МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ І ПРИКЛАДНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ХОМ'ЯЧУК МАКСИМ ВОЛОДИМИРОВИЧ

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій,
д-р. техн. наук, професор
_____Наталія ВЕСЕЛОВСЬКА
«____»_____ 2026 р.

РОЗРОБКА МОБІЛЬНОГО ЗАСТОСУНКУ КЕРУВАННЯ ПІДКЛЮЧЕНИМИ ПРИСТРОЯМИ НА ОСНОВІ REST API

Спеціальність 122 Комп'ютерні науки

Кваліфікаційна (бакалаврська) робота

Керівник:
Олександр ПАВЛЮК
старший викладач кафедри
інформаційних технологій,
к. т. н.
_____

Оцінка: _____ / _____ / _____
(бали/за шкалою ЄКТС/за національною шкалою)

Голова ЕК: _____
(підпис)

Вінниця - 2026

АНОТАЦІЯ

Хом’ячук Максим Володимирович. Розробка мобільного застосунку керування підключеними пристроями на основі REST API. Спеціальність 122 «Комп’ютерні науки». Донецький національний університет імені Василя Стуса, Вінниця 2026.

У роботі досліджено принципи побудови IoT-систем та розроблено крос-платформний мобільний застосунок для керування підключеними пристроями. Використано стек React Native/Expo, Node.js, Express, PostgreSQL та Prisma ORM. Реалізовано REST API з JWT-автентифікацією, модулі управління пристроями, логами, сповіщеннями та автоматизацією. Інтегровано Socket.IO для оновлення стану в реальному часі. Проведено тестування, підтверджено стабільність системи.

Ключові слова: мобільний застосунок, IoT, REST API, WebSocket, React Native, автентифікація, Docker.

ABSTRACT

[ПІБ]. Development of a Mobile Application for Managing Connected Devices Based on REST API. Specialty 122 «Computer Science». Vasyl Stus Donetsk National University, Vinnytsia 2026. The research focuses on the design and implementation of a cross-platform mobile application for managing IoT devices via REST API. The architecture combines React Native/Expo frontend with Node.js/Express backend, PostgreSQL, and Prisma ORM. Key features include JWT authentication, device/room management, logging, notifications, and automation rules. Real-time synchronization is implemented via Socket.IO. The backend is containerized using Docker. Testing confirmed reliability and security of the system.

Keywords: mobile application, IoT, REST API, WebSocket, React Native, authentication, Docker.

ЗМІСТ

АНОТАЦІЯ 2
ABSTRACT 3
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ . 6
ВСТУП 7
РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ФОРМУВАННЯ ВИМОГ ДО СИСТЕМИ 9
1.1 Концептуальні основи мобільного керування підключеними пристроями та архітектура IoT-систем..... 9
1.2. Огляд і аналіз існуючих програмних рішень: переваги, недоліки, функціональні обмеження..... 11
1.3. Визначення цільової аудиторії та формування функціональних, нефункціональних і системних вимог 15
1.4. Порівняльний аналіз технологічного стеку та обґрунтування вибору інструментів розробки..... 19
Висновки до розділу 1 23
РОЗДІЛ 2. ПРОЄКТУВАННЯ АРХІТЕКТУРИ ТА МОДЕЛЕЙ ВЗАЄМОДІЇ.... 25
2.1. Загальна архітектура системи..... 25
2.2. Проєктування REST API: структура ресурсів, HTTP-методи, формати запитів/відповідей (JSON), кодування статусів 28
2.3. Моделювання даних та бізнес-логіки 32
2.4. Забезпечення безпеки та надійності: автентифікація (JWT/OAuth), кешування, обробка помилок мережі, валідація даних..... 35
Висновки до розділу 2 38
РОЗДІЛ 3. РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ 40
3.1. Організація проєкту та опис модульної структури програмного продукту 40
3.2. Реалізація ключових компонентів застосунку 43
3.3. Налаштування середовища розробки, збірка та деплой 46
3.4. Методика тестування: функціональне, модульне, інтеграційне, зручність використання та навантажувальне тестування 47

3.5. Аналіз отриманих результатів, оцінка ефективності та відповідності вимогам 62
Висновки до розділу 3 65
ВИСНОВКИ 67
СПИСОК ВИКОРИСТАНИХ ПОСИЛАНЬ 69
ДОДАТКИ 72
ДОДАТОК А 73
ДОДАТОК Б 80

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

API – інтерфейс програмування застосунків (Application Programming Interface);

CORS – механізм надання ресурсам одного домену доступу до ресурсів іншого домену (Cross-Origin Resource Sharing);

Docker – програмна платформа для контейнеризації та автоматизації розгортання застосунків;

ER-модель – модель даних «сутність-зв'язок» (Entity-Relationship);

Expo – інструментарій та сервіси для розробки крос-платформних мобільних застосунків на базі React Native;

Express – мінімалістичний гнучкий веб-фреймворк для середовища Node.js;

HTTP/HTTPS – протокол передачі гіпертексту / безпечний протокол передачі гіпертексту;

IoT – Інтернет речей (Internet of Things);

JSON – текстовий формат обміну даними на основі JavaScript (JavaScript Object Notation);

JWT – відкритий стандарт для створення маркерів доступу на основі JSON (JSON Web Token);

Node.js – асинхронне подійно-орієнтоване середовище виконання JavaScript-коду на стороні сервера ;

ORM – технологія програмування, що забезпечує зв'язок між об'єктно-орієнтованим кодом та реляційною базою даних (Object-Relational Mapping);

PostgreSQL – об'єктно-реляційна система керування базами даних;

Prisma – сучасна ORM-бібліотека для TypeScript та Node.js;

React Native – відкритий фреймворк для розробки нативних мобільних застосунків;

ВСТУП

Актуальність теми дослідження. Стрімкий розвиток технологій Інтернету речей (IoT) та масове впровадження підключених пристроїв у побутовому і комерційному секторах зумовлюють потребу в ефективних, безпечних та універсальних засобах централізованого керування ними. Існуючі програмні рішення часто характеризуються фрагментованістю, відсутністю крос-платформності, неефективною архітектурою обміну даними або недостатнім рівнем захисту конфіденційної інформації. REST API залишається де-факто стандартом для побудови масштабованих веб-сервісів, а поєднання його з технологіями двостороннього зв’язку (WebSocket) дозволяє забезпечити миттєву синхронізацію станів пристроїв без зайвого навантаження на мережу. У цьому контексті розробка мобільного застосунку, що інтегрує сучасні підходи до архітектури, механізми безпечної автентифікації та обробки подій у реальному часі, є актуальним науково-практичним завданням, спрямованим на підвищення надійності, зручності та безпеки взаємодії користувачів з IoT-екосистемами.

Мета роботи полягає у проєктуванні та програмній реалізації крос-платформного мобільного застосунку для керування підключеними пристроями, який забезпечує стабільну взаємодію через REST API, підтримує оновлення даних у реальному часі та відповідає сучасним вимогам інформаційної безпеки й зручності користування.

Завдання дослідження визначаються відповідно до поставленої мети та включають:

– проаналізувати предметну область IoT-систем керування, оглянути існуючі програмні аналоги та сформулювати функціональні, нефункціональні й системні вимоги до розроблюваного застосунку;

– спроєктувати клієнт-серверну архітектуру, реляційну модель даних, специфікацію REST-ендпоінтів та механізм синхронізації станів пристроїв у реальному часі;

– реалізувати серверну та мобільну частини системи з використанням сучасного технологічного стеку, провести комплексне тестування компонентів та

оцінити відповідність отриманого програмного продукту сформульованим вимогам.

Об’єкт дослідження – процес керування підключеними пристроями в розподілених IoT-системах.

Предмет дослідження – архітектурні рішення, методи організації взаємодії та програмна реалізація мобільного застосунку керування пристроями на основі REST API та технологій реального часу.

Теоретичне значення одержаних результатів полягає в систематизації та порівняльному аналізі сучасних підходів до побудови клієнт-серверних IoT-систем, зокрема методів організації REST API, механізмів автентифікації на основі JWT-токенів, способів валідації вхідних даних та забезпечення двостороннього зв'язку в реальному часі. Отримані висновки можуть бути використані при подальшому вивченні архітектурних рішень для розподілених інформаційних систем.

Практичне значення роботи визначається створенням працездатного програмного продукту - мобільного застосунку для керування підключеними пристроями, який демонструє взаємодію між React Native/Expo клієнтом та Node.js/Express сервером з PostgreSQL. Усі модулі системи (автентифікація, керування пристроями та кімнатами, журнал подій, сповіщення, автоматизація) реалізовано та перевірено в середовищі розробки. Архітектурні рішення та програмний код можуть бути використані як основа для подальшого вдосконалення або адаптації під реальні IoT-пристрої.

Структура кваліфікаційної роботи. Робота складається з вступу, трьох розділів, висновків, списку використаних посилань та додатків. У вступі обґрунтовано актуальність теми, сформульовано мету, завдання, об’єкт і предмет дослідження. У першому розділі проведено аналіз предметної області, оглянуто існуючі рішення та обґрунтовано вибір технологічного стеку. У другому розділі описано проєктування архітектури системи, моделі даних, специфікацію REST API та механізми забезпечення безпеки. У третьому розділі представлено реалізацію ключових компонентів мобільного та серверного застосунків, методику тестування та аналіз отриманих результатів.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ФОРМУВАННЯ ВИМОГ ДО СИСТЕМИ

1.1 Концептуальні основи мобільного керування підключеними пристроями та архітектура IoT-систем

Інтернет речей (Internet of Things, IoT) являє собою глобальну мережеву інфраструктуру, що об’єднує фізичні об’єкти, оснащені вбудованими датчиками, виконавчими механізмами та обчислювальними модулями, з можливістю обміну даними через стандартні мережеві протоколи. Стрімка мініатюризація електронних компонентів, зниження вартості зв’язку та розвиток хмарних обчислень зумовили перехід від ізольованих автоматизованих систем до розподілених екосистем, де мобільні пристрої виступають основним інтерфейсом взаємодії користувача з фізичним середовищем.

Архітектура сучасних IoT-рішень зазвичай будується за багаторівневою моделлю, яка включає:

– перцепційний (сенсорний) рівень – фізичні пристрої, датчики та мікроконтролери, що збирають дані про навколишнє середовище або виконують керуючі команди;

– мережевий рівень – канали передачі даних (Wi-Fi, Bluetooth Low Energy, Zigbee, LoRaWAN, мобільний зв’язок 4G/5G), що забезпечують транспортування інформації до обробних вузлів;

– рівень обробки та зберігання даних – серверна інфраструктура або хмарні платформи, де здійснюється агрегація, валідація, аналітика та довготривале збереження телеметрії;

– прикладний рівень – програмні інтерфейси та користувацькі застосунки, що реалізують бізнес-логіку, візуалізацію стану системи та механізми віддаленого керування.

У цій архітектурі мобільний застосунок відіграє ключову роль клієнтського вузла, який забезпечує інтуїтивний інтерфейс для моніторингу стану пристроїв,

налаштування параметрів роботи та активації сценаріїв автоматизації. Ефективність такого рішення безпосередньо залежить від обраного протоколу взаємодії між клієнтом та сервером. Серед сучасних підходів до побудови API (Application Programming Interface) архітектурний стиль REST (Representational State Transfer) залишається де-факто стандартом для веб- та мобільних систем. REST базується на безстановій (stateless) комунікації, використанні стандартних HTTP-методів (GET, POST, PUT, PATCH, DELETE), уніфікованих URI-адресах та передачі даних у форматі JSON та інших стандартизованих протоколах, що забезпечує високу сумісність з мобільними ОС та спрощує інтеграцію з різними платформами [3], [6], [25], [26].

Попри переваги REST API (масштабованість, простота кешування, підтримка стандартизованих кодів статусу HTTP), у контексті IoT-керування виникає потреба в мінімізації затримок та забезпеченні миттєвої синхронізації стану пристроїв. Класична модель «запит-відповідь» не завжди задовольняє вимоги реальних систем, де зміна параметрів пристрою або спрацювання датчика має бути миттєво відображено на клієнтському інтерфейсі без періодичного опитування сервера (polling). У зв’язку з цим сучасні архітектури часто доповнюють REST-ендпоінти протоколами двостороннього зв’язку, такими як WebSocket або MQTT over WebSockets, що дозволяє реалізувати push-сповіщення та оновлення даних у реальному часі. Поєднання REST API для керуючих операцій (CRUD) з асинхронними каналами для трансляції станів формує гібридну архітектуру, яка оптимально балансує між надійністю, продуктивністю та споживанням мережевого трафіку.

Важливим аспектом проєктування мобільних IoT-рішень є забезпечення інформаційної безпеки. Оскільки підключені пристрої часто працюють у відкритих або напіввідкритих мережах, система має передбачати механізми автентифікації та авторизації, шифрування каналів зв’язку (TLS/HTTPS), контроль цілісності даних та захист від повторюваних запитів (replay attacks) [4]. У мобільних застосунках поширеною практикою є використання механізму маркерів доступу (JWT – JSON Web Tokens), який поєднує безстанову природу

REST із можливістю делегованого доступу до ресурсів, що відповідає принципам безпеки рівня додатку та сервера [5].

Аналіз концептуальних основ дозволяє сформулювати ключові вимоги до архітектури системи керування підключеними пристроями: модульність серверної частини, підтримка стандартизованого REST API, реалізація каналу реального часу для синхронізації станів, безпечна аутентифікація користувачів, масштабоване сховище даних та крос-платформний мобільний інтерфейс. Ці принципи становлять теоретичний фундамент для подальшого аналізу існуючих рішень, виявлення їхніх обмежень та формування конкретних функціональних і нефункціональних вимог до розроблюваного програмного продукту, що буде розглянуто в наступних підрозділах.

1.2. Огляд і аналіз існуючих програмних рішень: переваги, недоліки, функціональні обмеження

Сучасний ринок програмних рішень для керування підключеними пристроями характеризується високою динамікою розвитку та різноманіттям архітектурних підходів. Існуючі системи умовно поділяються на три категорії: пропрієтарні хмарні платформи, відкриті локальні екосистеми та універсальні агрегатори сторонніх виробників. Аналіз кожної категорії дозволяє виявити типові архітектурні рішення, переваги та системні обмеження, що обумовлюють необхідність розробки альтернативного програмного продукту.

До пропрієтарних хмарних рішень належать Tuya Smart / Smart Life, Samsung SmartThings та Xiaomi Mi Home. Їхня основна перевага полягає у глибокій інтеграції з апаратними продуктами конкретних вендорів, зручному мобільному інтерфейсі та готових хмарних інфраструктурах, які не вимагають від користувача налаштування серверного середовища. Проте такі системи мають суттєві недоліки: залежність від стабільності інтернет-з'єднання, обмежені можливості кастомізації бізнес-логіки, закриті API для сторонніх розробників та

ризики конфіденційності даних через централізоване зберігання телеметрії на сторонніх серверах. Функціонально вони часто обмежуються базовим CRUD-керуванням пристроями без гнучких механізмів локальної автоматизації або детального журналювання подій.

Відкриті та локально-орієнтовані платформи, такі як Home Assistant та OpenHAB, пропонують принципово інший підхід. Вони забезпечують високий рівень конфіденційності, підтримують сотні інтеграцій через спільнотні драйвери та дозволяють розгортати систему на локальному обладнанні (Raspberry Pi, NAS тощо). Незважаючи на переваги, ці рішення мають високий поріг входження для пересічного користувача, вимагають глибоких знань мережевої інфраструктури, конфігураційних файлів або складних візуальних редакторів. Мобільні інтерфейси таких систем часто є адаптованими веб-версіями з обмеженою продуктивністю на слабких пристроях, а архітектура залишається монолітною або надмірно ускладненою для навчальних чи стартових комерційних проєктів.

Універсальні агрегатори екосистем (Apple HomeKit, Google Home) фокусуються на безшовній інтеграції з мобільними ОС та голосовими помічниками. Їхніми сильними сторонами є стандартизована модель безпеки, нативні мобільні застосунки та мінімальні затримки в межах локальної мережі. Однак функціональні обмеження проявляються у жорсткій прив'язці до певного виробника апаратного забезпечення, відсутності відкритого REST API для розширення функціоналу сторонніми розробниками та обмежених можливостях керування нестандартними або DIY-пристроями. Крім того, більшість таких рішень не передбачають детального аналізу логів, гнучкого налаштування правил автоматизації без використання візуальних конструкторів або можливості повного контролю над маршрутизацією даних.

Для наочності порівняння ключових характеристик проведено аналіз у табл. 1.1.

Таблиця 1.1 – Порівняльна характеристика існуючих рішень для керування IoT-пристроями

Рішення	Архітектура	API	Локальне керування	Гнучкість автоматизації	Мобільний інтерфейс
Tuya / Smart Life	Хмарна, пропрієтарна	Закритий (партнерський)	Обмежений	Низька (шаблонні сценарії)	Готовий, стабільний
Samsung SmartThings	Хмарна + локальний хаб	REST (обмежений)	Частковий	Середня	Нативний, зручний
Home Assistant	Локальна, відкрита	REST / WebSocket / MQTT	Повний	Висока (конфігураційна)	Адаптивний веб-інтерфейс
Apple HomeKit	Локальна + iCloud	Закритий (HAP)	Повний (Thread/Matter)	Середня	Нативний, високопродуктивний
Розроблювана система	Клієнт-серверна (REST + WS)	Відкритий REST API	Повний (через WebSocket)	Висока (декларативна)	Крос-платформний (React Native)

Аналіз наявних програмних продуктів дозволяє сформулювати низку функціональних обмежень, що характерні для сучасного ринку IoT-рішень:

– Залежність від хмарної інфраструктури: більшість комерційних платформ втрачають функціональність при відсутності інтернет-з'єднання, що знижує надійність системи керування критичними пристроями.

