Хмельницький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерної інженерії та інформаційних систем

КВАЛІФІКАЦІЙНА РОБОТА

Програмно-апаратна система приватного файлового сховища з контролем доступу
Назва теми

Рівень вищої освіти *перший (бакалаврський)*

Галузь знань *12 «Інформаційні технології»*
Шифр, назва

Спеціальність *123 «Комп'ютерна інженерія»*
Шифр, назва

Освітня програма *«Комп'ютерна інженерія та програмування»*
Назва

Шифр КвРКІ 022071.22.03.34 ПЗ

Виконав здобувач IV курсу, група КІ2-22-3 ______ Андрій ПІКАЛОВ
Підпис Ініціали, прізвище

Керівник ______ Володимир КИСІЛЬ
Науковий ступінь, учене звання Підпис Ініціали, прізвище

Нормоконтролер канд.фіз.-мат.наук,доцент ______ Тетяна КИСІЛЬ
Науковий ступінь, учене звання Підпис Ініціали, прізвище

До захисту допускаю:
завідувач кафедри КІІС ______ Ольга ПАВЛОВА
Підпис Ініціали, прізвище
«01» червня 2026 р.
дата

Хмельницький 2026

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій

Кафедра Комп'ютерної інженерії та інформаційних систем

Рівень вищої освіти перший (бакалаврський)

Галузь знань 12 Інформаційні технології

Спеціальність 123 Комп'ютерна інженерія

Освітня програма «Комп'ютерна інженерія та програмування»

ЗАТВЕРДЖУЮ
Завідувачка кафедри КІІС
Ольга ПАВЛОВА

" 10 " 01 2026 р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Пікалов Андрію Руслановичу
Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Програмно-апаратна система приватного файлового сховища з контролем доступу

Керівник проекту (роботи) Кисіль Володимир Володимирович
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 20.01.2026 р. № 7

2. Термін подання здобувачем роботи на кафедру 01.06.2026 р.

3. Вихідні дані до роботи Завдання на кваліфікаційну роботу

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)

Дослідження програмно-апаратної системи приватного файлового сховища з контролем доступу та постановка задачі

Проєктування архітектури та засобів захисту інформації у програмно-апаратній системі приватного файлового сховища

Програмно-апаратна реалізація системи приватного файлового сховища з контролем доступу

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

Схема багатофакторної автентифікації

Специфікація структури бази даних

Апаратне забезпечення проєкту

Алгоритм функціонування підсистеми апаратної автентифікації

6. Консультанти розділів кваліфікаційної роботи

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

7. Дата видачі завдання «_10_»__________01__________2026 р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів (розділів) дипломного проекту (роботи)	Термін виконання етапів проекту (роботи)	Примітка
1	Вибір напряму дослідження та узгодження тематики кваліфікаційної роботи з керівником	10.01.2026	виконано
2	Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження	01.02.2026	виконано
3	Робота над розділом 1 – дослідження методів організації приватного файлового сховища та постановка задачі	01.03.2026	виконано
4	Робота над розділом 2 – проєктування архітектури та механізмів контролю доступу приватного файлового сховища	01.04.2026	виконано
5	Робота над розділом 3 – програмно-апаратна реалізація системи приватного файлового сховища з контролем доступу	29.04.2026	виконано
6	Оформлення пояснювальної записки згідно вимог	25.05.2026	виконано
7	Попередній захист ВКР	26.05.2026	виконано
8	Захист ВКР на засіданні ЕК	Червень 2026 року	

Здобувач__________________ ____Андрій ПІКАЛОВ___
Підпис Імя, ПРІЗВИЩЕ

Керівник кваліфікаційної роботи__________________ ____Володимир КИСІЛЬ____
Підпис Імя, ПРІЗВИЩЕ

№ рядка	формат	Позначення	Найменування	Кіл. листів	№ екз	Примітка
			Текстові документи			
1		КвРКІ 022071.22.03.34 ПЗ	Пояснювальна записка	64		
			Графічні матеріали			
2		КвРКІ 022071.22.03.34 Е8	Схема багатофакторної автентифікації	1		
3		КвРКІ 022071.22.03.34 Е8	Специфікація структури бази даних	1		
4		КвРКІ 022071.22.03.34 Е8	Апаратне забезпечення	1		
			проєкту			
		КвРКІ 022071.22.03.34 Е8	Алгоритм функціонування підсистеми апаратної автентифікації	1		
		КвРКІ 022071.22.03.34 Е8	Код прошивки ESP-32	1		
		КвРКІ 022071.22.03.34 Е8	Дерево файлів бекенд частини	1		

Зм	Арк	№ докум	Підпис	Дата	КвРКІ 022071.22.03.34 ВП		
Розробив		Пікалов			Відомість проекту	Літера	Аркуш	Аркушів
Перевір.		Кисіль				У	1	64
Н. контр.		Кисіль				ХНУ, КІ2-22-3		
Затв.		Павлова		01.06				

АНОТАЦІЯ

Тема кваліфікаційної роботи: «Програмно-апаратна система приватного файлового сховища з контролем доступу».

Автор роботи: Андрій Пікалов.

Керівник роботи: Володимир Кисіль.

Пояснювальна записка: 64 с., 22 рис., 5 табл., 6 дод., 50 джерел.

Графічна частина: 4 креслень.

АВТЕНТИФІКАЦІЯ, АРХІТЕКТУРА, БАЗА ДАНИХ, КОНТРОЛЬ ДОСТУПУ, ЛОГУВАННЯ, ОБ'ЄКТНЕ СХОВИЩЕ, ПРИВАТНЕ СХОВИЩЕ, ESP32, HMAC-SHA256.

Кваліфікаційна робота бакалавра присвячена розробці програмно-апаратної системи приватного файлового сховища з використанням фізичного фактора автентифікації на базі мікроконтролера ESP32. Актуальність теми зумовлена необхідністю створення систем з високим рівнем довіри, де доступ до конфіденційних даних захищений не лише паролем, а й апаратним ключем, що виключає можливість несанкціонованого входу навіть у разі компрометації облікових даних користувача.

Метою роботи є проєктування та реалізація комплексу для безпечного зберігання файлів з підсистемою двофакторної автентифікації. Для досягнення мети було розроблено архітектуру на базі NestJS та MinIO, а також спроєктовано апаратний модуль на базі ESP32. У роботі впроваджено криптографічний протокол Challenge-Response з використанням алгоритму HMAC-SHA256, що забезпечує захист від атак повторного відтворення. Реалізовано механізм синхронізації апаратного ключа та веб-клієнта через Redis. Програмна частина включає багатошарову структуру бекенду, систему журналювання подій та Docker-інфраструктуру. Апаратна частина включає розробку прошивки з реалізацією криптографічних функцій та контролю фізичної присутності користувача

______________ 30.05.2026

Підпис здобувача Дата

ЗМІСТ

Вступ 4
1 Дослідження програмно-апаратної системи приватного файлового сховища з контролем доступу та постановка задачі 5
1.1 Змістовний аналіз предметної області приватного файлового сховища, її структурних та функціональних особливостей 5
1.2 Аналіз програмно-апаратного забезпечення обробки даних та систем багатофакторного контролю доступу в приватному файловому сховищі 11
1.3 Визначення вимог до програмно-апаратної системи приватного сховища та розробка технічного завдання на проєктування 18
2 Проєктування архітектури та моделей взаємодії компонентів програмно-апаратної системи приватного сховища 22
2.1 Архітектурна модель розподіленої системи та принципи багатошарової побудови ПЗ 22
2.2 Проєктування апаратного модуля автентифікації на базі мікроконтролера ESP32 25
2.3 Розробка протоколу взаємодії «Сервер – Апаратний токен» 36
2.4 Функціональна декомпозиція та паралельне виконання завдань 40
3 Програмно-апаратна реалізація та тестування системи приватного сховища 43
3.1 Програмна реалізація та архітектурна структура бекенд-сервера 43
3.2 Програмна реалізація механізмів апаратно-залежної автентифікації ... 51
3.3. Тестування та верифікація програмно-апаратного комплексу 58
Висновки 70
Перелік джерел посилань 71
Додаток А Схема багатофакторної автентифікації 77

Зм.	Арк.	№докум.	Підпис	Дата	КвРКІ. 022071.22.03. 34 ПЗ		
Виконав		Андрій ПІКАЛОВ			Програмно-апаратна система приватного файлового сховища з контролем доступу Пояснювальна записка	Літера	Аркуш	Аркушів
Перевір.		Володимир Кисіль				у	2	64
Н.контр.		Тетяна КИСІЛЬ				ХНУ КІ2-22-3		
Затвер.		Ольга ПАВЛОВА						

Додаток Б Специфікація структури бази даних 79
Додаток В Апаратне забезпечення проєкту 79
Додаток Г Алгоритм функціонування підсистеми апаратної автентифікації 80
Додаток Д Код прошивки ESP-32 81
Додаток Е Дерево файлів бекенд частини 85

ВСТУП

Актуальність дослідження. У сучасну епоху цифрової трансформації надійне зберігання та захист інформації стали критично важливими аспектами життєдіяльності суспільства. Повсюдне використання публічних хмарних сервісів створює серйозні ризики компрометації конфіденційних даних через загрози витоку, несанкціонованого доступу та залежність від сторонніх провайдерів. Окрему проблему становить недостатня надійність традиційної однофакторної автентифікації (паролів), що потребує впровадження фізичних факторів підтвердження доступу. У даному дослідженні ми розглянемо реалізацію програмно-апаратного комплексу приватного сховища, що базується на принципах «нульової довіри» (Zero Trust) та поєднує об’єктне S3-зберігання з апаратним криптографічним захистом на базі мікроконтролерів.

Метою дипломної роботи є проєктування, програмна реалізація та апаратна інтеграція системи приватного файлового сховища, що забезпечує високий рівень контролю над даними завдяки використанню об’єктного зберігання та багатофакторного контролю доступу з підтвердженням через апаратний ключ на базі ESP32.

Об’єктом дослідження є процеси автентифікації, зберігання та управління доступом до цифрових ресурсів у децентралізованих обчислювальних системах.

Предметом дослідження є алгоритми криптографічного протоколу Challenge-Response, методи архітектурної побудови S3-сумісних сервісів та апаратно-програмна взаємодія мікроконтролера ESP32 з сервером сховища для забезпечення цілісності та конфіденційності даних.

1 ДОСЛІДЖЕННЯ ПРОГРАМНО-АПАРАТНОЇ СИСТЕМИ ПРИВАТНОГО ФАЙЛОВОГО СХОВИЩА З КОНТРОЛЕМ ДОСТУПУ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Змістовний аналіз предметної області приватного файлового сховища, її структурних та функціональних особливостей

Необхідність проєктування та впровадження автономних систем зберігання стає стратегічно важливим завданням для сучасних підприємств, наукових установ та індивідуальних розробників. Предметна область дослідження охоплює сукупність методів організації мережевого доступу до даних, криптографічних протоколів та архітектурних рішень для управління неструктурованою інформацією [1]. У сучасну епоху тотальної цифровізації обсяги генерованих даних зростають експоненціально, що висуває нові вимоги до надійності та методології їх збереження [2]. Повсюдне використання публічних хмарних сервісів, попри їхню зручність, створює низку серйозних ризиків: загрози конфіденційності через архітектурні вразливості провайдерів, можливість несанкціонованого доступу з боку третіх осіб та повна залежність користувача від стабільності та політик сторонніх сервісів [3].

Приватне сховище, реалізоване як локальне (On-premise) рішення, дозволяє власнику або організації здійснювати повний контроль над процесами обробки, збереження та передачі цифрової інформації [9]. Такий архітектурний підхід забезпечує реалізацію концепції «цифрового суверенітету», що є важливою умовою для секторів з високими вимогами до конфіденційності та цілісності даних [23]. Використання власних апаратно-програмних засобів дозволяє мінімізувати ризики, пов’язані із санкційною політикою, зміною умов обслуговування сторонніми провайдерами або неконтрольованою передачею конфіденційної інформації за межі підконтрольного периметру.

Приклади галузей, де впровадження такої системи є найбільш затребуваним, наведено у таблиці 1.1.

Таблиця 1.1 – Приклади практичного застосування приватного сховища

Сфера застосування	Опис практичного використання системи
Корпоративний сектор	Збереження внутрішньої документації та фінансових звітів із суворим контролем доступу
IT-розробка	Надійне зберігання початкового коду проектів та дистрибутивів програмного забезпечення
Медичні установи	Безпечне архівування цифрових карток пацієнтів та результатів медичної діагностики
Освітні платформи	Розміщення відео-лекцій та навчальних посібників для студентів із великою кількістю звернень
Системи відеонагляду	Довгострокове збереження великих масивів відеоінформації з автоматичним очищенням

Незважаючи на наявність широкого спектра готових програмних та апаратно-програмних рішень, вони часто характеризуються надмірною ресурсомісткістю для специфічних сценаріїв автоматизації або відсутністю вбудованої підтримки низькорівневих апаратних факторів автентифікації на базі мікроконтролерів [8, 27]. Це зумовлює доцільність розробки спеціалізованого комплексу, що інтегрує сучасні вебтехнології з криптографічним захистом на фізичному рівні. Актуальність проєктування програмно-апаратної системи приватного сховища на базі фреймворку NestJS та мікроконтролера ESP32 полягає у можливості створення високоефективного рішення з гнучкою модульною архітектурою та багаторівневим контролем доступу [4, 12].

Ефективність функціонування автономного сховища визначається не лише його фізичним розміщенням та потужністю серверного обладнання, а й архітектурними особливостями організації даних. Функціональною особливістю сучасної предметної області зберігання є поступовий відхід від традиційних

ієрархічних файлових систем (таких як NTFS або EXT4) на користь об'єктно-орієнтованих структур [13, 16].

Традиційні файлові системи, зокрема NTFS та EXT4, організовують дані у вигляді ієрархічного дерева каталогів [1]. Такий підхід є зручним для локального управління файлами, однак при побудові масштабованих мережевих сховищ він створює суттєві структурні обмеження: зі збільшенням кількості об'єктів експоненціально зростає складність індексації, що призводить до значних затримок під час пошуку інформації [16]. Для вирішення зазначених проблем у межах даного проєкту застосовується технологія об'єктного зберігання даних (Object Storage). На відміну від класичного підходу, у цій моделі кожен файл розглядається як окремий об'єкт, що зберігається у спеціальному логічному контейнері – бакеті (Bucket) [6]. Структурно кожен об'єкт ідентифікується за унікальним ключем (Key), що забезпечує високу швидкість доступу до даних незалежно від загальної кількості об'єктів у системі або глибини вкладеності віртуальних каталогів [14, 15].

Додатковою перевагою об'єктного підходу є можливість інтегрованого зберігання розширеного набору метаданих безпосередньо разом із об'єктом [16]. Метадані можуть містити вичерпну інформацію про тип контенту, часові мітки, контрольні суми та параметри прав доступу, що суттєво спрощує автоматизовану організацію та подальшу обробку даних у складних програмних комплексах. Окрім цього, об'єктне сховище демонструє високу адаптивність до функціонування у розподілених архітектурах. Завдяки механізмам реплікації об'єктів між кількома фізичними вузлами забезпечується високий рівень відмовостійкості та цілісності інформації, що дозволяє системі зберігати працездатність навіть у разі виходу з ладу окремих компонентів серверного обладнання [18, 33].

Основним стандартом взаємодії з об'єктними сховищами на сучасному етапі розвитку інформаційних систем є протокол S3 (Simple Storage Service) [13]. Спочатку розроблений компанією Amazon для внутрішніх потреб хмарної

інфраструктури, він набув статусу універсального галузевого стандарту завдяки своїй надійності та простоті інтеграції. Взаємодія із системою за протоколом S3 ґрунтується на використанні архітектурного стилю REST та мережевого протоколу HTTP(S). Це дозволяє виконувати повний цикл операцій над об'єктами (завантаження, отримання метаданих, модифікацію та видалення) за допомогою стандартних методів GET, POST, PUT і DELETE [15, 35]. Така безстанова (stateless) природа взаємодії забезпечує виключно високу сумісність із різними мовами програмування та програмними платформами, дозволяючи розробникам інтегрувати файлове сховище у будь-яку сучасну серверну архітектуру без необхідності використання специфічних низькорівневих драйверів.

Використання S3 API суттєво спрощує процеси інтеграції об'єктного сховища з сучасними вебзастосунками та складними мікросервісними архітектурами [13, 20]. Наявність офіційних SDK (Software Development Kits) для більшості високорівневих мов програмування дозволяє уніфікувати методи маніпуляції даними та значно скоротити терміни розробки і тестування прикладного програмного забезпечення [12]. Такий стандартизований підхід забезпечує прозорість взаємодії між клієнтськими програмами та серверною частиною, гарантуючи високу стабільність роботи системи при динамічному розширенні її функціональних можливостей.

У межах даного проєкту для реалізації серверної складової зберігання даних обрано систему MinIO, яка є високоефективним об'єктним сховищем із відкритим початковим кодом [10, 34]. Головною інженерною перевагою MinIO є повна сумісність із протоколом S3 та оптимізація для роботи у хмарно-орієнтованих (cloud-native) середовищах на базі технології контейнеризації Docker [5]. Застосування цього інструментарію дозволяє абстрагувати логіку управління файловими об'єктами від специфіки апаратної конфігурації сервера, забезпечуючи високу швидкість обробки неструктурованої інформації та надійну інтеграцію із бекенд-системою через уніфіковані програмні інтерфейси

[8, 16]. Такий підхід гарантує цілісність даних та високу доступність сервісу навіть при значних навантаженнях, що є важливим для функціонування приватного файлового сховища.

Система MinIO характеризується високими показниками продуктивності та нативною підтримкою горизонтального масштабування, що є критично важливою умовою для функціонування сучасних інформаційних систем, які оперують значними масивами неструктурованих даних [34]. Окрім цього, в архітектурі MinIO реалізовано передові механізми забезпечення цілісності інформації, зокрема технологію «Erasure Coding» (надлишкове кодування), яка дозволяє автоматично відновлювати об'єкти навіть у разі фізичної відмови декількох накопичувачів одночасно [31]. Це суттєво підвищує загальну відмовостійкість програмно-апаратного комплексу та гарантує надійність збереження конфіденційної інформації [18]. Для обґрунтування доцільності переходу до об'єктної моделі було проведено порівняльний аналіз характеристик традиційних мережевих протоколів (SMB, FTP) та сучасного S3-сумісного сховища, результати якого наведено в табл. 1.2 [14].

Таблиця 1.2 – Порівняльна таблиця традиційного зберігання та об'єктного сховища MinIO

Ознака порівняння	Традиційні мережеві папки (SMB/FTP)	Об'єктне сховище (MinIO + NestJS)
Спосіб доступу	Пряме підключення у вигляді мережевого диска	Програмне керування через сучасне S3 API
Відмовостійкість	Залежить від конкретного сервера	Підтримка реплікації та відновлення даних

Кінець таблиці 1.2

Гнучкість системи	Обмежена налаштуваннями ОС	Дуже висока завдяки використанню власного коду
Масштабованість	Складно розширювати без значної втрати швидкості	Легко масштабується до петабайтів даних
Безпека та доступ	Базова авторизація на рівні операційної системи	Сучасна авторизація через токени JWT та ролі
Автоматизація	Майже відсутня без використання сторонніх утиліт	Повне програмування будь яких сценаріїв обробки
Вартість розширення	Висока через потребу у специфічному залізі	Низька завдяки використанню звичайних серверів

Узагальнюючи вищевикладене, можна стверджувати, що об’єктна модель зберігання у поєднанні зі стандартом S3 формує надійний технологічний та архітектурний фундамент для побудови сучасних автономних сховищ [1]. Такий підхід забезпечує не лише високу швидкість обробки безстанових HTTP-запитів та практично необмежене горизонтальне масштабування, а й відкриває широкі можливості для впровадження спеціалізованих систем багатофакторної безпеки та гнучкого контролю доступу [18]. Використання сучасних засобів розробки із суворою типізацією гарантує стабільність програмних інтерфейсів та цілісність

логіки автентифікації на всіх рівнях взаємодії компонентів програмно-апаратного комплексу [50].

1.2 Аналіз програмно-апаратного забезпечення обробки даних та систем багатофакторного контролю доступу в приватному файловому сховищі

Метою даного аналізу є дослідження досвіду провідних розробників систем автоматизації та засобів захисту інформації у сфері мережевого зберігання даних. Під час проєктування було проаналізовано низку існуючих рішень для організації приватних сховищ, що дозволило визначити їхні основні переваги та недоліки в контексті забезпечення цифрового суверенітету та гнучкості системних налаштувань [23].

Проєкт Nextcloud (Nextcloud GmbH) є найбільш розповсюдженим програмним рішенням з відкритим вихідним кодом. Дане ПЗ пропонує широкий спектр функцій для управління файлами та спільної роботи, проте архітектурно Nextcloud базується на монолітній структурі, що робить його надмірно вимогливим до обчислювальних ресурсів сервера. Окрім цього, складність внутрішнього ядра системи не дозволяє гнучко інтегрувати низькорівневі апаратні фактори автентифікації на базі мікроконтролерів без значної модифікації базового програмного коду [12].

Система Synology Drive (Synology Inc.) представляє собою комерційне апаратно-програмне рішення, що забезпечує високу стабільність роботи за принципом «готового продукту». Проте закритість екосистеми (proprietary software), обмеженість вибору апаратної платформи та висока вартість спеціалізованого обладнання суттєво обмежують використання даного продукту у специфічних інженерних сценаріях. Зокрема, у випадках, де необхідний повний контроль над протоколами передачі даних, методами шифрування та впровадженням кастомних криптографічних засобів [21, 22].

Апаратні рішення компанії Yubico (YubiKey) на сьогодні вважаються промисловим стандартом у сфері фізичних засобів автентифікації. Незважаючи на високу криптографічну стійкість, інтеграція даних пристроїв у власні автономні інфраструктури часто супроводжується необхідністю використання зовнішніх хмарних сервісів верифікації (наприклад, YubiCloud). Така залежність від сторонньої валідації частково суперечить концепції повного цифрового суверенітету, оскільки створює критичну залежність від доступності сторонніх серверів у момент авторизації користувача в приватному сховищі [8, 23].

Таблиця 1.3 – Порівняння витрат на впровадження систем зберігання

Критерій порівняння	Nextcloud	Synology Drive	Розроблювана система
Тип зберігання	Файловий	Файловий / Блочний	Об’єктний
Апаратний ключ	Тільки через FIDO2	Опціонально	Гнучка інтеграція ESP32
Контроль коду	Відкритий	Закритий	Повний
Масштабованість	Середня	Платні керовані сервіси баз даних	Висока

Традиційна однофакторна модель автентифікації, що базується лише на використанні пароля, сьогодні демонструє вразливість перед спектром сучасних загроз. Використання GPU-прискорення для атак Brute-force, поширення методів «credential stuffing» та складні сценарії фішингу, що атакують людський фактор, роблять парольний захист недостатнім для систем із високими вимогами до конфіденційності [3, 38].

Окрім того, компрометація баз даних та застосування «райдужних таблиць» (Rainbow Tables) для реверс-інжинірингу застарілих хешів (MD5, SHA-1) призводять до повної втрати контролю над даними [1]. Для мінімізації цих

ризиків у сучасній інженерії впроваджується багатофакторна автентифікація (MFA), яка базується на використанні декількох незалежних категорій факторів [8, 29]:

1. Фактор знання – пароль або PIN-код.
2. Фактор володіння – смартфон, апаратний ключ або смарт-картка.
3. Фактор властивості – біометричні характеристики користувача.

У межах даного проєкту було проведено аналіз найбільш поширених методів реалізації другого фактора автентифікації. Одним із найпростіших варіантів є використання SMS-повідомлень або одноразових кодів через електронну пошту. Перевагою таких методів є простота впровадження, проте вони мають низку суттєвих недоліків. Зокрема, існує ризик перехоплення SMS-повідомлень, заміни SIM-картки користувача або проведення атак типу «людина посередині» (MitM-атаки).

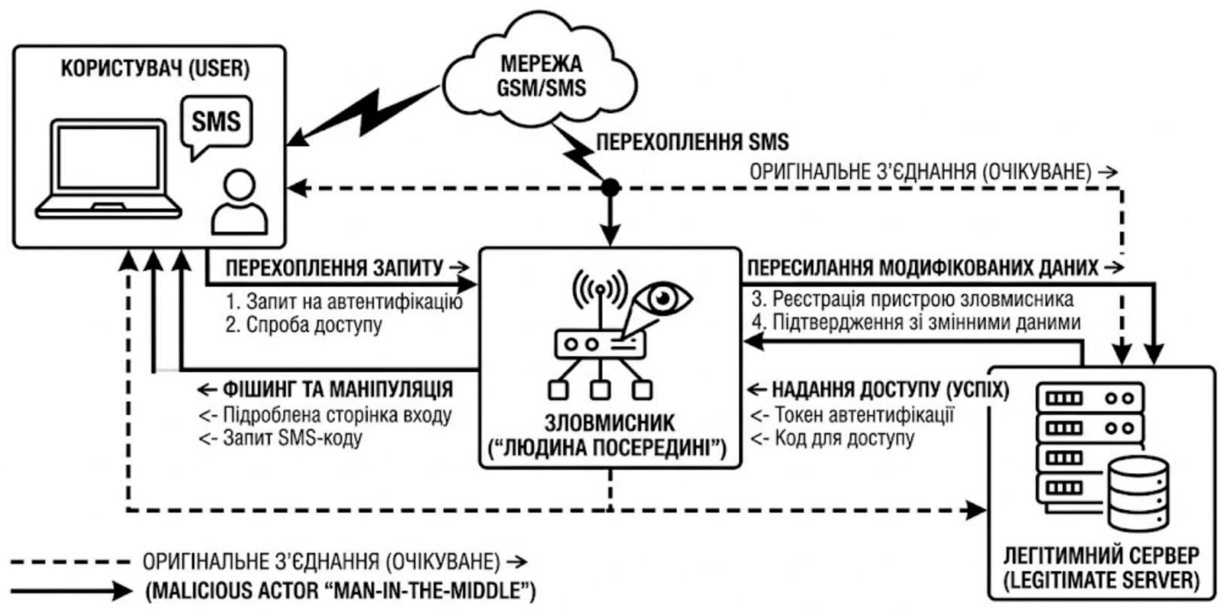


Рисунок 1.1 – Схема атаки типу «людина посередині» (Man-in-the-Middle)

Іншим поширеним інженерним рішенням є програмні автентифікатори, наприклад Google Authenticator або Microsoft Authenticator. Вони функціонують на базі алгоритму TOTP (Time-based One-Time Password) та забезпечують

генерацію тимчасових одноразових кодів доступу на основі криптографічного методу спільного секрету (shared secret). Такі засоби забезпечують суттєво вищий рівень безпеки порівняно з SMS-повідомленнями, оскільки генерація коду відбувається локально на пристрої користувача без передачі через відкриті мережі зв’язку [8].

