КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БУДІВНИЦТВА І АРХІТЕКТУРИ

Автоматизації і інформаційних технологій

(факультет)

Кібербезпеки та комп'ютерної інженерії

(назва випускової кафедри)

КВАЛІФІКАЦІЙНА РОБОТА
ЗДОБУВАЧА СТУПЕНЯ ВИЩОЇ ОСВІТИ МАГІСТР

на тему:

Система моніторингу та керування IoT-пристроями (веб-додаток)

Козлюк Ілля Юрійович

(прізвище, ім’я та по батькові здобувача повністю)

Київ 2025 р.

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БУДІВНИЦТВА І АРХІТЕКТУРИ

Автоматизації і інформаційних технологій

(факультет)

Кібербезпеки та комп'ютерної інженерії

(назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____

„___” _____ 20 __ року

КВАЛІФІКАЦІЙНА РОБОТА
ЗДОБУВАЧА СТУПЕНЯ ВИЩОЇ ОСВІТИ МАГІСТР

Система моніторингу та керування
IoT-пристроями (веб-додаток)

(назва)

Я як здобувач вищої освіти КНУБА розумію і підтримую політику закладу з академічної доброчесності. Я не надавав і не одержував недозволену допомогу під час підготовки цієї роботи. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

Здобувач
Козлюк Ілля Юрійович
(прізвище, ім’я та по батькові повністю)
123 «Комп`ютерна інженерія»
(спеціальність)
Комп’ютерні системи і мережі
(освітня програма)
Група КСМм-24
Керівник Шабала Є.Є.
(прізвище та ініціали)
канд. тех. наук, доцент
(вчене звання, науковий ступінь)
Рецензент _____
(прізвище та ініціали)
Ідентичність підтверджую

Київ 2025 р.

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БУДІВНИЦТВА І АРХІТЕКТУРИ

Факультет: Автоматизації і інформаційних технологій .
Випускова кафедра: Кібербезпеки та комп'ютерної інженерії .
Ступінь вищої освіти: Магістр .
Спеціальність: 123 «Комп`ютерна інженерія» .
Освітня програма: Комп'ютерні системи і мережі .

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____
„___”_____ 20 __ року

З А В Д А Н Н Я
ДО ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ ЗДОБУВАЧА СТУПЕНЯ ВИЩОЇ ОСВІТИ МАГІСТР

Козлюк Ілля Юрійович .
.
(прізвище, ім'я та по батькові здобувача)

1. Тема роботи Система моніторингу та керування IoT-пристроями (веб-додаток)
.
.
затверджена наказом ректора КНУБА №1636/23.2/25 від «30» вересня 2025 року.
Керівник роботи
Шабала Євгенія Євгенівна .
кандидат технічних наук, доцент .
(прізвище, ім'я та по батькові, науковий ступінь, вчене звання)
3. Термін подання здобувачем роботи до захисту _____
4. Зміст пояснювальної записки за розділами:
Р. 1. Аналіз предметної області та сучасних рішень у сфері інтернету речей .
Р. 2. Обґрунтування вибору методів та засобів для розробки системи .
Р. 3. Проєктно-програмні рішення та результати їх дослідження .
5. Графічний матеріал за розділами:

Р. 1. 2 рисунка .
Р. 2. 3 рисунка .
Р. 3. 13 рисунків .

6. Консультанти розділів кваліфікаційної випускної роботи

Розділи	Прізвища, ініціали та посади консультанта	Перевірив	
		дата	підпис
Розділ 1.			
Розділ 2.			
Розділ 3.			

7. Календарний план виконання роботи:

Види робіт та їх зміст	Дата виконання
Р.1. Аналіз предметної області та сучасних рішень у сфері інтернету речей	Жовтень 2025 р.
Р.2. Обґрунтування вибору методів та засобів для розробки системи	Листопад 2025 р.
Р.3. Проєктно-програмні рішення та результати їх дослідження	Грудень 2025 р.
Висновок	Грудень 2025 р.
Остаточне оформлення роботи	Грудень 2025р.
Попередній захист роботи на кафедрі	Грудень 2025 р.
Направлення роботи на рецензування	Грудень 2025 р.

7. Дата видачі завдання _____

Керівник Шабала Є.Є. _____

Здобувач Козлюк І.Ю. _____

РЕЗЮМЕ (SUMMARY) *до кваліфікаційної випускової роботи здобувача*	ПІБ *Козлюк Ілля Юрійович* *Kozliuk Illia*		
ЗВО	Київський національний університет будівництва і архітектури		
Тема *(українською та англійською)*	Система моніторингу та керування IoT-пристроями (веб-додаток) IoT Device Monitoring and Management System (Web Application)		
Освітній ступінь	Магістр		
Факультет	Автоматизації і інформаційних технологій		
Випускова кафедра	Кібербезпеки та комп'ютерної інженерії		
Спеціальність	123 Комп'ютерна інженерія		
Освітня програма	Комп'ютерні системи і мережі		
Керівник	Шабала Євгенія Євгенівна		
Обсяг роботи:	*Пояснювальна записка, стор.*	*Розділів*	*Презентація, кількість слайдів*
	102(110 з додатками)	3	15
Розділ 1	Аналіз предметної області у сфері Інтернету речей		