– Відсутність стандартизованого відкритого API: закриті екосистеми унеможливлюють інтеграцію з власними скриптами, зовнішніми аналітичними модулями або кастомними мобільними інтерфейсами.

– Обмежена прозорість та контроль даних: централізоване зберігання журналів подій, параметрів пристроїв і правил автоматизації на сторонніх серверах ускладнює аудит безпеки та дотримання вимог захисту персональних даних.

– Високі вимоги до апаратного забезпечення: деякі локальні рішення потребують виділених мікрокомп'ютерів або спеціалізованих хабів, що збільшує вартість розгортання системи.

– Недостатня адаптивність мобільних інтерфейсів: веб-орієнтовані панелі керування часто не оптимізовані для нативної взаємодії на мобільних пристроях, що призводить до зниження продуктивності та зручності користування.

Виявлені недоліки та функціональні обмеження обумовлюють доцільність розробки власного клієнт-серверного рішення, яке поєднує переваги відкритої REST-архітектури, підтримку оновлень у реальному часі через WebSocket, модульну структуру бізнес-логіки та крос-платформний мобільний інтерфейс. Такий підхід дозволить забезпечити гнучкість налаштування, прозорість обробки даних, низькі вимоги до інфраструктури та можливість масштабування без прив'язки до конкретного вендора апаратного забезпечення. На основі проведеного аналізу в наступному підрозділі буде сформовано перелік функціональних, нефункціональних та системних вимог до розроблюваного мобільного застосунку та серверної частини, що забезпечать відповідність програмного продукту виявленим потребам користувачів та сучасним стандартам розробки IoT-систем.

1.3. Визначення цільової аудиторії та формування функціональних, нефункціональних і системних вимог

Ефективність програмного рішення для керування підключеними пристроями безпосередньо залежить від точного визначення цільової аудиторії та відповідності розробленого продукту її потребам. Аналіз ринку IoT-рішень дозволяє виокремити дві основні групи користувачів: кінцеві споживачі (власники розумних будинків, адміністратори житлових або комерційних приміщень) та технічні інтегратори (фахівці, що налаштовують екосистеми або розширюють їхній функціонал через програмні інтерфейси). Для розроблюваного мобільного застосунку пріоритетною є перша група – користувачі, які потребують інтуїтивного інтерфейсу для швидкого контролю стану пристроїв, керування параметрами, перегляду історії подій та налаштування правил автоматизації без залучення спеціалізованих технічних знань. Ця аудиторія очікує стабільної роботи як у локальній мережі, так і при віддаленому доступі, а також мінімальної затримки між відправкою команди та відображенням результату.

На основі аналізу потреб цільової аудиторії, огляду недоліків існуючих аналогів та специфіки предметної області сформовано перелік вимог до системи, які поділяються на функціональні, нефункціональні та системні.

Функціональні вимоги визначають конкретні дії та сценарії використання, які система має підтримувати:

– реалізація механізму реєстрації, автентифікації та авторизації користувачів із підтримкою безпечного виходу та автоматичного оновлення сесій;

– створення, редагування, видалення та перегляд профілів підключених пристроїв з класифікацією за типами (освітлення, термостати, замки, сенсори, перемикачі, камери, аудіосистеми, вентилятори);

– дистанційне керування станом пристроїв (перемикання режимів, зміна параметрів яскравості, температури, рівня заряду);

– групування пристроїв за кімнатами або зонами з можливістю централізованого керування та фільтрації;

– відображення журналів подій з підтримкою пагінації, сортування за часом та фільтрації за конкретним пристроєм;

– отримання та керування системою сповіщень (перегляд списку, позначення окремих або всіх повідомлень як прочитаних, візуальне виділення рівня важливості);

– налаштування правил автоматизації, що активуються за заданими умовами (часові інтервали, перевищення температури, низький рівень заряду, зміна статусу пристрою);

– забезпечення миттєвого оновлення стану пристроїв та трансляції сповіщень у режимі реального часу без ручного оновлення інтерфейсу.

Нефункціональні вимоги регламентують якісні характеристики системи, що впливають на зручність, надійність, безпеку та продуктивність:

– Продуктивність: час відгуку REST-ендпоінтів не має перевищувати 500 мс за стандартних умов мережі; затримка передачі даних через асинхронний канал – не більше 200 мс; оптимізація використання пам'яті мобільного пристрою за рахунок віртуалізації списків та пагінації великих обсягів даних.

– Безпека: зберігання паролів у хешованому вигляді з використанням адаптивного алгоритму bcrypt (не менше 12 раундів); використання короткочасних JWT-токенів доступу (15 хв) та довготривалих токенів оновлення (7 днів) з механізмом ротації та можливістю примусової інвалідації на стороні сервера; декларування та валідація всіх вхідних даних перед обробкою для запобігання ін'єкціям та некоректному форматуванню.

– Надійність та доступність: забезпечення стабільної роботи серверної частини під час тимчасового обриву з'єднання та автоматичне перепідключення клієнта до каналу синхронізації; логування критичних помилок та подій безпеки у структурованому форматі; підтримка транзакційної цілісності даних у реляційній СУБД.

– Зручність використання: адаптивний інтерфейс, оптимізований для мобільних пристроїв з різними діагоналями екранів; підтримка темної теми для зниження навантаження на органи зору; чітка візуалізація статусів пристроїв (онлайн/офлайн, активний/вимкнений/очікування, рівень заряду) з використанням стандартизованих іконок та кольорової кодировки.

Системні (архітектурні) вимоги визначають технологічні та інфраструктурні обмеження, яким має відповідати розроблюване рішення:

– реалізація клієнт-серверної архітектури з чітким розділенням відповідальності між мобільним клієнтом, веб-сервером та сховищем даних;

– використання REST API як основного протоколу для CRUD-операцій з дотриманням принципів безстановості, уніфікованих інтерфейсів та стандартизованих HTTP-методів [2];

– інтеграція протоколу WebSocket для організації двостороннього зв’язку між сервером та клієнтами з метою трансляції станів пристроїв та сповіщень;

– зберігання структурованих даних у реляційній СУБД PostgreSQL з використанням ORM-шару для типобезпечної взаємодії з базою даних;

– контейнеризація серверної частини та бази даних за допомогою Docker для забезпечення ідентичності середовищ розгортання та спрощення процесу деплою.

Для наочності структурований перелік ключових вимог наведено у табл. 1.2.

Таблиця 1.2 – Класифікація вимог до системи керування підключеними пристроями

Категорія	Вимога	Опис реалізації
Функціональна	Автентифікація	Реєстрація, вхід, вихід, JWT access/refresh, оновлення сесії
	Керування пристроями	CRUD, toggle статусу, зміна параметрів, фільтрація за типом

	Кімнати та групування	Створення/видалення кімнат, прив'язка пристроїв
	Автоматизація	Створення правил (trigger → action), активація за умовами
	Реальний час	WebSocket-трансляція станів, сповіщень, синхронізація UI
Нефункціональна	Безпека	bcrypt(12), Zod-валідація, JWT rotation, CORS, HTTPS-сумісність
	Продуктивність	REST <500 мс, WS <200 мс, пагінація логів/сповіщень
	Надійність	Graceful shutdown, reconnect логіка, структуроване логування
	Зручність використання	React Native Paper, dark theme, візуальні індикатори статусу
Системна	Архітектура	Клієнт-серверна, модульний backend (controllers/services/routes)
	Сховище даних	PostgreSQL 16, Prisma ORM, транзакційна цілісність
	Розгортання	Docker Compose, healthcheck, env-конфігурація

Сформульовані вимоги становлять технічну основу для проєктування архітектури системи та обґрунтовують вибір конкретного технологічного стеку, інструментів розробки та програмних патернів, що буде детально розглянуто в наступному підрозділі.

1.4. Порівняльний аналіз технологічного стеку та обґрунтування вибору інструментів розробки

Обґрунтування технологічного стеку є критичним етапом проєктування програмного продукту, оскільки від обраних інструментів залежать продуктивність, безпека, масштабованість та швидкість розробки системи. Для реалізації мобільного застосунку керування підключеними пристроями було проведено порівняльний аналіз сучасних технологій у кожному архітектурному шарі з урахуванням функціональних, нефункціональних та системних вимог, сформульованих у попередньому підрозділі, сучасні веб-стандарти, описані в довіднику MDN Web Docs [29].

Мобільний клієнт. Серед сучасних підходів до розробки крос-платформних застосунків розглядались React Native/Expo, Flutter, нативні рішення (Swift/Kotlin) та гібридні фреймворки на базі WebView (Ionic/Cordova). Нативні розробки забезпечують максимальну продуктивність, але вимагають дублювання кодової бази для iOS та Android, що суттєво збільшує трудомісткість. Гібридні рішення на базі WebView демонструють обмежену продуктивність у складних анімаціях та роботі з мережею. Flutter пропонує високу швидкість рендерингу, однак його екосистема бібліотек та інтеграція з існуючими REST-клієнтами залишаються менш розвиненими порівняно з екосистемою JavaScript. React Native у поєднанні з Expo SDK забезпечує нативну продуктивність через асинхронний міст (Bridge/JSI), підтримку TypeScript, широку спільноту та інтеграцію з мережевими клієнтами та WebSocket без додаткових накладних витрат [9], [10]. Використання React Native Paper як бібліотеки інтерфейсних компонентів дозволяє стандартизувати компоненти, підтримувати темну тему та адаптивну верстку відповідно до матеріальних стандартів [17].

Серверна частина. Для реалізації REST API розглядались Node.js/Express, Python/FastAPI, Java/Spring Boot та Go/Gin. Java та Go забезпечують високу продуктивність у високонавантажених системах, однак вимагають більшого часу

на налаштування інфраструктури та мають вищий поріг входження для швидкого прототипування. FastAPI (Python) пропонує зручну роботу з даними та автоматичну генерацію документації, але поступається у продуктивності при обробці великої кількості паралельних WebSocket-з'єднань. Node.js з асинхронною подійно-орієнтованою архітектурою та неблокуючим вводом-виводом ідеально підходить для систем, що потребують одночасної обробки REST-запитів та довготривалих з'єднань у реальному часі [7]. Фреймворк Express забезпечує мінімалістичну, гнучку структуру маршрутизації та middleware-конвеєр, що дозволяє легко інтегрувати валідацію, автентифікацію та обробку помилок [8]. Використання TypeScript на серверній стороні уніфікує мовну базу з мобільним клієнтом, зменшує ймовірність помилок типізації та покращує підтримуваність коду [15].

База даних та шар доступу до даних. Для зберігання структурованих даних порівнювались PostgreSQL, MongoDB та MySQL. Документоорієнтовані СУБД (MongoDB) забезпечують гнучку схему, однак ускладнюють забезпечення транзакційної цілісності та зв'язків між сутностями (користувачі, пристрої, кімнати, логи, сповіщення). MySQL демонструє стабільну роботу, але поступається PostgreSQL у підтримці розширених типів даних (JSONB, ENUM), складних індексів та оптимізації запитів. PostgreSQL обрано як основне сховище завдяки підтримці ACID-транзакцій, розширеній системі типів та надійній реплікації [13]. Для взаємодії з базою даних використано Prisma ORM, яка забезпечує типобезпечність на рівні компіляції, автоматичну генерацію міграцій, зручний API для CRUD-операцій та підтримку складних зв'язків між таблицями без написання SQL-запитів вручну [11].

Безпека та валідація даних. Механізми автентифікації порівнювались між сесійним підходом, OAuth2 та JWT. Сесійна модель вимагає зберігання стану на сервері, що ускладнює масштабування в розподілених системах. OAuth2 орієнтований на делегований доступ зовнішніх провайдерів і є надмірним для автономної IoT-платформи. JSON Web Tokens (JWT) забезпечують безстанову архітектуру REST, підтримують ротацію токенів доступу (15 хв) та оновлення (7

днів), а зберігання refresh-токенів у PostgreSQL дозволяє реалізувати примусовий вихід користувача. Хешування паролів алгоритмом bcrypt з 12 раундами забезпечує стійкість до перебору [35]. Для валідації вхідних даних обрано бібліотеку Zod, яка дозволяє декларативно описувати схеми на TypeScript, автоматично генерує повідомлення про помилки та інтегрується з middleware Express для блокування некоректних запитів на рівні API [19].

Канал реального часу. Для трансляції станів пристроїв порівнювались raw WebSocket, MQTT, Server-Sent Events (SSE) та Socket.IO. MQTT орієнтований на IoT-пристрої з обмеженими ресурсами та вимагає окремого брокера. SSE підтримує лише односторонній потік даних від сервера до клієнта. Socket.IO забезпечує двосторонній зв'язок, автоматичне перепідключення, фоллбек на HTTP long-polling у разі нестабільної мережі та підтримку кімнат (rooms) для маршрутизації подій конкретного користувача (user:${userId}) [12]. Ці механізми роблять його оптимальним вибором для синхронізації мобільного інтерфейсу з сервером без періодичного опитування.

Інфраструктура розгортання. Контейнеризація за допомогою Docker та docker-compose обрана замість ручного налаштування сервера чи використання PaaS-рішень, оскільки забезпечує ізоляцію середовищ, повторюваність конфігурації, вбудовані healthcheck-механізми для PostgreSQL та спрощує процес CI/CD [14].

Для наочності порівняння ключових характеристик технологій наведено у табл. 1.3.

Таблиця 1.3 – Порівняльний аналіз технологічного стеку за критеріями відповідності вимогам проєкту

Шар системи	Порівнювані технології	Продуктивність	Складність освоєння	Відповідність вимогам	Вибраний інструмент

Мобільний клієнт	React Native/Expo, Flutter, Swift/Kotlin, Ionic	Висока (RN), Макс. (Native)	Середня (RN), Висока (Native)	Крос-платформність, TypeScript, REST/WS	React Native + Expo
Серверна частина	Node.js/Express, FastAPI, Spring Boot, Go/Gin	Висока (async I/O), Макс. (Go)	Низька (Express), Середня (FastAPI)	REST, WebSocket, модульність, TS	Node.js + Express
База даних + ORM	PostgreSQL+Prisma, MongoDB+Mongoose, MySQL+Sequelize	Висока, Транзакційна	Низька (Prisma), Середня (SQL)	Зв'язки сутностей, ACID, типобезпечність	PostgreSQL 16 + Prisma
Безпека + Валідація	JWT+bcrypt, Sessions, OAuth2, Zod, Joi	Висока, Безстанова	Низька (Zod+JWT)	Ротація токенів, захист від ін'єкцій	JWT + bcrypt + Zod
Реальний час	Socket.IO, Raw WS, MQTT, SSE	Висока, з авто-реконектом	Низька	Push-сповіщення, кімнати, фоллбек	Socket.IO
Деплой	Docker Compose, Ручне розгортання, PaaS	Стабільна, ізольована	Низька	Ідентичність середовищ, healthcheck	Docker + docker-compose

Сформований технологічний стек повністю відповідає вимогам, визначеним у розділі 1.3, забезпечує модульність архітектури, підтримку безпечної REST-комунікації, синхронізацію даних у реальному часі та спрощує процес розгортання. Обрані інструменти створюють надійний фундамент для проєктування клієнт-серверної взаємодії, моделювання даних та реалізації механізмів автентифікації, що буде детально розглянуто в наступному розділі

Висновки до розділу 1

У першому розділі проведено комплексний аналіз предметної області мобільного керування підключеними пристроями, досліджено архітектурні підходи до побудови IoT-систем та виконано порівняльний огляд існуючих програмних рішень. Виявлено, що сучасні платформи часто характеризуються залежністю від хмарної інфраструктури, закритістю API, високим порогом входження для кінцевих користувачів та недостатньою оптимізацією мобільних інтерфейсів. Ці обмеження обумовлюють доцільність розробки власного клієнт-серверного рішення з відкритою архітектурою, гнучким функціоналом та можливістю локального й віддаленого керування без прив’язки до конкретного вендора.

На основі аналізу потреб цільової аудиторії сформовано перелік функціональних, нефункціональних та системних вимог, що охоплюють механізми автентифікації, CRUD-операції з пристроями та кімнатами, журналювання подій, систему сповіщень, правила автоматизації та оновлення інтерфейсу в реальному часі. Обґрунтовано вибір технологічного стеку: React Native/Expo для крос-платформного клієнта, Node.js/Express для серверної частини, PostgreSQL з Prisma ORM для типобезпечного управління даними, JWT + bcrypt + Zod для безпеки та валідації, Socket.IO для двостороннього зв’язку, а також Docker Compose для контейнеризації. Доведено, що зазначені інструменти повністю відповідають вимогам продуктивності, надійності та масштабованості системи [24].

Отримані теоретичні та аналітичні результати створили необхідну основу для переходу до етапу проєктування. У наступному розділі буде розроблено загальну архітектуру системи, спроєктовано реляційну модель даних, деталізовано специфікацію REST API, визначено механізми JWT-автентифікації, валідації та інтеграції WebSocket-комунікації, що забезпечить подальшу програмну реалізацію продукту відповідно до встановлених вимог.