Проте надійність даних засобів захисту безпосередньо корелює із цілісністю операційної системи мобільного пристрою. У разі зараження смартфона спеціалізованим шкідливим програмним забезпеченням (наприклад, банківськими троянами або шпигунським ПЗ), спільний секретний ключ може бути викрадений безпосередньо з ізольованої пам'яті додатка [38]. Це робить програмний метод автентифікації лише «умовно надійним», оскільки захист конфіденційних даних у сховищі залишається залежним від стороннього пристрою загального призначення, який постійно підключений до мережі інтернет [3].

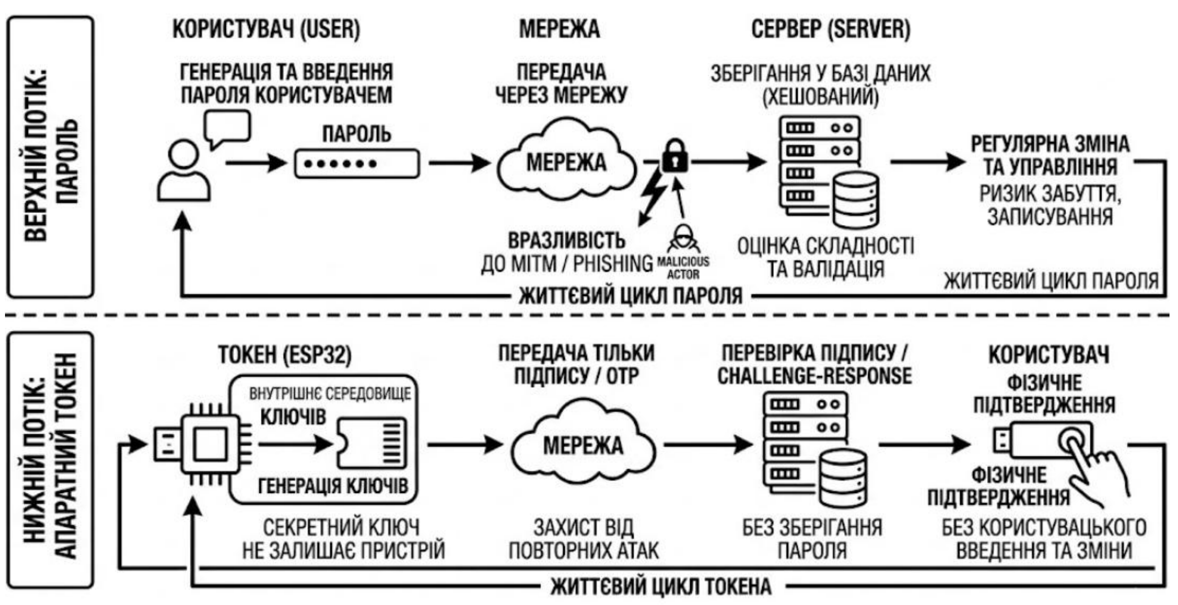


Рисунок 1.2 – Порівняння життєвих циклів пароля та апаратного токена

На основі порівняльного аналізу, наведеного на рис. 1.2, можна встановити, що традиційний пароль (верхній потік) проходить тривалий шлях передачі через відкриті канали мережі та потребує постійного зберігання в базах

даних (навіть у хешованому вигляді), залишаючись вразливим на кожному етапі – від моменту введення користувачем до фінальної валідації на сервері [15]. Натомість апаратний токен (нижній потік) базується на використанні секретного ключа, який за принципом ізоляції ніколи не залишає внутрішнє середовище мікроконтролера. Замість передачі облікових даних через мережу транслюється лише результат обчислення криптографічного протоколу у вигляді цифрового підпису повідомлення, що практично нівелює ризики перехоплення автентифікаційної інформації.

Додатковим інженерним фактором захисту в проєктованій системі є обов'язкова фізична взаємодія з пристроєм – натискання механічної кнопки. Це дозволяє реалізувати концепцію «User Presence» (фізичної присутності користувача), що унеможливлює автоматичне спрацювання шкідливих сценаріїв зловмисника, навіть за умови отримання ним несанкціонованого доступу до мережевого інтерфейсу ключа [8]. Таким чином, поєднання програмного NestJS-бекенда з апаратним компонентом на базі ESP32 забезпечує рівень захисту промислового класу, роблячи систему стійкою до Replay-атак та несанкціонованого віддаленого входу.

Впровадження у структуру приватного сховища спеціалізованого пристрою на базі мікроконтролера ESP32 дозволяє кардинально змінити модель загроз, оскільки головною відмінністю апаратного токена від традиційного пароля є зміна самого принципу життєвого циклу облікових даних. Дане апаратне рішення від компанії Espressif Systems було обране завдяки наявності вбудованого криптографічного співпроцесора, що забезпечує апаратне прискорення алгоритмів AES та SHA-256 [24, 41]. Це дозволяє ефективно реалізувати механізм Challenge-Response, де сервер формує унікальний випадковий набір даних («виклик»), який надсилається на пристрій для обчислення підпису без загрози компрометації основного секрету.

Внутрішній обчислювальний модуль ESP32 генерує цифровий підпис повідомлення за допомогою закритого ключа, після чого результат передається

серверу для верифікації. При такому алгоритмі взаємодії закритий ключ за принципом ізоляції ніколи не залишає межі енергонезалежної пам'яті пристрою, що гарантує високий рівень безпеки навіть під час передачі даних через незахищені або відкриті мережі зв'язку.

Синергія апаратного ключа та серверної частини, реалізованої на базі NestJS, забезпечує суттєве підвищення загального рівня відмовостійкості та захищеності системи. Навіть за умови повної компрометації пароля користувача, зловмисник позбавлений можливості успішного проходження автентифікації без безпосереднього володіння фізичним пристроєм. Наявність кнопки підтвердження на корпусі апаратного ключа дозволяє реалізувати перевірку фізичної присутності користувача (User Presence) під час ініціалізації сеансу, що повністю виключає можливість проведення автоматизованих атак типу «brute-force» або віддаленого перехоплення сесії.

Окрім підвищення рівня інформаційної безпеки, застосування відкритої апаратної платформи сприяє значній мінімізації витрат на впровадження системи та забезпечує повну прозорість функціонування всього програмно-апаратного комплексу. Таким чином, інтеграція багатофакторної автентифікації з використанням власного апаратного компонента є найбільш доцільним інженерним рішенням для захисту приватного сховища та конфіденційних даних [19, 43, 45].

Окрему увагу під час аналізу предметної області було приділено економічній доцільності впровадження системи. Розробка власного програмно-апаратного комплексу дозволяє повністю уникнути регулярних фінансових витрат на оплату підписок у публічних хмарних провайдерів [21]. Окрім фінансової вигоди, стратегічною перевагою On-premise рішення є отримання повного контролю над фізичним розміщенням даних та внутрішніми політиками конфіденційності, що дозволяє уникнути ризиків, пов'язаних зі зміною тарифних планів або умов обслуговування сторонніми організаціями [23].

Для кількісної оцінки економічної ефективності було проведено порівняльний аналіз витрат на розгортання власної інфраструктури та використання комерційних хмарних платформ (табл. 1.4) [22]. Результати розрахунків свідчать, що власне рішення є економічно виправданим у довгостроковій перспективі та забезпечує значне скорочення витрат на підтримку системи вже після першого року її активної експлуатації.

Таблиця 1.4 – Порівняння витрат на впровадження систем зберігання

Компонент витрат	Приватне сховище (NestJS + MinIO)	Публічні хмарні платформи (AWS/Azure)
Програмне забезпечення	Відкритий код та повністю безкоштовні ліцензії	Регулярна оплата за використання сервісів
Апаратне забезпечення	Використання наявного сервера або ПК	Входить у ціну оренди, але вам не належить
База даних метаданих	PostgreSQL та Prisma (безкоштовно)	Платні керовані сервіси баз даних
Мережевий трафік	Безкоштовно в межах локальної інфраструктури	Оплата за кожне завантаження та API запит
Масштабування простору	Купівля звичайних накопичувачів HDD чи SSD	Перехід на значно дорожчі тарифні плани
Адміністрування	Власними силами розробників системи	Потребує сертифікованих хмарних інженерів
Складність оновлення системи	Повний контроль над версіями	Без можливості впливу на процес

Кінець таблиці 1.4

Підсумкова вартість	Мінімальна, складається лише з ціни заліза	Висока та постійно зростаюча абонентська плата
Програмне забезпечення	Відкритий код та безкоштовні ліцензії	Щомісячна оплата за використання сервісів

1.3 Визначення вимог до програмно-апаратної системи приватного сховища та розробка технічного завдання на проєктування

Важливим етапом проєктування є обґрунтування вибору технологій реалізації серверної логіки, оскільки від цього залежать показники продуктивності, масштабованості та стабільності всього програмного забезпечення. У межах роботи проведено аналіз сучасних засобів розробки, зокрема розглянуто використання мов програмування Python (фреймворк FastAPI), Go (Golang) та TypeScript (фреймворк NestJS) [15, 17]. Хоча мова Go характеризується високою швидкістю виконання бінарного коду, а Python – швидкістю розробки, для даного проєкту було обрано NestJS. Даний фреймворк забезпечує строгу архітектурну структуру, базується на модульній побудові та підтримує сучасні принципи інверсії залежностей, що є критичним для створення безпечних систем із розгалуженим контролем доступу [4, 12].

Під час аналізу засобів розробки встановлено, що мова Go характеризується високою швидкодією та ефективною роботою з паралельними процесами, що робить її придатною для створення низькорівневих високонавантажених сервісів. Проте, Go має нижчий рівень абстракції, що ускладнює розробку розгалужених бізнес-моделей. У свою чергу, фреймворк FastAPI забезпечує швидку розробку API на мові Python, але поступається NestJS у потужності інструментів для структурування великих корпоративних проєктів. Для реалізації даної системи було обрано NestJS, оскільки цей фреймворк

базується на модульній архітектурі та підтримує передові інженерні принципи побудови програмного забезпечення. Зокрема, використання механізму інверсії залежностей (Dependency Injection) дозволяє декуплеризувати компоненти системи, що спрощує взаємодію між окремими модулями та суттєво полегшує процеси подальшої підтримки та автоматизованого тестування коду [4, 12].

Значною перевагою NestJS є нативна підтримка мови TypeScript, яка забезпечує строгу статичну типізацію даних [50]. Це дозволяє виявляти значну частину архітектурних та логічних помилок ще на етапі компіляції програмного коду, що суттєво підвищує загальну надійність серверної частини. Для файлового сховища, де здійснюються операції з багатьма профілями користувачів, складними ієрархічними правами доступу та різнотипними метаданими об'єктів, строгий контроль типів виступає базовим фактором забезпечення стабільності та передбачуваності функціонування системи.

Для організації надійного зберігання та структурування інформації в межах проєкту було обрано реляційну систему керування базами даних PostgreSQL [28]. Вона використовується для накопичення та індексації даних про облікові записи користувачів, конфігурації прав доступу (ACL), файлові метадані та системні журнали аудиту безпеки. PostgreSQL характеризується високою стабільністю, повною відповідністю принципам ACID (Atomicity, Consistency, Isolation, Durability) та можливістю ефективної обробки складних реляційних запитів при роботі з великими масивами взаємопов'язаної інформації.

Для забезпечення надійної взаємодії серверної логіки з базою даних застосовується сучасний інструментарій ORM Prisma [17]. Використання ORM дозволяє повністю абстрагуватися від низькорівневих SQL-запитів і оперувати даними через програмні моделі, що гарантує «Type-Safety» (типобезпеку) на рівні всього прикладного коду. Такий підхід суттєво прискорює цикл розробки та підвищує безпеку системи завдяки автоматичному формуванню

параметризованих запитів, що практично нівелює можливість проведення атак типу SQL-ін’єкцій [17, 38].

Для обробки тимчасових даних та управління активними сесіями користувачів у системі впроваджено високопродуктивне сховище Redis [32]. Застосування даного рішення забезпечує збереження інформації безпосередньо в оперативній пам’яті сервера, що гарантує мінімальні часові затримки (low latency) при доступі до даних. Redis виступає інструментом для швидкої обробки токенів автентифікації, тимчасових криптографічних «викликів» (Challenges) для апаратного ключа та інших службових даних, які потребують високої швидкості опрацювання в режимі реального часу.

Для забезпечення високого рівня довіри до системи приватного сховища необхідно сформулювати суворі технічні вимоги, що базуються на математично обґрунтованих методах захисту інформації. Впровадження апаратного компонента на базі мікроконтролера ESP32 у систему контролю доступу зумовлено необхідністю ізоляції вузлів автентифікації від вразливого середовища операційної системи загального призначення. Проте надійність такого рішення безпосередньо залежить від стійкості протоколу взаємодії між ключем та сервером [8].

Головною загрозою для апаратних факторів доступу є атака повторного відтворення (Replay Attack), за якої зловмисник перехоплює легітимний сигнал від пристрою та використовує його для несанкціонованого входу. Для запобігання подібним ризикам технічні вимоги до системи передбачають використання криптографічної моделі Challenge-Response (Виклик-Відповідь). Суть даного підходу полягає у тому, що сервер при кожній спробі входу генерує унікальне математичне завдання («виклик»), результат розв’язання якого є унікальним для кожної окремої сесії, що нівелює можливість повторного використання перехоплених даних [38].

Математичним ядром підсистеми захисту обрано алгоритм HMAC-SHA256 (Hash-based Message Authentication Code). На відміну від стандартних

методів хешування, HMAC забезпечує автентифікацію повідомлення шляхом комбінування криптографічної хеш-функції SHA-256 із секретним ключем (Shared Secret). З інженерної точки зору цей процес описується наступною моделлю:

$$HMAC(K, m) = H((K \oplus opad) \parallel H((K \oplus ipad) \parallel m)),$$

де H – хеш-функція SHA-256;
K – секретний ключ, що зберігається в енергонезалежній пам'яті ESP32 та у захищеному сховищі сервера;
m – випадковий рядок, згенерований сервером;
opad та *ipad* – зовнішня та внутрішня константи заповнення (маски), що використовуються для захисту від специфічних атак на хеш-функції.

Функціональні вимоги до управління даними:

1. Реалізація об’єктної моделі зберігання файлів із підтримкою S3-сумісного API (MinIO).
2. Створення гнучкого механізму управління віртуальною ієрархією папок у межах плоского адресного простору об’єктів.
3. Впровадження алгоритму прямого завантаження даних у сховище для мінімізації навантаження на основний серверний вузол.
4. Підтримка комбінованого контролю доступу через рольову модель та індивідуальні списки прав на конкретні об’єкти.

Серверна частина має бути реалізована на базі фреймворку NestJS із використанням мови TypeScript, що забезпечує модульність архітектури та виявлення помилок на етапі розробки [4, 50]. Для забезпечення незалежності від операційного середовища всі компоненти (база метаданих PostgreSQL, S3-сховище MinIO та API-сервіс) мають бути контейнеризовані за допомогою Docker [5, 44].

2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ ТА МОДЕЛЕЙ ВЗАЄМОДІЇ КОМПОНЕНТІВ ПРОГРАМНО-АПАРАТНОЇ СИСТЕМИ ПРИВАТНОГО СХОВИЩА

2.1 Архітектурна модель розподіленої системи та принципи багатошарової побудови ПЗ

Проєктована програмно-апаратна система приватного сховища розглядається як єдиний функціональний комплекс, побудований на принципах розподілених обчислень. На відміну від монолітних архітектур, у яких усі процеси управління даними та безпеки зосереджені в одному потоці виконання, запропонована система розподіляє основні функції між спеціалізованими вузлами. Це дозволяє оптимізувати використання ресурсів сервера та підвищити відмовостійкість за рахунок ізоляції підсистем.

Архітектура системи включає чотири функціональні вузли. Центральним логічним компонентом є NestJS API, який виконує роль оркестратора бізнес-логіки, керування метаданими та контролю доступу без обробки бінарних даних. Вузол зберігання реалізовано на базі MinIO як об’єктного сховища, оптимізованого для високонавантажених I/O-операцій за протоколом S3. Асинхронну синхронізацію станів між веб-сесіями та фізичними діями користувача забезпечує Redis, що виконує функцію швидкого сховища станів. Окремим компонентом є апаратний вузол на базі ESP32, який виступає зовнішнім довіреним агентом і виконує криптографічні операції в ізольованому середовищі, недоступному для компрометації з боку програмного середовища клієнта або сервера.

Багатошарова модель також забезпечує чітке розділення відповідальності між компонентами системи, що зменшує складність супроводу та тестування програмного забезпечення. Кожен із шарів може розвиватися незалежно, не впливаючи на функціонування інших рівнів

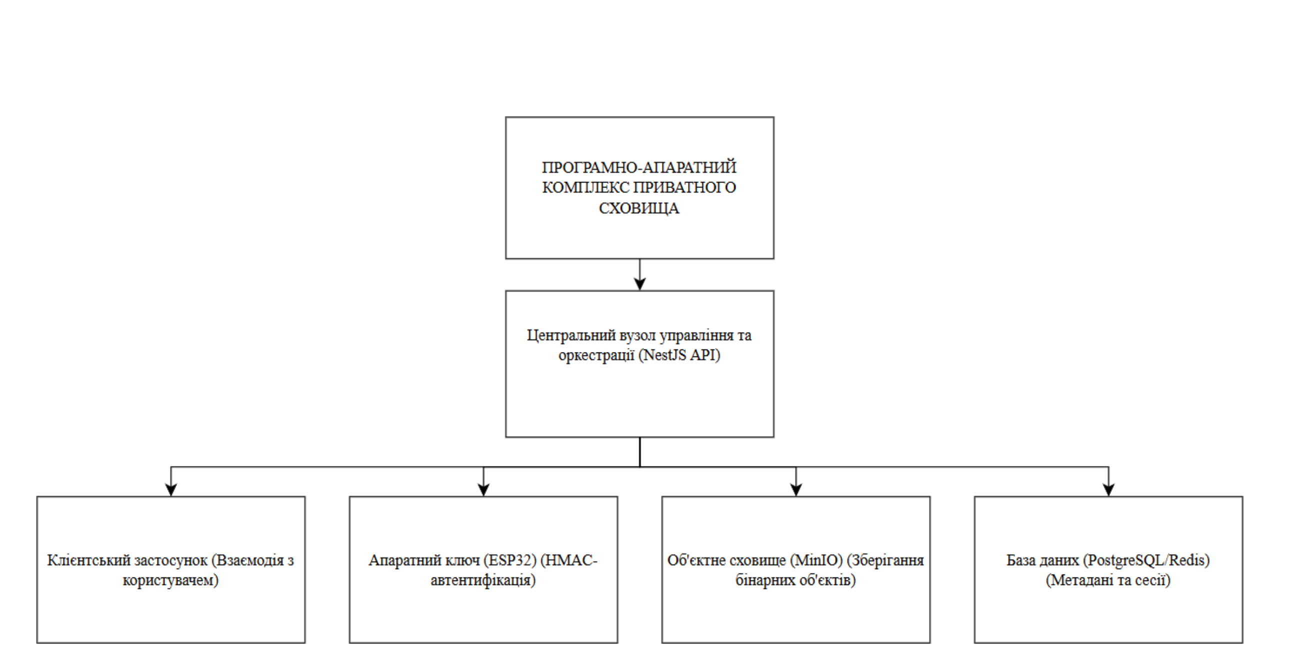


Рисунок 2.1 – Структурна схема взаємодії компонентів розподіленої системи

На рис. 2.1 представлена структурна схема взаємодії компонентів системи. Центральним елементом є NestJS API, який виконує функцію координатора. Особливістю архітектури є пряме передавання бінарних даних від клієнта до MinIO (Direct Data Stream) без проходження через серверну пам’ять, що знижує навантаження на бекенд. Вузол Redis забезпечує асинхронну взаємодію між клієнтськими діями та підтвердженням на ESP32, виконуючи роль спільного сховища стану сесій.

Інтеграція компонентів реалізована через трирівневу модель, що охоплює інфраструктурний, логічний та комунікаційний рівні. Інфраструктурний рівень базується на контейнеризації Docker, яка забезпечує ізоляцію API, бази даних і об’єктного сховища, а також стабільний розподіл ресурсів і стійкість до локальних збоїв. На рівні даних використано гібридний підхід: структурована інформація зберігається в PostgreSQL із Prisma ORM, а бінарні файли — у MinIO. Узгодження даних забезпечується через унікальний ідентифікатор storageKey. Комунікація між компонентами реалізована через HTTP/S і REST, що забезпечує універсальність взаємодії між веб-клієнтом та ESP32.

Апаратний модуль ESP32 виконує роль «gatekeeper» і працює як автономний мережевий агент. Така архітектура забезпечує ізоляцію криптографічних ключів, які зберігаються безпосередньо на пристрої, знижуючи ризики компрометації при атаках на робочу станцію. Використання Redis дозволяє реалізувати асинхронний сценарій підтвердження сесії, коли ініціація входу виконується на клієнті, а підтвердження — на фізичному пристрої. Це підвищує рівень безпеки та забезпечує готовність системи до IoT-сценаріїв.

Архітектура системи базується на принципі слабкої зв’язності компонентів (Loose Coupling), що забезпечує гнучкість та масштабованість. Усі взаємодії реалізуються через стандартизовані API, що дозволяє замінювати окремі компоненти (MinIO, ESP32) без змін у логічному ядрі системи. Використання NestJS забезпечує модульність, а Prisma ORM — типобезпечну роботу з реляційними зв’язками між користувачами, пристроями та файловими метаданими. Таким чином, запропонована архітектура є універсальною та придатною для розгортання в різних інфраструктурних середовищах.

Окрему увагу при проєктуванні приділено внутрішній багатошаровій архітектурі програмного забезпечення бекенд-сервера. Для забезпечення слабкої зв'язності та високої тестованості коду, ПЗ розділено на три логічні рівні. Рівень представлення (Presentation Layer) реалізований через REST-контролери, які відповідають за валідацію вхідних даних та маршрутизацію. Рівень бізнес-логіки (Domain Layer) ізолює правила обробки файлів, ієрархію папок та алгоритми криптографічної перевірки підписів. Інфраструктурний шар (Infrastructure Layer) містить адаптери для роботи з MinIO SDK, Redis та Prisma ORM. Така декомпозиція дозволяє вносити зміни у роботу окремих модулів (наприклад, змінити СУБД або тип об'єктного сховища) без необхідності модифікації ядра системи, що є дуже важливим для універсальних розподілених систем.

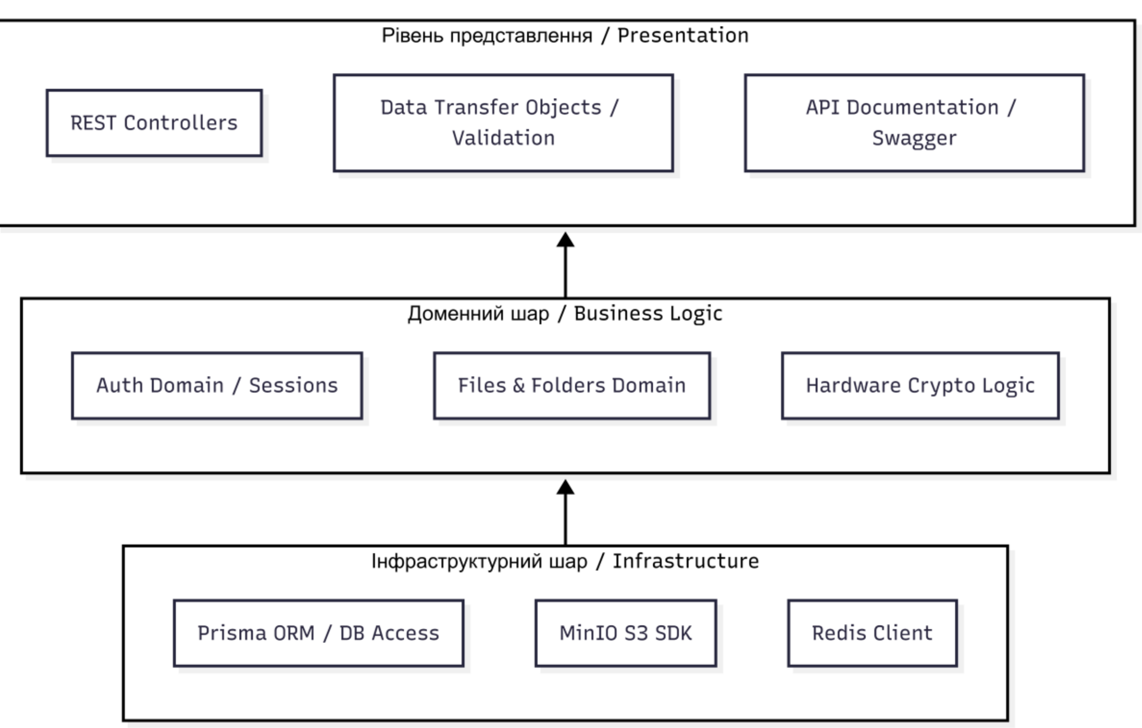


Рисунок 2.2 – Принципи багатошарової побудови ПЗ

На рис. 2.2 зображено ієрархічну модель шарів програмного забезпечення. В основі лежить інфраструктурний шар, що забезпечує низькорівневу взаємодію з базами даних та фізичним сховищем. Доменний шар містить ключові бізнес-правила системи та є незалежним від зовнішніх інтерфейсів. Верхній рівень представлення відповідає за обробку вхідних HTTP-запитів та повернення результатів користувачу. Такий підхід гарантує ізольованість логіки та спрощує подальшу підтримку програмного комплексу.

