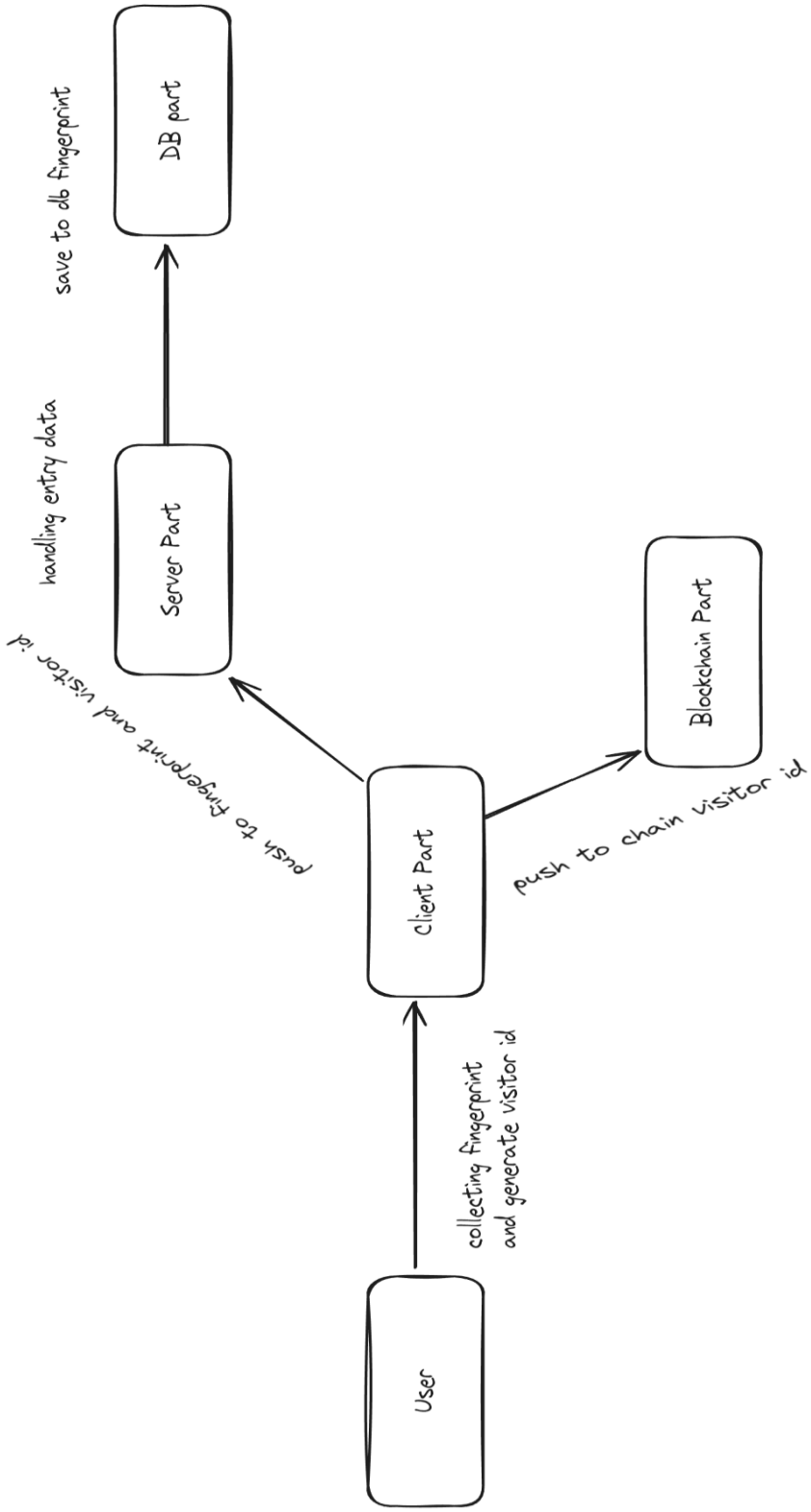

Рисунок В.1 — Блок схема програмного застосунку

ДОДАТОК Г
ІЛЮСТРАТИВНА ЧАСТИНА

Інтерфейс користувача

ПРОГРАМНИЙ ЗАСІБ ЗБОРУ ІНФОРМАЦІЇ ПРО КОРИСТУВАЧІВ У МЕРЕЖІ WEB3

Рисунок Г 1 - Інтерфейс користувача

ДОДАТОК Д

ЛІСТИНГ КОДУ

Ендпойнти:

```
using System.Threading.Channels;
using Newtonsoft.Json.Linq;
using Web3Collecting.Endpoints.Contracts;
using Web3Collecting.Services;

namespace Web3Collecting.Endpoints.UserEndpoints;

public static class UserEndpoints
{
    public static void RegisterUserEndpoints(this IEndpointRouteBuilder app)
    {
        var users = app.MapGroup("/api/users");


        users.MapGet("/{visitorId}", (string visitorId, UserService userService)
=>
        {
            var currentUser = userService.GetCurrentUserInfo(visitorId);
            return Results.Ok(currentUser);
        });



        users.MapPost("/create", async (UserInfoDto userInfoDto,UserService
userService) =>
        {
            JObject  userInfoJObj =
JObject.FromObject(userInfoDto.ComponentInfo);
            string userInfoStr = userInfoJObj.ToString();
            JObject tested = JObject.Parse(userInfoStr);
            ComponentsDto comp = tested.ToObject<ComponentsDto>();

            bool createdResult = await
userService.CreateUserInfoAsync(userInfoDto.VisitorId, userInfoJObj.ToString());
            if (createdResult)
            {
                return Results.Ok();
            }

            return Results.Problem();
        });
    }
}
```