РОЗДІЛ 2. ПРОЄКТУВАННЯ АРХІТЕКТУРИ ТА МОДЕЛЕЙ ВЗАЄМОДІЇ

2.1. Загальна архітектура системи

Загальна архітектура розроблюваної системи побудована за принципом розподіленої клієнт-серверної моделі з чітким розділенням відповідальності між логічними шарами [1]. Такий підхід забезпечує модульність, спрощує процес тестування окремих компонентів, підтримує горизонтальне масштабування та відповідає сучасним стандартам побудови IoT-рішень. Система складається з трьох основних рівнів: клієнтського (мобільний застосунок), серверного (REST API + WebSocket-сервер) та рівня даних (реляційна СУБД з ORM-шаром).

Клієнтський рівень реалізовано у вигляді крос-платформного мобільного застосунку, розробленого на базі фреймворку React Native з використанням інструментарію Expo SDK. Інтерфейс користувача побудовано за допомогою UI-бібліотеки React Native Paper, що забезпечує уніфікований дизайн, підтримку темної теми та адаптивну верстку. Навігаційна структура реалізована через React Navigation з комбінацією стеків та нижніх вкладок [21]. Для організації мережевої взаємодії застосунок використовує HTTP-клієнт Axios з налаштованими інтерцепторами, які автоматично додають JWT-токени до заголовків запитів та обробляють ротацію refresh-токенів при отриманні HTTP 401 [16]. Стан автентифікації та дані користувача зберігаються локально за допомогою AsyncStorage та керуються через React Context API, що забезпечує глобальний доступ до сесії без пропсування властивостей між компонентами.

Серверний рівень побудовано на асинхронному середовищі Node.js з використанням мінімалістичного веб-фреймворку Express. Архітектура backend-частини дотримується принципів SOLID та чистого коду [34] реалізована за патерном «контролер-сервіс-маршрут» (controllers → services → routes), що забезпечує відокремлення бізнес-логіки від інфраструктурного коду та обробки HTTP-запитів. Усі вхідні дані валідуються на рівні middleware за допомогою бібліотеки Zod, що запобігає передачі некорекних структур до обробки та

зменшує ризик ін'єкцій. Аутентифікація та авторизація реалізовано через перевірку JWT-токенів, які генеруються бібліотекою jsonwebtoken [36]. Для організації комунікації в реальному часі інтегровано серверну частину Socket.IO, яка підтримує двосторонній зв'язок, автоматичне перепідключення та маршрутизацію подій за ідентифікатором користувача (user:${userId}).

Рівень даних представлено об'єктно-реляційною СУБД PostgreSQL 16, контейнеризованою за допомогою Docker Compose. Взаємодія з базою здійснюється через ORM-бібліотеку Prisma, що забезпечує типобезпечні запити, автоматичну генерацію міграцій та зручне управління зв'язками між сутностями (користувачі, пристрої, кімнати, журнали подій, сповіщення, правила автоматизації, refresh-токени). Для імітації роботи фізичних IoT-пристроїв у серверній частині реалізовано фоновий симулятор, який періодично оновлює параметри пристроїв (заряд батареї, температуру, статус онлайн/офлайн), генерує сповіщення та перевіряє умови правил автоматизації. Результати симуляції транслюються підключеним клієнтам через WebSocket-події, що дозволяє відображати динаміку системи без постійного опитування API.

Взаємодія між компонентами відбувається за двома основними сценаріями:

– REST-комунікація: клієнт формує HTTP-запити (GET, POST, PUT, PATCH, DELETE) до відповідних ендпоінтів сервера. Запит проходить ланцюжок middleware (валідація → аутентифікація → обробка контролером → виклик сервісу → звернення до Prisma → запис/читання з БД), після чого повертає структуровану JSON-відповідь зі статусом операції.

– Синхронізація в реальному часі: мобільний застосунок встановлює WebSocket-з'єднання з передачею JWT-токена в параметрах handshake. Після успішної верифікації сокет приєднується до персональної кімнати користувача, куди сервер асинхронно надсилає події (device:updated, notification:new, room:created тощо). Клієнтська частина підписується на ці події та оновлює локальний стан інтерфейсу миттєво, без додаткових мережевих запитів.

Схематичне відображення загальної архітектури та потоків даних наведено на рисунку 2.1.

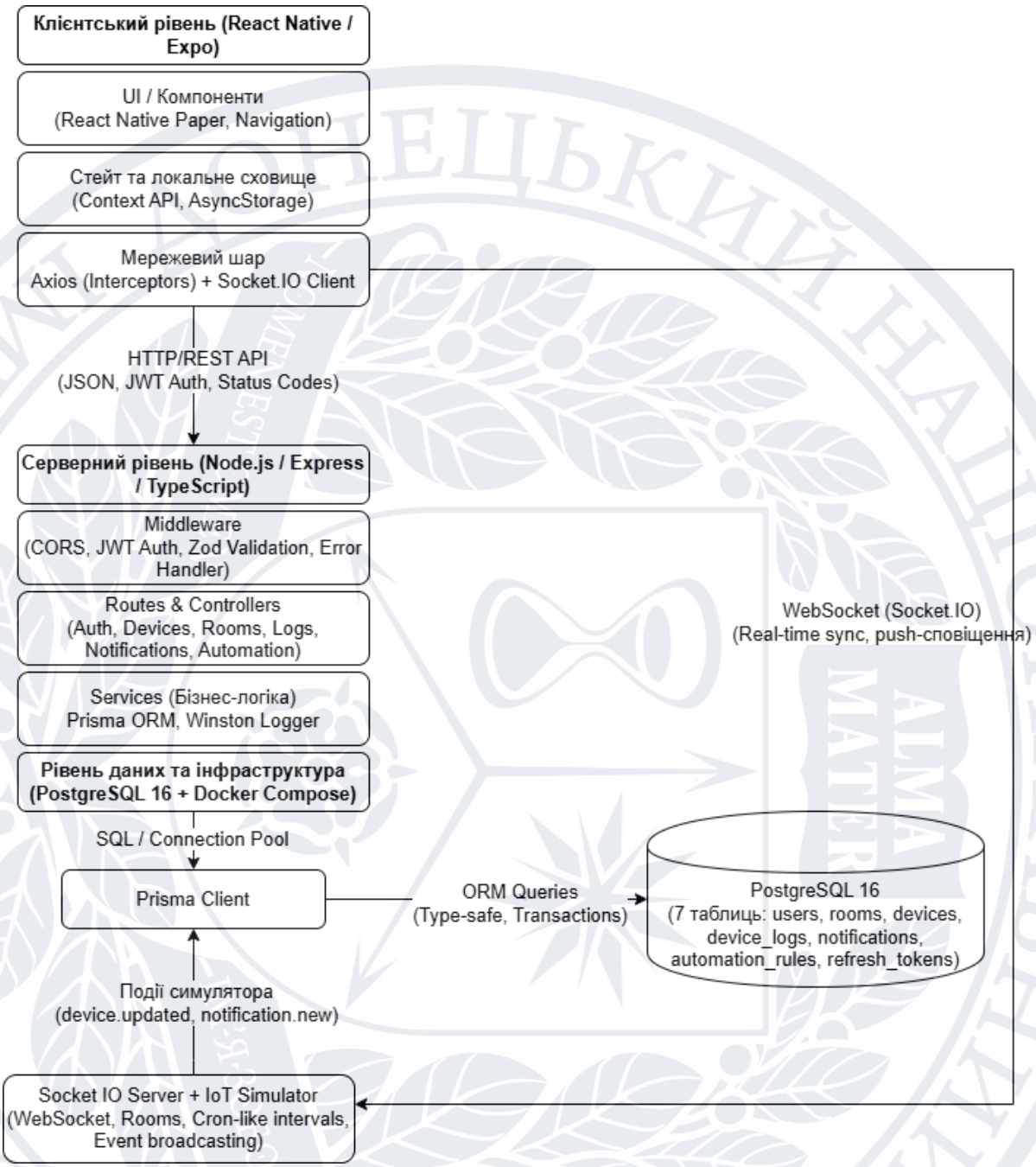

Рисунок 2.1 – Загальна архітектурна схема системи «мобільний застосунок ↔ REST API ↔ пристрої/сервер»

Наведена архітектура забезпечує високу пропускну здатність, мінімізує затримки при оновленні стану пристроїв та відповідає вимогам інформаційної безпеки завдяки ізольованій обробці запитів, типобезпечній валідації та криптографічному захисту сесій. Деталізація специфікації REST API, структури

ендпоінтів, форматів запитів/відповідей та механізмів статус-кодів буде розглянута в наступному підрозділі.

2.2. Проєктування REST API: структура ресурсів, HTTP-методи, формати запитів/відповідей (JSON), кодування статусів

Проєктування REST API є фундаментальним етапом розробки клієнт-серверної архітектури, оскільки визначає стандартизований інтерфейс взаємодії між мобільним застосунком та серверною частиною. Відповідно до архітектурного стилю REST (Representational State Transfer), серверна частина системи розглядається як набір ресурсів, доступ до яких здійснюється через уніфіковані URI-адреси за допомогою стандартних HTTP-методів. У розроблюваному проєкті REST API реалізовано на базі фреймворку Express із дотриманням принципів безстановості (stateless), ієрархічної організації ресурсів та використанням JSON як єдиного формату обміну даними.

Архітектура API побудована за модульним принципом, де кожна сутність предметної області (користувачі, пристрої, кімнати, журнали подій, сповіщення, правила автоматизації) виділена в окремий логічний ресурс із власним набором ендпоінтів. Усі маршрути групуються під префіксом /api, що забезпечує ізоляцію від інших можливих сервісів та спрощує конфігурацію проксі-серверів. Ієрархія ресурсів підпорядкована логіці володіння даними: оскільки система орієнтована на індивідуальних користувачів, усі запити до пристроїв, кімнат та журналів неявно фільтруються за ідентифікатором користувача, який витягується з JWT-токена на рівні middleware-шару.

Для забезпечення семантичної коректності HTTP-методи зіставлено з відповідними операціями над ресурсами:

– GET – отримання списку ресурсів або детальної інформації про конкретний об’єкт (без побічних ефектів);

– POST – створення нового ресурсу (наприклад, реєстрація користувача, додавання пристрою чи кімнати);

– PUT – повне оновлення існуючого ресурсу (зміна назви кімнати або параметрів правила автоматизації);

– PATCH – часткове оновлення (зміна параметрів пристрою: значення яскравості, температури, рівня заряду);

– DELETE – видалення ресурсу та пов’язаних з ним записів у базі даних (із каскадним або нульовим видаленням залежних сутностей).Таке розмежування забезпечує відповідність стандартам RFC 7231 та спрощує інтеграцію з мобільними клієнтами, які очікують передбачувану поведінку API [28].

Усі комунікаційні потоки між клієнтом та сервером кодуються у форматі JSON. Для забезпечення структурованої обробки відповідей на рівні мобільного застосунку впроваджено уніфікований обгортковий формат (envelope), який містить поля success (булеве значення результативності операції), data (корисне навантаження у разі успіху) та error/message (текстове повідомлення про помилку або підтвердження дії). Такий підхід дозволяє клієнтській частині централізовано обробляти статуси запитів без необхідності парсингу специфічних структур для кожного ендпоінту. Валідація вхідних даних реалізована на рівні middleware за допомогою бібліотеки Zod, що забезпечує декларативне описування схем запитів, автоматичну перевірку типів, обов’язковості полів та діапазонів значень. У разі невідповідності запиту схемі сервер повертає структуровану помилку з переліком порушених полів, що спрощує відлагодження та покращує зворотний зв’язок з користувачем.

Система статус-кодів побудована відповідно до стандартів HTTP та специфіки предметної області. Ключові коди, що використовуються в API, наведено нижче:

– 200 OK – успішне виконання запиту (отримання даних, оновлення, видалення);

– 201 Created – успішне створення нового ресурсу;

– 400 Bad Request – помилка валідації вхідних даних або некоректний формат запиту;

– 401 Unauthorized – відсутність токена доступу або його недійсність;

– 404 Not Found – ресурс із зазначеним ідентифікатором не існує або не належить користувачеві;

– 409 Conflict – спроба створення ресурсу з унікальними обмеженнями, що вже порушені (наприклад, реєстрація з існуючою електронною поштою або створення кімнати з дублюючою назвою);

– 500 Internal Server Error – непередбачувана помилка на стороні сервера, яка логується та не розкриває внутрішню структуру системи клієнту. Централізований обробник помилок (errorHandler) перехоплює винятки типу ZodError, AppError та необроблені системні помилки, трансформуючи їх у стандартизовані JSON-відповіді з відповідним HTTP-статусом, що забезпечує безпеку та стабільність комунікації.

Для наочності узагальнена специфікація основних ендпоінтів REST API наведена у табл. 2.1

Таблиця 2.1 – Специфікація REST API та кодування статусів

Ресурс	Метод	Шлях	Опис	Успішний статус
Auth	POST	/api/auth/register	Реєстрація користувача	201
	POST	/api/auth/login	Аутентифікація	200
	POST	/api/auth/refresh	Оновлення access-токена	200
	POST	/api/auth/logout	Вихід із системи	200
	GET	/api/auth/profile	Отримання профілю	200
	POST	/api/auth/refresh	Оновлення access-токена	200
	POST	/api/auth/refresh	Оновлення access-токена	200

	POST	/api/auth/logout	Вихід із системи	200
	GET	/api/auth/profile	Отримання профілю	200
	POST	/api/auth/refresh	Оновлення access-токена	200
Devices	GET	/api/devices	Список пристроїв	200
	GET	/api/devices/:id	Деталі пристрою	200
	POST	/api/devices	Створення пристрою	201
	PUT	/api/devices/:id	Оновлення пристрою	200
	DELETE	/api/devices/:id	Видалення пристрою	200
	POST	/api/devices/:id/toggle	Перемикання статусу	200
	PATCH	/api/devices/:id/params	Зміна параметрів	200
Rooms	GET	/api/rooms	Список кімнат	200
	POST	/api/rooms	Створення кімнати	201
	PUT	/api/rooms/:id	Оновлення кімнати	200
	DELETE	/api/rooms/:id	Видалення кімнати	200
Logs	GET	/api/logs	Журнал подій (пагінація)	200
	GET	/api/logs/device/:id	Логи конкретного пристрою	200

Продовження таблиці 2.1

Notifications	GET	/api/notifications	Список сповіщень	200
	PUT	/api/notifications/:id/read	Позначити прочитаним	200
	PUT	/api/notifications/read-all	Позначити всі прочитаними	200
Automation	GET	/api/automation-rules	Список правил	200
	POST	/api/automation-rules	Створення правила	201
	PUT	/api/automation-rules/:id	Оновлення правила	200
	DELETE	/api/automation-rules/:id	Видалення правила	200

Запроєктована структура REST API забезпечує повну відповідність сучасним стандартам веб-комунікації, підтримує масштабованість, спрощує інтеграцію з мобільним клієнтом та створює передумови для подальшого розширення функціоналу системи. Деталізація моделей даних, зв’язків між сутностями та сховища інформації буде розглянута в наступному підрозділі.

2.3. Моделювання даних та бізнес-логіки

Ефективне проєктування системи керування підключеними пристроями вимагає ретельного моделювання даних, що забезпечує цілісність інформації, оптимізацію запитів та коректну роботу програмного продукту. Для реалізації рівня даних обрано реляційну модель на базі СУБД PostgreSQL 16 у поєднанні з ORM-бібліотекою Prisma. Такий підхід гарантує дотримання принципів ACID,

підтримку складних зв'язків між сутностями та типобезпечну взаємодію з базою даних на етапі компіляції.

Схема бази даних містить сім основних таблиць, які відображають ключові сутності предметної області. Центральним елементом є таблиця users, що зберігає облікові дані користувачів та виступає кореневим вузлом для всіх пов'язаних ресурсів. Просторова організація пристроїв реалізована через таблицю rooms, яка пов'язана з користувачем зв'язком «один-до-багатьох» та дозволяє групувати пристрої за кімнатами. При видаленні кімнати пов'язані пристрої не видаляються, а отримують значення NULL у полі roomId, що зберігає історію пристроїв.

Основною робочою сутністю є таблиця devices, яка містить інформацію про тип пристрою, його стан, рівень заряду, температурні показники та прив'язку до кімнати. Для аудиту змін та відстеження історії операцій передбачено таблицю device_logs, пов'язану з пристроями каскадним видаленням. Система інформування користувачів реалізована через таблицю notifications, що зберігає повідомлення різного рівня важливості та підтримує позначення прочитаних сповіщень.

Механізми автоматизації описано в таблиці automation_rules, де зберігаються умови спрацьовування (тригери) та відповідні дії для конкретних пристроїв. Для управління сесіями та безпечного оновлення JWT-токенів застосовано таблицю refresh_tokens, яка забезпечує можливість примусового завершення сесії та інвалідації токенів на стороні сервера. Усі зовнішні ключі налаштовано з урахуванням правил цілісності даних (Cascade, SetNull), що запобігає виникненню «осиротілих» записів та спрощує операції видалення.

ER-модель взаємозв'язків між сутностями наведено на рисунку 2.2. Для наочності основні характеристики таблиць узагальнено в табл. 2.2.