2.2 Проєктування апаратного модуля автентифікації на базі мікроконтролера ESP32

Важливим етапом проєктування програмно-апаратної системи є розробка автономного довіреного вузла, який виконуватиме роль фізичного ідентифікатора користувача. Вибір елементної бази для цього модуля базується на необхідності поєднання високої обчислювальної потужності, підтримки мережевих стеків та наявності апаратних засобів захисту інформації. У межах

даного проєкту в якості основного керуючого контролера обрано мікроконтролер архітектури Tensilica Xtensa LX6 – ESP32 (WROOM-32). Дане рішення є оптимальним завдяки інтегрованому модулю Wi-Fi/Bluetooth, що дозволяє реалізувати концепцію «мережевого ключа», та наявності розгалуженої периферії GPIO для організації індикації та фізичних органів управління. Головною перевагою ESP32 перед аналогами (наприклад, серією ESP8266) є наявність вбудованого апаратного криптографічного прискорювача, що забезпечує виконання операцій хешування та шифрування на рівні кремнію, а не програмної емуляції.

Використання апаратного криптографічного прискорювача мікроконтролера ESP32 є фундаментальним інженерним рішенням для гарантування цілісності процесів автентифікації. В архітектурі обраного чіпа реалізовано незалежний обчислювальний блок (hardware crypto engine), який підтримує стандарти SHA-1, SHA-256, SHA-384 та SHA-512 на рівні логічних вентилів. На відміну від програмної реалізації алгоритмів хешування, де обчислення виконуються загальним центральним процесором (CPU) через набір інструкцій, апаратний прискорювач працює автономно. Це дозволяє системі проводити складні математичні перетворення паралельно з обробкою мережевого стека Wi-Fi, що важливо для стабільного підтримання HTTP-сесій. З точки зору комп'ютерної інженерії, апаратне виконання SHA-256 забезпечує стійкість до атак по сторонніх каналах (side-channel attacks), таких як аналіз часових затримок (timing analysis) або моніторинг енергоспоживання, оскільки операції в крипто-модулі відбуваються за детерміновану кількість тактів незалежно від вхідного набору даних.

Окремим аспектом проєктування є організація системи візуального зворотного зв'язку, що дозволяє користувачу контролювати етапи проходження протоколу Challenge-Response. Для цього до схеми інтегровано три світловипромінювальні діоди (LED) різних кольорів. Кожен світлодіод виконує роль індикатора стану скінченного автомата прошивки: жовтий колір сигналізує

про процес ініціалізації Wi-Fi та очікування відповіді від сервера; червоний – вказує на стан готовності системи або виникнення помилки автентифікації; зелений – підтверджує успішну верифікацію підпису та надання доступу. Такий підхід до побудови людино-машинного інтерфейсу (HMI) забезпечує інформативність пристрою без необхідності використання складних дисплейних модулів, що знижує загальне енергоспоживання апаратного ключа.

Вибір мікроконтролера ESP32 (модифікація WROOM-32) в якості обчислювального ядра системи обумовлений його високою продуктивністю та розвиненою архітектурною периферією. Дана система на кристалі (SoC) базується на двох 32-бітних ядрах Xtensa® LX6 із максимальною тактовою частотою до 240 МГц. Така конфігурація дозволяє ефективно реалізувати багатопотокову обробку завдань на апаратному рівні: поки одне ядро забезпечує стабільне Wi-Fi з’єднання та обробку мережевого стека TCP/IP для взаємодії з бекендом, друге ядро займається виключно криптографічними перетвореннями та логікою скінченного автомата прошивки.

Окремим аспектом проєктування апаратного модуля є організація внутрішнього простору Flash-пам’яті для надійного збереження криптографічних секретів. У межах проєкту реалізовано спеціалізовану схему розмітки пам'яті (Partition Table), де виділено ізольований розділ NVS (Non-Volatile Storage). На відміну від стандартної файлової системи SPIFFS, NVS базується на архітектурі ключ-значення та використовує механізми вирівнювання зносу (wear leveling). Це гарантує, що «Shared Secret» буде захищений від пошкодження навіть при багаторазовому перезаписуванні конфігурацій. Під час ініціалізації пристрою доступ до цього розділу здійснюється через зашифровані дескриптори, що створює додатковий шар захисту від зчитування дампа пам'яті за допомогою зовнішніх програматорів.

З точки зору енергоефективності, проєкт враховує особливості споживання мікроконтролера ESP32 у різних режимах роботи. Оскільки апаратний ключ більшу частину часу перебуває в очікуванні дій користувача, спроєктована

модель передбачає оптимізацію роботи Wi-Fi модема. Використання протоколу Modern Sleep дозволяє мікроконтролеру відключати радіочастотний блок у паузах між маяковими кадрами (DTIM), що знижує середній струм споживання з 160-240 мА до 20-30 мА. При цьому зберігається миттєва готовність до відправки криптографічного підпису. Таке рішення дозволяє в майбутньому перевести пристрій на автономне живлення від літій-полімерного акумулятора без значного збільшення габаритів корпусу.

Внутрішня архітектура пам'яті пристрою включає 520 КБ статичної оперативної пам'яті (SRAM), що забезпечує достатній ресурс для буферизації вхідних HTTP-пакетів та виконання ресурсомісткого парсингу JSON-відповідей від NestJS сервера в режимі реального часу. Для надійного збереження керуючої програми та конфігураційних параметрів пристрою використовується 4 МБ енергонезалежної Flash-пам'яті. Особливу увагу при проєктуванні приділено використанню захищеного розділу NVS (Non-Volatile Storage) для збереження унікального криптографічного ключа (Shared Secret). Це гарантує, що конфіденційна інформація не буде втрачена при відключенні живлення та залишиться ізольованою від зовнішнього доступу.

Важливим для спеціальності «Комп'ютерна інженерія» є наявність в ESP32 апаратного генератора справді випадкових чисел (Hardware RNG) та виділених прискорювачів для стандартів AES, SHA-2 та RSA. Використання апаратного SHA-256 прискорювача дозволяє проводити обчислення HMAC-підпису безпосередньо на рівні логічних вентилів, що значно швидше та безпечніше за програмну емуляцію. Розгалужена система введення-виведення, яка налічує 34 програмовані GPIO-піни, надає високу інженерну гнучкість при проєктуванні схемотехнічного рішення, зокрема для підключення трирівневої системи світлодіодної індикації та фізичних органів управління, що забезпечують підтвердження присутності користувача.

Схемотехнічне рішення апаратного модуля розроблено з урахуванням вимог стабільності логічних рівнів та енергоефективності. Для візуалізації станів

системи до мікроконтролера підключено три індикаторні світлодіоди через струмообмежувальні резистори номіналом 220 Ом, що забезпечує оптимальну яскравість при номінальній напрузі 3.3 В. Жовтий світлодіод підключено до порту GPIO 14, червоний – до GPIO 12, а зелений – до GPIO 13. Використання широтно-імпульсної модуляції (PWM) на цих портах дозволяє в майбутньому реалізувати плавні переходи індикації для покращення людино-машинної взаємодії.

Для реалізації фізичного фактора захисту використано тактову кнопку, підключену до порту GPIO 27. З метою спрощення апаратної частини та підвищення надійності, вхід кнопки сконфігуровано в режимі INPUT_PULLUP із використанням внутрішнього підтягуючого резистора мікроконтролера. Це дозволяє уникнути плаваючого потенціалу на вході та гарантує чітке спрацювання логічного нуля при замиканні контакту на землю (GND). Спільна шина заземлення об'єднує всі компоненти схеми, створюючи єдиний контур повернення струму для стабільної роботи бездротового модуля Wi-Fi.

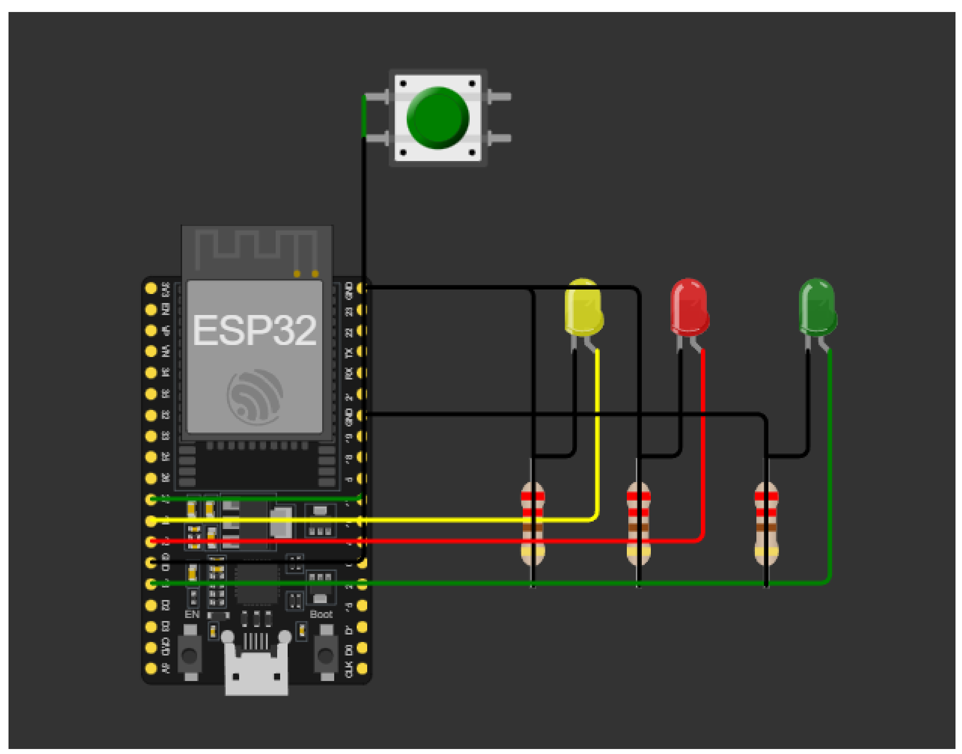


Рисунок 2.3 – Монтажна схема

Алгоритмічна база прошивки апаратного ключа розроблена на основі концепції скінченного автомата (Finite State Machine, FSM), що забезпечує високу стабільність роботи та детермінованість переходів між станами пристрою. Використання моделі FSM дозволяє чітко розділити процеси керування периферією та виконання мережевих протоколів, виключаючи виникнення «гонок станів» (race conditions). Програмна логіка реалізована мовою C++ з використанням фреймворку Arduino в середовищі PlatformIO, що надає доступ до низькорівневих системних викликів операційної системи реального часу FreeRTOS, яка інтегрована в ядро ESP32.

На першому етапі функціонування пристрій переходить у стан ініціалізації. У цьому режимі виконується конфігурація GPIO-портів, ініціалізація послідовного порту для логування та запуск Wi-Fi стеку. Протягом процесу встановлення з’єднання з точкою доступу жовтий світлодіод працює в режимі блимання, сигналізуючи про відсутність мережевої готовності. Після отримання локальної IP-адреси пристрій автоматично переходить у стан очікування . У цьому стані активується червоний індикатор, а мікроконтролер переходить у режим опитування стану вхідного порту GPIO 27, очікуючи на фізичну взаємодію з користувачем.

При механічному натисканні кнопки ініціюється перехід до стану отримання виклику. Апаратний модуль формує асинхронний HTTP-запит до бекенд-сервера за унікальним ендпоінтом /auth/hardware/challenge. У відповідь сервер генерує та надсилає випадковий 32-байтний рядок (криптографічний Nonce). Важливою особливістю реалізації є те, що цей рядок зберігається лише в оперативній пам’яті мікроконтролера на час поточної сесії, що мінімізує ризики його компрометації.

Основним етапом роботи є перехід до стану криптографічного підпису. Тут активується апаратний прискорювач ESP32 для виконання алгоритму HMAC-SHA256. Для реалізації цієї функції використано спеціалізовану бібліотеку mbedtls, яка інтегрована безпосередньо в SDK мікроконтролера.

Процес обчислення базується на комбінуванні отриманого від сервера «виклику» та спільного секретного ключа, який вилучається із захищеного енергонезалежного сховища NVS. Завдяки використанню апаратних регістрів, обчислення відбувається поза загальним адресним простором CPU, що унеможливлює зчитування проміжних значень підпису іншими програмними потоками.

Завершальний етап включає стан верифікації. Сформований HMAC-підпис упаковується в JSON-структуру за допомогою бібліотеки ArduinoJson та надсилається на сервер методом HTTP POST. У разі отримання статусу «200 OK», пристрій переходить у стан успіху, активуючи зелений світлодіод на фіксований час, після чого повертається в режим очікування. У разі виникнення мережевої помилки або відмови у верифікації, система переходить у стан помилки, що супроводжується інтенсивним блиманням червоного індикатора. Така логіка гарантує користувачу повну візуальну прозорість процесів захищеного обміну даними.

Додатковим рівнем захисту апаратного модуля є механізм програмного блокування повторних спроб автентифікації. У разі багаторазового введення некоректного криптографічного підпису або виникнення серії мережевих помилок мікроконтролер переходить у режим тимчасового блокування, під час якого нові запити до сервера не виконуються протягом визначеного інтервалу часу. Такий підхід дозволяє мінімізувати ризики автоматизованих атак типу brute force та зменшує навантаження на бекенд-сервер у випадку несанкціонованої активності. Реалізація подібного механізму на рівні прошивки забезпечує автономність системи захисту навіть при частковій недоступності серверної інфраструктури.

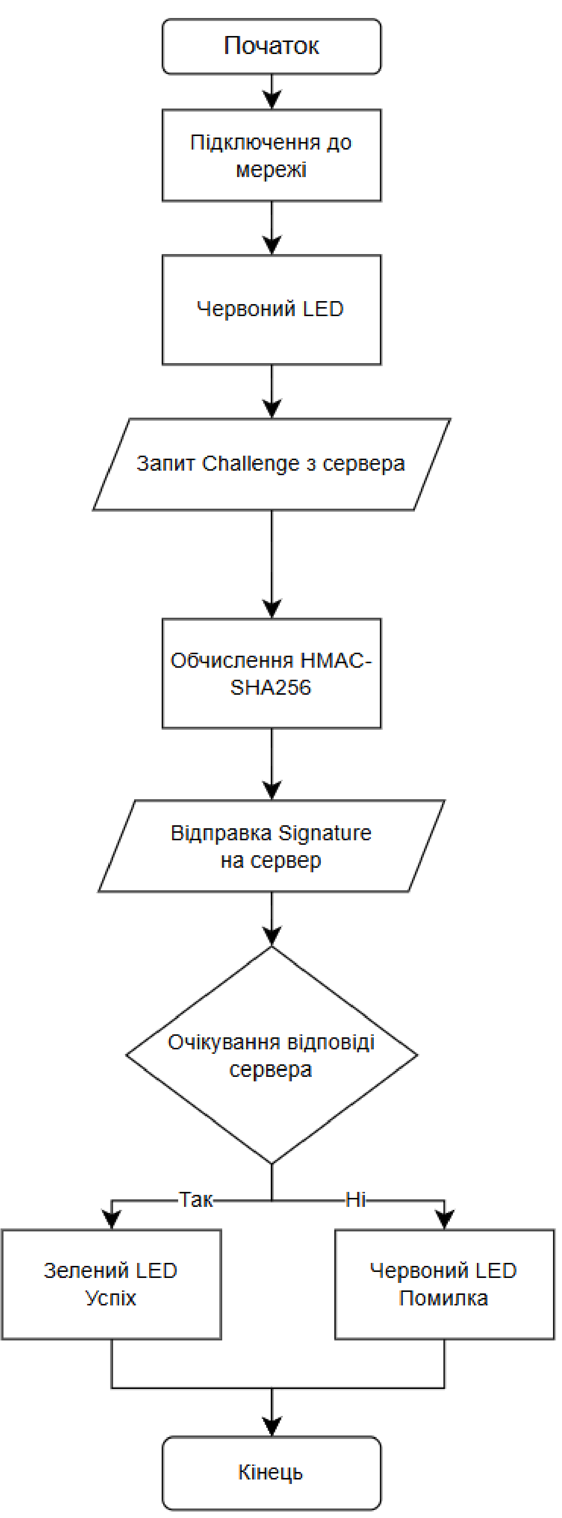


Рисунок 2.4. – Блок-схема алгоритму роботи прошивки апаратного модуля

Для систематизації відомостей про фізичну конфігурацію пристрою та забезпечення прозорості подальшої збірки апаратного прототипу, детальний розподіл пінів (Pinout Table) мікроконтролера ESP32 наведено в табл. 2.1.

Таблиця 2.1 – Специфікація підключення компонентів до GPIO ESP32

Компонент	Пін ESP32	Тип сигналу	Призначення
Button	GPIO 27	Input (Digital)	Тригер початку автентифікації
Yellow LED	GPIO 14	Output (PWM)	Індикація обробки даних
Red LED	GPIO 12	Output (PWM)	Режим очікування / Помилка
Green LED	GPIO 13	Output (PWM)	Доступ дозволено

Вибір пінів GPIO 12-14 та 27 зумовлений їхньою приналежністю до блоку RTC (Real-Time Clock), що в майбутньому дозволяє переводити пристрій у режими глибокого сну для економії енергії, зберігаючи при цьому можливість пробудження за зовнішнім перериванням від кнопки.

Криптографічний підрозділ прошивки є найбільш відповідальною частиною проєктування, оскільки саме тут реалізується логіка формування цифрового доказу володіння ключем. В основі захисту лежить алгоритм HMAC-SHA256, який поєднує високу стійкість до колізій із можливістю апаратного прискорення на рівні SoC.

Процес криптографічного перетворення в межах апаратного модуля ілюструється схемою (рис. 2.5), де чітко розмежовано джерела вхідних даних.

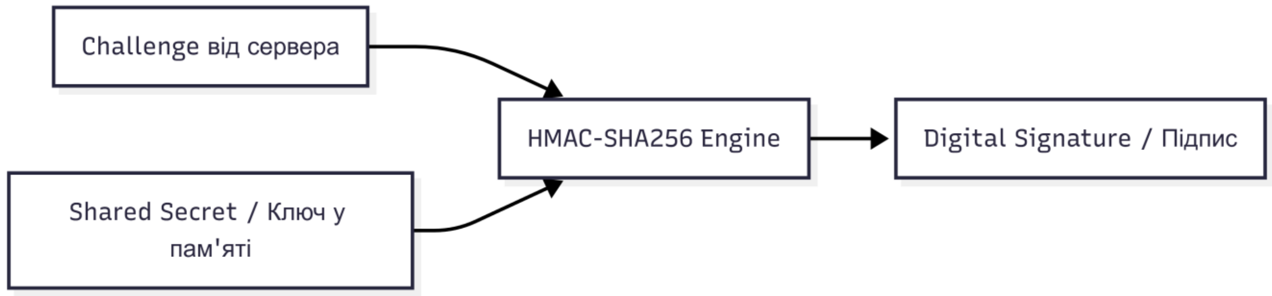


Рисунок 2.5 – Схема логічного криптографічного перетворення на стороні пристрою

Як показано на рис. 2.5, обчислення підпису базується на взаємодії двох незалежних параметрів. Першим параметром є «Shared Secret» – унікальний 256-бітний ключ, що записується в пам’ять пристрою на етапі ініціалізації та ніколи не передається за межі мікроконтролера. Для його безпечного зберігання використано механізм NVS (Non-Volatile Storage), який реалізує захищене розділення пам'яті Flash-накопичувача. Другим параметром є динамічний «Challenge», який генерується сервером для кожної окремої спроби входу.

Математична стійкість даної моделі полягає у тому, що результат є незворотним. Зловмисник, перехопивши підпис у відкритій Wi-Fi мережі, не зможе відновити Shared Secret або використати підпис повторно, оскільки для наступної сесії сервер згенерує новий випадковий виклик. Використання апаратних регістрів ESP32 для зберігання проміжних значень хеш-функції SHA-256 гарантує, що конфіденційні дані не будуть доступні стороннім програмним потокам, що важливо для систем із архітектурою «нульової довіри».

Розробка програмного забезпечення (Firmware) для апаратного модуля здійснювалася з використанням об’єктно-орієнтованого підходу мовою C++ у середовищі PlatformIO. Для забезпечення високої надійності та безпеки мережевої взаємодії було задіяно набір спеціалізованих бібліотек, які дозволяють ефективно використовувати ресурси мікроконтролера ESP32. Основним інструментом для виконання криптографічних операцій обрано бібліотеку mbedtls, яка забезпечує інтерфейс до апаратного прискорювача чіпа. Використання методів даної бібліотеки дозволяє проводити ініціалізацію контексту HMAC та послідовне опрацювання блоків даних безпосередньо у захищених регістрах крипто-модуля. Такий підхід гарантує, що «Shared Secret» не потрапляє у відкриту область оперативної пам’яті у відкритому вигляді під час обчислень.

Для обробки вхідних та вихідних повідомлень від бекенд-сервера застосовується бібліотека ArduinoJson, яка оптимізована для роботи в системах з обмеженим обсягом ОЗП. Процес десеріалізації JSON-відповідей сервера

(наприклад, при отриманні криптографічного «виклику») реалізовано з використанням механізму статичного розподілу пам'яті (StaticJsonDocument). Це дозволяє уникнути фрагментації купи (heap fragmentation) та забезпечує стабільну роботу пристрою при тривалій експлуатації без необхідності перезавантаження. Кожна транзакція супроводжується перевіркою цілісності структури повідомлення та валідацією полів, що запобігає збоям прошивки при отриманні некоректних даних з мережі.

Мережевий стек пристрою базується на використанні бібліотек WiFi.h та HTTPClient.h. Під час проєктування логіки взаємодії особливу увагу було приділено механізмам обробки виняткових ситуацій та затримок виконання. У разі втрати Wi-Fi з'єднання або відсутності відповіді від NestJS API протягом встановленого ліміту часу (10 секунд), скінченний автомат прошивки ініціює примусовий перехід у стан ERROR. Це супроводжується очищенням буферів із тимчасовими токенами та поверненням до стану ініціалізації. Така архітектурна особливість реалізує принцип самоорганізації апаратної частини: пристрій самостійно намагається відновити зв'язок із сервером, не потребуючи втручання адміністратора або користувача.

На завершення опису апаратного модуля варто відзначити високий рівень інтеграції між низькорівневим кодом прошивки та високорівневою логікою бекенда. Використання стандартизованих REST-інтерфейсів дозволяє ESP32 виступати повноцінним учасником розподіленої системи, а апаратна реалізація HMAC-SHA256 перетворює мікроконтролер на надійне джерело довіри. Спроєктована програмно-апаратна конфігурація модуля автентифікації є повністю завершеною і готовою до інтеграції у загальну схему взаємодії з файловим сховищем, опис якої буде наведено у наступному підрозділі.

2.3 Розробка протоколу взаємодії «Сервер – Апаратний токен»

Основним аспектом проєктування розподіленої системи є розробка захищеного протоколу взаємодії між двома незалежними клієнтами: веб-інтерфейсом користувача та апаратним ключем на базі мікроконтролера ESP32. Оскільки ці пристрої працюють асинхронно, виникає необхідність у створенні проміжного шару синхронізації станів, який би дозволив об'єднати результати програмної та апаратної перевірки в межах єдиної сесії автентифікації. У даному проєкті роль такого координатора виконує система Redis, а логіка взаємодії реалізована за протоколом Challenge-Response (Виклик-Відповідь).

Алгоритмічна послідовність проходження багатофакторної автентифікації та обміну даними між вузлами системи детально представлена на аркуші графічної частини «Схема багатофакторної автентифікації» (див. Додаток А). Процес базується на механізмі відкладеного підтвердження (deferred acceptance), де веб-сесія залишається у стані очікування до моменту отримання валідного криптографічного підпису від апаратного пристрою.

На першому етапі взаємодії, після валідації стандартної пари «логін-пароль», NestJS API не генерує фінальні JWT-токени, а створює в базі Redis тимчасовий запис, пов’язаний із Email користувача. Веб-клієнт отримує статус HARDWARE_AUTH_REQUIRED і переходить у режим циклічного опитування стану цієї сесії. Використання Redis як «моста» синхронізації дозволяє уникнути тривалого блокування потоків виконання на сервері та забезпечує високу швидкість реакції системи на зовнішні події [32].

Другий етап протоколу передбачає генерацію криптографічного «виклику» (Nonce) на запит від ESP32. Важливою особливістю розробленого протоколу є те, що «виклик» є абсолютно випадковим для кожної спроби входу та має обмежений час життя (TTL) у 120 секунд. Це гарантує захист від атак повторного відтворення: навіть якщо зловмисник перехопить підписаний пакет

даних, він не зможе використати його для входу в іншу сесію, оскільки Nonce уже буде недійсним.

Апаратний ключ, отримавши виклик, проводить обчислення HMAC-SHA256, використовуючи Shared Secret, що зберігається в його енергонезалежній пам'яті. Сформований підпис передається на сервер, де відбувається його фінальна верифікація. У разі успіху, сервер оновлює стан сесії в Redis, записуючи туди сформовані JWT-токени (accessToken та refreshToken). При наступному запиті веб-клієнта до ендпоінта перевірки статусу, він отримує готові токени і завершує процедуру входу.