Вхідний файл:

```
using Web3Collecting.DataAccess;
using Web3Collecting.Endpoints.UserEndpoints;
using Web3Collecting.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddControllers();
builder.Services.AddSwaggerGen();
```

```
builder.Services.AddScoped<UserInfoDbContext>();
builder.Services.AddScoped<UserService>();
builder.Services.AddCors(options =>
{
    options.AddDefaultPolicy(policy =>
    {
        policy.AllowAnyOrigin();
        policy.AllowAnyHeader();
        policy.AllowAnyMethod();
    });
});


var app = builder.Build();

using var scope = app.Services.CreateScope();
await using var dbContext =
scope.ServiceProvider.GetService<UserInfoDbContext>();
await dbContext.Database.EnsureCreatedAsync();
var service = scope.ServiceProvider.GetService<UserService>();
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}


app.MapControllers();
app.UseHttpsRedirection();
app.RegisterUserEndpoints();
app.UseCors();

app.Run();
```

Сервісний файл:

```
using Microsoft.EntityFrameworkCore;
using Web3Collecting.DataAccess;
using Web3Collecting.Models.User;

namespace Web3Collecting.Services;

public class UserService
{
    private UserInfoDbContext _dbContext;
    private List<UserInfo> _userInfos;

    public UserService(UserInfoDbContext dbContext)
    {
        _dbContext = dbContext;
        _userInfos = _dbContext.UserInfos.ToList();
    }

    public async Task<bool> CreateUserInfoAsync(string visitorId, string
componentInfo)
    {
        var createdUser = new UserInfo(visitorId, componentInfo);
        _dbContext.UserInfos.Add(createdUser);
        await _dbContext.SaveChangesAsync();
        return true;
    }

    public UserInfo GetCurrentUserInfo(string visitorId)
```

```
    {
        var currentUser = _userInfos.Single(u =>
u.VisitorId.ToLower().Equals(visitorId.ToLower()));
        if (currentUser == null)
        {
            throw new ArgumentException("Not found this user");
        }
        return currentUser;
    }

    public async Task<bool> UserInfoUpdateAsync(string visitorId, string
componentInfo)
    {
        var currentUser = GetCurrentUserInfo(visitorId);
        currentUser.ComponentInfo = componentInfo;
        await _dbContext.SaveChangesAsync();
        return true;
    }

    public async Task<bool> DeleteUserInfoAsync(string visitorId)
    {
        var currentUser = GetCurrentUserInfo(visitorId);
        _dbContext.UserInfos.Remove(currentUser);
        await _dbContext.SaveChangesAsync();
        return true;
    }
}
```

Вхідний файл інтерфейсу:

```
import { useState, useEffect } from 'react';
import './App.css';
import Header from './components/Header';
import InformationComponent from './components/Information';
import AttentionPopup from './components/AttentionPopup';
import TransactionForm from './components/TransactionForm';
import FingerprintJS from '@fingerprintjs/fingerprintjs';
import { use } from 'react';


function App() {
  const [walletConnected, setWalletConnected] = useState(false);
  const [account, setAccount] = useState(null);
  const [fpHash, setFpHash] = useState('');
  const [componentInfo, setComponentInfo] = useState('');
  const [popupClosed, setPopupClosed] = useState(false);

  const toggleWalletConnection = async () => {
    if (!walletConnected) {
      try {
        if (window.ethereum) {
```

```
          const accounts = await window.ethereum.request({ method:
"eth_requestAccounts" });
          setAccount(accounts[0]);
          setWalletConnected(true);
        } else {
          alert("Don't find wallet");
        }
      } catch (err) {
        console.error("Error", err);
      }
    } else {
      setAccount(null);
      setWalletConnected(false);
    }
  };
  const disconnectWallet = () => {
    setAccount(null);
    setWalletConnected(false);
    console.log("Wallet disconected!");
  };

  useEffect(()=>{

  },[]  )

  useEffect(() => {
    if (popupClosed || localStorage.getItem('popupShown') === 'true') {
      const setFp = async () => {
        const agent = await FingerprintJS.load({ debug: true });
        const compon = await agent.getComponents();
        console.log('Fingerprint Components:', compon);
        const allAgentInfo  = await agent.get();
        console.log('Visitor ID:', allAgentInfo.result.visitorId);
        setFpHash(allAgentInfo.result.visitorId);
        setComponentInfo(compon);
      };
      setFp();
    }
  }, [popupClosed]);

  return (
    <>
      <AttentionPopup onClose={() => setPopupClosed(true)} />
```

```
      <Header
        walletConnected={walletConnected}
        toggleWalletConnection={toggleWalletConnection}
        disconnectWallet={disconnectWallet}
        account={account}
      />
      <TransactionForm visitorId={fpHash} walletConnected={walletConnected}
componentInfo={componentInfo}/>
      <InformationComponent fpHash={fpHash} />
    </>
  );
}
export default App;
```