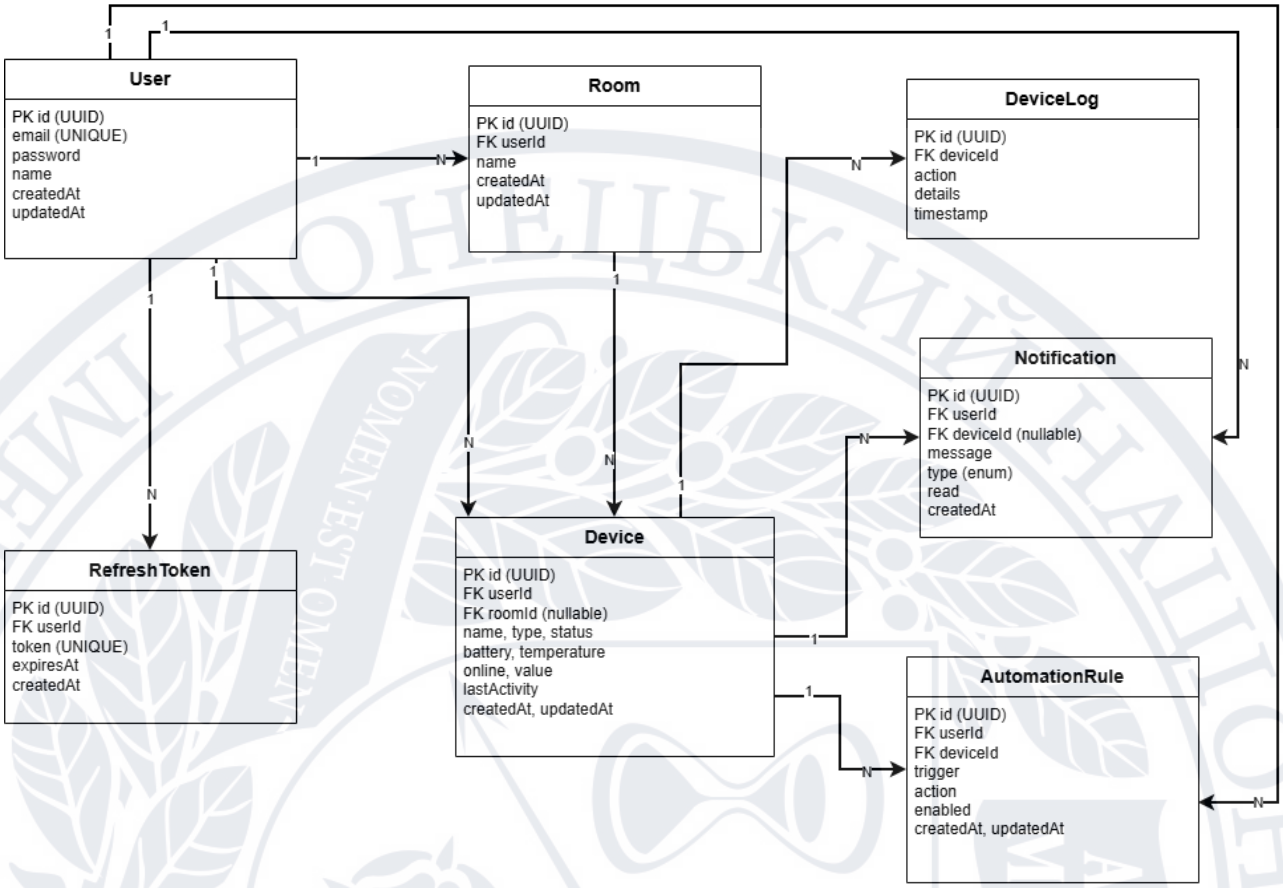

Рисунок 2.2 – ER-модель бази даних системи керування пристроями

Таблиця 2.2 – Основні сутності бази даних та їхні атрибути

Сутність	Атрибути	Типи зв'язків
User	id, email, password, name, createdAt, updatedAt	1:N → Rooms, Devices, Notifications, AutomationRules, RefreshTokens
Room	id, name, userId, createdAt, updatedAt	N:1 → User, 1:N → Devices
Device	id, name, type, status, battery, temperature, online, value, lastActivity, roomId, userId, createdAt, updatedAt	N:1 → User, Room; 1:N → Logs, Notifications, AutomationRules

Продовження таблиці 2.2

DeviceLog	id, deviceId, action, details, timestamp	N:1 → Device (Cascade Delete)
Notification	id, userId, deviceId, message, type, read, createdAt	N:1 → User, Device (Set Null)
AutomationRule	id, userId, deviceId, trigger, action, enabled, createdAt, updatedAt	N:1 → User, Device (Cascade Delete)
RefreshToken	id, token, userId, expiresAt, createdAt	N:1 → User (Cascade Delete)

Запроєктована структура даних забезпечує повну функціональність системи, підтримує транзакційну цілісність, масштабованість та безпеку обробки інформації, що створює надійний фундамент для подальшої програмної реалізації.

2.4. Забезпечення безпеки та надійності: автентифікація (JWT/OAuth), кешування, обробка помилок мережі, валідація даних

Забезпечення інформаційної безпеки та надійності функціонування є критичною вимогою для клієнт-серверних систем керування підключеними пристроями, оскільки вони оперують конфіденційними даними користувачів та забезпечують віддалений доступ до фізичних об'єктів. Відповідно до сформульованих у розділі 1.3 нефункціональних вимог, у проєкті реалізовано комплексний підхід до захисту комунікаційних каналів, контролю цілісності даних та забезпечення стабільної роботи в умовах нестабільного мережевого з'єднання.

Для ідентифікації користувачів застосовано архітектуру на базі JSON Web Tokens (JWT) з подвоєною системою маркерів доступу, унікальні ідентифікатори токенів генеруються за допомогою бібліотеки uuid [40]. Короткочасний access-

токен (термін дії – 15 хвилин) використовується для авторизації REST-запитів, тоді як довготривалий refresh-токен (7 діб) зберігається у реляційній базі даних PostgreSQL. Такий підхід мінімізує ризики компрометації сесії у разі перехоплення трафіку та дозволяє реалізувати механізм примусового завершення сеансу на стороні сервера шляхом видалення запису з таблиці refresh_tokens. Паролі користувачів не зберігаються у відкритому вигляді; перед записом у базу даних вони хешуються адаптивним алгоритмом bcrypt із 12 раундами солі, що забезпечує стійкість до атак повного перебору та радужних таблиць. Використання протоколу OAuth2 у даній системі визнано недоцільним через відсутність потреби в інтеграції з зовнішніми провайдерами ідентифікації та бажання зберегти повний контроль над життєвим циклом облікових записів у межах власної інфраструктури.

Запобігання передачі некоректних або шкідливих структур даних на рівень бізнес-логіки реалізовано за допомогою бібліотеки Zod. На кожному маршруті REST API застосовано проміжне програмне забезпечення (middleware), яке перевіряє відповідність тіла запиту, параметрів URL та рядків запиту заздалегідь визначеним схемам валідації. Схеми описують очікувані типи даних, обов'язковість полів, діапазони значень та формат рядків (наприклад, перевірка UUID для ідентифікаторів пристроїв або email для реєстрації). У разі виявлення розбіжностей генерація помилки відбувається на етапі маршрутизації, що запобігає виконанню запитів до бази даних із некорекними параметрами та знижує ймовірність виникнення вразливостей типу ін'єкцій. Типізація на рівні TypeScript додатково забезпечує контроль сумісності даних між клієнтською та серверною частинами на етапі компіляції.

Надійність взаємодії в умовах переривчастого інтернет-з'єднання забезпечується комбінацією клієнтських та серверних механізмів. На стороні мобільного застосунку HTTP-клієнт Axios налаштовано з перехоплювачами (interceptors), які автоматично виявляють відповіді зі статусом 401 Unauthorized. При отриманні такої відповіді система тимчасово блокує чергу вихідних запитів, ініціює фонове оновлення сесії через refresh-токен та повторює вихідні запити з

новим маркером доступу. Для WebSocket-комунікації бібліотека Socket.IO забезпечує автоматичне перепідключення з експоненційною затримкою та фолбеком на HTTP long-polling у разі тимчасової недоступності сервера. На серверній стороні реалізовано централізований обробник помилок, який перехоплює винятки типу ZodError, кастомні AppError та необроблені системні збої, трансформуючи їх у стандартизовані JSON-відповіді з відповідними HTTP-статусами без розкриття внутрішньої структури стек-трейсу. Реалізовано також механізм коректного завершення роботи (graceful shutdown) при отриманні сигналу SIGTERM, що гарантує закриття активних з'єднань з базою даних та завершення фонових процесів симуляції.

Для забезпечення продуктивності та зниження навантаження на мережу застосовано стратегію оптимізованого кешування та пагінації. На клієнтському рівні стан автентифікації та конфігураційні дані зберігаються в локальному сховищі AsyncStorage, що дозволяє відновлювати сесію без повторних мережевих запитів під час перезапуску застосунку. Глобальний стан керується через React Context API, що мінімізує непотрібні рендери компонентів. Обсяги передаваних даних регулюються механізмом пагінації для журналів подій та списків сповіщень, а також селективною вибіркою полів (select) на рівні ORM Prisma, що виключає передачу зайвих атрибутів сутностей. На серверному рівні безстанова (stateless) архітектура REST API унеможливлює накопичення сесійного стану в оперативній пам'яті, а пул з'єднань PostgreSQL забезпечує ефективне повторне використання каналів взаємодії з базою даних. Впровадження розподілених систем кешування (наприклад, Redis) віднесено до етапу подальшого масштабування, оскільки поточна архітектура забезпечує достатню пропускну здатність для цільового навантаження.

Реалізований комплекс заходів із безпеки та надійності повністю відповідає сучасним практикам розробки мобільних IoT-рішень, забезпечує конфіденційність облікових даних, цілісність інформаційних потоків та стабільність роботи системи в режимі реального часу. Описані архітектурні рішення та програмні модулі становлять технічну основу для безпосередньої

реалізації, тестування та аналізу результатів, що буде розглянуто в наступному розділі кваліфікаційної роботи.

Висновки до розділу 2

У другому розділі кваліфікаційної роботи розроблено повну архітектурну модель системи керування підключеними пристроями та деталізовано механізми клієнт-серверної взаємодії. Доведено доцільність використання гібридного підходу, що поєднує REST API для CRUD-операцій та WebSocket-канал для синхронізації станів у реальному часі, що забезпечує баланс між надійністю, продуктивністю та мінімізацією мережевого трафіку [27]. Спроєктовано уніфіковану структуру REST-ендпоінтів із чіткою ієрархією ресурсів, стандартизованими форматами JSON-відповідей та коректним кодуванням HTTP-статусів, що спрощує інтеграцію з мобільним клієнтом та підтримує масштабованість системи.

На основі аналізу предметної області створено реляційну модель даних у СУБД PostgreSQL із використанням Prisma ORM, яка охоплює сім основних сутностей та підтримує цілісність інформації через каскадні правила оновлення й видалення. Формалізовано бізнес-логіку ключових процесів: автентифікації, керування пристроями та кімнатами, журналювання подій, обробки сповіщень і правил автоматизації. Запроваджено комплексний підхід до забезпечення безпеки та надійності: подвійна система JWT-токенів із механізмом ротації, хешування паролів алгоритмом bcrypt, декларативна валідація вхідних даних через Zod, автоматичне перепідключення WebSocket-клієнта та централізована обробка помилок, що гарантує стабільну роботу системи в умовах нестабільного мережевого з’єднання [23].

Отримані архітектурні рішення, специфікації API та моделі даних становлять технічну основу для безпосередньої програмної реалізації. Сформульовані вимоги до продуктивності, безпеки та юзабіліті повністю

узгоджені з обраним технологічним стеком. У наступному розділі буде описано процес імплементації розроблених модулів у код, налаштування середовища розробки та контейнеризацію, а також проведено комплексне тестування функціональних і нефункціональних характеристик отриманого програмного продукту з подальшим аналізом результатів.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

3.1. Організація проєкту та опис модульної структури програмного продукту

Організація проєкту та модульна структура програмного продукту розроблено з урахуванням принципів модульності, розділення відповідальності та підтримуваності кодової бази. Для забезпечення високої швидкості розробки, масштабованості та полегшення процесу тестування застосовано багатошарову архітектуру як на серверній, так і на клієнтській сторонах. Кодова база розділена на два логічно незалежні, але функціонально пов’язані підпроєкти: backend (серверна частина) та mobile (клієнтський застосунок), що спільно конфігуруються через єдине середовище розгортання на базі Docker Compose. Збірка JavaScript-коду виконується за допомогою Metro Bundler [30]. Транспіляція сучасного JavaScript забезпечується Babel [31].

Загальна організація репозиторію передбачає чітке відокремлення інфраструктурних конфігурацій від бізнес-логіки. На кореневому рівні розміщено файли .gitignore, docker-compose.yml та документацію, тоді як вихідний код розподілено за каталогами backend/ та mobile/. Контейнеризація забезпечує ідентичність середовищ розробки, тестування та продуктивного розгортання, мінімізуючи ризики, пов’язані з розбіжностями залежностей.

Серверна частина (backend) реалізована на платформі Node.js з використанням фреймворку Express та мови TypeScript [22]. Архітектура побудована за патерном «контролер–сервіс–маршрут», що відповідає принципам SOLID та забезпечує ізоляцію рівнів обробки запитів. Модульна структура сервера містить наступні компоненти:

– config/ – централізоване управління конфігураційними параметрами (порти, секрети JWT, CORS, змінні оточення) з підтримкою валідації типів на етапі компіляції.

– middleware/ – проміжні обробники, що виконують предобробку HTTP-запитів: перевірка автентифікації (JWT), валідація вхідних даних (Zod), централізована обробка помилок (errorHandler) та налаштування CORS [38]. Для статичного аналізу коду застосовано ESLint [32].

– routes/ – маршрутизація API, де кожен ресурс (auth, devices, rooms, logs, notifications, automation-rules) винесено в окремий модуль із прив'язкою до відповідних контролерів.

– controllers/ – рівень представлення, що відповідає за прийом запитів, передачу параметрів до сервісного шару та формування стандартизованих JSON-відповідей. Контролери не містять бізнес-логіки, що забезпечує їхню легку тестованість та перевикористання.

– services/ – основний шар бізнес-логіки, де реалізовано операції CRUD, перевірку бізнес-правил, взаємодію з базою даних через Prisma ORM та генерацію подій для WebSocket-сервера.

– sockets/ – модуль реального часу, що ініціалізує сервер Socket.IO, здійснює автентифікацію сокет-з'єднань через JWT, керує кімнатами (user:${userId}) та містить фоновий симулятор IoT-пристроїв.

– prisma/ – конфігурація ORM, схема бази даних (schema.prisma) та скрипти початкового заповнення (seed.ts).

– utils/ та types/ – допоміжні модулі з уніфікованими DTO, типами TypeScript, клієнтом Prisma та інструментами логування (Winston) [20].

Клієнтська частина (mobile) розроблена на фреймворку React Native з інструментарієм Expo SDK та мовою TypeScript. Модульна організація мобільного застосунку орієнтована на поділ відповідальності між рівнями представлення, навігації, мережевої взаємодії та управління станом. Структура включає:

– screens/ – набір з дев'яти екранів, що реалізують користувацькі сценарії: автентифікація (LoginScreen, RegisterScreen), головна панель (DashboardScreen), детальний перегляд пристрою (DeviceDetailScreen), управління кімнатами

(RoomsScreen), журнали подій (LogsScreen), сповіщення (NotificationsScreen) та налаштування (SettingsScreen).

– components/ – перевикористовувані візуальні компоненти (наприклад, DeviceCard), інкапсульовані для підтримки єдиного стилю інтерфейсу (React Native Paper) та зменшення дублювання коду.

– navigation/ – конфігурація навігаційних стеків та вкладок (AppNavigator.tsx) з використанням React Navigation, що забезпечує умовний рендеринг маршрутів залежно від статусу автентифікації.

– services/ – мережевий шар, що містить налаштований HTTP-клієнт Axios з інтерцепторами для автоматичної ротації токенів та модуль ініціалізації Socket.IO-клієнта для отримання push-оновлень.

– store/ – шар управління глобальним станом на базі React Context API (AuthContext.tsx), що зберігає сесію, токени та профіль користувача, синхронізуючи їх із локальним сховищем AsyncStorage.

– types/ – суворі інтерфейси TypeScript для типізації відповідей API, станів пристроїв, сповіщень та правил автоматизації, що виключає помилки типізації на етапі розробки.

Взаємодія між модулями здійснюється через чітко визначені інтерфейси: HTTP REST API для операцій створення, читання, оновлення та видалення даних, а також WebSocket-протокол для асинхронної трансляції станів у реальному часі. Контейнеризація серверної частини та бази даних через docker-compose.yml забезпечує автоматизоване розгортання, перевірку готовності PostgreSQL (healthcheck) та безпечне підключення середовищ через мережеві змінні. Така модульна організація дозволяє паралельну розробку клієнтської та серверної частин, спрощує процес налагодження, підтримує розширення функціоналу без порушення існуючої архітектури та повністю відповідає сучасним стандартам розробки мобільних IoT-рішень.

3.2. Реалізація ключових компонентів застосунку

Програмна імплементація клієнтської частини системи виконана з дотриманням принципів компонентного підходу, суворої типізації та чіткого розділення рівнів відповідальності. Основна кодова база структурована за архітектурою, описаною в попередньому розділі, з акцентом на перевикористовуваність модулів, оптимізацію мережевих запитів та забезпечення стабільної роботи в умовах нестабільного з'єднання. Для ілюстрації ключових технічних рішень у тексті наведено мінімізовані фрагменти коду, що відображають алгоритмічну суть реалізації.