Застосування такої моделі взаємодії дозволяє реалізувати концепцію фізичної ізоляції секретів. Апаратний ключ ESP32 виступає як незалежний довірений агент (Trusted Agent), який спілкується з сервером через захищене API, але при цьому не має прямого зв'язку з веб-браузером користувача. Це забезпечує стійкість системи до компрометації робочої станції користувача: навіть за наявності шкідливого ПЗ на ПК, зловмисник не зможе отримати доступ до файлового сховища без фізичної дії на апаратному ключі. Розроблений протокол є універсальним інженерним рішенням, що дозволяє інтегрувати в єдину систему безпеки різнотипне обладнання та клієнтські додатки.

Для забезпечення асинхронної взаємодії між веб-клієнтом та апаратним модулем, у системі спроєктовано спеціалізовану структуру ключів у сховищі Redis. Використання in-memory бази даних обумовлено необхідністю мінімального часу відгуку (Latency < 1 мс) при високій частоті запитів Polling-циклу. Ключ у Redis формується динамічно за шаблоном auth_session:{email}, де значенням виступає JSON-об'єкт, що містить поточний статус сесії, згенерований Nonce та тимчасове сховище для JWT-токенів.

Важливою інженерною особливістю є використання механізму Atomic Operations у Redis. При отриманні підпису від ESP32, сервер NestJS виконує операцію SETEX (Set with Expiry), яка одночасно оновлює статус автентифікації та встановлює жорсткий ліміт часу життя запису (Time-To-Live, TTL). Це

гарантує самоочищення системи: якщо користувач натиснув кнопку на ключі, але не завершив вхід у браузері протягом 120 секунд, дані автоматично видаляються з пам'яті, що запобігає витоку токенів та захаращенню оперативної пам'яті сервера.

Протокол взаємодії передбачає сувору послідовність криптографічних перетворень на стороні мікроконтролера ESP32. Отриманий від сервера «виклик» (Challenge) має довжину 32 байти і є результатом роботи криптографічно стійкого генератора псевдовипадкових чисел (CSPRNG) на NestJS.

При обчисленні підпису алгоритм HMAC-SHA256 виконує подвійне хешування, що ілюструється внутрішньою логікою прошивки. Секретний ключ спочатку комбінується з внутрішньою маскою ipad (0x36), після чого результат конкатенується з повідомленням (Challenge) і хешується. Отриманий проміжний результат повторно конкатенується з результатом комбінування ключа із зовнішньою маскою opad (0x5c) і проходить фінальне хешування. Такий складний ланцюжок перетворень, реалізований на рівні апаратних регістрів ESP32, унеможливлює проведення атак на основі подовження повідомлення (Length Extension Attacks), які є вразливістю для звичайних функцій SHA-256.

Оскільки передача даних відбувається через Wi-Fi канали, протокол взаємодії має враховувати можливі мережеві затримки та втрати пакетів. У межах проєкту реалізовано стратегію «Exponential Backoff» для повторних спроб з’єднання на стороні ESP32 та систему таймаутів на стороні NestJS.

Якщо апаратний ключ надсилає підпис, але сервер не може його верифікувати (наприклад, через невідповідність Nonce), сесія негайно блокується в Redis із міткою FAILED_ATTEMPT. Веб-клієнт, отримавши такий статус під час чергового запиту /status, припиняє опитування та виводить повідомлення про помилку. Це створює механізм самоорганізації безпеки: система автоматично перериває потенційну атаку в момент виявлення першої ж

невідповідності даних, не дозволяючи зловмиснику продовжувати підбір підпису в межах тієї самої сесії.

Під час проєктування було проведено порівняння розробленого HMAC-протоколу з існуючими стандартами безпеки, такими як FIDO2. Хоча FIDO2 базується на криптографії з відкритим ключем (Asymmetric Cryptography), він потребує підтримки браузером специфічних API та часто залежить від сторонніх драйверів операційної системи.

Перевагою розробленого в даній роботі протоколу «Server – ESP32» є повна автономність від клієнтського ПЗ. Оскільки апаратний токен є мережевим агентом, він може підтверджувати вхід незалежно від того, з якого пристрою (ПК, смартфон чи IoT-панель) користувач ініціював сесію. Це реалізує концепцію «Out-of-band Authentication» (позасмугова автентифікація), де підтвердження відбувається через канал зв’язку, фізично відмінний від основного каналу передачі пароля. Таке розділення каналів вважається найвищим стандартом безпеки в комп'ютерній інженерії, оскільки навіть повна компрометація одного каналу (наприклад, перехоплення трафіку в браузері) не дає зловмиснику контролю над другим (апаратним ключем).

Проєктування протоколу також враховувало часові характеристики взаємодії. Теоретичний розрахунок показує, що повний цикл автентифікації (від натискання кнопки до отримання доступу в браузері) займає не більше 1.5–2 секунд, враховуючи час проходження сигналу через мережу та обчислення HMAC.

На апаратному рівні реалізовано синхронізацію станів індикації з етапами протоколу. Коли ESP32 перебуває у стані обчислення HMAC, активується жовтий світлодіод, що візуалізує для користувача внутрішній обчислювальний процес. Це знижує ймовірність повторних натискань кнопки та робить систему більш зручною в експлуатації. Таким чином, розроблений протокол є не лише засобом захисту, а й повноцінним комунікаційним шаром, що об’єднує

програмні ресурси сервера з фізичними можливостями мікроконтролера в єдину відмовостійку систему.

2.4 Функціональна декомпозиція та паралельне виконання завдань

Процес проєктування складної програмно-апаратної системи передбачає проведення функціональної декомпозиції – розподілу загальної мети системи на окремі незалежні завдання, що можуть виконуватися паралельно або автономно. Це дозволяє усунути вузькі місця в архітектурі, забезпечуючи високу швидкість відгуку інтерфейсу та стабільність роботи апаратної частини. У межах даної системи виділено чотири критичні процеси, що реалізують принцип паралельної обробки даних.

Оркестрація двоетапного завантаження файлів. Це найбільш ресурсомістке завдання, яке в традиційних системах часто призводить до блокування потоків виконання API-сервера. У розробленій архітектурі реалізовано паралельне розділення потоку управління та потоку даних. Коли користувач завантажує файл, NestJS API виконує лише легку операцію перевірки прав та генерації підписаного S3-посилання. Безпосередня передача бінарних байтів відбувається паралельно, минаючи бекенд-сервер, напряму до контейнера MinIO. Таке рішення дозволяє основному серверному процесору зосередитися на обробці логічних запитів, тоді як мережевий інтерфейс та дискова підсистема сховища працюють автономно. Це дозволяє системі підтримувати завантаження гігабайтних файлів без ризику вичерпання оперативної пам'яті на рівні Node.js додатка.

Забезпечення інформаційної безпеки потребує фіксації кожної дії в системі, що створює додаткове навантаження на базу даних. Для оптимізації цього процесу впроваджено асинхронну модель на базі EventEmitter. При виконанні будь-якої операції (наприклад, видалення файлу), основний потік виконання негайно повертає відповідь користувачу, тоді як завдання запису в

журнал аудиту (ActivityLog) випромінюється як внутрішня подія. Спеціалізований лістенер (Listener) перехоплює цю подію та проводить запис у PostgreSQL у фоновому режимі. Таке розпаралелювання гарантує, що час відгуку системи не залежить від швидкості запису логів у базу даних, що критично для масштабованих інфраструктур.

Паралельна синхронізація апаратних та веб-сесій через Redis. Протокол автентифікації передбачає одночасне функціонування двох незалежних обчислювальних процесів: циклічного опитування (Polling) з боку веб-браузера та криптографічного обчислення на стороні ESP32. Використання Redis як проміжного шару дозволяє цим процесам працювати у паралельних часових доменах. Бекенд-сервер обробляє запити від апаратного ключа та оновлює стан сесії атомарно, не блокуючи вхідні запити від веб-клієнта. Це демонструє переваги розподіленої архітектури, де стан системи (State) відокремлений від логіки обробки (Logic), що дозволяє обслуговувати тисячі одночасних спроб входу без виникнення взаємних блокувань.

Динамічна мікросервісна обробка медіаконтенту. Для покращення продуктивності клієнтських інтерфейсів система потребує генерації зменшених копій завантажених зображень. Це завдання є обчислювально важким для CPU. В архітектурі проєкту це завдання повністю делеговано окремому вузлу – мікросервісу imgproxy. Коли клієнт запитує зображення, API генерує URL, який перенаправляє запит на imgproxy. Таким чином, процеси ресайзу та трансформації зображень відбуваються паралельно до роботи основного API, в окремому Docker-контейнері з власними лімітами ресурсів. Це забезпечує ізоляцію навантаження: навіть при масовому завантаженні медіафайлів, критичні функції автентифікації та управління доступом залишатимуться доступними та швидкими.

Важливою складовою функціональної декомпозиції є проєктування віртуальної мережевої топології всередині середовища контейнеризації. Вся система розгорнута в межах єдиної ізольованої мережі Docker Bridge Network,

що реалізує концепцію «Network Encapsulation». Це дозволяє повністю закрити порти бази даних PostgreSQL (5432) та кешу Redis (6379) від зовнішнього доступу. Прямий доступ з мережі інтернет дозволено лише для API-шлюзу та веб-інтерфейсу MinIO, що мінімізує «поверхню атаки» на програмно-апаратний комплекс .

Реалізація взаємодії між вузлами базується на механізмі Service Discovery (автоматичне виявлення сервісів). Замість використання статичних IP-адрес, які можуть змінюватися при перезапуску контейнерів, компоненти системи звертаються один до одного за внутрішніми іменами хостів, наприклад, db:5432 або storage:9000. Docker-оркестратор виконує роль внутрішнього DNS-сервера, який динамічно резолвить ці імена. Такий підхід забезпечує високу гнучкість системи: при необхідності масштабування сховища, а саме додавання нових вузлів MinIO, логічне ядро NestJS продовжує працювати без зміни конфігураційних файлів, автоматично отримуючи доступ до розширених ресурсів через віртуальний комутатор мережі. Це перетворює набір ізольованих програм на цілісну самоорганізовану інфраструктуру, здатну до адаптації під мінливі апаратні умови.

Самоорганізація в архітектурі реалізована на декількох рівнях:

1. На рівні контейнеризації: Docker Engine автоматично моніторить стан здоров'я (Health Checks) кожного вузла. У разі критичного збою бекенда або сховища MinIO, система оркестрації ініціює перезапуск відповідного контейнера, автоматично відновлюючи зв'язки через внутрішню віртуальну мережу.

2. На рівні управління сесіями: Механізм TTL (Time-To-Live) у Redis забезпечує автоматичне очищення пам'яті від застарілих криптографічних викликів та токенів, що не були використані.

3. На рівні апаратного ключа: Прошивка ESP32 містить логіку автономного перепідключення до Wi-Fi та скидання станів при помилках зв'язку, що гарантує готовність пристрою до роботи без необхідності його ручного перезавантаження.

3 ПРОГРАМНО-АПАРАТНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ ПРИВАТНОГО СХОВИЩА

3.1 Програмна реалізація та архітектурна структура бекенд-сервера

Ефективність функціонування програмно-апаратного комплексу приватного сховища безпосередньо залежить від якості проєктування схеми бази даних, яка має забезпечувати цілісність метаданих, гнучке керування правами доступу та надійний зв'язок із апаратними токенами. Для реалізації рівня персистентності даних було обрано реляційну СУБД PostgreSQL, а взаємодія з нею реалізована за допомогою сучасного інструментарію Prisma ORM. Повна логічна структура бази даних із зазначенням усіх зв'язків, індексів та типів полів наведена у Додатку Б [17, 28].

Важливою інженерною особливістю реалізації є використання модульної структури схем Prisma. Замість створення єдиного громіздкого файлу конфігурації, моделі розподілені за логічними доменами (access, credentials, hardware, files тощо). Це дозволило досягти високої читабельності коду та спростити процеси CI/CD. Всі сутності використовують універсальні унікальні ідентифікатори (UUID) як первинні ключі. На відміну від автоінкрементних цілочисельних значень, UUID забезпечують вищий рівень безпеки, оскільки унеможливлюють атаку шляхом перебору (enumeration) ідентифікаторів об’єктів або профілів користувачів у системі [12].

Одним із архітектурних рішень стало розділення профілю користувача та його конфіденційних даних для автентифікації. Це реалізовано через суворий зв'язок «один-до-одного» (1:1) між моделями User та Credential. Таке розділення відповідальності дозволяє ізолювати технічну інформацію (хешовані паролі, солі) від бізнес-даних профілю, що є стандартом при побудові захищених систем.

Оскільки об’єктне сховище MinIO за своєю природою має «пласку» структуру, реалізація звичної для користувача ієрархії каталогів була перенесена на рівня бази даних. Це досягнуто через використання рекурсивного зв'язку (self-

relation) у моделі Folder. Кожна папка може мати посилання на батьківський елемент (parentId), що дозволяє будувати віртуальні дерева каталогів необмеженої вкладеності без дублювання фізичних даних у S3-сховищі.

Для реалізації багатофакторної автентифікації спроєктовано таблицю hardware, що включає моделі Device та HardwareChallenge. Таблиця Device виконує роль реєстру довірених апаратних ключів, де для кожного мікроконтролера ESP32 зберігається унікальний «Shared Secret». У свою чергу, модель HardwareChallenge забезпечує роботу протоколу Challenge-Response, фіксуючи видані сервером випадкові «виклики», час їхньої ек expiration (expiresAt) та статус використання, що гарантує захист від повторних атак.

Особливу увагу приділено механізму делегування прав доступу (ACL), який реалізовано через сутність AccessGrant. Дана модель є сполучною ланкою між користувачами та об’єктами (файлами чи папками). Вона дозволяє призначати ролі VIEWER (тільки перегляд) або EDITOR (повний доступ) для конкретних суб’єктів. Така гнучкість дозволяє реалізувати складні сценарії спільної роботи над документами в межах приватного сховища.

Для забезпечення простежуваності всіх процесів у системі впроваджено модель ActivityLog. З метою оптимізації структури БД та підтримки динамічних метаданих, поле details у цій моделі має тип JSON. Це інженерне рішення дозволяє зберігати в одному стовпці різнорідну інформацію (наприклад, IP-адресу входу, розмір завантаженого файлу або помилки верифікації HMAC) без необхідності модифікації схеми таблиці при додаванні нових типів подій.

Практична реалізація спроєктованої структури бази даних базується на автоматизованому циклі синхронізації моделей Prisma із фізичною схемою PostgreSQL. Для цього в межах розробки використовується набір інструментальних скриптів, інтегрованих у менеджер пакетів npm. Головну роль відіграє команда prisma generate, яка на основі описаних вище моделей генерує типобезпечний (Type-Safe) клієнт для NestJS, що дозволяє працювати з БД без ризику виникнення помилок невідповідності типів на рівні коду. Безпосередня

зміна структури таблиць у PostgreSQL виконується командою prisma db push, яка синхронізує схему без необхідності ручного написання SQL-міграцій, що суттєво прискорює ітеративну розробку програмно-апаратного комплексу.

Для забезпечення стабільного функціонування та портативності всі вузли зберігання та обробки даних контейнеризовані за допомогою Docker. Це дозволяє розгорнути ідентичне оточення як на локальній машині розробника, так і на цільовому сервері. Інфраструктура описується конфігураційним файлом docker-compose.yml, де визначено взаємодію трьох основних сервісів:

1. Сервіс postgres:15 – забезпечує персистентне зберігання метаданих. Для запобігання втраті інформації при перезапуску контейнерів використано механізм Docker Volumes (./storage/postgres_data), що монтує фізичну папку сервера до внутрішньої файлової системи бази даних.

2. Сервіс redis:7-alpine – реалізує високошвидкісний кеш для управління сесіями та тимчасовими криптографічними викликами апаратного ключа. Доступ до Redis захищено паролем на рівні командного рядка запуску сервера.

3. Сервіс minio – виконує роль S3-сумісного об’єктного сховища. Конфігурація передбачає автоматичне створення бакетів за допомогою допоміжного контейнера createbuckets на базі minio/mc. Це реалізує принцип самоорганізації системи: при першому запуску комплекс самостійно ініціалізує простір зберігання (mc mb) та встановлює необхідні політики доступу.

Усі критичні параметри доступу та секретні ключі винесені у файл конфігурації середовища .env. Такий підхід забезпечує високий рівень безпеки, оскільки дозволяє гнучко змінювати токени та паролі без модифікації вихідного коду програми. Структура змінних оточення розділена на функціональні блоки:

1. Параметри БД: DATABASE_URL, що містить облікові дані для підключення Prisma до PostgreSQL.

2. Параметри S3: MINIO_ACCESS_KEY та MINIO_SECRET_KEY для авторизації запитів до сховища.

3. Криптографічні секрети: LOCAL_HASH_SALT для «соління» паролів та JWT_KEY для підпису токенів доступу.

4. Інтеграційні параметри: TELEGRAM_BOT_API_KEY та TELEGRAM_CHANNEL_ID для забезпечення роботи модуля оповіщень та відновлення доступу через відправку кодів. В майбутньому це буде здійснювати через сервіси віпдравки електронних писем прямо на пошту користувача.

Використання вказаних інструментів дозволяє створити цілісну, ізольовану від зовнішнього впливу інфраструктуру, де кожен компонент працює за суворими правилами взаємодії, визначеними на рівні конфігураційних файлів та змінних середовища. Таким чином, розроблена реляційна структура на базі PostgreSQL та Prisma ORM створює цілісний фундамент для програмно-апаратного комплексу. Вона забезпечує Type-Safe взаємодію на рівні коду, гарантує безпечне зберігання криптографічних секретів та підтримує складні механізми ієрархічного зберігання та контролю доступу.

Процес розробки програмного забезпечення серверної частини вимагав створення масштабованої, безпечної та зручної для подальшої підтримки кодової бази. Для реалізації логічного ядра системи було обрано фреймворк NestJS, архітектура якого нативно підтримує принципи модульності та інверсії залежностей (Dependency Injection). З метою забезпечення високої якості коду та дотримання принципів SOLID, програмну архітектуру проєкту побудовано на основі концепції предметно-орієнтованого проєктування (Domain-Driven Design, DDD). Такий підхід дозволив розділити вихідний код на чітко ізольовані логічні шари, кожен з яких виконує єдину відповідальність: інфраструктурний шар, шар бізнес-логіки, шар представлення та шар наскрізного функціоналу.

Фізична структура каталогів розробленого програмного забезпечення відображає прийняті архітектурні рішення та складається з наступних основних директорій: libs (інфраструктура), domain (предметна область), rest (API інтерфейси), а також common та shared (допоміжний функціонал).

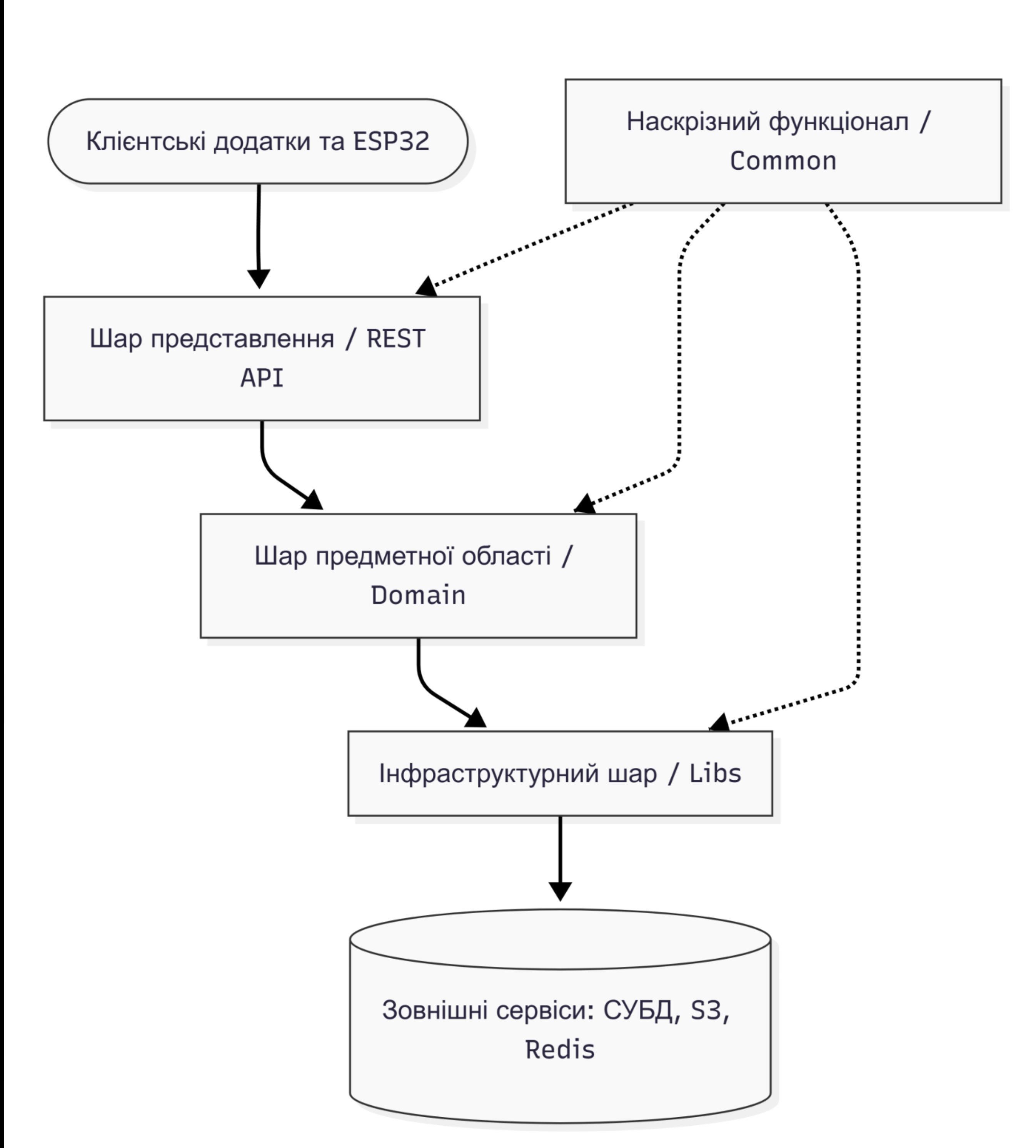


Рисунок 3.1 – Загальна багаторівнева архітектура бекенд-сервера

В основу взаємодії системи із зовнішнім середовищем покладено інфраструктурний шар (libs), який повністю абстрагує бізнес-правила від конкретних реалізацій баз даних чи мережевих протоколів. У межах цього шару реалізовано адаптери для роботи з головними технологічними вузлами. Зокрема,

модуль бази даних використовує Prisma ORM для забезпечення типобезпечної взаємодії з PostgreSQL. Модуль файлового сховища інтегрує MinIO SDK, реалізуючи механізм генерації попередньо підписаних посилань (Presigned URLs). Це інженерне рішення дозволяє клієнтським додаткам завантажувати файли безпосередньо у фізичне сховище, минаючи оперативну пам'ять бекенд-сервера, що критично важливо для обробки великих масивів даних. Додатково на цьому рівні реалізовано модулі кешування на базі Redis для управління життєвим циклом сесій та модуль криптографії для генерації і розшифрування JWT-токенів із застосуванням симетричного шифрування AES-256. Схему підключення інфраструктурних адаптерів зображено на рисунку 3.2.

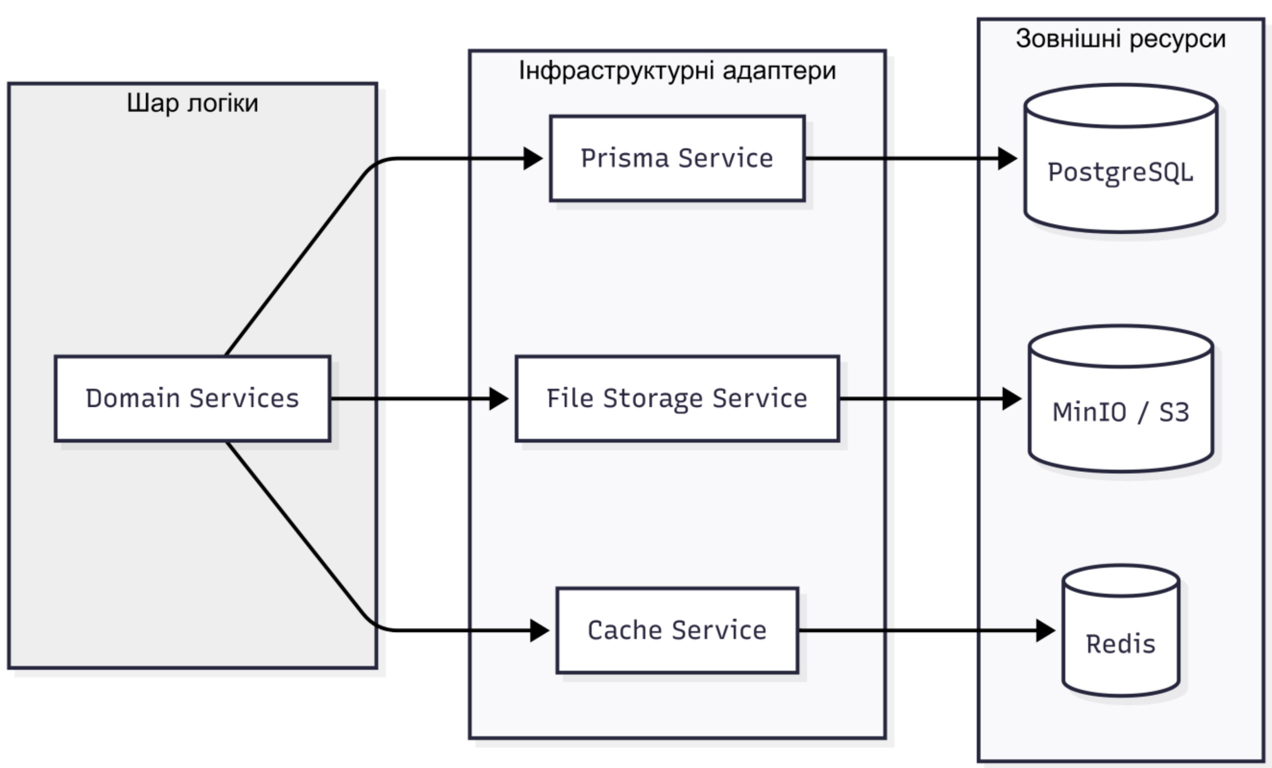


Рисунок 3.2 – Схема взаємодії бізнес-логіки з інфраструктурними сервісами

Центральним елементом програмної системи є шар предметної області (domain), який ізолює внутрішню логіку роботи приватного сховища від зовнішніх інтерфейсів. Цей шар декомпозитовано на незалежні модулі відповідно до їх функціонального призначення. Модуль управління файлами

(files) реалізує логіку перевірки списків контролю доступу (ACL), побудову віртуальної ієрархії каталогів та механізми спільного використання об'єктів (надання ролей VIEWER або EDITOR). Модуль апаратної безпеки (hardware) виступає ядром підсистеми двофакторної автентифікації. Саме тут реалізовано алгоритм генерування випадкових криптографічних викликів (Challenge) та математичну верифікацію HMAC-SHA256 підписів, отриманих від мікроконтролера ESP32. Управління станами користувачів зосереджено в модулі sessions, який забезпечує автоматичне закриття застарілих підключень та механізм відклику компрометованих токенів. Окрему увагу приділено системі аудиту (logs): для фіксації системних подій застосовано подієво-орієнтовану модель (на базі EventEmitter2), що дозволяє асинхронно записувати логі у фоновому режимі, не збільшуючи час очікування відповіді для користувача.

Зовнішній доступ до реалізованої бізнес-логіки забезпечується через шар представлення (rest), який відповідає за маршрутизацію (Routing) вхідних HTTP-запитів. Для гарантування безпеки даних на етапі їх надходження до контролерів, у системі широко застосовуються об’єкти передачі даних (Data Transfer Objects, DTO) у поєднанні з бібліотекою class-validator. Наявність таких структур, як HardwareVerifyDto чи SignUpDto, забезпечує сувору типізацію та валідацію вхідних параметрів. Це унеможливлює передачу некоректних або шкідливих даних у шар предметної області, виступаючи ефективним методом захисту від ін'єкцій та маніпуляцій з API. Додатково в архітектурі передбачено рівень доступу до даних, що реалізує взаємодію з базою через ORM та репозиторії, ізольовані від бізнес-логіки. Усі операції читання і запису виконуються з урахуванням цілісності даних та транзакційного контролю. Обробка помилок централізована і стандартизована, що забезпечує єдиний формат відповідей API.

Такий підхід підвищує стабільність системи та спрощує її подальший розвиток. Повний життєвий цикл обробки HTTP-запиту в системі відображено на рисунку 3.3.

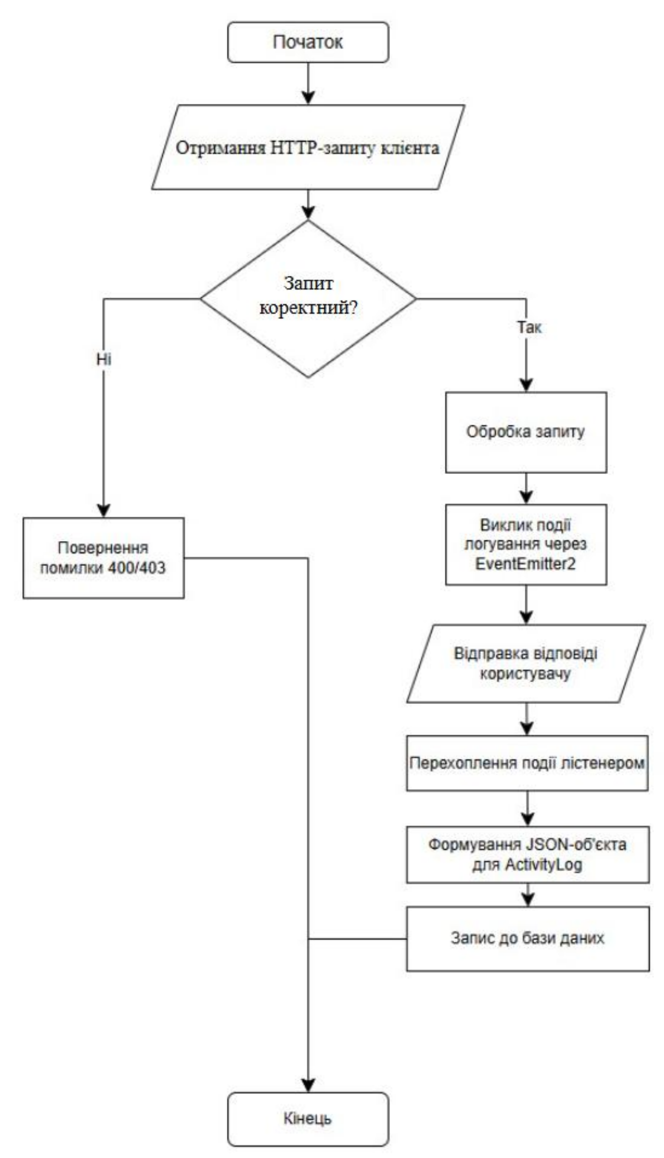


Рисунок 3.3 – Алгоритм обробки вхідного запиту та асинхронного логування

Для стандартизації обробки наскрізного функціоналу системи виділено модулі загального призначення (common та shared). Одним із архітектурних рішень на цьому рівні є впровадження глобального фільтра винятків DomainExceptionsFilter. Він автоматично перехоплює всі помилки бізнес-логіки (наприклад, невірні облікові дані або спроби несанкціонованого доступу) і трансформує їх у стандартизовані HTTP-відповіді. Такий підхід запобігає витоку системної інформації (stack trace) у разі критичних збоїв. Крім того, на рівні наскрізного функціоналу реалізовано захисники маршрутів (Guards), зокрема

AuthGuard, який здійснює первинну перевірку валідності JWT-токена ще до того, як запит досягне логічного ядра системи.

Отже, застосована модульна архітектура бекенд-сервера формує надійний фундамент для функціонування програмно-апаратного комплексу. Чітке розмежування відповідальності між шарами інфраструктури, бізнес-логіки та API гарантує високу безпеку обробки даних, стійкість до відмов та здатність системи до гнучкого масштабування чи заміни окремих компонентів у майбутньому без руйнування загальної логіки роботи.

3.2 Програмна реалізація механізмів апаратно-залежної автентифікації

Основним завданням реалізованої підсистеми безпеки є повне унеможливлення несанкціонованого доступу до приватного сховища навіть у разі повної компрометації пароля користувача. Для досягнення цієї мети впроваджено механізм багатофакторної автентифікації (2FA), де вирішальним фактором виступає фізичне володіння апаратним ключем на базі мікроконтролера ESP32. У цій архітектурі ESP32 діє як Trusted Hardware Agent – незалежний довірений вузол, що виконує криптографічні операції в ізольованому середовищі.

Алгоритм функціонування підсистеми охоплює два етапи: реєстрацію пристрою та безпосередню автентифікацію. Детальна схема інформаційних потоків між веб-клієнтом, NestJS API, сховищем Redis та апаратним ключем наведена у Додатку Г. Реалізована модель базується на принципі відкладеного підтвердження (deferred acceptance): сервер ініціює сесію, але не видає токени доступу до моменту отримання валідного криптографічного доказу від фізичного пристрою.

Програмна реалізація автентифікації на боці сервера розділена між AuthService та HardwareService. При отриманні запиту на вхід (POST /auth/sign-in), система проводить валідацію першого фактора. Якщо за обліковим записом закріплено апаратний пристрій, сервер повертає статус

HARDWARE_AUTH_REQUIRED. З цього моменту веб-клієнт переходить у стан очікування, періодично опитуючи статус сесії в Redis. Такий підхід дозволяє уникнути використання ресурсномістких WebSockets і робить систему стійкою до розривів мережевого з’єднання.

Криптографічна взаємодія базується на протоколі Challenge-Response з використанням алгоритму HMAC-SHA256. Сервер генерує випадковий 32-байтний рядок (Nonce), який апаратний ключ підписує за допомогою спільного секрету (Shared Secret). Логіку роботи алгоритму верифікації апаратного ключа можна представити у вигляді наступної послідовності:

1. Отримати від пристрою: ID_пристрою, Код_виклику (Challenge) та отриманий_підпис.
2. Знайти в базі даних пристрій за його ідентифікатором та вилучити його Секретний_ключ.
3. Перевірити стан пристрою: якщо він заблокований – зупинити з помилкою.
4. Знайти запис про Код_виклику у таблиці тимчасових завдань:
5. Якщо код уже використаний або термін дії вичерпано – відхилити запит.
6. Розрахувати «еталонний підпис» на стороні сервера:
7. Еталонний_підпис = HMAC_SHA256(Секретний_ключ, Код_виклику).
8. Порівняти Еталонний_підпис із Отриманим_підписом:
9. Якщо вони ідентичні – позначити код виклику як використаний та дозволити вхід.
10. Якщо є розбіжність – зафіксувати спробу зламу в логах та відмовити в доступі.

Алгоритм апаратного обчислення підпису:

1. Створити порожній буфер пам'яті для результату (32 байти).

2. Ініціалізувати криптографічний контекст для роботи з алгоритмом SHA256.

3. Звернутися до апаратного модуля шифрування ESP32 та завантажити в нього Секретний_ключ.

4. Подати на вхід крипто-модуля Дані_виклику (Challenge), отримані від сервера.

5. Виконати операцію фіналізації хешування для отримання бінарного підпису.

6. Звільнити використані ресурси крипто-модуля для запобігання витоку пам'яті.

7. Для кожного байта отриманого бінарного результату:

8. Перетворити байт у його шістнадцяткове текстове представлення (Hex).

9. Додати отриманий текст до загального рядка результату.

10. Повернути готовий текстовий рядок підпису.

Для координації асинхронних дій веб-браузера та ESP32 використано систему Redis. Після успішної верифікації апаратного підпису сервер зберігає згенеровані JWT-токени в кеші з терміном життя (TTL) 120 секунд. Це створює безпечне «вікно» для завершення входу. Веб-клієнт, отримавши токени через ендпоінт статусу, завершує процедуру автентифікації. Така реалізація забезпечує високий рівень ізоляції факторів захисту та робить систему стійкою до атак типу Man-in-the-Middle та Replay-атак.

Практична реалізація підсистеми автентифікації базується на модульній структурі NestJS, де логіка взаємодії з апаратним ключем винесена в окремий сервіс HardwareService. Основним завданням реалізації є забезпечення суворого контролю доступу через комбінацію програмного пароля та апаратного HMAC-підпису.

Процес розробки серверної частини почався зі створення об’єктів передачі даних для валідації вхідних HTTP-запитів. Використання декораторів class-validator (наприклад, @IsString, @IsUUID) дозволяє відсікати некоректні запити на етапі їх надходження до контролера, що запобігає спробам експлуатації вразливостей через маніпуляції з полями JSON. Кожна спроба входу ініціюється методом signIn, який при виявленні прив’язаного пристрою генерує виключення UnauthorizedException із кастомним кодом статусу. Це слугує сигналом для клієнтського додатку про необхідність переходу до етапу апаратного підтвердження.

Для реалізації криптографічного протоколу на боці сервера використано вбудований модуль crypto середовища Node.js. Процедура верифікації HMAC-підпису, отриманого від мікроконтролера ESP32, базується на детермінованому алгоритмі, який враховує часові обмеження сесії та стан апаратного токена. Логіку роботи методу верифікації (verifySignature) можна представити у вигляді наступної послідовності кроків:

1. Ініціалізація: отримання вхідного пакета даних, що містить унікальний ідентифікатор пристрою (deviceId), рядок-виклик (challenge) та отриманий від пристрою підпис (signature).

2. Ідентифікація пристрою: пошук запису про пристрій у базі даних PostgreSQL. Якщо пристрій не зареєстрований або має статус «відкликано» (revoked), виконання переривається з поверненням помилки доступу.

3. Валідація виклику: пошук метаданих «виклику» у таблиці HardwareChallenge. Система перевіряє, чи відповідає цей виклик вказаному пристрою, чи не був він використаний раніше та чи не закінчився термін його дії (TTL).

4. Генерація еталонного підпису: на основі секретного ключа пристрою (device.secret), який зберігається в БД у зашифрованому вигляді, та отриманого виклику сервер проводить обчислення очікуваного HMAC-SHA256 підпису.

5. Атомарне оновлення: незалежно від результату порівняння, виклик позначається в базі даних як «використаний», що виключає можливість проведення повторних атак (Replay attacks) з цим самим значенням Nonce.

6. Фінальна верифікація: порівняння отриманого від ESP32 підпису з еталонним обчисленим значенням. У разі повної ідентичності байтів – сесія вважається підтвердженою.

Апаратна частина реалізована як автономна прошивка. У практичній реалізації на ESP32 особливу увагу приділено надійності Wi-Fi з'єднання. Програма містить цикл автоматичного перепідключення, що забезпечує готовність ключа до роботи навіть при нестабільному сигналі мережі. Після встановлення зв'язку пристрій переходить у режим низького енергоспоживання, активуючи основний цикл обробки лише при детекції переривання від фізичної кнопки (GPIO 27).

Для виконання криптографічних обчислень на мікроконтролері використано бібліотеку mbedtls. Практична цінність цього рішення полягає у використанні апаратних регістрів ESP32, що унеможливлює зчитування Shared Secret через програмні помилки. Алгоритм верифікації криптографічного підпису на сервері:

1. Прийняти вхідні дані: унікальний ідентифікатор пристрою, код виклику (Challenge) та отриманий від мікроконтролера підпис.

2. Звернутися до реєстру пристроїв у базі даних та отримати збережений секретний ключ, закріплений за цим ідентифікатором.

3. Провести валідацію статусу пристрою: якщо ключ позначено як відкликаний – припинити обробку з помилкою доступу.

4. Код має належати саме цьому пристрою.

5. Прапор «використано» має бути відсутнім.

6. Поточний час не має перевищувати час життя коду (Expiration Time).

7. Ініціалізувати криптографічний об'єкт HMAC-SHA256, використовуючи секретний ключ із бази даних як майстер-ключ.

8. Обчислити очікуваний підпис (Expected Signature) на основі коду виклику.

9. Виконати запис у базу даних про використання даного коду виклику (захист від повторних спроб входу).

10. Порівняти очікуваний підпис із підписом, який надав пристрій:

11. При ідентичності значень – дозволити генерацію сесійних токенів.

12. При розбіжності – відмовити в авторизації.

Синхронізація станів між асинхронними вузлами системи реалізована через базу даних Redis. Після успішної перевірки підпису від ESP32, сервер записує сформовані JWT-токени за ключем auth_pending:{email} із встановленням прапорця isVerified: true. Веб-клієнт проводить періодичне опитування сервера з інтервалом 2 секунди. Як тільки запис у кеші оновлюється, сервер повертає токени клієнту, і сесія вважається встановленою. Термін життя такого запису в кеші обмежений 120 секундами, що є практичним заходом запобігання витоку прав доступу при незавершеній автентифікації.

Така багаторівнева практична реалізація дозволяє досягти балансу між зручністю користувача та максимальною захищеністю даних, перетворюючи звичайний мікроконтролер на надійний інструмент контролю доступу промислового рівня.

Центральним компонентом забезпечення фізичного рівня безпеки приватного сховища є апаратний ключ, розроблений на базі мікроконтролера архітектури ESP32-WROOM-32. Проєктне рішення передбачає реалізацію автономного мережевого вузла, який виступає в ролі довіреного агента автентифікації. Процес розробки модуля охоплював етапи схемотехнічного проєктування та створення алгоритмічного забезпечення прошивки для реалізації захищеного протоколу взаємодії з бекенд-сервером.

Схемотехнічна база пристрою спроєктована з урахуванням візуалізації станів системи та фізичного підтвердження дій користувача. Обчислювальним ядром обрано модуль ESP32 завдяки наявності апаратного криптографічного

прискорювача для SHA-256. Для забезпечення взаємодії з користувачем до схеми інтегровано тактову кнопку (GPIO 27) та трирівневу світлодіодну індикацію. Електричні з’єднання реалізовано через струмообмежувальні резистори 220 Ом, а вхід кнопки налаштовано з внутрішньою підтяжкою до високого рівня. Повна принципова схема наведена у Додатку В.

Алгоритмічне забезпечення мікроконтролера базується на моделі скінченного автомата, що забезпечує детермінованість переходів між станами пристрою. Логіку функціонування апаратної частини в межах сесії автентифікації можна представити у вигляді наступної послідовності кроків:

1. Етап ініціалізації: конфігурація режимів роботи периферії та встановлення захищеного з’єднання з локальною мережею Wi-Fi. На цьому етапі активовано режим візуального сповіщення через переривчасту роботу жовтого індикатора.

2. Режим очікування (Idle): перехід системи до стану готовності, що супроводжується активацією червоного світлодіода. Пристрій перебуває у циклі опитування стану фізичного тригера, очікуючи на дію користувача.

3. Фаза криптографічного запиту: при детекції натискання кнопки пристрій ініціює вихідний HTTP-запит до API-сервера для отримання унікального криптографічного «виклику» (Challenge).

4. Обчислення підпису: на основі отриманого Nonce-значення та секретного ключа (Shared Secret), що ізольовано зберігається в енергонезалежній пам'яті (NVS), мікроконтролер обчислює HMAC-SHA256 підпис. Використання бібліотеки mbedtls дозволяє виконувати цю операцію безпосередньо в регістрах апаратного прискорювача.

5. Верифікація та фідбек: розрахований підпис передається на сервер методом POST. При отриманні підтвердження успішної автентифікації пристрій активує зелений індикатор, що сигналізує про надання доступу до сесії сховища. У разі мережевої помилки або некоректного підпису активується режим аварійної індикації.

6. Програмна реалізація мережевого стеку на мікроконтролері забезпечує надійну обробку HTTP-статусів та парсинг вхідних даних у форматі JSON.

Особливу увагу в архітектурі прошивки приділено безпеці: секретний ключ завантажується в пам'ять пристрою одноразово і ніколи не транслюється через мережеві інтерфейси. Використання фізичної кнопки як обов'язкового етапу алгоритму реалізує концепцію «User Presence», що створює непереборний бар'єр для автоматизованих віддалених атак. Повний опис логічних структур та вихідний код прошивки представлений у Додатку Д. Спроєктований апаратний ключ є цілісним інженерним вузлом, що гарантує цілісність та конфіденційність процесів доступу до приватного файлового сховища.

3.3. Тестування та верифікація програмно-апаратного комплексу

Для підтвердження працездатності, безпеки та відмовостійкості розробленої системи було проведено комплексне тестування її компонентів. З огляду на розподілену архітектуру проєкту, процес верифікації охоплював перевірку API, апаратно-серверної взаємодії та моделювання користувацьких сценаріїв роботи з файлами. Основними інструментами тестування виступили Swagger та Postman.

Тестування логічного ядра бекенд-сервера здійснювалося через Swagger UI, який автоматично генерується на основі декораторів NestJS (@ApiTags, @ApiOperation, @ApiOkResponse). Це забезпечило ізольоване тестування ендпоінтів. У ході перевірки було верифіковано генерацію JWT-токенів при /auth/sign-in, роботу системи авторизації на основі Guards для захищених маршрутів (зокрема /files/my-folders), а також коректність глобального обробника винятків. У разі невалідних запитів система стабільно повертала HTTP-статуси 400 Bad Request або 401 Unauthorized без розкриття внутрішньої реалізації.

Перевірка безпекової складової полягала у тестуванні апаратного модуля автентифікації на базі ESP32 та протоколу Challenge–Response. Після введення облікових даних сервер повертав статус HARDWARE_AUTH_REQUIRED, ініціюючи апаратний етап. Мікроконтролер підключався до Wi-Fi, отримував криптографічний виклик через /auth/hardware/challenge та переходив у режим очікування фізичної взаємодії. Після її виконання ESP32 обчислював HMAC-SHA256 підпис за допомогою mbedtls і надсилав його на сервер (/auth/hardware/verify). Перевірка статусу (/auth/hardware/status) завершувалася видачею JWT-токенів доступу.

Додатково було проведено тестування стійкості до атак повторного відтворення (Replay Attacks). Повторна відправка застарілого HMAC-підпису відхилялася сервером із блокуванням сесії в Redis, що підтверджує коректну обробку невалідних запитів.

Процес реєстрації передбачає створення облікового запису та прив’язку профілю до фізичного пристрою як основу двофакторної автентифікації. Через POST /auth/sign-up передаються облікові дані та параметри апаратного ключа (hardwareDeviceId, hardwareSecret). У відповідь сервер повертає статус 201 Created і первинну пару JWT-токенів (accessToken, refreshToken), що підтверджує успішне створення сесії та готовність системи до подальшої роботи.

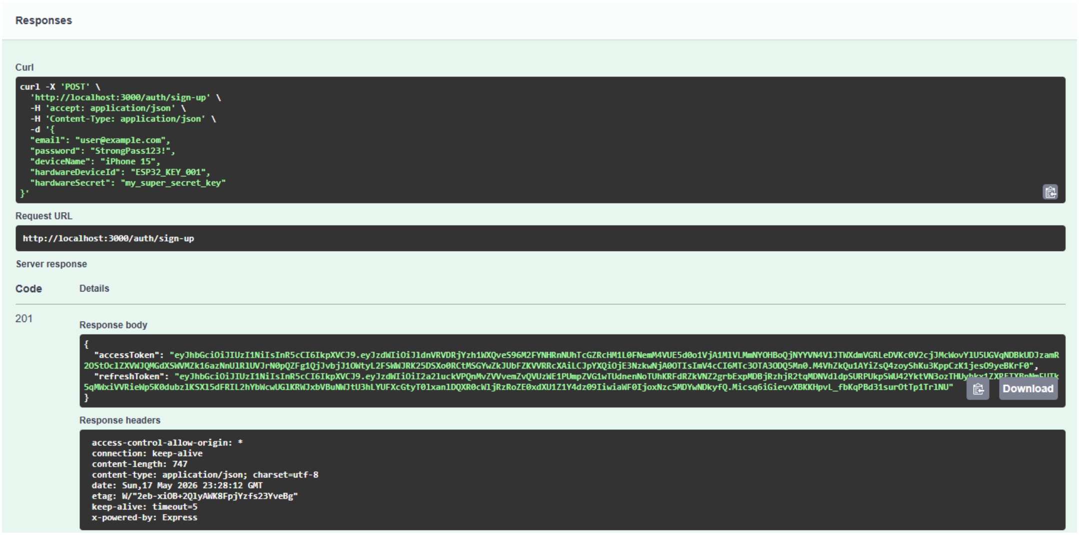


Рисунок 3.4 – Успішна відповідь сервера зі згенерованими JWT-токенами сесії

Після завершення реєстрації проводиться перевірка багаторівневого протоколу автентифікації, який моделює реальний сценарій доступу до захищеного сховища. Через Swagger UI надсилається запит із електронною поштою та паролем, після чого сервер перевіряє хеш пароля. Якщо профіль пов'язаний з апаратним пристроєм, NestJS API блокує миттєву видачу сесійних токенів і повертає статус про необхідність активації апаратного фактора захисту разом з ідентифікатором пристрою. Це підтверджує функціонування механізму посиленої автентифікації та додаткового захисту даних.

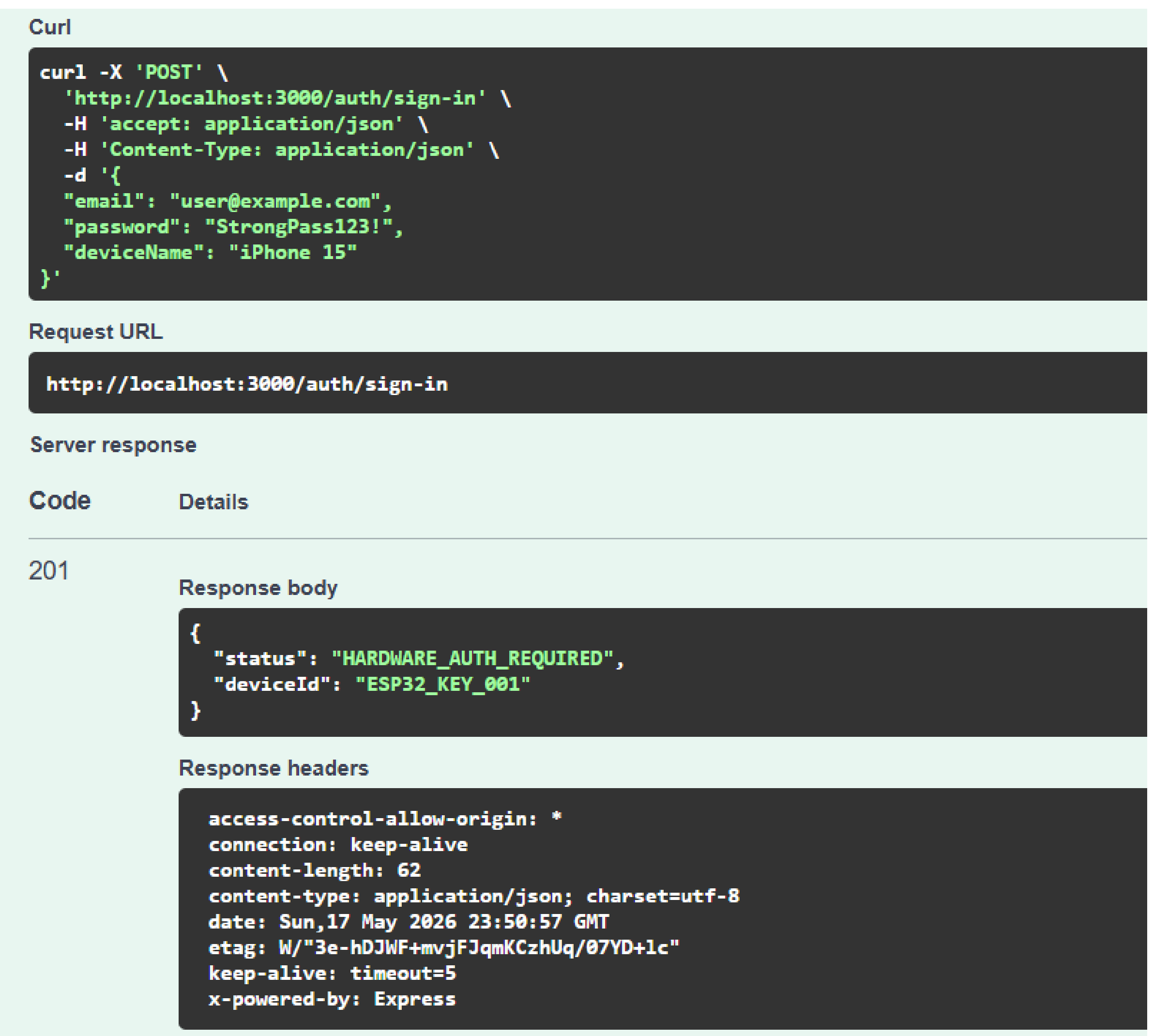


Рисунок 3.5 – Результат первинної перевірки пароля з вимогою апаратного підтвердження

Наступна фаза тестування переноситься у середовище апаратної симуляції, де мікроконтролер ESP32 перебуває у стані очікування зовнішнього переривання. При натисканні фізичної кнопки на платі активується алгоритм мережевого запиту криптографічного виклику, що фіксується у Serial Monitor симулятора.