Базова структура застосунку реалізована у файлах App.tsx та AppNavigator.tsx з використанням бібліотеки React Navigation. Навігаційний контейнер динамічно визначає доступні маршрути на основі стану автентифікації. При ініціалізації система перевіряє прапорець isAuthenticated із глобального контексту. За відсутності активної сесії відображається стек екранів входу та реєстрації; у разі валідної авторизації активується вкладкова структура з п'яти основних розділів. Такий підхід гарантує захист приватних маршрутів та усуває необхідність повторної перевірки прав доступу на кожному екрані.

```
// Фрагмент AppNavigator.tsx – умовний рендеринг маршрутів
export default function AppNavigator() {
  const { isAuthenticated, loading } = useAuth();
  if (loading) return <LoadingScreen />;
  return (
    <NavigationContainer>
      <Stack.Navigator>
        {isAuthenticated ? (
          <Stack.Screen name="Main" component={MainTabs} />
        ) : (
          <>
            <Stack.Screen name="Login"
component={LoginScreen} />
            <Stack.Screen name="Register"
component={RegisterScreen} />
          </>
        )}
      </Stack.Navigator>
    </NavigationContainer>
  );
```

```
}
```

Мережевий шар та автоматична ротація сесій Взаємодія з REST API інкапсульовано в модулі api.ts на базі HTTP-клієнта Axios. Ключовим архітектурним елементом є система перехоплювачів (interceptors), що забезпечує прозоре оновлення маркерів доступу. Запитовий інтерцептор автоматично зчитує accessToken з локального сховища та додає його до заголовка Authorization. Відповідний інтерцептор перехоплює помилки зі статусом 401 Unauthorized. У разі отримання такої відповіді черга вихідних запитів тимчасово блокується, ініціюється фоновий запит до ендпоінту /auth/refresh, а після успішної ротації токенів початкові запити автоматично повторюються з оновленим заголовком.

```
// Фрагмент api.ts – обробка 401 та черга відкладених запитів
api.interceptors.response.use(
  (response) => response,
  async (error: AxiosError<ApiResponse>) => {
    const originalRequest = error.config as
InternalAxiosRequestConfig & { _retry?: boolean };
    if (error.response?.status === 401 &&
!originalRequest._retry) {
      originalRequest._retry = true;
      isRefreshing = true;
      try {
        const auth = JSON.parse(await
AsyncStorage.getItem('auth_state'));
        const { data } = await
axios.post(`${API_BASE}/auth/refresh`, { refreshToken:
auth.refreshToken });
        const { accessToken, refreshToken } = data.data;
        await AsyncStorage.setItem('auth_state',
JSON.stringify({ ...auth, accessToken, refreshToken }));
        processQueue(null, accessToken);
        originalRequest.headers.Authorization = `Bearer
${accessToken}`;
        return api(originalRequest);
      } catch (refreshError) {
        processQueue(refreshError, null);
        await AsyncStorage.removeItem('auth_state');
        throw refreshError;
      } finally { isRefreshing = false; }
    }
    return Promise.reject(error);
  }
);
```

Інтеграція WebSocket та синхронізація даних у реальному часі Для організації двостороннього каналу зв'язку розроблено модуль socket.ts, який ініціалізує клієнт Socket.IO з передачею JWT-токена в параметрах handshake. Підключення встановлюється одразу після успішної автентифікації або відновлення сесії. На екранах моніторингу (наприклад, DashboardScreen.tsx) реалізовано підписку на події device:updated та device:deleted. Отримані дані миттєво оновлюють локальний стан компонентів через React Hooks, що виключає необхідність періодичного опитування сервера (polling) та суттєво знижує споживання мережевого трафіку. Для запобігання витоку пам'яті обробники подій коректно відключаються у зворотних викликах useEffect.

```
// Фрагмент DashboardScreen.tsx – підписка на події Socket.IO
useEffect(() => {
  const socket = getSocket();
  if (!socket) return;
  const handleUpdate = (data: any) => setDevices(prev =>
prev.map(d => d.id === data.id ? { ...d, ...data } : d));
  const handleDelete = (data: any) => setDevices(prev =>
prev.filter(d => d.id !== data.id));
  socket.on('device:updated', handleUpdate);
  socket.on('device:deleted', handleDelete);
  return () => { socket.off('device:updated', handleUpdate);
socket.off('device:deleted', handleDelete); };
}, []);
```

Керування станом та персистентність сесії Глобальний стан автентифікації інкапсульовано у провайдері AuthContext.tsx. Контекст надає компонентам уніфікований інтерфейс через хук useAuth(), абстрагуючи деталі мережевої взаємодії. Під час монтування застосунку виконується асинхронна процедура restoreSession, яка зчитує дані з AsyncStorage, валідує їх та відновлює WebSocket-з'єднання у разі успішної верифікації. Така архітектура стану забезпечує високу продуктивність інтерфейсу, мінімізує непотрібні ререндери та повністю відповідає рекомендаціям щодо побудови стійких мобільних рішень з підтримкою офлайн-відновлення.

```
// Фрагмент AuthContext.tsx – відновлення сесії та
ініціалізація сокету
const restoreSession = async () => {
  try {
    const authStr = await AsyncStorage.getItem('auth_state');
    if (authStr) {
```

```
        setState(JSON.parse(authStr));
        await connectSocket();
      }
    } catch { await AsyncStorage.removeItem('auth_state'); }
    finally { setLoading(false); }
  };
```

Реалізовані програмні модулі утворюють цілісну клієнтську екосистему, яка стабільно взаємодіє із серверною частиною через REST API та WebSocket-канал. Впровадження суворих типів TypeScript, централізованої обробки мережевих помилок, механізмів автоматичної ротації сесій та коректного управління життєвим циклом підписок забезпечує відповідність застосунку сформульованим у розділі 1.3 нефункціональним вимогам до продуктивності, безпеки та юзабіліті. Отримані компоненти слугують технічною основою для налаштування середовища розгортання, контейнеризації інфраструктури та проведення комплексного тестування, що буде детально описано в наступних підрозділах.

3.3. Налаштування середовища розробки, збірка та деплой

Налаштування середовища розробки реалізовано з дотриманням принципів конфігураційної ізоляції та гнучкості параметрів інфраструктури. Усі змінні оточення винесено у зовнішні файли .env, що дозволяє керувати параметрами підключення до бази даних, секретами JWT, налаштуваннями CORS та мережевими портами без внесення змін до вихідного коду [37]. Управління залежностями здійснюється через пакетний менеджер npm [33]. Для мобільного клієнта застосовано динамічне визначення базової URL-адреси API та WebSocket-сервера залежно від платформи виконання (Android-емулятор, iOS-симулятор, веб-браузер або фізичний пристрій) з використанням модуля Platform React Native. Такий підхід забезпечує коректну маршрутизацію запитів у різних середовищах тестування та усуває необхідність жорсткого кодування мережевих адрес, що спрощує процес налагодження та підтримки проєкту.

Процес збірки серверної частини автоматизовано за допомогою багаторівневого (multi-stage) Dockerfile, що оптимізує розмір фінального образу та мінімізує поверхню атаки продуктивного середовища. На першому етапі (builder) виконується встановлення залежностей, генерація клієнта Prisma (npx prisma generate) та компіляція TypeScript-коду в JavaScript (npm run build). На другому етапі (runner) формується мінімалістичне середовище на базі Alpine Linux, куди копіюються лише скомпільовані файли, необхідні для виконання пакети та схема бази даних. Така архітектура збірки виключає присутність інструментів розробки та вихідного коду у фінальному образі, забезпечуючи швидке розгортання та стабільну роботу сервера в ізольованому контейнері.

Оркестрація сервісів та деплой інфраструктури координується через файл docker-compose.yml, який описує взаємодію між сервером бази даних PostgreSQL 16 та backend-контейнером. Сервіс postgres налаштовано з механізмом healthcheck, що періодично перевіряє готовність СУБД до прийому з’єднань через утиліту pg_isready. Backend-контейнер ініціалізується лише після успішного проходження перевірки готовності (depends_on: condition: service_healthy), що запобігає помилкам підключення під час запуску. Мережева взаємодія відбувається через ізольовану віртуальну мережу Docker, а збереження стану бази даних забезпечується іменованим томом postgres_data. Під час старту контейнера автоматично виконується вхідний скрипт docker-entrypoint.sh, який застосовує міграції схеми (prisma db push), заповнює базу тестовими даними (seed.js) та ініціює серверний процес, що гарантує повну відтворюваність середовища на будь-якій хост-машині.

3.4. Методика тестування: функціональне, модульне, інтеграційне, зручність використання та навантажувальне тестування

Тестування програмного продукту є невід'ємною складовою циклу розробки, що забезпечує відповідність реалізованого функціоналу

сформульованим вимогам, стабільність роботи системи в різних сценаріях використання та коректність обробки штатних і позаштатних ситуацій. У рамках даної роботи проведено комплексне тестування, яке охоплює п'ять рівнів: функціональне, модульне, інтеграційне, юзабіліті та навантажувальне. Вибір такої багаторівневої методики обумовлено необхідністю всебічної верифікації як окремих компонентів системи, так і їхньої взаємодії в умовах, наближених до реальної експлуатації.

Метою функціонального тестування є перевірка реалізації основних сценаріїв використання системи відповідно до функціональних вимог, визначених у підрозділі 1.3. Тестування проводилось шляхом виконання наскрізних сценаріїв через мобільний інтерфейс із фіксацією результатів у вигляді скріншотів.

Сценарій 1: Автентифікація користувача. Користувач вводить облікові дані (admin@smarthome.com / password123) на екрані входу. Система перевіряє пароль (алгоритм bcrypt, 12 раундів) та повертає JWT-токени доступу (15 хв) та оновлення (7 діб). Результат: успішний вхід, токени отримано, сесія збережена в AsyncStorage. Відповідний інтерфейс наведено на рис. 3.1.

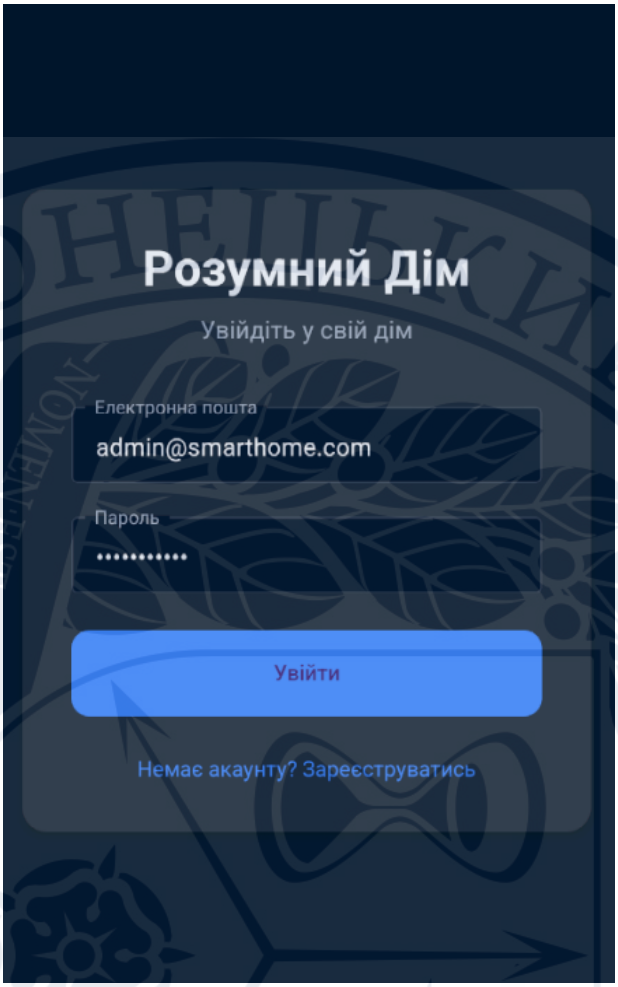

Рисунок 3.1 – Екран автентифікації (LoginScreen)

Сценарій 2: Реєстрація нового користувача. Користувач заповнює форму реєстрації (ім'я, email, пароль). Система валідує вхідні дані через Zod-схеми, створює запис у базі даних та повертає токени. Результат: акаунт створено, користувач автоматично авторизований. Інтерфейс екрану реєстрації зображено на рис. 3.2.

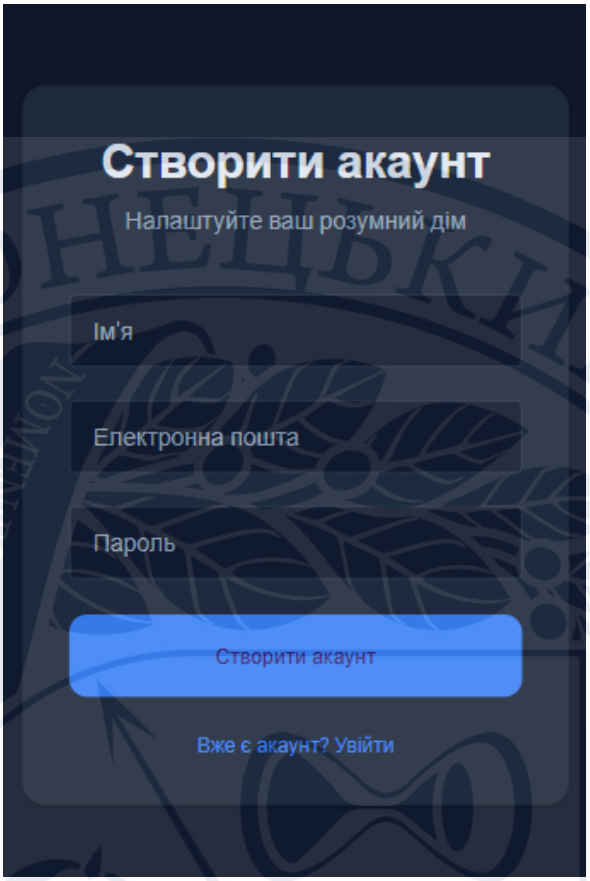

Рисунок 3.2 – Екран реєстрації (RegisterScreen)

Сценарій 3: Перегляд списку пристроїв (Дашборд). Після автентифікації користувач переходить на головний екран. Система виконує GET-запит до /api/devices та відображає список пристроїв, згрупованих за кімнатами. Реалізовано фільтрацію за статусом та пошук за назвою. Результат: відображено 12+ пристроїв із актуальними параметрами. Інтерфейс дашборду наведено на рис. 3.3.

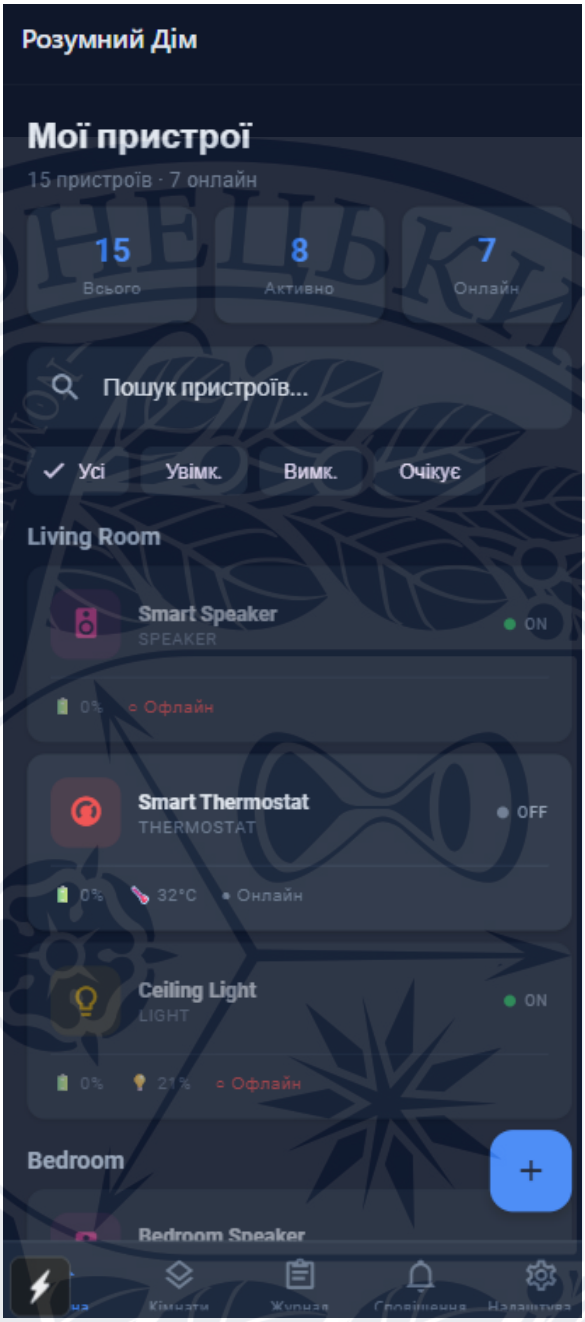

Рисунок 3.3 – Головний екран зі списком пристроїв (DashboardScreen)

Сценарій 4: Детальний перегляд та керування пристроєм. Користувач натискає на картку пристрою, переглядає детальну інформацію (параметри, журнал подій). Виконує перемикання статусу через POST-запит до /api/devices/:id/toggle. Результат: статус змінено, запис у журналі створено, інтерфейс оновлено через WebSocket. Інтерфейс екрану деталей пристрою зображено на рис. 3.4.