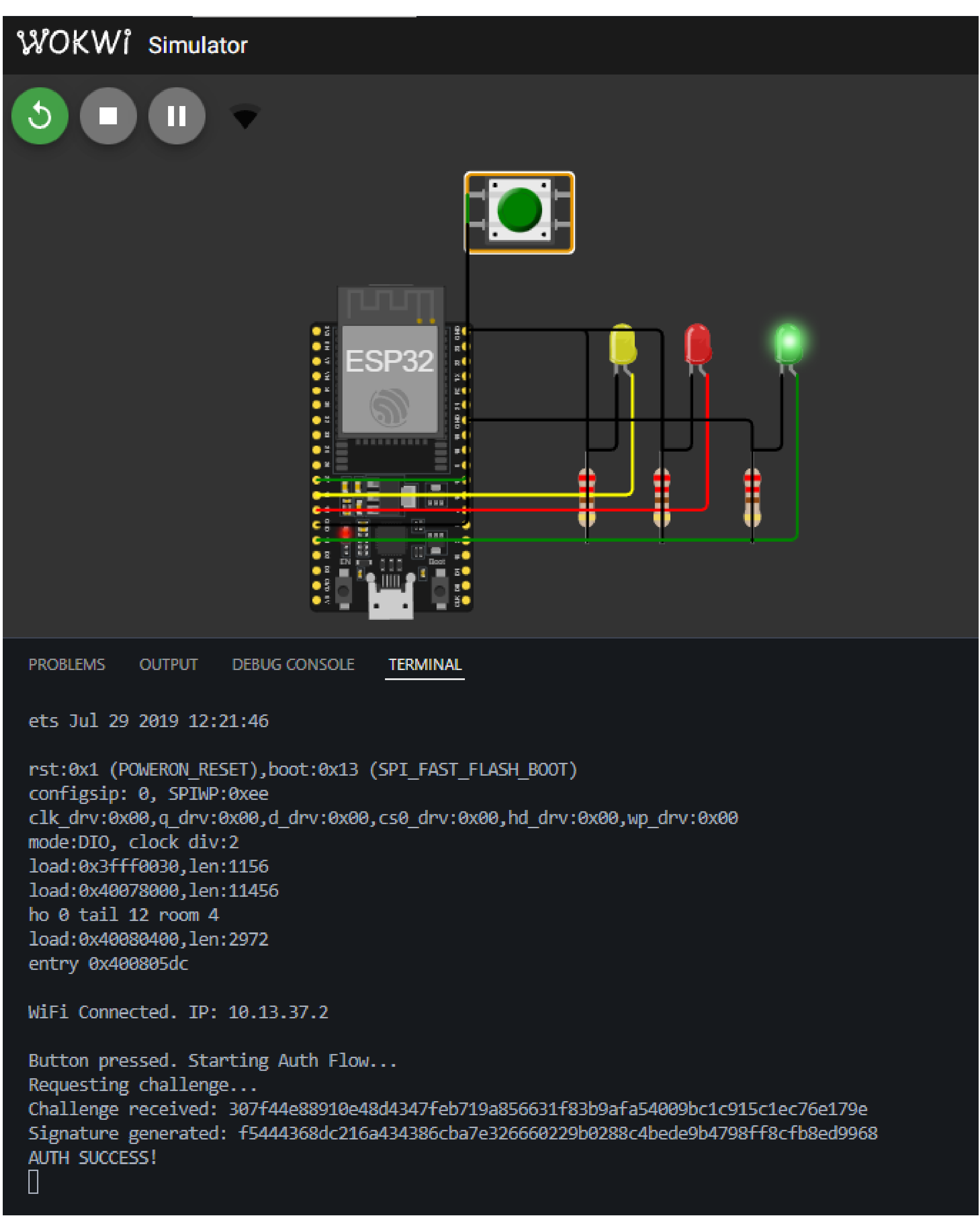


Рисунок 3.6 – Лог роботи мікроконтролера ESP32 під час обчислення криптографічного підпису

Отриманий від сервера випадковий рядок обробляється мікроконтролером із використанням спільного секретного ключа, у результаті чого формується HMAC-SHA256 підпис. Одночасно бекенд-сервер фіксує спробу апаратної верифікації в журналі аудиту.

Після надсилання підпису система виконує фінальну перевірку та, у разі збігу значень, зберігає JWT-сесію в Redis з обмеженим часом дії. Завершальним етапом є звернення клієнта до ендпоінта перевірки статусу, після чого користувач отримує фінальну пару токенів доступу. Це підтверджує успішне завершення процесу двофакторної автентифікації.

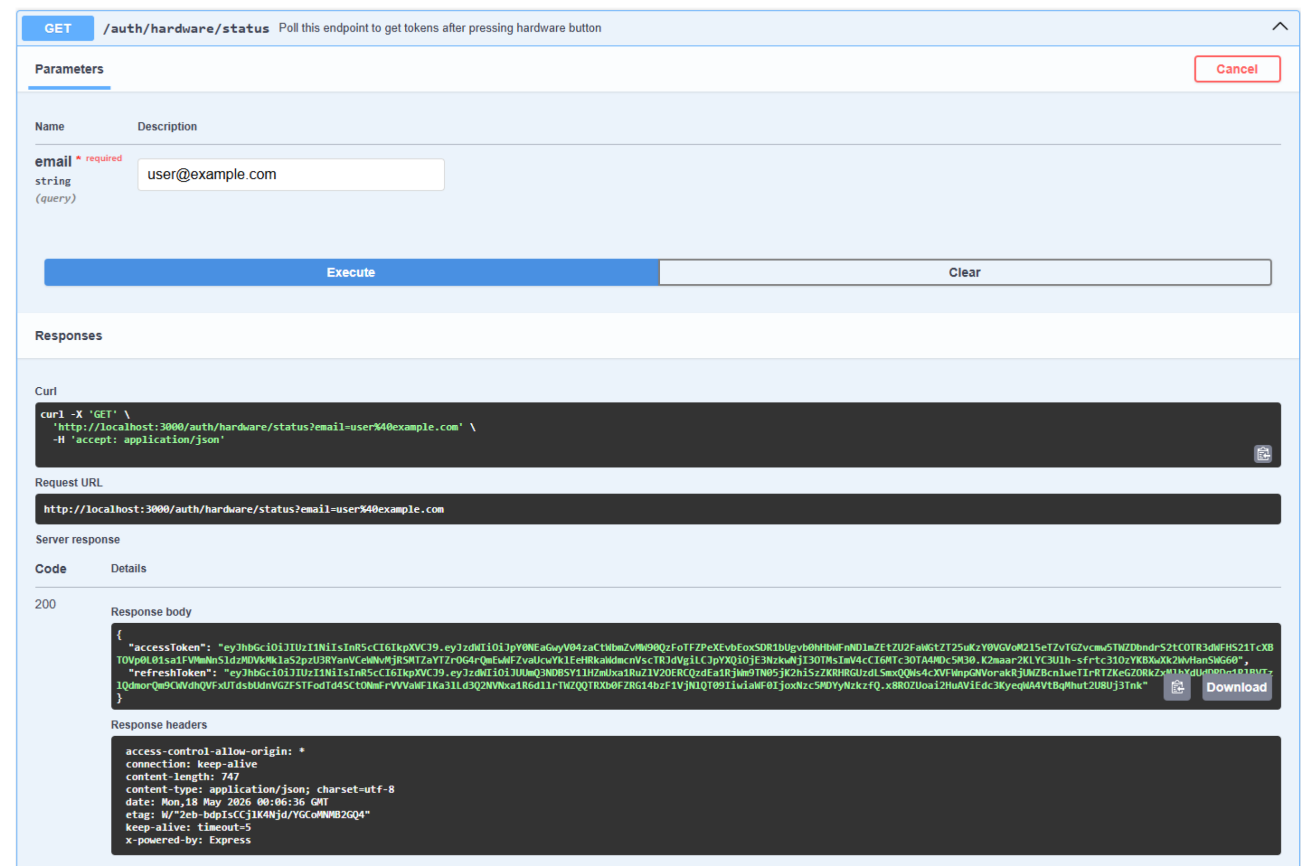


Рисунок 3.7 – Фінальна видача токенів авторизації після успішної двофакторної перевірки

Подальший етап перевірки підсистеми управління об’єктним сховищем та ієрархічної моделі зберігання даних розпочинається зі створення віртуальної

структури каталогів, які використовуються для подальшої навігації та завантаження файлів.

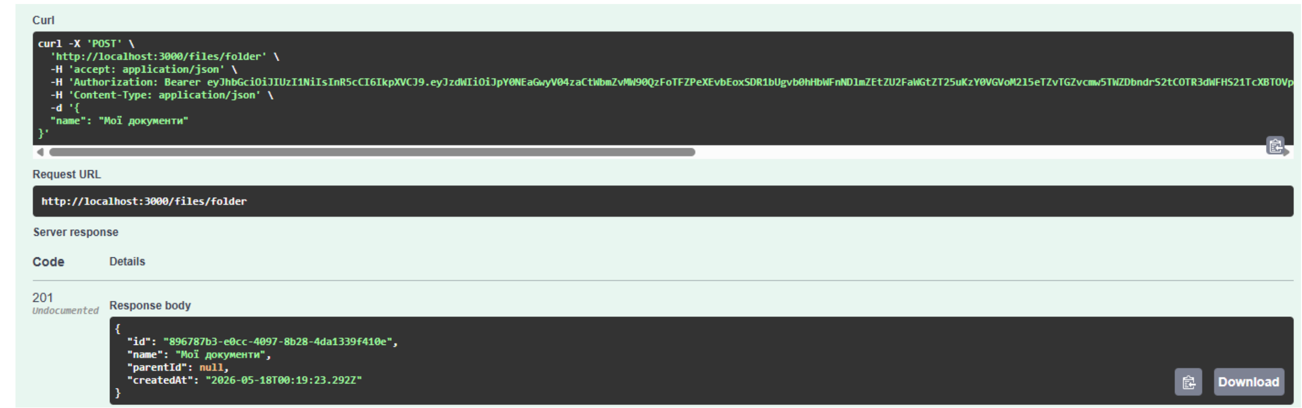


Рисунок 3.8 – Створення папки в корені віртуального каталогу

Особливу увагу приділено верифікації двоетапного протоколу завантаження об’єктів, який забезпечує пряме передавання даних між клієнтом і сховищем з мінімальним навантаженням на сервер. Процес ініціюється через Swagger UI шляхом виклику методу отримання попередньо підписаного URL, де передаються назва файлу та ідентифікатор цільової папки. У відповідь сервер формує ідентифікатор транзакції та тимчасове підписане посилання, згенероване на основі S3-ключів доступу.

Рисунок 3.9 – Генерація попередньо підписаного URL для прямого доступу до S3-вузла

Наявність цих даних дозволяє перейти до фізичної передачі бінарного контенту із використанням Postman. Виконання HTTP PUT-запиту за отриманим посиланням моделює поведінку клієнтського застосунку при роботі з потоковими даними, при цьому інформація передається безпосередньо до бакета MinIO без залучення оперативної пам’яті NestJS.

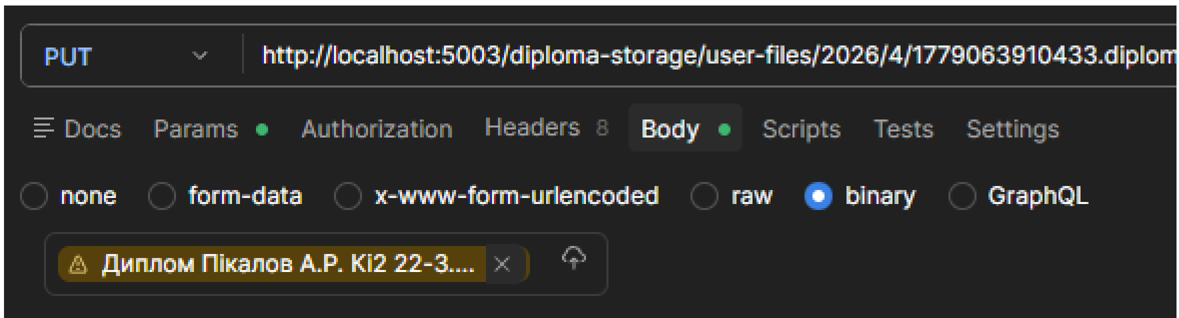


Рисунок 3.10 – Симуляція завантаження бінарних даних файлу безпосередньо у MinIO через Postman

Фіналізація процесу завантаження здійснюється шляхом передачі ідентифікатора транзакції назад до бекенд-сервера, що ініціює процедуру синхронізації станів. На цьому етапі NestJS API звертається до MinIO SDK для виконання методу перевірки характеристик об’єкта, що дозволяє отримати достовірну інформацію про реальний розмір файлу та його MIME-тип безпосередньо від файлової системи. Тільки після успішного підтвердження фізичної наявності даних у сховищі система фіксує фінальні метадані у PostgreSQL та генерує публічне посилання на файл. Перевірка результату через ендпоінт перегляду вмісту демонструє цілісну картину, де завантажені файли чітко структуровані за відповідними папками, що підтверджує коректність роботи алгоритмів віртуалізації файлової системи та надійність двоетапної моделі передачі даних.

Curl

```
curl -X 'POST' \
  'http://localhost:3000/files/upload/finish' \
  -H 'accept: application/json' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJpY0NEaGwyV04zaCtWbmZvMW90QzFoTFZPeXEv
  -H 'Content-Type: application/json' \
  -d '{
  "uploadId": "LHEMJNUQQPHEXIEHDNPQZTMU"
}'
```

Request URL

```
http://localhost:3000/files/upload/finish
```

Server response

Code	Details
201 *Undocumented*	Response body


```
{
  "id": "b36c8a2f-ad51-4940-afe0-f3470e30b8d2",
  "name": "diplom",
  "size": 6056896,
  "mimetype": "application/octet-stream",
  "publicUrl": "http://localhost:5003/diploma-storage/user-files/2026/4/1779065174922.diplom",
  "createdAt": "2026-05-18T00:46:34.531Z"
}
```


Рисунок 3.11 – Реєстрація метаданих файлу після успішної верифікації його наявності в сховищі

Curl

```
curl -X 'GET' \
  'http://localhost:3000/files/content?folderId=37011c7a-ac8e-4c85-be56-79aa976a5897' \
  -H 'accept: */*' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJpY0NEaGwyV04zaCtWbmZvMW90QzFoTFZPeXEv
```

Request URL

```
http://localhost:3000/files/content?folderId=37011c7a-ac8e-4c85-be56-79aa976a5897
```

Server response

Code	Details
200	Response body


```
{
  "folders": [],
  "files": [
    {
      "id": "b36c8a2f-ad51-4940-afe0-f3470e30b8d2",
      "name": "diplom",
      "size": 6056896,
      "mimetype": "application/octet-stream",
      "publicUrl": "http://localhost:5003/diploma-storage/user-files/2026/4/1779065174922.diplom",
      "createdAt": "2026-05-18T00:46:34.531Z"
    }
  ]
}
```


Рисунок 3.12 – Перегляд вмісту папки із зареєстрованим файловим об’єктом

Важливою функціональною перевагою розробленої системи є механізм делегування прав доступу, що дозволяє організувати спільну роботу користувачів над об’єктами сховища. Процес тестування цієї підсистеми розпочинається з ідентифікації цільового суб’єкта в загальному реєстрі користувачів. Використання ендпоінта отримання списку користувачів із застосуванням фільтрації за поштовою адресою дозволяє ініціатору запиту отримати унікальний ідентифікатор (UUID) іншого учасника системи. На цьому етапі бекенд-сервер виконує запит до PostgreSQL, повертаючи публічні профілі, що відповідають заданим критеріям пошуку, що є необхідною умовою для точного призначення прав доступу в межах розподіленої архітектури.

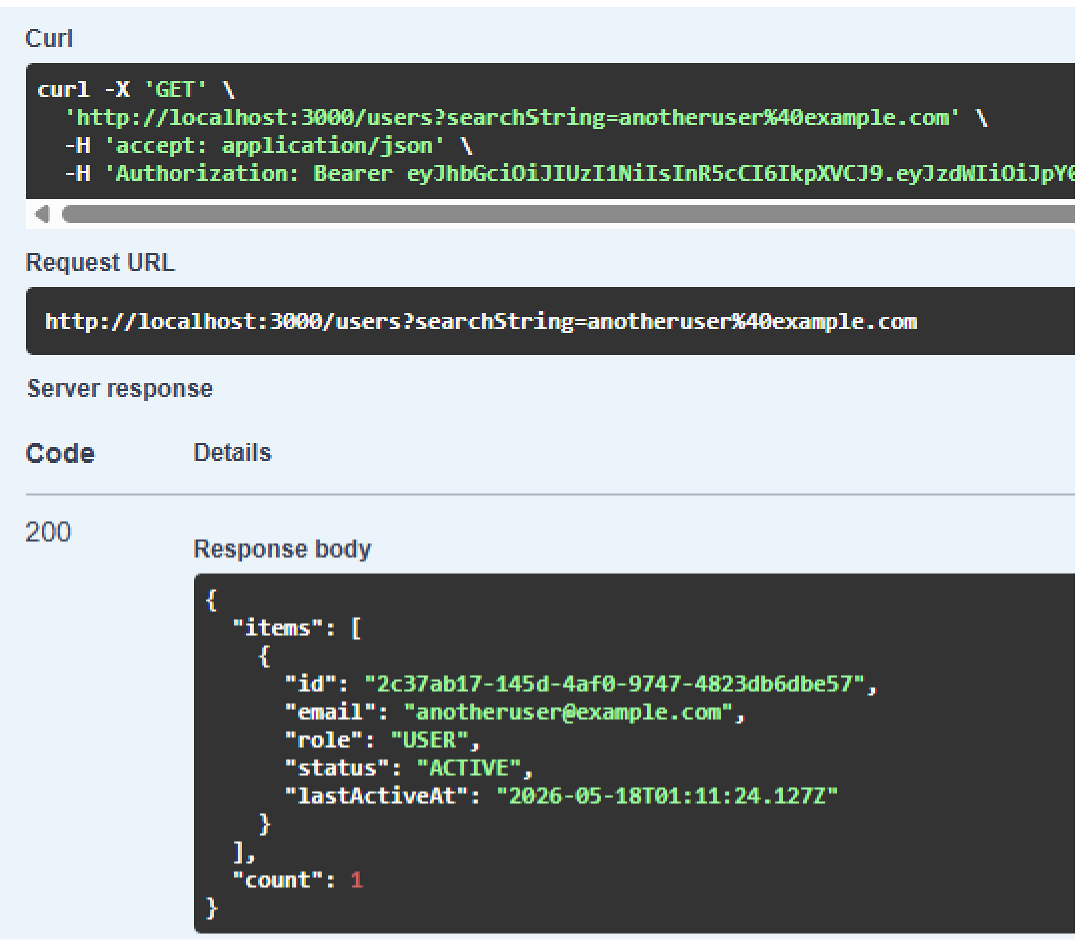


Рисунок 3.13 – Ідентифікація цільового користувача через системний реєстр для подальшого надання доступу

Безпосереднє надання прав реалізується через метод реляційного зв'язування об'єктів у таблиці прав доступу. Під час тестування було змодельовано ситуацію передачі прав на рівні «Редактор» (EDITOR) для конкретної папки. При виконанні запиту на поширення об'єкта NestJS API фіксує новий запис у сутності AccessGrant, створюючи стійкий логічний зв'язок між ідентифікатором користувача та файловою структурою власника, що виключає необхідність фізичного дублювання даних.

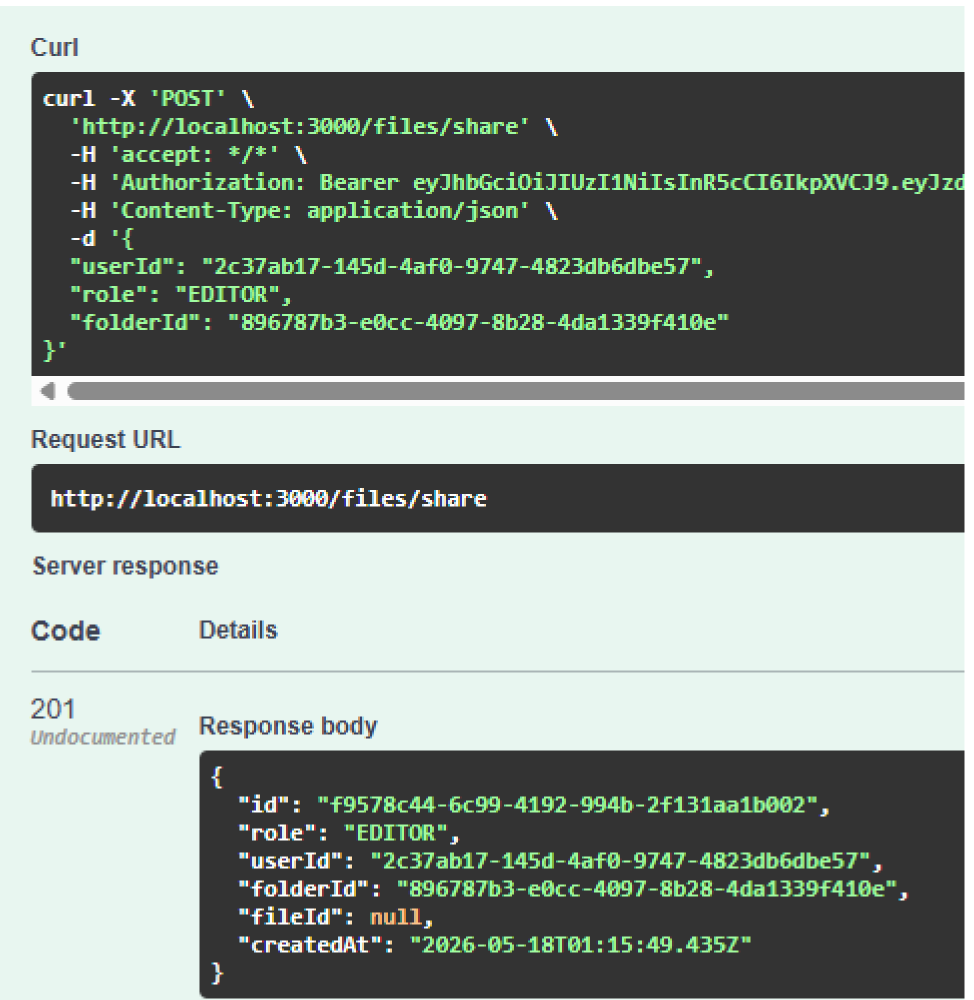


Рисунок 3.14 – Надання прав доступу із призначенням ролі «EDITOR»

Верифікація успішності проведеної операції здійснюється з боку отримувача доступу через звернення до спеціалізованого вузла перегляду спільних ресурсів. Тестування підтвердило, що після отримання відповідних

повноважень, цільовий об’єкт (папка або файл) коректно відображається у вибірці shared-контенту нового власника. Внутрішня логіка доменного шару при цьому автоматично фільтрує результати, базуючись на ідентифікаторі поточної сесії, що гарантує ізоляцію приватних даних та доступність лише тих вузлів сховища, на які було отримано явний дозвіл. Такий підхід забезпечує гнучкість управління цифровим периметром без компромісів у питанні інформаційної безпеки.