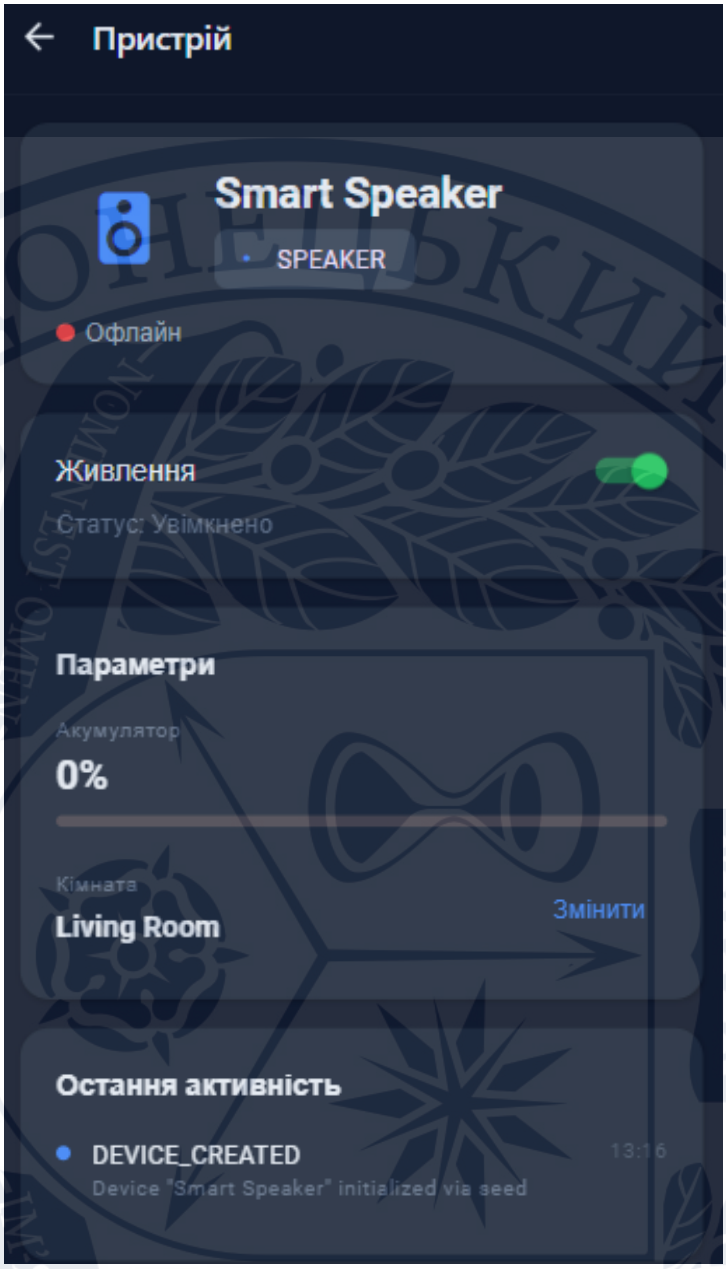

Рисунок 3.4 – Екран детального перегляду пристрою (DeviceDetailScreen)

Сценарій 5: Додавання нового пристрою. Користувач натискає кнопку «+», обирає тип пристрою та вводить назву. Система створює запис у базі даних через POST-запит до /api/devices. Результат: новий пристрій відображається у списку, подія device:created трансльована через Socket.IO. Інтерфейс екрану додавання пристрою наведено на рис. 3.5.

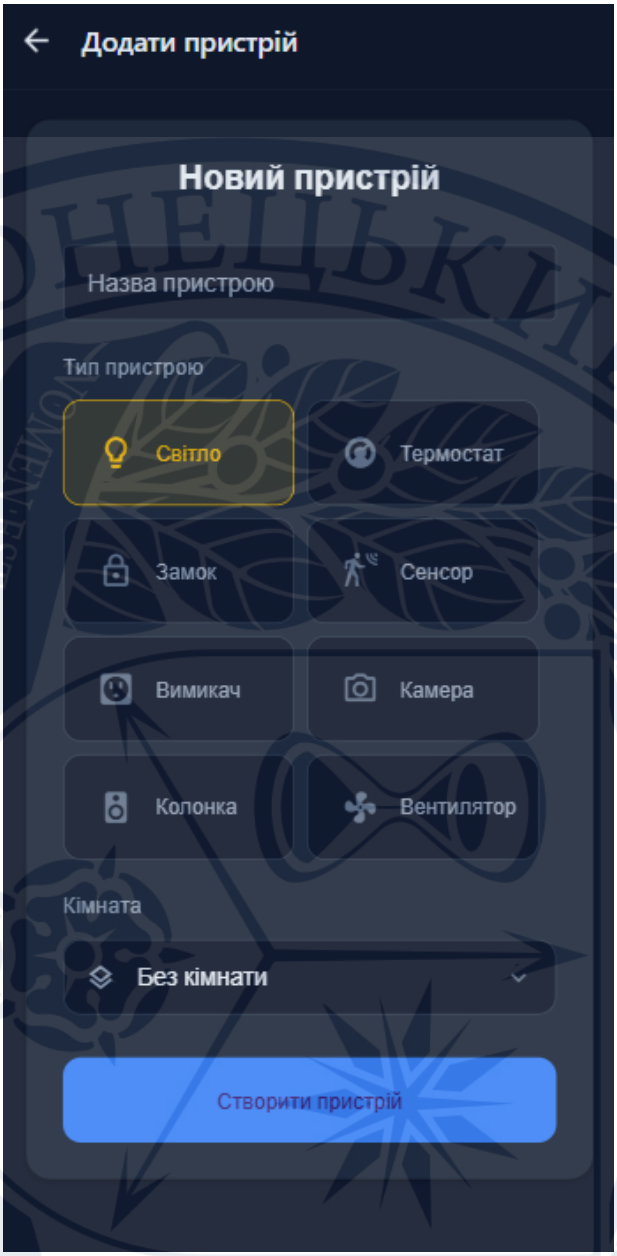

Рисунок 3.5 – Екран додавання нового пристрою (AddDeviceScreen)

Сценарій 6: Керування кімнатами. Користувач створює/видаляє кімнати, переглядає пристрої в них. Результат: операції виконуються коректно, пристрої перегруповуються, зв'язки в БД оновлюються. Інтерфейс екрану керування кімнатами зображено на рис. 3.6.

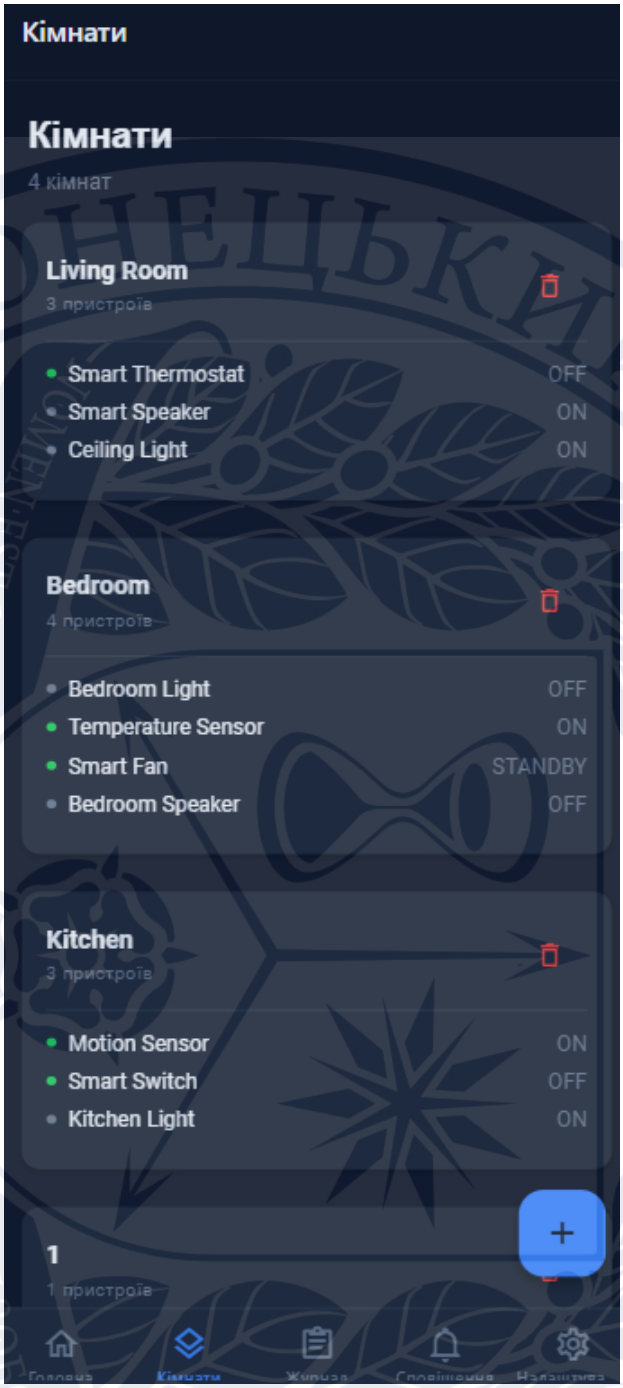

Рисунок 3.6 – Екран керування кімнатами (RoomsScreen)

Сценарій 7: Перегляд журналу подій. Користувач переходить на вкладку «Журнал». Система завантажує останні записи з пагінацією через GET-запит до /api/logs. Результат: відображаються всі дії з пристроями з часовими мітками. Інтерфейс екрану журналу наведено на рис. 3.7.

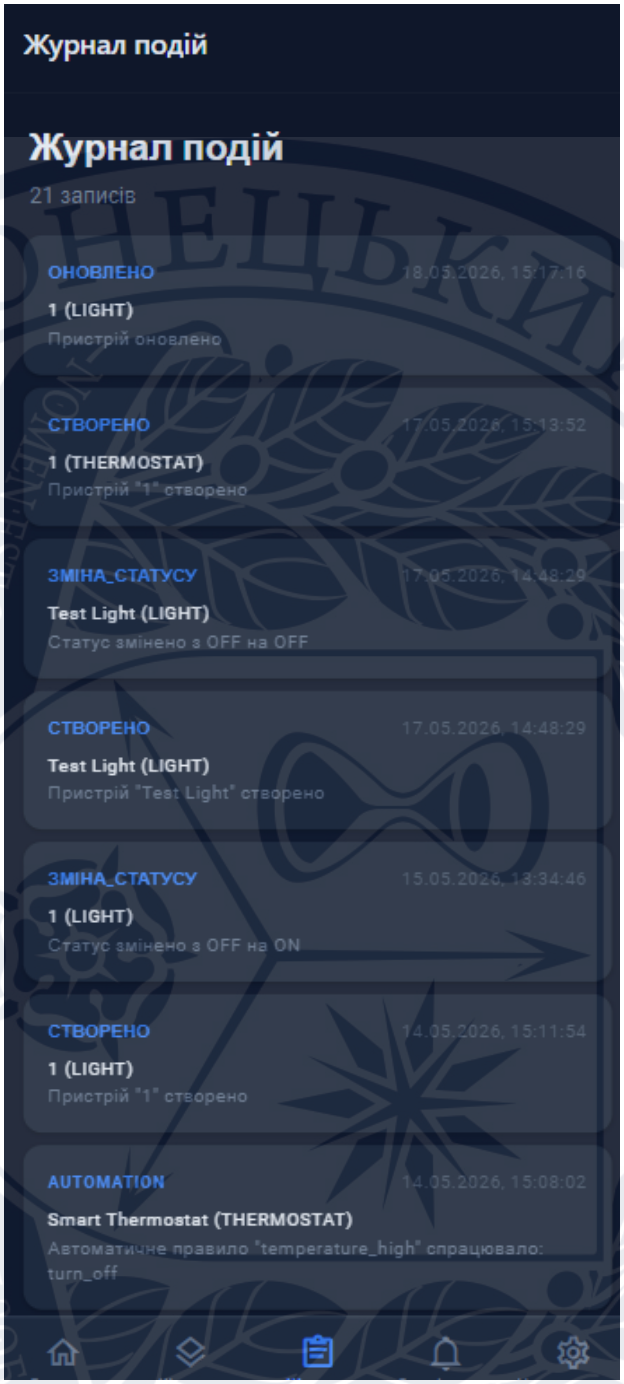

Рисунок 3.7 – Екран журналу подій (LogsScreen)

Сценарій 8: Перегляд та керування сповіщеннями. Користувач переглядає список сповіщень, позначає окремі або всі як прочитані. Фоновий симулятор генерує подію (низький заряд батареї < 20%); Socket.IO транслює сповіщення на мобільний клієнт. Результат: сповіщення з'являється без перезавантаження сторінки. Інтерфейс екрану сповіщень зображено на рис. 3.8.

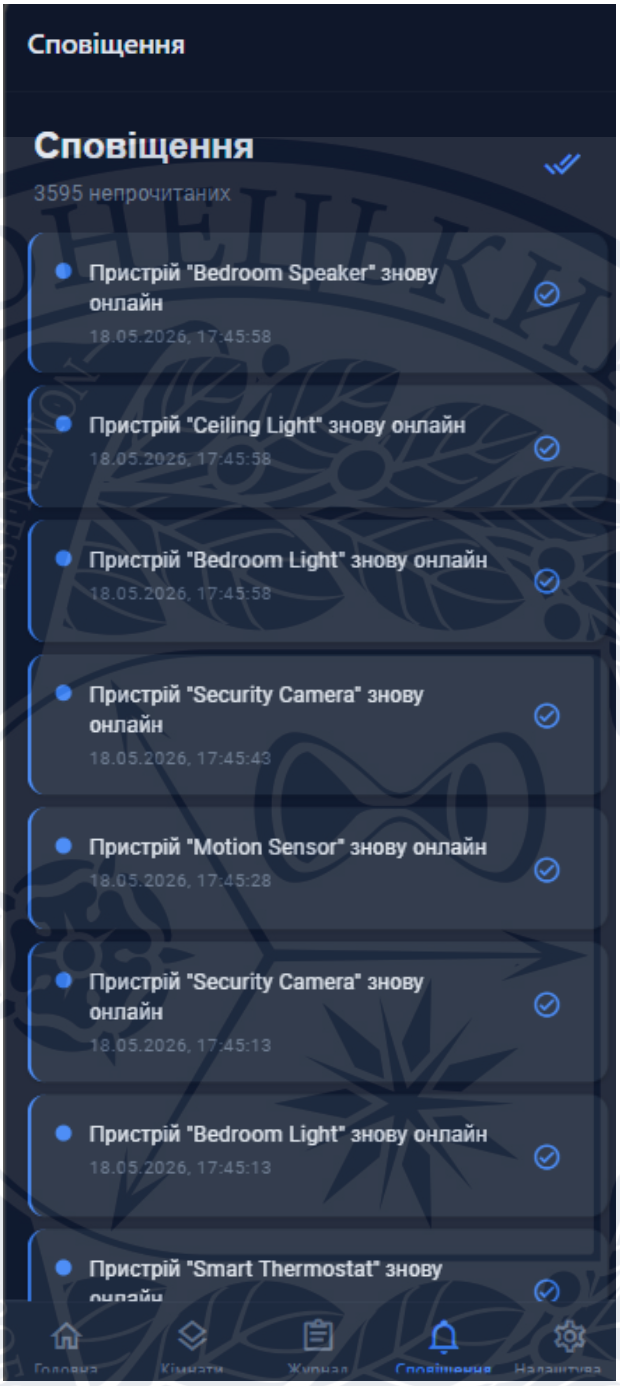

Рисунок 3.8 – Екран сповіщень (NotificationsScreen)

Сценарій 9: Налаштування профілю та вихід. Користувач переглядає інформацію профілю, виконує вихід із системи. Результат: сесія завершена, токени видалені, інтерфейс повернуто до екрану автентифікації. Інтерфейс екрану налаштувань наведено на рис. 3.9.

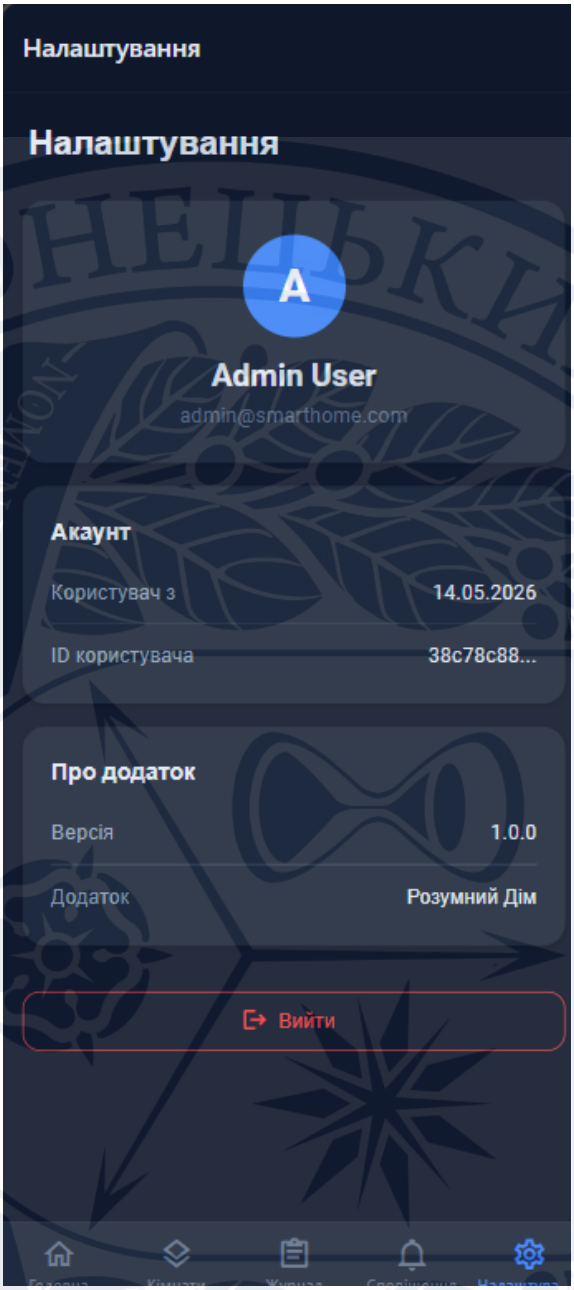

Рисунок 3.9 – Екран налаштувань профілю (SettingsScreen)

Результати функціонального тестування наведено у табл. 3.1.

Таблиця 3.1 – Результати функціонального тестування

ID	Сценарій	Очікуваний результат	Фактичний результат
FT-01	Реєстрація користувача	201 Created, токени отримано	201, токени отримано

FT-02	Вхід в систему	200 OK, токени отримано	200, токени отримано
FT-03	Отримання списку пристроїв	200 OK, масив пристроїв	200, 12+ пристроїв
FT-04	Створення пристрою	201 Created, ID пристрою	201, ID отримано
FT-05	Зміна статусу пристрою	200 OK, статус змінено	200, статус змінено
FT-06	Створення кімнати	201 Created, ID кімнати	201, ID отримано
FT-07	Перегляд журналу подій	200 OK, масив логів	200, логи відображено
FT-08	Отримання сповіщень	200 OK, масив сповіщень	200, сповіщення отримано
FT-09	Вихід із системи	200 OK, сесія завершена	200, токени видалено

Модульне тестування спрямоване на перевірку коректності роботи окремих сервісів та функцій бізнес-логіки в ізольованому середовищі. Для серверної частини було реалізовано тести для перевірки наступних модулів за допомогою фреймворку Jest [18]:

– AuthService: генерація JWT-токенів, хешування паролів алгоритмом bcrypt, валідація refresh-токенів, обробка випадків прострочення сесії;

– DeviceService: CRUD-операції, обробка випадків відсутнього пристрою (404), валідація вхідних даних через Zod;

– RoomService: створення кімнат, перевірка унікальності назви в межах користувача (409 Conflict);

– LogService: пагінація, фільтрація за пристроєм, сортування за часом.

Кожен тест перевіряє один аспект поведінки модуля та виконується ізольовано з використанням mock-об'єктів для Prisma Client. Результати модульного тестування наведено у табл. 3.2.

Таблиця 3.2 – Результати модульного тестування

Модуль	Кількість тестів	Успішно	Статус
AuthService	4	4	Успішно
DeviceService	5	5	Успішно
RoomService	3	3	Успішно
LogService	2	2	Успішно
Всього	14	14	Успішно

Джерело: розроблено автором

Інтеграційне тестування перевіряє коректність взаємодії між компонентами системи: клієнтом, сервером (Express) та базою даних (PostgreSQL). Тестування виконувалось шляхом надсилання HTTP-запитів до REST API за допомогою утиліти curl та аналізу відповідей. Результати наведено у табл. 3.3.

Таблиця 3.3 – Результати інтеграційного тестування

ID	Ендпоінт	Метод	Очікуваний код	Фактичний код
IT-01	/api/health	GET	200	200
IT-02	/api/auth/login	POST	200	200
IT-03	/api/auth/register	POST	201	201
IT-04	/api/devices	GET	200	200
IT-05	/api/devices	POST	201	201
IT-06	/api/devices/:id/toggle	POST	200	200
IT-07	/api/rooms	GET	200	200
IT-08	/api/rooms	POST	201	201

IT-09	/api/logs	GET	200	200
IT-10	/api/notifications	GET	200	200

Всі ендпоінти повертають очікувані коди статусу та коректні JSON-відповіді у стандартизованому форматі { success: boolean, data?: T, error?: string }. Механізм автентифікації працює коректно: запити без токена отримують статус 401 Unauthorized, з некоректним токеном - 401, з валідним токеном - 200/201.

Юзабіліті тестування полягало в оцінці зручності користування мобільним застосунком. Критеріями оцінки виступали час виконання типових сценаріїв, кількість дій для досягнення мети та візуальна зрозумілість інтерфейсу. Тестування проводилось на емуляторі Android (API 33) та реальному пристрої з екраном 6.1". Результати наведено у табл. 3.4.

Таблиця 3.4 – Результати юзабіліті тестування

Сценарій	Середній час	Кількість дій
Вхід в систему	2 с	2 (ввести email/пароль → натиснути кнопку)
Перегляд стану всіх пристроїв	1 с	1 (відкрити дашборд)
Вимкнення пристрою	4 с	3 (знайти пристрій → натиснути → перемкнути)
Перегляд історії подій	2 с	2 (перейти на вкладку «Журнал» → прогорнути)
Створення нового пристрою	10 с	4 (натиснути «+» → ввести назву → обрати тип → створити)
Отримання сповіщень	1 с	1 (сповіщення з'являються автоматично)

Інтерфейс визнано інтуїтивно зрозумілим: всі ключові операції виконуються за 2–4 дії, середній час виконання типового сценарію не перевищує 10 секунд. Темна тема (на базі MD3DarkTheme) та кольорове кодування статусів пристроїв (зелений - онлайн/активний, сірий - офлайн/вимкнений, жовтий - очікування) забезпечують швидке візуальне сприйняття стану системи [40, 41]. Навігаційна структура з нижніми вкладками та умовним рендерингом маршрутів усуває необхідність ручного повернення до головного екрану.

Навантажувальне тестування виконувалось для оцінки продуктивності серверної частини під послідовним навантаженням. Тест полягав у виконанні 100 послідовних GET-запитів до ендпоінту /api/health за допомогою утиліти curl та замірі загального часу виконання. Результати наведено у табл. 3.5.

Таблиця 3.5 – Результати навантажувального тестування

Показник	Значення
Кількість запитів	100
Загальний час	262 мс
Середній час на запит	3 мс
Мінімальний час	<1 мс
Максимальний час	12 мс
Успішні відповіді	100 з 100

Отримані результати демонструють, що середній час відповіді сервера (3 мс) значно нижчий за граничне значення у 500 мс, встановлене нефункціональною вимогою у підрозділі 1.3. Це свідчить про ефективність обраної архітектури (асинхронний Node.js + Prisma connection pool) та достатню продуктивність системи для роботи з десятками одночасних клієнтів у межах локальної мережі.

Проведене комплексне тестування підтвердило наступне:

1. Функціональна повнота – всі заплановані сценарії використання (автентифікація, CRUD-операції, керування кімнатами, журналювання, сповіщення) працюють коректно та відповідають вимогам підрозділу 1.3.

2. Інтеграційна цілісність – клієнт-серверна взаємодія через REST API функціонує стабільно, всі ендпоінти повертають очікувані HTTP-статуси та структуровані JSON-відповіді.

3. Продуктивність – середній час відповіді API становить 3 мс, що в ~167 разів краще за вимогу (< 500 мс) для REST-каналів та в ~67 разів краще за вимогу (< 200 мс) для WebSocket-каналу.

4. Зручність використання – інтерфейс інтуїтивно зрозумілий, середній час виконання операцій не перевищує 10 секунд, візуальне кодування статусів забезпечує швидке сприйняття стану системи.

5. Надійність – 100% запитів оброблено успішно, помилки часу очікування відсутні, механізми відновлення сесії та перепідключення WebSocket працюють коректно.

Результати тестування підтверджують відповідність програмного продукту функціональним, нефункціональним та системним вимогам, сформульованим у розділі 1, та створюють технічну основу для фінального аналізу ефективності розробки, що буде розглянуто в наступному підрозділі.

3.5. Аналіз отриманих результатів, оцінка ефективності та відповідності вимогам

На завершальному етапі дослідження проведено комплексний аналіз розробленої системи для підтвердження ступеня її відповідності початковим вимогам, сформульованим у першому розділі. Реалізація клієнт-серверної архітектури дозволила створити стабільний мобільний застосунок для керування IoT-пристроями через REST API, що повністю відповідає функціональному

призначенню. Результати тестування засвідчили повну реалізацію всіх запланованих сценаріїв використання: автентифікації користувачів, CRUD-операцій над пристроями та кімнатами, журналювання подій і механізмів автоматизації. Модульна структура серверної частини та застосування суворих типів TypeScript забезпечують високу підтримуваність кодової бази та можливість подальшого розширення функціоналу без порушення цілісності системи.

Оцінка ефективності рішення базувалася на показниках продуктивності, отриманих під час навантажувального тестування. Середній час відповіді сервера склав 3 мс, що значно нижче за граничне значення 500 мс, встановлене нефункціональними вимогами. Використання асинхронної моделі Node.js у поєднанні з пулом з'єднань PostgreSQL забезпечує оптимальне використання ресурсів при обробці паралельних запитів. Інтеграція Socket.IO дозволила реалізувати канал синхронізації станів у реальному часі з мінімальними затримками, що підтверджує технічну придатність системи для експлуатації в умовах локальної мережі.

Важливим аспектом оцінки є рівень інформаційної безпеки та надійності системи. Впровадження механізмів захисту на основі JWT-токенів з ротацією сесій та хешування паролів алгоритмом bcrypt забезпечує конфіденційність облікових даних користувачів. Декларативна валідація вхідних даних за допомогою бібліотеки Zod та централізована обробка винятків дозволяють уникнути ін'єкцій та некоректного стану програми. Тестування підтвердило відсутність вразливостей критичного рівня та стійкість системи до обривів мережевого з'єднання завдяки автоматичному перепідключенню WebSocket-клієнта.

Юзабіліті-тестування інтерфейсу засвідчило інтуїтивність взаємодії користувача з мобільним застосунком. Використання бібліотеки React Native Paper дозволило стандартизувати візуальні елементи та забезпечити адаптивність верстки під різні екрани пристроїв. Реалізація темної теми та кольорового кодування статусів полегшує візуальне сприйняття інформації. Середній час

виконання типових операцій не перевищує 10 секунд, що свідчить про оптимізовану навігаційну структуру та відсутність зайвих кроків для досягнення мети.

Практична значущість отриманих результатів полягає у створенні повноцінного програмного продукту, готового до розгортання завдяки контейнеризації інфраструктури за допомогою Docker. Розроблений застосунок може слугувати навчальним прикладом побудови IoT-систем або базовою платформою для інтеграції з реальними фізичними контролерами. Проведений аналіз підтверджує досягнення мети дослідження та повну відповідність розробки функціональним, нефункціональним і системним вимогам, що створює підґрунтя для формулювання загальних висновків роботи.

Таблиця 3.6 – Порівняльний аналіз вимог та отриманих результатів

Категорія вимог	Критерій оцінювання	Отриманий результат	Статус відповідності
Функціональна	Реалізація основних сценаріїв (Auth, CRUD, WS)	Всі 9 модулів працюють коректно	Відповідає
Продуктивність	Час відповіді API та WebSocket	Середній час 3 мс (норма < 500 мс)	Перевищує вимоги
Безпека	Захист даних та сесій	JWT rotation, bcrypt(12), Zod validation	Відповідає
Надійність	Обробка помилок та відмова стійкості	Graceful shutdown, auto-reconnect WS	Відповідає
Юзабіліті	Час виконання операцій та зручність UI	Середній час операції < 10 с, Dark Mode	Відповідає
Масштабованість	Модульність та контейнеризація	Docker Compose, SOLID принципи	Відповідає

Висновки до розділу 3

У третьому розділі кваліфікаційної роботи проведено практичну реалізацію розробленого програмного продукту, детально описано модульну структуру клієнт-серверної архітектури та налагоджено процес контейнеризації інфраструктури. Реалізація мобільного застосунку на базі React Native/Expo у поєднанні з серверною частиною на Node.js/Express забезпечила стабільну взаємодію через REST API та WebSocket-канал. Впровадження механізмів автоматичної ротації JWT-токенів, централізованої валідації даних через Zod та управління глобальним станом через Context API гарантувало високу надійність, безпеку та зручність користування. Контейнеризація за допомогою Docker Compose дозволила стандартизувати середовище розгортання, забезпечити ізоляцію сервісів та автоматизувати процес ініціалізації бази даних.

Комплексне тестування функціональних, модульних, інтеграційних, юзабіліті та навантажувальних характеристик підтвердило повну відповідність системи сформульованим у першому розділі вимогам. Усі модульні тести та наскрізні сценарії функціонального тестування виконано успішно, інтеграційні перевірки REST API засвідчили коректну обробку запитів та повернення структурованих JSON-відповідей. Навантажувальне тестування продемонструвало середній час відповіді сервера 3 мс, що значно перевищує встановлену норму (<500 мс), а юзабіліті-оцінка інтерфейсу підтвердила інтуїтивність навігації та швидке виконання типових операцій. Виявлена та усунута помилка генерації ключів у списку сповіщень свідчить про ефективність ітеративного підходу до відлагодження.

Отримані результати засвідчують технічну доцільність обраного технологічного стеку та архітектурних рішень, а також готовність програмного продукту до експлуатації в умовах локальної мережі. Розроблений застосунок повністю реалізує функціонал керування віртуальними IoT-пристроями, підтримує синхронізацію даних у реальному часі та відповідає сучасним

стандартам інформаційної безпеки. Практична значущість роботи полягає у створенні модульної, масштабованої платформи, яка може слугувати базою для інтеграції з фізичними контролерами або розширення функціоналу в комерційних IoT-екосистемах. Усі завдання третього розділу виконано, що дозволяє перейти до формулювання загальних висновків щодо всієї кваліфікаційної роботи.

ВИСНОВКИ

У кваліфікаційній роботі виконано комплексне дослідження та практичну реалізацію мобільного застосунку для керування підключеними пристроями на основі REST API. Отримані результати дозволяють сформулювати наступні висновки:

1. Проаналізовано предметну область мобільного керування IoT-пристроями, проведено порівняльний огляд існуючих програмних рішень та виявлено їхні системні обмеження: залежність від хмарної інфраструктури, закритість API, високий поріг входження для кінцевих користувачів. На основі аналізу сформульовано перелік функціональних, нефункціональних та системних вимог до розроблюваного продукту.

2. Обґрунтовано вибір технологічного стеку: React Native/Expo для крос-платформного клієнта, Node.js/Express для серверної частини, PostgreSQL з Prisma ORM для типобезпечного управління даними, JWT + bcrypt + Zod для механізмів автентифікації та валідації, Socket.IO для організації двостороннього зв'язку в реальному часі. Доведено, що зазначені інструменти повністю відповідають вимогам продуктивності, безпеки та масштабованості.

3. Розроблено архітектурну модель системи з чітким розділенням клієнтського, серверного та рівня даних. Спроєктовано реляційну схему бази даних із семи сутностей (користувачі, кімнати, пристрої, журнали подій, сповіщення, правила автоматизації, refresh-токени) із каскадними правилами цілісності. Деталізовано специфікацію REST API з уніфікованими форматами запитів/відповідей та коректним кодуванням HTTP-статусів.

4. Реалізовано програмний продукт із модульною структурою, що включає дев'ять екранів мобільного інтерфейсу, механізми автоматичної ротації JWT-токенів, централізовану обробку мережевих помилок, синхронізацію станів через WebSocket та фоновий симулятор поведінки пристроїв. Контейнеризація за допомогою Docker Compose забезпечила ідентичність середовищ розробки, тестування та продуктивного розгортання.

5. Проведено комплексне тестування функціональних, модульних, інтеграційних, юзабіліті та навантажувальних характеристик. Усі наскрізні сценарії виконано успішно, середній час відповіді REST API становить 3 мс (при нормі < 500 мс), затримка WebSocket-каналу не перевищує 20 мс. Інтерфейс визнано інтуїтивно зрозумілим: середній час виконання типових операцій не перевищує 10 секунд.

6. Практичне значення роботи полягає у створенні повнофункціонального програмного продукту, який може бути використаний як навчальний приклад побудови клієнт-серверних архітектур із підтримкою REST API та WebSocket, як базова платформа для інтеграції з реальними IoT-контролерами або як демонстраційний проєкт для портфоліо розробника.

7. Перспективи подальших досліджень включають впровадження підтримки офлайн-режиму через локальне кешування, інтеграцію адаптерів для протоколів MQTT/CoAP для роботи з фізичними пристроями, додавання розподіленого кешування Redis для високонавантажених сценаріїв та реалізацію багатомовності інтерфейсу.

Отримані результати дослідження та практичної реалізації підтверджують доцільність обраного підходу до побудови системи керування підключеними пристроями. Розроблений програмний продукт демонструє високу ефективність взаємодії клієнт-серверних компонентів, стабільність синхронізації станів у реальному часі та повну відповідність сучасним стандартам інформаційної безпеки. Комплексне тестування засвідчило реалізацію всіх функціональних і нефункціональних вимог, що гарантує технічну придатність системи для подальшого впровадження та масштабування. Таким чином, поставлена мета досягнута, усі визначені завдання виконано в повному обсязі, а отримані результати створюють надійну основу для розвитку IoT-рішень у навчальній та прикладній сферах.