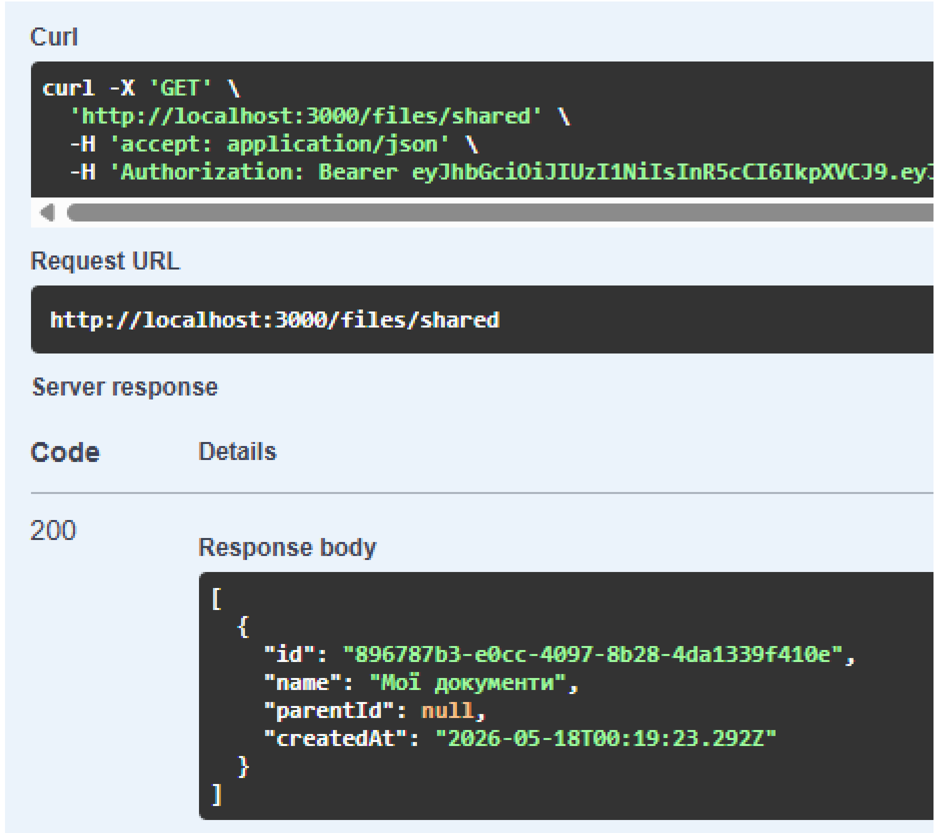


Рисунок 3.15 – Відображення переліку спільних об’єктів у інтерфейсі користувача-отримувача

Завершальний етап експериментальної перевірки присвячено верифікації життєвого циклу об’єктів та працездатності підсистеми подієво-орієнтованого аудиту. Для підтвердження коректності механізмів очищення дискового простору та цілісності даних у реляційній базі було проведено тестування операцій видалення на різних рівнях ієрархії. Сценарій передбачав створення

тимчасового тестового каталогу в межах існуючої структури з подальшим виконанням операції його повного видалення. Під час виконання цієї операції система забезпечує синхронне видалення записів у PostgreSQL та, у разі наявності вкладених об’єктів, ініціює запити до MinIO для фізичного видалення даних. Успішне завершення операції підтверджується стандартним HTTP-кодом відповіді, що свідчить про коректне спрацювання механізмів очищення та відсутність залишкових метаданих у системі.

Паралельно було проведено перевірку підсистеми Activity Logs, яка забезпечує прозорість дій користувачів та контроль доступу. Завдяки архітектурі на основі EventEmitter2 кожна дія користувача, зокрема створення та видалення об’єктів, автоматично формує подію, що фіксується в журналі аудиту. Для верифікації коректності логування було виконано запит на отримання історії дій для тестового каталогу, результати якого містили повну хронологію операцій із зазначенням ідентифікаторів користувачів, часових міток та метаданих у форматі JSONB. Це підтверджує реалізацію принципу простежуваності дій (auditability), що дозволяє здійснювати аналіз інцидентів безпеки та відстеження змін у файловій структурі системи.

Додатково під час тестування було оцінено поведінку системи в умовах паралельних запитів, що дозволило підтвердити коректність обробки конкурентних операцій над спільними ресурсами. Також було протестовано механізми відновлення після збоїв, зокрема коректність повторного підключення сесій та збереження цілісності транзакцій.

Окремо перевірено стійкість підсистеми зберігання до часткових відмов, включно з некоректними або перерваними операціями запису в об’єктне сховище. Результати експериментів підтвердили, що система коректно завершує або відкочує незавершені операції без втрати даних. Таким чином, комплексне тестування засвідчило стабільність роботи основних підсистем та їх узгоджену взаємодію в межах загальної архітектури.

ВИСНОВКИ

У роботі за результатами теоретичних і практичних досліджень спроєктовано та реалізовано програмно-апаратну систему приватного файлового сховища, що базується на технологіях об'єктного зберігання даних та використовує апаратний токен на базі ESP32 для забезпечення багатофакторної автентифікації.

У першому розділі проведено аналіз предметної області автономного мережевого зберігання даних та існуючих архітектурних рішень. Обґрунтовано переваги об'єктної моделі S3 (MinIO) над традиційними файловими системами, а також необхідність використання фізичних засобів захисту на основі аналізу вразливостей парольної автентифікації. Сформульовано технічні вимоги до системи, зокрема впровадження протоколу Challenge–Response із використанням HMAC-SHA256 для захисту від атак повторного відтворення.

У другому розділі виконано проєктування архітектури та моделей взаємодії компонентів розподіленої системи. Запропоновано багаторівневу серверну архітектуру, що включає API-оркестратор, базу метаданих, об'єктне сховище та in-memory кеш. Спроєктовано алгоритмічне та схемотехнічне забезпечення мікроконтролера ESP32 у вигляді скінченного автомата. Розроблено асинхронний протокол взаємодії між веб-клієнтом і пристроєм через Redis, що забезпечує ізоляцію криптографічних секретів та фізичне підтвердження дій користувача.

У третьому розділі виконано практичну реалізацію та тестування системи. Розроблено бекенд частину для керування даними і правами доступу. Реалізовано прошивку для ESP32, що виконує апаратне обчислення цифрових підписів із використанням вбудованого криптомодуля. Впроваджено двоетапний механізм завантаження файлів. Тестування із застосуванням Swagger UI та Postman підтвердило стабільність HTTP-обробки, коректність роботи файлових операцій та стійкість підсистеми апаратної автентифікації.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Білоконь А. С., Борисов С. О., Усатенко М. В., Федорченко В. М. Аналіз функціонування розподілених систем обробки та зберігання даних. Системи управління, навігації та зв'язку. 2024. Вип. 3 (77). С. 84–88.

2. Mell P., Grance T. The NIST Definition of Cloud Computing. *NIST Special Publication*. 2011. No. 800-145. 7 p.

3. Рибальченко О. Людський фактор у системі забезпечення безпеки корпоративних баз даних: соціально-поведінковий аналіз ризиків. *Кібербезпека: освіта, наука, техніка*. 2026. Т. 4, № 32. С. 747–756.

4. Myśliwiec K. NestJS: A Progressive Node.js Framework. URL: https://docs.nestjs.com. (дата звернення: 10.02.2026).

5. Containers and Orchestration. Docker Documentation. URL: https://docs.docker.com. (дата звернення: 12.02.2026).

6. Жарчинський С. М. Система зберігання даних на основі OpenStack Object Storage : кваліфікаційна робота магістра : 123 Комп'ютерна інженерія. С. М. Жарчинський ; Хмельниц. нац. ун-т. – Хмельницький, 2025 – 105 с.

7. Bernstein D. Containers and Cloud: From LXC to Docker to Kubernetes. IEEE Cloud Computing. 2014. Vol. 1, No 3. P. 81–84. DOI: https://doi.org/10.1109/MCC.2014.51. (дата звернення: 12.02.2026).

8. Батаєв С. В., Федорчук А. Л., Багнюк О. М. Zero Trust у вебархітектурі: нові підходи до безпеки в умовах розподілених систем. Наука і техніка сьогодні. 2025. № 52. С. 1844-1863.

9. Shackelford A. Beginning Amazon Web Services with Node.js. New York : Apress, 2015. 280 p.

10. Erasure coding. MinIO Blog. 2023. https://www.min.io/blog/erasure-coding. (дата звернення: 28.05.2026).

11. Freeman A. Pro Angular. Berkeley, CA : Apress, 2022. https://doi.org/10.1007/978-1-4842-8176-5 (дата звернення: 19.02.2026).

12. Золотарьов В., Скворцов В. Порівняння монолітної та мікросервісної архітектури у розробці вебзастосунків. Радіоелектроніка та молодь у XXI столітті. Т. 4 : Конференція "Перспективи розвитку інфокомунікацій та інформаційно-вимірювальних технологій". Харків, 2024. https://doi.org/10.30837/iyf.pdicimt.2024.152.(дата звернення: 27.02.2026).

13. Dubey P., Kumar Tiwari A., Raja R. Amazon Simple Storage Service. *Amazon Web Services: the Definitive Guide for Beginners and Advanced Users*. 2023. P. 97–114. https://doi.org/10.2174/9789815165821123010007 (дата звернення: 27.05.2026).

14. Персунов Д. Технологія контролю та відновлення цілісності даних у розподілених сховищах. Information Technology and Society. 2025. № 4 (19). С. 140-143. https://doi.org/10.32689/maup.it.2025.4.22. (дата звернення: 15.02.2026).

15. Chung Y. T., Nakamura T. Optimizing Object Storage Performance for Large File Uploading under Small-start Environments. *Journal of Information Processing*. 2026. Vol. 34. P. 86–94. https://doi.org/10.2197/ipsjjip.34.86 (дата звернення: 13.02.2026).

16. LogCIoud: Fast Search of Compressed Logs on Object Storage / Z. Wang et al. *Proceedings of the VLDB Endowment*. 2025. Vol. 18, no. 8. P. 2362–2370. https://doi.org/10.14778/3742728.3742733 (дата звернення: 17.02.2026).

17. Data Modeling with TypeScript. Prisma ORM Documentation. URL: https://www.prisma.io/docs (дата звернення: 16.02.2026).

18. Григоренко К. С., Кузєв І. О. Класифікація інформаційних систем. Матеріали XXXII Міжнародної науково-практичної конференції студентів, аспірантів та молодих учених «Актуальні проблеми життєдіяльності суспільства»,. 2025. С. 135–136. https://doi.org/10.32782/2079-5009.krnu25.5.6 (дата звернення: 13.02.2026).

19. Сіренко О. Дослідження моделей та засобів збереження та обробки конфиденційної інформації у хмарі. *Theoretical and practical aspects of modern*

scientific research. 2021. https://doi.org/10.36074/logos-30.04.2021.v1.57 (дата звернення: 27.02.2026).

20. Taibi D., Lenarduzzi V., Pahl C. Microservices Anti-patterns: A Taxonomy. *Microservices*. Cham, 2019. P. 111–128. https://doi.org/10.1007/978-3-030-31646-4_5 (date of access: 22.02.2026).

21. Краліна Г., Баков Н. Проблеми якості програмного забезпечення. *Débats scientifiques et orientations prospectives du développement scientifique*. 2021. https://doi.org/10.36074/logos-05.02.2021.v3.29 (дата звернення: 12.02.2026).

22. Kosheliuk V., Koren V., Tymchuk V. Архітектурні моделі мультитенантності в сучасних SaaS-системах: порівняльний аналіз та методологія вибору. *Computer-integrated technologies: education, science, production*. 2025. № 61. С. 98–103. https://doi.org/10.36910/6775-2524-0560-2025-61-14 (дата звернення: 13.02.2026).

23. Smith H. Reliability and Server Failover. *Data Center Storage*. 2016. P. 161–172. https://doi.org/10.1201/b10798-19 (дата звернення: 17.02.2026).

24. Merkel D. Docker: lightweight Linux containers for consistent development and deployment. Linux Journal. 2014. No 239.

25. Tahaei M., Vaniea K. A survey on developer-centred security. 2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). 2019. P. 129-138.

26. Основи управління іт-проектами: принципи, методи та інструменти. С. Паращук та ін. *Наука і техніка сьогодні*. 2025. № 5(46). https://doi.org/10.52058/2786-6025-2025-5(46)-1925-1933 (дата звернення: 27.05.2026).

27. PostgreSQL 16 Documentation. PostgreSQL Global Development Group. URL: https://www.postgresql.org/docs (дата звернення: 11.02.2026).

28. Ferraiolo D. F., Kuhn D. R., Chandramouli R. Role-Based Access Control. Artech House Publishers, 2003. 338 p.

29. Borovskova Y. A. Determining optimal caching strategies for improving performance of server applications on NestJS. Scientific notes of Taurida National V.I.

Vernadsky University. Series: Technical Sciences. 2024. Vol. 2, no. 6. P. 18-26. https://doi.org/10.32782/2663-5941/2024.6.2/03. (дата звернення: 11.03.2026).

30. Buuck B. Object Storage Optimized Databases: Trends & Industry Leaders. MinIO Blog. 2025. https://www.min.io/blog/object-storage-optimized-databases-trends-industry-leaders. (дата звернення: 28.05.2026).

31. Almeida D. et al. Performance comparison of redis, memcached, mysql, and postgresql: A study on key-value and relational databases. 2023 Second International Conference On Smart Technologies For Smart Nation (SmartTechCon). 2023. P. 902-907.

32. Ткаченко Т. А., Мартовицький В. О., Бологова Н. М. Методи оптимізації запитів у розподілених базах даних. Вісник Херсонського національного технічного університету. 2025. Т. 3, № 4 (95). С. 245-251. https://doi.org/10.35546/kntu2078-4481.2025.4.3.28. (дата звернення: 24.02.2026).

33. Thakkar J. B. Designing an In-Memory Metadata Cache to Accelerate Object Storage Operations : master’s thesis. Arizona State University, 2025. 78 p.

34. Gowda P., Gowda A. N. Best Practices in REST API Design for Enhanced Scalability and Security. *Journal of Artificial Intelligence, Machine Learning and Data Science*. 2024. Vol. 2, no. 1. P. 827-830. https://doi.org/10.51219/jaimld/priyanka-gowda/202. (дата звернення: 27.05.2026).

35. Goodfellow I. Deep learning of representations and its application to computer vision : thèse. 2014. http://hdl.handle.net/1866/11674. (дата звернення: 18.03.2026).

36. Трофименко О. Г. та ін. Штучний інтелект у системному аналізі. Таврійський науковий вісник. Серія: Технічні науки. 2024. № 5. С. 85-97. https://doi.org/10.32782/tnv-tech.2024.5.9. (дата звернення: 05.03.2026).

37. Wilhelm T., Engebretson P. OWASP Top 10. The Basics of Hacking and Penetration Testing. 2026. P. 65-110. https://doi.org/10.1016/b978-0-443-43886-8.00003-5. (дата звернення: 03.03.2026).

38. Коробейнікова Т., Кравчук Н. Організація захищеного доступу до web-серверів засобами машинного навчання. SWorldJournal. 2023. № 20-01. С. 52-59. https://doi.org/10.30888/2663-5712.2023-20-01-041. (дата звернення: 17.02.2026).

39. Peak P. P. Key Control Log Book - Key Inventory Checkout System Register: Sign in/Out for Business and Personal Use. Independently Published, 2022. 120 p.

40. Shettar G., Sataraddi R., Dalawai Y. S. Edge Computing Challenges. *International Journal of Research in Engineering, Science and Management*. 2024. Vol. 7, no. 7. P. 92-95. https://doi.org/10.5281/zenodo.13908920. (дата звернення: 26.02.2026).

41. Нежуренко О. Г. Оптимізація зберігання даних IoT-пристроїв у розподілених базах даних: виклики масштабування та ефективності. Наука і техніка сьогодні. 2025. № 48. С. 1739-1753.

42. Anjos J. C. S. D. et al. A survey on collaborative learning for intelligent autonomous systems. ACM Computing Surveys. 2023. Vol. 56, no. 4. P. 1-37.

43. Reis D. et al. Developing docker and docker-compose specifications: A developers' survey. *IEEE Access*. 2021. Vol. 10. P. 2318-2329.

44. Рязанцев О. І. та ін. Віртуалізація та контейнерізація як засоби організації обчислювальних середовищ. Вісник Східноукраїнського національного університету імені Володимира Даля. 2026. № 1 (299). С. 16-27.

45. Svendsrud D. S., Smith P., Hydle K. Network orchestration: managing the scaling of platform-based ecosystems. 2023. 45 p.

46. Felani R. et al. Optimizing virtual resources management using Docker on cloud applications. Indonesian Journal of Computing and Cybernetics Systems. 2020. Vol. 14, no. 3. P. 227-238.

47. Олійник П., Мартинюк В. Удосконалений метод роботи з метриками покриття коду для забезпечення ефективного оцінювання результатів тестування програмного забезпечення. Вимірювальна та обчислювальна техніка в технологічних процесах. 2023. № 3. С. 138-143.

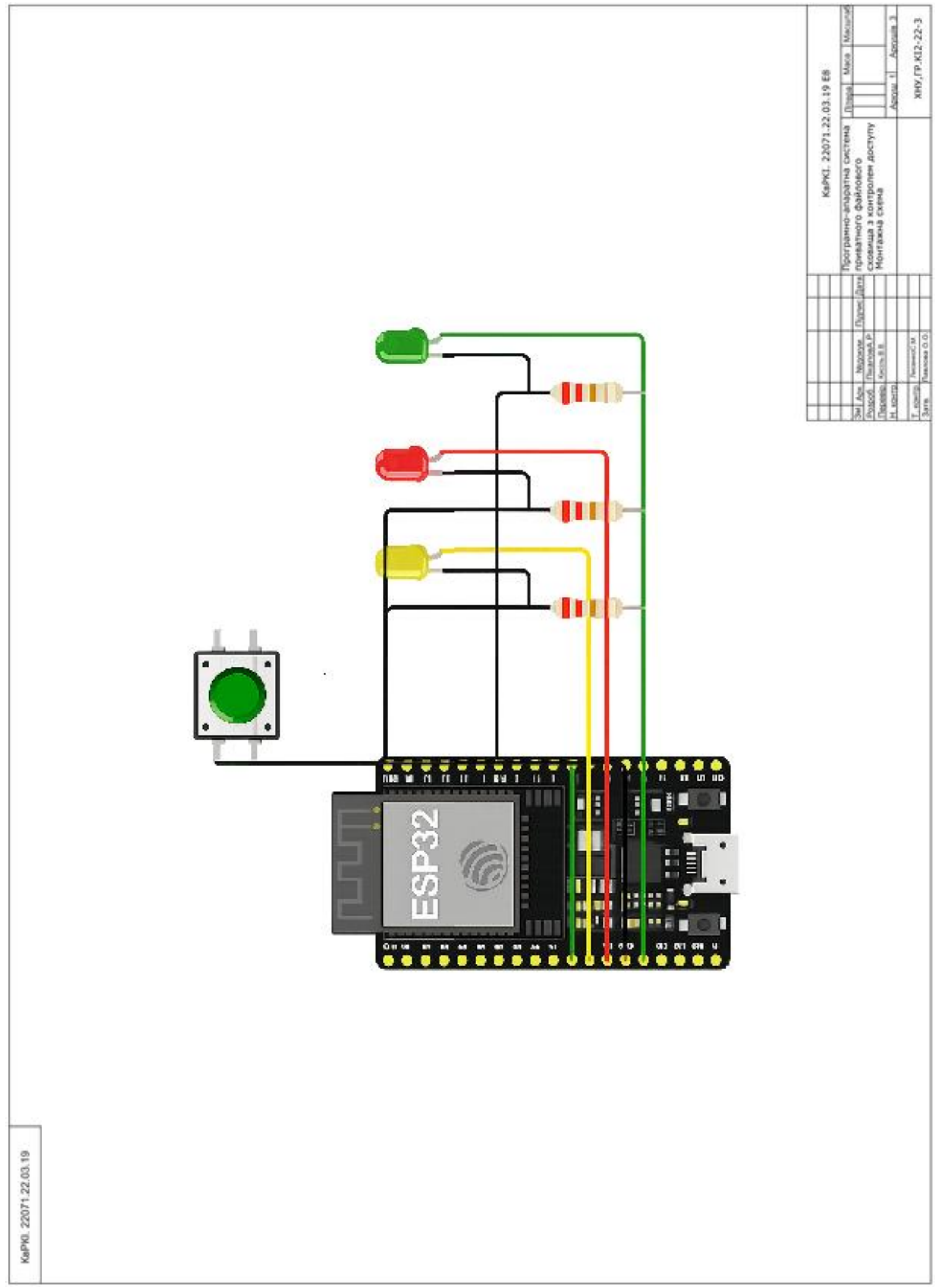
48. Leahy D., Thorpe C. Zero Trust Container Architecture (ZTCA): A Framework for Applying Zero Trust Principals to Docker Containers. *International Conference on Cyber Warfare and Security*. 2022. Vol. 17. P. 111-120. https://doi.org/10.34190/iccws.17.1.35. (дата звернення: 11.03.2026).

49. Despoudis T. TypeScript 5 Design Patterns and Best Practices: Build clean and scalable apps with proven patterns and expert insights. Packt Publishing Ltd, 2025. 380 p.

ДОДАТОК А

(обов'язковий)

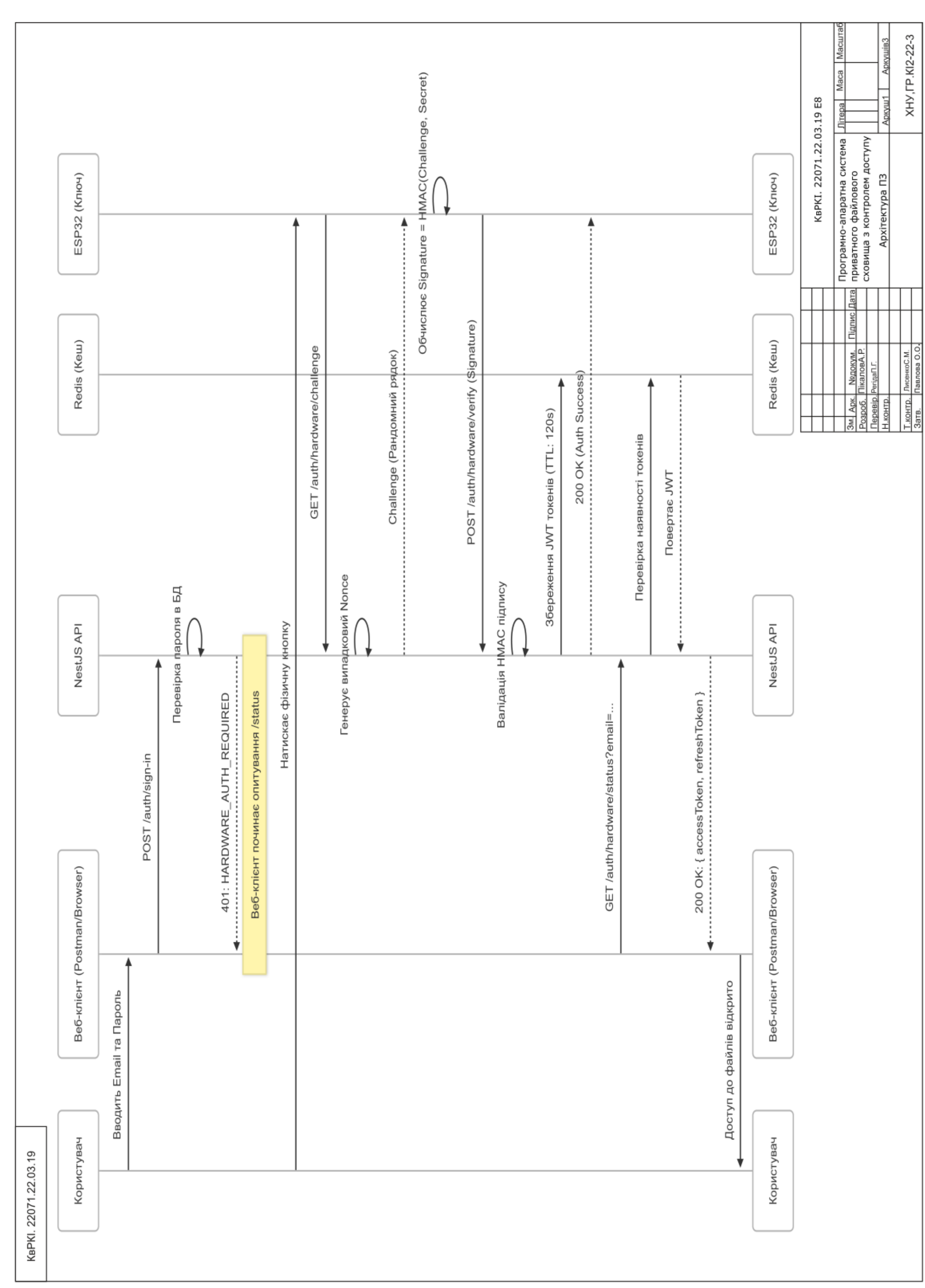
Копія креслення «Монтажна схема»



ДОДАТОК В

(обов’язковий)