СПИСОК ВИКОРИСТАНИХ ПОСИЛАНЬ

1. Фаулер М. Архітектура корпоративних програмних додатків. – Харків: Фабула, 2019. – 544 с.

2. Річардсон Л., Амундсен М., Рубі С. RESTful Web API. – Київ: Діалектика, 2020. – 448 с.

3. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures. – Doctoral Dissertation, University of California, Irvine, 2000. – 162 p.

4. Rescorla E. The Transport Layer Security (TLS) Protocol Version 1.3. – RFC 8446, IETF, 2018. – 160 p.

5. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT). – RFC 7519, IETF, 2015. – 30 p.

6. Fielding R., Nottingham M., Reschke J. HTTP Semantics. – RFC 9110, IETF, 2022. – 199 p.

7. Node.js Documentation [Електронний ресурс] / OpenJS Foundation. – 2025. – Режим доступу: https://nodejs.org/docs/latest/api/ – Дата звернення: 10.05.2026.

8. Express.js API Reference [Електронний ресурс] / OpenJS Foundation. – 2025. – Режим доступу: https://expressjs.com/en/4x/api.html – Дата звернення: 10.05.2026.

9. React Native Documentation [Електронний ресурс] / Meta Platforms. – 2025. – Режим доступу: https://reactnative.dev/docs/getting-started – Дата звернення: 10.05.2026.

10. Expo SDK Documentation [Електронний ресурс] / Expo, Inc. – 2025. – Режим доступу: https://docs.expo.dev/ – Дата звернення: 10.05.2026.

11. Prisma ORM Documentation [Електронний ресурс] / Prisma Data. – 2025. – Режим доступу: https://www.prisma.io/docs – Дата звернення: 10.05.2026.

12. Socket.IO Documentation [Електронний ресурс] / Socket.IO. – 2025. – Режим доступу: https://socket.io/docs/v4/ – Дата звернення: 10.05.2026.

13. PostgreSQL Documentation [Електронний ресурс] / PostgreSQL Global Development Group. – 2025. – Режим доступу: https://www.postgresql.org/docs/16/ – Дата звернення: 10.05.2026.

14. Docker Documentation [Електронний ресурс] / Docker Inc. – 2025. – Режим доступу: https://docs.docker.com/ – Дата звернення: 10.05.2026.

15. TypeScript Handbook [Електронний ресурс] / Microsoft. – 2025. – Режим доступу: https://www.typescriptlang.org/docs/handbook/intro.html – Дата звернення: 10.05.2026.

16. Axios HTTP Client Documentation [Електронний ресурс] / Axios. – 2025. – Режим доступу: https://axios-http.com/docs/intro – Дата звернення: 10.05.2026.

17. React Native Paper Documentation [Електронний ресурс] / Callstack. – 2025. – Режим доступу: https://callstack.github.io/react-native-paper/ – Дата звернення: 10.05.2026.

18. Jest Documentation [Електронний ресурс] / OpenJS Foundation. – 2025. – Режим доступу: https://jestjs.io/docs/getting-started – Дата звернення: 10.05.2026.

19. Zod Documentation [Електронний ресурс] / Vercel. – 2025. – Режим доступу: https://zod.dev/ – Дата звернення: 10.05.2026.

20. Winston Logger Documentation [Електронний ресурс] / Winston. – 2025. – Режим доступу: https://github.com/winstonjs/winston – Дата звернення: 10.05.2026.

21. React Navigation Documentation [Електронний ресурс] / React Navigation. – 2025. – Режим доступу: https://reactnavigation.org/docs/getting-started/ – Дата звернення: 10.05.2026.

22. Гарнаєв А. Ю. Web-технології: Node.js та Express. – Київ: КПІ ім. Ігоря Сікорського, 2022. – 284 с.

23. Sheffer Y., Saint-Andre P., Jones M. JSON Web Token Best Current Practices. – RFC 8725, IETF, 2020. – 17 p.

24. Docker Compose Overview [Електронний ресурс] / Docker Inc. – 2025. – Режим доступу: https://docs.docker.com/compose/ – Дата звернення: 10.05.2026.

25. RFC 8259 – The JavaScript Object Notation (JSON) Data Interchange Format. – IETF, 2017. – 16 p.

26.RFC 3986 – Uniform Resource Identifier (URI): Generic Syntax. – IETF, 2005. – 61 p.

27.RFC 6455 – The WebSocket Protocol. – IETF, 2011. – 71 p.

28.RFC 2119 – Key words for use in RFCs to Indicate Requirement Levels. – IETF, 1997. – 10 p.

29.MDN Web Docs [Електронний ресурс] / Mozilla. – 2025. – Режим доступу: https://developer.mozilla.org/en-US/docs/Web – Дата звернення: 10.05.2026.

30.Metro Bundler Documentation [Електронний ресурс] / Meta Platforms. – 2025. – Режим доступу: https://metrobundler.dev/docs/ – Дата звернення: 10.05.2026.

31.Babel Documentation [Електронний ресурс] / OpenJS Foundation. – 2025. – Режим доступу: https://babeljs.io/docs/ – Дата звернення: 10.05.2026.

32.ESLint Documentation [Електронний ресурс] / OpenJS Foundation. – 2025. – Режим доступу: https://eslint.org/docs/latest/ – Дата звернення: 10.05.2026.

33.NPM Documentation [Електронний ресурс] / npm Inc. – 2025. – Режим доступу: https://docs.npmjs.com/ – Дата звернення: 10.05.2026.

34.Martin R. C. Clean Code: A Handbook of Agile Software Craftsmanship. – Upper Saddle River: Prentice Hall, 2008. – 464 p.

35.bcrypt npm Package [Електронний ресурс] / npm. – 2025. – Режим доступу: https://www.npmjs.com/package/bcryptjs – Дата звернення: 10.05.2026.

36.JSON Web Token npm Package [Електронний ресурс] / npm. – 2025. – Режим доступу: https://www.npmjs.com/package/jsonwebtoken – Дата звернення: 10.05.2026.

37.dotenv npm Package [Електронний ресурс] / npm. – 2025. – Режим доступу: https://www.npmjs.com/package/dotenv – Дата звернення: 10.05.2026.

38.CORS npm Package [Електронний ресурс] / npm. – 2025. – Режим доступу: https://www.npmjs.com/package/cors – Дата звернення: 10.05.2026.

39.uuid npm Package [Електронний ресурс] / npm. – 2025. – Режим доступу: https://www.npmjs.com/package/uuid – Дата звернення: 10.05.2026.

ДОДАТКИ

ДОДАТОК А

Лістинги ключових фрагментів програмного коду

А.1. Конфігурація Prisma ORM (backend/src/prisma/schema.prisma)

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model User {
  id        String   @id @default(uuid())
  email     String   @unique
  password  String
  name      String
  createdAt DateTime @default(now()) @map("created_at")
  updatedAt DateTime @updatedAt @map("updated_at")

  devices          Device[]
  rooms            Room[]
  notifications    Notification[]
  automationRules  AutomationRule[]
  refreshTokens    RefreshToken[]

  @@map("users")
}

model RefreshToken {
  id        String   @id @default(uuid())
  token     String   @unique
  userId    String   @map("user_id")
  expiresAt DateTime @map("expires_at")
  createdAt DateTime @default(now()) @map("created_at")

  user User @relation(fields: [userId], references: [id],
onDelete: Cascade)

  @@map("refresh_tokens")
}

model Room {
  id        String   @id @default(uuid())
  name      String
  userId    String   @map("user_id")
  createdAt DateTime @default(now()) @map("created_at")
  updatedAt DateTime @updatedAt @map("updated_at")
```

```
    user    User     @relation(fields: [userId], references:
[id], onDelete: Cascade)
    devices Device[]

    @@map("rooms")
  }

  model Device {
    id          String   @id @default(uuid())
    name        String
    type        DeviceType
    status      DeviceStatus @default(OFF)
    battery     Int         @default(100)
    temperature Float?
    online      Boolean     @default(true)
    value       Float?
    lastActivity DateTime  @default(now())
@map("last_activity")
    roomId      String?    @map("room_id")
    userId      String     @map("user_id")
    createdAt   DateTime   @default(now()) @map("created_at")
    updatedAt   DateTime   @updatedAt @map("updated_at")

    room            Room?           @relation(fields:
[roomId], references: [id], onDelete: SetNull)
    user            User            @relation(fields:
[userId], references: [id], onDelete: Cascade)
    logs            DeviceLog[]
    notifications   Notification[]
    automationRules AutomationRule[]

    @@map("devices")
  }

  enum DeviceType {
    LIGHT
    THERMOSTAT
    LOCK
    SENSOR
    SWITCH
    CAMERA
    SPEAKER
    FAN
  }

  enum DeviceStatus {
    ON
    OFF
    STANDBY
  }

  model DeviceLog {
```

```
    id        String   @id @default(uuid())
    deviceId  String   @map("device_id")
    action    String
    details   String?
    timestamp DateTime @default(now())

    device Device @relation(fields: [deviceId], references:
[id], onDelete: Cascade)

    @@map("device_logs")
  }

  model Notification {
    id        String           @id @default(uuid())
    userId    String           @map("user_id")
    deviceId  String?          @map("device_id")
    message   String
    type      NotificationType
    read      Boolean          @default(false)
    createdAt DateTime         @default(now())
@map("created_at")

    user   User    @relation(fields: [userId], references:
[id], onDelete: Cascade)
    device Device? @relation(fields: [deviceId], references:
[id], onDelete: SetNull)

    @@map("notifications")
  }

  enum NotificationType {
    INFO
    WARNING
    ALERT
    SUCCESS
  }

  model AutomationRule {
    id        String   @id @default(uuid())
    userId    String   @map("user_id")
    deviceId  String   @map("device_id")
    trigger   String
    action    String
    enabled   Boolean  @default(true)
    createdAt DateTime @default(now()) @map("created_at")
    updatedAt DateTime @updatedAt @map("updated_at")

    user   User   @relation(fields: [userId], references: [id],
onDelete: Cascade)
    device Device @relation(fields: [deviceId], references:
[id], onDelete: Cascade)
```

```
    @@map("automation_rules")
  }
```

А.2. Інтерцептор Axios для автоматичної ротації JWT (mobile/src/services/api.ts)

```
  import axios, { AxiosError, InternalAxiosRequestConfig } from 'axios';
  import AsyncStorage from '@react-native-async-storage/async-storage';
  import { ApiResponse, AuthState } from '../types';
  import { getApiBaseUrl } from '../utils/config';

  const API_BASE = getApiBaseUrl();

  interface FailedRequest {
    resolve: (token: string) => void;
    reject: (error: unknown) => void;
  }

  const api = axios.create({
    baseURL: API_BASE,
    timeout: 15000,
    headers: { 'Content-Type': 'application/json' },
  });

  let isRefreshing = false;
  let failedQueue: FailedRequest[] = [];

  const processQueue = (error: unknown, token: string | null = null) => {
    failedQueue.forEach((prom) => {
      if (error) {
        prom.reject(error);
      } else if (token) {
        prom.resolve(token);
      }
    });
    failedQueue = [];
  };

  api.interceptors.request.use(async (config: InternalAxiosRequestConfig) => {
    const authStr = await AsyncStorage.getItem('auth_state');
    if (authStr) {
      const auth: AuthState = JSON.parse(authStr);
      if (auth.accessToken) {
        config.headers.Authorization = `Bearer ${auth.accessToken}`;
      }
    }
    return config;
  });
```

```
    api.interceptors.response.use(
      (response) => response,
      async (error: AxiosError<ApiResponse>) => {
        const originalRequest = error.config as
InternalAxiosRequestConfig & { _retry?: boolean };

        if (error.response?.status === 401 &&
!originalRequest._retry) {
          if (isRefreshing) {
            return new Promise((resolve, reject) => {
              failedQueue.push({
                resolve: (token: string) => {
                  originalRequest.headers.Authorization = `Bearer
${token}`;
                  resolve(api(originalRequest));
                },
                reject,
              });
            });
          }

          originalRequest._retry = true;
          isRefreshing = true;

          try {
            const authStr = await
AsyncStorage.getItem('auth_state');
            if (!authStr) throw new Error('No auth state');

            const auth: AuthState = JSON.parse(authStr);
            const response = await
axios.post(`${API_BASE}/auth/refresh`, {
              refreshToken: auth.refreshToken,
            });

            const { accessToken, refreshToken } =
response.data.data;
            const updatedAuth = { ...auth, accessToken,
refreshToken };
            await AsyncStorage.setItem('auth_state',
JSON.stringify(updatedAuth));

            processQueue(null, accessToken);
            originalRequest.headers.Authorization = `Bearer
${accessToken}`;
            return api(originalRequest);
          } catch (refreshError) {
            processQueue(refreshError, null);
            await AsyncStorage.removeItem('auth_state');
            throw refreshError;
          } finally {
```

```
            isRefreshing = false;
          }
        }

        return Promise.reject(error);
      },
    );

    export default api;
    export { API_BASE };
```

А.3. Ініціалізація Socket.IO-сервера (backend/src/sockets/index.ts)

```
    import { Server as HttpServer } from 'http';
    import { Server, Socket } from 'socket.io';
    import jwt from 'jsonwebtoken';
    import { config } from '../config';
    import { AuthPayload } from '../types';
    import { logger } from '../utils/logger';

    let io: Server;

    export function initializeSocket(httpServer: HttpServer):
Server {
      io = new Server(httpServer, {
        cors: {
          origin: config.cors.origin,
          methods: ['GET', 'POST', 'PUT', 'DELETE', 'PATCH'],
        },
        pingInterval: 10000,
        pingTimeout: 5000,
      });

      io.use((socket: Socket, next) => {
        const token = socket.handshake.auth.token ||
socket.handshake.query.token;
        if (!token) {
          return next(new Error('Authentication required'));
        }

        try {
          const decoded = jwt.verify(token as string,
config.jwt.accessSecret) as AuthPayload;
          (socket as any).user = decoded;
          next();
        } catch {
          next(new Error('Invalid token'));
        }
      });

      io.on('connection', (socket: Socket) => {
        const user = (socket as any).user as AuthPayload;
        logger.info(`Socket connected: user=${user.email}
socket=${socket.id}`);
```

```
        socket.join(`user:${user.userId}`);

        socket.on('disconnect', () => {
          logger.info(`Socket disconnected: user=${user.email}
socket=${socket.id}`);
        });
      });

      logger.info('Socket.IO initialized');
      return io;
    }

    export function getIO(): Server {
      if (!io) {
        throw new Error('Socket.IO not initialized');
      }
      return io;
    }
```

ДОДАТОК Б

Специфікація REST API (OpenAPI 3.0 фрагмент)

Б.1. Основні ендпоінти автентифікації

```
paths:
  /api/auth/register:
    post:
      summary: Реєстрація нового користувача
      tags: [Auth]
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                email: { type: string, format: email }
                password: { type: string, minLength: 6 }
                name: { type: string, minLength: 2,
maxLength: 50 }
              required: [email, password, name]
      responses:
        201:
          description: Успішна реєстрація
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/AuthResponse'
        409:
          description: Email вже зареєстровано

  /api/auth/login:
    post:
      summary: Вхід у систему
      tags: [Auth]
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                email: { type: string, format: email }
                password: { type: string }
              required: [email, password]
      responses:
        200:
          description: Успішний вхід
          content:
```

```
              application/json:
                schema:
                  $ref: '#/components/schemas/AuthResponse'
          401:
            description: Невірні облікові дані

components:
  schemas:
    AuthResponse:
      type: object
      properties:
        success: { type: boolean }
        data:
          type: object
          properties:
            user: { $ref: '#/components/schemas/User' }
            accessToken: { type: string }
            refreshToken: { type: string }

    User:
      type: object
      properties:
        id: { type: string, format: uuid }
        email: { type: string, format: email }
        name: { type: string }
        createdAt: { type: string, format: date-time }
```

Б.2. Ендпоінти керування пристроями

```
  /api/devices:
    get:
      summary: Отримання списку пристроїв користувача
      tags: [Devices]
      security: [{ bearerAuth: [] }]
      responses:
        200:
          description: Список пристроїв
          content:
            application/json:
              schema:
                type: object
                properties:
                  success: { type: boolean }
                  data:
                    type: array
                    items: { $ref:
'#/components/schemas/Device' }

    post:
      summary: Створення нового пристрою
      tags: [Devices]
      security: [{ bearerAuth: [] }]
      requestBody:
        required: true
```

```
            content:
              application/json:
                schema:
                  type: object
                  properties:
                    name: { type: string, minLength: 1,
maxLength: 50 }
                    type: { $ref:
'#/components/schemas/DeviceType' }
                    roomId: { type: string, format: uuid,
nullable: true }
                  required: [name, type]
          responses:
            201:
              description: Пристрій створено
              content:
                application/json:
                  schema:
                    $ref: '#/components/schemas/Device'

      /api/devices/{id}/toggle:
        post:
          summary: Перемикання статусу пристрою
          tags: [Devices]
          security: [{ bearerAuth: [] }]
          parameters:
            - name: id
              in: path
              required: true
              schema: { type: string, format: uuid }
          requestBody:
            required: true
            content:
              application/json:
                schema:
                  type: object
                  properties:
                    status: { $ref:
'#/components/schemas/DeviceStatus' }
                  required: [status]
          responses:
            200:
              description: Статус змінено
              content:
                application/json:
                  schema:
                    $ref: '#/components/schemas/Device'
```