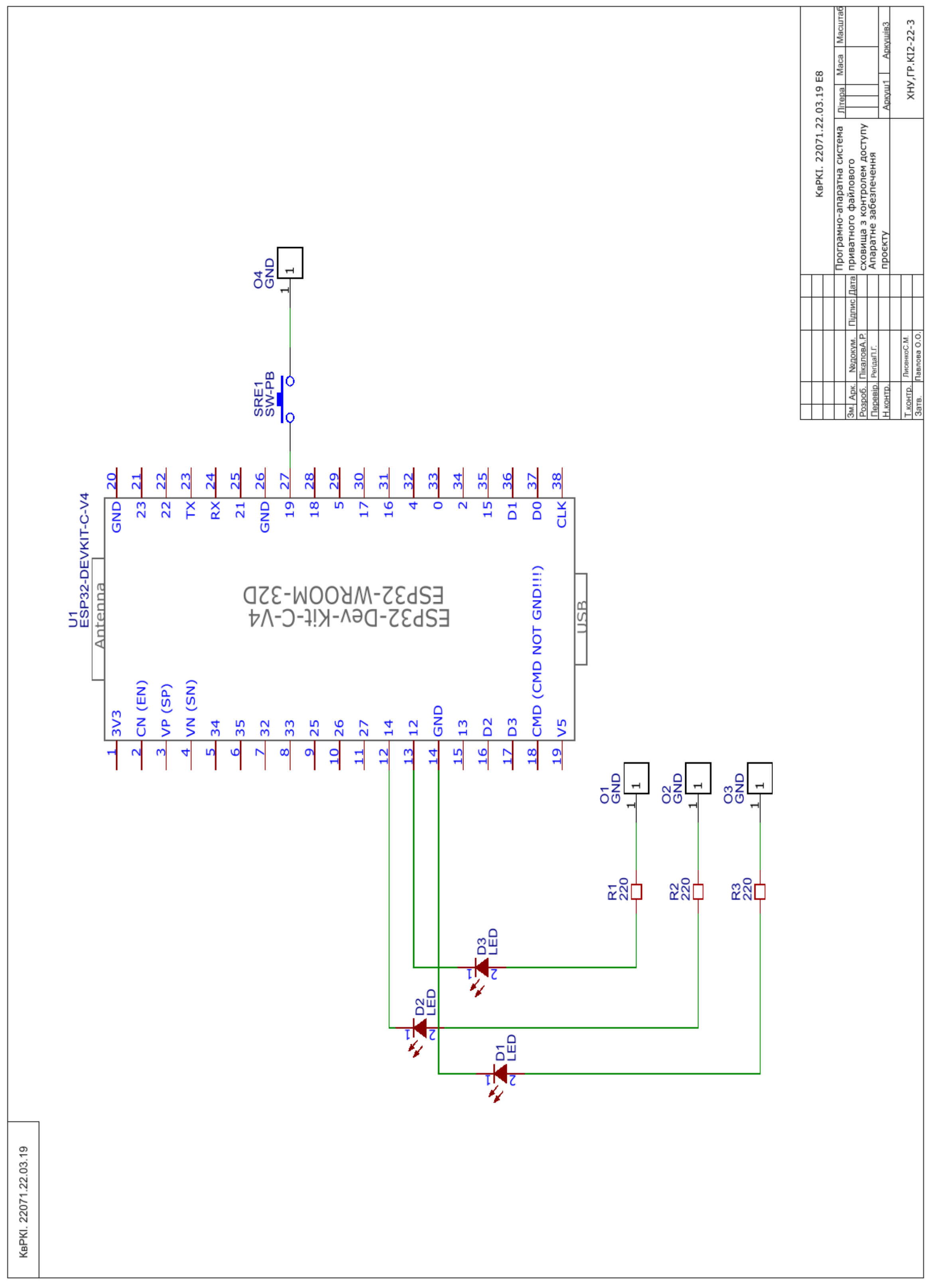
Копія креслення «Схема багатофакторної автентифікації»



ДОДАТОК Б

(обов'язковий)

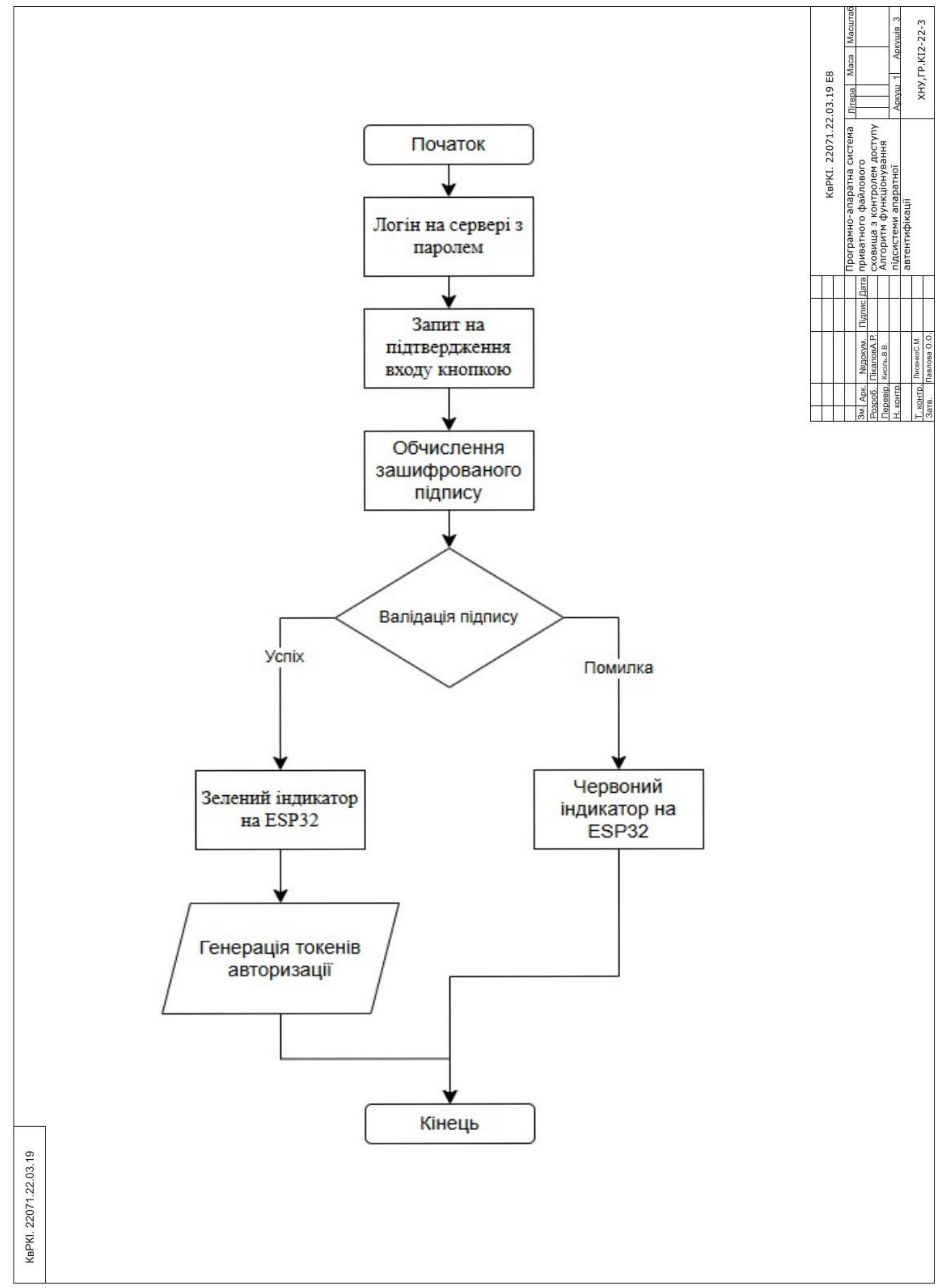
Копія креслення «Апаратне забезпечення проєкту»



ДОДАТОК Г

(обов’язковий)

Копія креслення «Алгоритм функціонування підсистеми апаратної автентифікації»



ДОДАТОК Д

(обов’язковий)

Код прошивки ESP-32

```
#include <Arduino.h>
#include <WiFi.h>
#include <HTTPClient.h>
#include <ArduinoJson.h>
#include <mbedtls/md.h>

const char* WIFI_SSID = "Wokwi-GUEST";
const char* WIFI_PASS = "";
const String SERVER_URL =
"http://192.168.0.101:3000/auth/hardware";

const String DEVICE_ID = "ESP32_KEY_001";
const char* SECRET_KEY = "super_secret_shared_key_123";

const int BUTTON_PIN = 27;
const int LED_YELLOW = 14;
const int LED_RED = 12;
const int LED_GREEN = 13;

void setLeds(int yellow, int red, int green) {
  digitalWrite(LED_YELLOW, yellow);
  digitalWrite(LED_RED, red);
  digitalWrite(LED_GREEN, green);
}

String computeHMAC(String payload, const char* key) {
  byte hmacResult[32];
  mbedtls_md_context_t ctx;
  mbedtls_md_type_t md_type = MBEDTLS_MD_SHA256;
```

```
    mbedtls_md_init(&ctx);
    mbedtls_md_setup(&ctx, mbedtls_md_info_from_type(md_type), 1);
    mbedtls_md_hmac_starts(&ctx, (const unsigned char*)key,
strlen(key));
    mbedtls_md_hmac_update(&ctx, (const unsigned
char*)payload.c_str(), payload.length());
    mbedtls_md_hmac_finish(&ctx, hmacResult);
    mbedtls_md_free(&ctx);

    String hash = "";
    for (int i = 0; i < 32; i++) {
      char hex[3];
      sprintf(hex, "%02x", hmacResult[i]);
      hash += hex;
    }
    return hash;
  }

  void processAuthentication() {
    if (WiFi.status() != WL_CONNECTED) return;

    setLeds(HIGH, LOW, LOW); // yellow
    HTTPClient http;

    Serial.println("Requesting challenge...");
    http.begin(SERVER_URL + "/challenge?deviceId=" + DEVICE_ID);
    int httpCode = http.GET();

    if (httpCode == 200) {
      String payload = http.getString();
      JsonDocument doc;
      deserializeJson(doc, payload);
      String challenge = doc["challenge"];

      Serial.println("Challenge received: " + challenge);
```

```
      String signature = computeHMAC(challenge, SECRET_KEY);
      Serial.println("Signature generated: " + signature);

      http.begin(SERVER_URL + "/verify");
      http.addHeader("Content-Type", "application/json");

      JsonDocument outDoc;
      outDoc["deviceId"] = DEVICE_ID;
      outDoc["signature"] = signature;
      outDoc["challenge"] = challenge;

      String requestBody;
      serializeJson(outDoc, requestBody);

      int verifyCode = http.POST(requestBody);

      if (verifyCode == 200 || verifyCode == 201) {
        Serial.println("AUTH SUCCESS!");
        setLeds(LOW, LOW, HIGH); // green
        delay(3000);
      } else {
        Serial.println("AUTH FAILED: " + String(verifyCode));
        setLeds(LOW, HIGH, LOW); // red
        delay(2000);
      }
    } else {
      Serial.println("Error     getting     challenge:     "     +
String(httpCode));
      setLeds(LOW, HIGH, LOW);
      delay(2000);
    }

    http.end();
    setLeds(LOW, HIGH, LOW);
```

```
}


void setup() {
  Serial.begin(115200);
  pinMode(BUTTON_PIN, INPUT_PULLUP);
  pinMode(LED_YELLOW, OUTPUT);
  pinMode(LED_RED, OUTPUT);
  pinMode(LED_GREEN, OUTPUT);

  setLeds(LOW, HIGH, LOW); // Очікування

  WiFi.begin(WIFI_SSID, WIFI_PASS, 6);
  while (WiFi.status() != WL_CONNECTED) {
    delay(250);
    digitalWrite(LED_YELLOW, !digitalRead(LED_YELLOW));
  }
  Serial.println("\nWiFi Connected. IP: " + WiFi.localIP().toString());
  setLeds(LOW, HIGH, LOW);
}

void loop() {
  if (digitalRead(BUTTON_PIN) == LOW) {
    Serial.println("\nButton pressed. Starting Auth Flow...");
    processAuthentication();
  }
}
```

ДОДАТОК Е

(обов'язковий)

Дерево файлів бекенд частини

```
┌── app.module.ts
├── common
│   ├── abstract
│   │   ├── base-dto.abstract.ts
│   │   ├── index.ts
│   │   └── seeder.abstract.ts
│   ├── decorators
│   │   ├── api-implicit-pagination.decorator.ts
│   │   ├── auth-guard.decorator.ts
│   │   ├── dto-property.decorator.ts
│   │   ├── index.ts
│   │   ├── req-pagination.decorator.ts
│   │   ├── req-role.decorator.ts
│   │   ├── req-session.decorator.ts
│   │   ├── req-user.decorator.ts
│   │   ├── roles-guard.decorator.ts
│   │   └── transform-string-to-boolean.decorator.ts
│   ├── enums
│   │   ├── events.enum.ts
│   │   ├── exception-keys.enum.ts
│   │   └── index.ts
│   ├── exceptions
│   │   ├── confirmation-code.exception.ts
│   │   ├── domain.exception.ts
│   │   ├── dto
│   │   │   ├── exception.dto.ts
│   │   │   └── index.ts
│   │   ├── index.ts
│   │   ├── invalid-credentials.exception.ts
│   │   ├── not-found.exception.ts
```

```
│   │   ├── user-already-exists.exception.ts
│   │   ├── wrong-otp.exception.ts
│   │   └── wrong-refresh-token.exception.ts
│   ├── filters
│   │   ├── domain-exception.filter.ts
│   │   └── index.ts
│   ├── guards
│   │   ├── index.ts
│   │   └── roles.guard.ts
│   ├── index.ts
│   ├── interfaces
│   │   ├── domain-exception.interface.ts
│   │   ├── index.ts
│   │   ├── pagination-result.interface.ts
│   │   └── pagination.interface.ts
│   └── validation
│       ├── config-validation-schema.ts
│       └── index.ts
├── domain
│   ├── files
│   │   ├── files.module.ts
│   │   ├── index.ts
│   │   ├── services
│   │   │   ├── files.service.ts
│   │   │   └── index.ts
│   │   └── typing
│   │       ├── consts
│   │       │   └── index.ts
│   │       ├── index.ts
│   │       └── interfaces
│   │           ├── files-service.interface.ts
│   │           └── index.ts
│   ├── hardware
│   │   ├── hardware.module.ts
│   │   ├── index.ts
```

```
│   │   ├── services
│   │   │   ├── hardware.service.ts
│   │   │   └── index.ts
│   │   └── typing
│   │       ├── consts
│   │       │   └── index.ts
│   │       ├── index.ts
│   │       └── interfaces
│   │           ├── hardware-service.interface.ts
│   │           └── index.ts
│   ├── index.ts
│   ├── logs
│   │   ├── index.ts
│   │   ├── listeners
│   │   │   ├── index.ts
│   │   │   └── logs.listener.ts
│   │   ├── logs.module.ts
│   │   ├── services
│   │   │   ├── index.ts
│   │   │   └── logs.service.ts
│   │   └── typing
│   │       ├── consts
│   │       │   └── index.ts
│   │       ├── index.ts
│   │       └── interfaces
│   │           ├── index.ts
│   │           └── logs-service.interface.ts
│   ├── sessions
│   │   ├── guards
│   │   │   ├── auth.guard.ts
│   │   │   └── index.ts
│   │   ├── index.ts
│   │   ├── sessions.module.ts
│   │   ├── sessions.service.ts
│   │   └── typing
```

```
│   │       ├── consts
│   │       │   └── index.ts
│   │       ├── index.ts
│   │       └── interfaces
│   │           ├── index.ts
│   │           ├── sessions-service.interface.ts
│   │           └── token-pair.interface.ts
│   └── users
│       ├── index.ts
│       ├── services
│       │   ├── index.ts
│       │   ├── users-passwords.service.ts
│       │   └── users.service.ts
│       ├── typing
│       │   ├── consts
│       │   │   └── index.ts
│       │   ├── index.ts
│       │   └── interfaces
│       │       ├── index.ts
│       │       └── users-service.interface.ts
│       └── users.module.ts
├── libs
│   ├── cache
│   │   ├── cache.module.ts
│   │   ├── cache.service.ts
│   │   └── index.ts
│   ├── confirmation
│   │   ├── confirmation-codes.service.ts
│   │   ├── confirmation.module.ts
│   │   ├── index.ts
│   │   └── typing
│   │       ├── consts
│   │       │   └── index.ts
│   │       ├── index.ts
│   │       └── interfaces
```

```
│   │           ├── confirmation-service.interface.ts
│   │           └── index.ts
│   ├── database
│   │   ├── database.module.ts
│   │   ├── database.service.ts
│   │   └── index.ts
│   ├── file-storage
│   │   ├── file-storage.module.ts
│   │   ├── file-storage.service.ts
│   │   ├── index.ts
│   │   └── typing
│   │       ├── index.ts
│   │       └── interfaces
│   │           ├── index.ts
│   │           └── input-file.interface.ts
│   ├── file-url-builder
│   │   ├── file-url-builder.module.ts
│   │   ├── file-url-builder.service.ts
│   │   ├── helpers
│   │   │   ├── file-url.helper.ts
│   │   │   └── index.ts
│   │   └── index.ts
│   ├── index.ts
│   ├── jwt
│   │   ├── index.ts
│   │   ├── interfaces
│   │   │   ├── create-token-payload.interface.ts
│   │   │   └── index.ts
│   │   ├── jwt.module.ts
│   │   └── jwt.service.ts
│   └── mailer
│       ├── helpers
│       │   ├── email-template.helper.ts
│       │   └── index.ts
│       ├── index.ts
```

```
│       ├── mailer.module.ts
│       ├── services
│       │   ├── index.ts
│       │   ├── mailer.service.ts
│       │   └── telegram-mailer.service.ts
│       └── typing
│           ├── consts
│           │   └── index.ts
│           ├── index.ts
│           └── interfaces
│               ├── index.ts
│               └── mailer-service.interface.ts
├── main.ts
├── rest
│   ├── auth
│   │   ├── auth.controller.ts
│   │   ├── auth.module.ts
│   │   ├── dto
│   │   │   ├── check-email.dto.ts
│   │   │   ├── hardware-auth.dto.ts
│   │   │   ├── index.ts
│   │   │   ├── restore-password.dto.ts
│   │   │   ├── sign-in.dto.ts
│   │   │   └── sign-up.dto.ts
│   │   ├── index.ts
│   │   └── services
│   │       ├── auth.service.ts
│   │       ├── index.ts
│   │       └── registration.service.ts
│   ├── files
│   │   ├── dto
│   │   │   ├── create-folder.dto.ts
│   │   │   ├── file-view.dto.ts
│   │   │   ├── finish-upload.dto.ts
│   │   │   ├── folder-view.dto.ts
```

```
│   │   │   ├── get-upload-link.dto.ts
│   │   │   ├── index.ts
│   │   │   └── share-object.dto.ts
│   │   ├── files.controller.ts
│   │   ├── files.module.ts
│   │   ├── files.service.ts
│   │   └── index.ts
│   ├── index.ts
│   ├── logs
│   │   ├── dto
│   │   │   ├── get-logs-params.dto.ts
│   │   │   ├── index.ts
│   │   │   ├── log-view.dto.ts
│   │   │   └── logs-list.dto.ts
│   │   ├── index.ts
│   │   ├── logs.controller.ts
│   │   ├── logs.module.ts
│   │   └── logs.service.ts
│   └── users
│       ├── dto
│       │   ├── get-users-list-params.dto.ts
│       │   ├── index.ts
│       │   ├── user-view.dto.ts
│       │   └── users-list.dto.ts
│       ├── index.ts
│       ├── users.controller.ts
│       ├── users.module.ts
│       └── users.service.ts
└── shared
    ├── dto
    │   ├── index.ts
    │   ├── logout.dto.ts
    │   ├── refresh-token.dto.ts
    │   └── token-pair.dto.ts
    ├── helpers
```

```
│    ├── env.helper.ts
│    ├── index.ts
│    ├── pagination.helpers.ts
│    ├── provider.helper.ts
│    ├── string-to-boolean.helper.ts
│    └── tokens.helper.ts
├── index.ts
└── regex
└── index.ts
```

Thu May 28 13:57:11 EEST 2026, Медзатий Дмитро Миколайович, Хмельницький національний університет, ХНУ

Anti-Plagiarism (http://ap.km.ua) v-15.701

Максимальне співпадіння з одним документом 2.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 15%**

ID: 272628 Назва: БКР Програмно-апаратна система приватного файлового сховища з контролем доступу Додано в БД: 2026-05-28 Автора: Андрій ПІКАЛОВ Керівники: Володимир КИСІЛЬ Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	107053	793	3958 (4%)	50 (6%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

Кисіль

Протокол аналізу звіту подібності експертом

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Андрій ПІКАЛОВ

Співавтор:

Назва: Програмно-апаратна система приватного файлового сховища з контролем доступу

Експерт: Володимир КИСІЛЬ

Підрозділ: Кафедра комп’ютерної інженерії та інформаційних систем

Коефіцієнт подібності 1:2.74%

Коефіцієнт подібності 2:0.79%

Мікропробіли: 27

Заміна букв: 2

Інтервали: 0

Білі знаки: 0

Дата створення звіту: 2026-05-28 16:09:04.0

Після аналізу Звіту подібності констатую наступне:

☒ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

2026-05-28 Доцент Андрій Нічепорук

Дата експерт

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Дипломник: Пікалов Андрій Русланович

Тема: Програмно-апаратна система приватного файлового сховища з контролем доступу

Спеціальність: 123 «Комп'ютерна інженерія»

Обсяг кваліфікаційної роботи:

Кількість листів креслень 4 Кількість сторінок записки 64

1. Короткий зміст роботи та прийнятих рішень: Метою кваліфікаційної роботи є синтез та моделювання операційного автомату на основі автомату Мура

2. Висновок про відповідність роботи дипломному завданню: Робота повністю відповідає поставленому завданню.

3. Характеристика виконання кожного розділу, ступінь використання останніх досягнень науки і техніки і передових методів роботи: У першому розділі проведено аналіз предметної області автономного мережевого зберігання даних та існуючих архітектурних рішень. Обґрунтовано переваги об'єктної моделі S3 (MinIO) над традиційними файловими системами, а також необхідність використання фізичних засобів захисту на основі аналізу вразливостей парольної автентифікації. Сформульовано технічні вимоги до системи, зокрема впровадження протоколу Challenge–Response із використанням HMAC-SHA256 для захисту від атак повторного відтворення.

У другому розділі виконано проєктування архітектури та моделей взаємодії компонентів розподіленої системи. Запропоновано багаторівневу серверну архітектуру, що включає API-оркестратор, базу метаданих, об'єктне сховище та in-memory кеш. Спроєктовано архітектуру та алгоритм роботи апаратного модуля на базі ESP32. Розроблено асинхронний протокол взаємодії між веб-клієнтом і пристроєм через Redis, що забезпечує ізоляцію криптографічних секретів та фізичне підтвердження дій користувача.

У третьому розділі виконано практичну реалізацію та тестування системи. Розроблено бекенд частину для керування даними і правами доступу. Реалізовано прошивку для ESP32, що виконує апаратне обчислення цифрових підписів із використанням вбудованого криптомодуля. Впроваджено двоетапний механізм завантаження файлів. Тестування із застосуванням Swagger UI та Postman підтвердило стабільність HTTP-обробки, коректність роботи файлових операцій та стійкість підсистеми апаратної автентифікації.

4. Позитивні сторони роботи: висока практична цінність роботи.

5. Негативні сторони роботи:

6. Оцінка графічного оформлення та пояснювальної записки роботи: Пояснювальна записка оформлена коректно, згідно діючих стандартів оформлення документації.

7. Відгук про роботу в цілому: Робота виконана на належному науково-технічному рівні.

8. Інші зауваження: ______________________________

9. Оцінка дипломної роботи: відмінно (A / 90)

Рецензент (прізвище, ім'я, по батькові, посада, місце роботи) ____________

Чешун Віктор Миколайович

канд. техн. наук, доцент, доцент каф. кібербезпеки

"29" 05 2026 р. ____________ (підпис)

Зав. кафедри КІС
д-р. філософії Ользі ПАВЛОВІЙ

Андрій ПІКАЛОВ

ПІБ здобувача вищої освіти

ФІТ, 4 курсу, групи КІ2-22-3

ЗАЯВА

З правилами чинного Положення про систему забезпечення академічної доброчесності у Хмельницькому національному університеті, згідно з яким виявлення академічного плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів академічної відповідальності, ознайомлений (а). Про використання спеціалізованих програмних засобів (СПЗ) StrikePlagiarism та Anti-Plagiarism для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність академічного плагіату оповіщений (а). Надаю університету право на передачу моєї роботи для обробки та збереження в базах даних СПЗ і використання роботи для виявлення академічного плагіату в інших роботах, які перевіряються СПЗ.

Також надаю свою згоду на обробку й збереження університетом моєї роботи в Інституційному репозитарії Хмельницького національного університету.

Робота надається для перевірки в електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

1 травня 2026 року

РІШЕННЯ ЕКСПЕРТНОЇ КОМІСІЇ

КАФЕДРИ КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Назва кваліфікаційної роботи Програмно-апаратна система приватного файлового сховища з контролем доступу
Автор Андрій ПІКАЛОВ
Освітня програма Комп'ютерна інженерія та програмування
Рівень вищої освіти перший (бакалаврський)
Спеціальність 123 Комп'ютерна інженерія
Науковий керівник: асистент Володимир КИСІЛЬ

На основі аналізу кваліфікаційної роботи на дотримання вимог академічної доброчесності (у т.ч. відсутності ознак академічного плагіату) з урахуванням результатів перевірки роботи спеціалізованим програмним засобом(ами) комісія зробила такий висновок:

№	Висновок	Позначка про відповідність
1	Ознаки академічного плагіату	
1.1	Запозичення, виявлені в роботі, є законними і не є академічним плагіатом (далі – зазначаються підстави віднесення запозичень до правомірних, якщо потрібно). Робота приймається до захисту.	відповідає
1.2	Виявлені запозичення не є академічним плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи (далі – зазначаються детальні та аргументовані підстави віднесення запозичень до правомірних). Робота приймається до захисту, але має бути відкоригована.	
1.3	Виявлені запозичення не є академічним плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота може бути допущена до захисту після того як буде відкоригована та доопрацьована і успішно пройде повторну перевірку на академічний плагіат.	
1.4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття текстових запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
2	Інші види порушень академічної доброчесності	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) усі запозичення фрагментарні, або мають належним чином оформленні посилання;
2) окремі виявлені збіги є загальновживаними фразами або виразами, про що свідчить посилання системи на збіг з джерелами на один фрагмент речення;
3) всі зафіксовані системою ознаки модифікації тексту відносяться до комбінування латинських символів зі україномовними скороченнями індексів в формулах, що не є модифікацією тексту.
4) значна частина знайденого плагіату відноситься до списку використаних джерел

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів/ ідентичності/схожості StrikePlagiarism, складає 2.74%; та системою Anti-Plagiarism складає 0.79%, що, з урахуванням наведених обгрунтувань, відповідає характеру наукового дослідження і свідчить на користь кваліфікаційної роботи.

01.06.2026

Завідувач кафедри	Підпис	Ольга ПАВЛОВА Ім'я, ПРІЗВИЩЕ
Гарант освітньої програми	Підпис	Андрій НІЧЕПОРУК Ім'я, ПРІЗВИЩЕ
Керівник кваліфікаційної роботи	Підпис	Володимир КИСІЛЬ Ім'я, ПРІЗВИЩЕ