Розділ 2	Обгрунтування вибору методів та засобів для розробки системи
Розділ 3	Проєктування та реалізація системи моніторингу та керування IoT-пристроями.
Висновки по роботі	У ході виконання дипломної роботи проаналізовано проблему фрагментації ринку IoT та існуючі програмні платформи. Розроблено та досліджено гнучку мікросервісну архітектуру системи моніторингу. Створено програмний прототип веб-додатку та проведено валідацію запропонованих проєктних рішень.
Ключові слова: Keywords:	Інтернет речей (IoT), мікросервісна архітектура, веб-додаток, NestJS, React, gRPC, RabbitMQ, Docker. Internet of Things (IoT), microservice architecture, web application, NestJS, React, gRPC, RabbitMQ, Docker.

Здобувач _____/ _____

Керівник _____/ _____

АНОТАЦІЯ

Дана кваліфікаційна магістерська робота має назву «Система моніторингу та керування IoT-пристроями (веб-додаток)», автором роботи є Козлюк Ілля Юрійович. Дипломна робота виконана в рамках спеціальності 123 "Комп'ютерна інженерія" та присвячена актуальній задачі проєктування складних програмних систем для Інтернету речей. Робота відповідає освітньому рівню "магістр".

Метою роботи є розробка та дослідження гнучкої мікросервісної архітектури для універсальної веб-системи моніторингу та керування пристроями Інтернету речей. У роботі представлено комплексне дослідження, проєктування та валідацію архітектурних рішень, що дозволяють вирішити проблему фрагментації сучасних IoT-екосистем.

У першому розділі проведено глибокий аналіз предметної області, виявлено ключову проблему гетерогенності IoT-пристроїв та обґрунтовано актуальність її вирішення. Проведено огляд сучасних технологій, протоколів та архітектурних моделей, а також виконано порівняльний аналіз існуючих на ринку платформ-аналогів, що дозволило визначити мету та завдання дослідження.

Другий розділ присвячено аналізу та обґрунтуванню вибору методів і засобів для розробки системи. Проведено порівняльний аналіз альтернативних архітектурних підходів . На основі аналізу було обґрунтовано вибір комплексної методики проєктування та визначено конкретний технологічний стек, що включає NestJS, React, gRPC, RabbitMQ та Docker.

Третій розділ є вирішальним для виконання завдання роботи. Він містить детальний опис розроблених проєктно-програмних рішень, починаючи від загальної архітектури системи і закінчуючи глибоким проєктуванням ключових мікросервісів та клієнтської частини. Практична цінність архітектури підтверджується результатами функціональної валідації програмного прототипу.

Ключові слова: Інтернет речей (IoT), мікросервісна архітектура, веб-додаток, NestJS, React, gRPC, RabbitMQ, Docker.

ABSTRACT

This Master's Qualification Thesis, titled "IoT Device Monitoring and Management System (Web Application)", was authored by Illia Kozliuk. The thesis was completed within the specialty 123 "Computer Engineering" and is dedicated to the relevant task of designing complex software systems for the Internet of Things. The work corresponds to the "Master's" educational level.

The objective of the work is to develop and research a flexible microservice architecture for a universal web-based system for monitoring and managing Internet of Things devices. The thesis presents a comprehensive study, design, and validation of architectural solutions that address the problem of fragmentation in modern IoT ecosystems.

The first chapter provides a deep analysis of the subject domain, identifies the key problem of IoT device heterogeneity, and substantiates the relevance of solving it. A review of modern technologies, protocols, and architectural models was conducted, along with a comparative analysis of existing analogue platforms on the market, which allowed for the definition of the research goal and objectives.

The second chapter is dedicated to the analysis and justification of the choice of methods and tools for the system's development. A comparative analysis of alternative architectural approaches was performed. Based on this analysis, a comprehensive design methodology was justified, and a specific technology stack was defined, including NestJS, React, gRPC, RabbitMQ, and Docker.

The third chapter is decisive for the accomplishment of the work's objective. It contains a detailed description of the developed design and software solutions, starting from the overall system architecture and ending with the in-depth design of key microservices and the client-side application. The practical value of the architecture is confirmed by the results of the functional validation of the software prototype.

Keywords: Internet of Things (IoT), microservice architecture, web application, NestJS, React, gRPC, RabbitMQ, Docker

ЗМІСТ

ВСТУП

Стрімкий розвиток інформаційних технологій та здешевлення мікроелектроніки призвели до формування глобального феномену — Інтернету речей (IoT), який передбачає підключення до мережі мільярдів фізичних пристроїв для збору, обробки та обміну даними. Актуальність даного напрямку підтверджується масштабами його поширення: згідно з аналітичними прогнозами, до 2030 року кількість підключених IoT-пристроїв перевищить 29 мільярдів, а обсяг глобального ринку сягне декількох трильйонів доларів. Інтеграція IoT-технологій у побут («розумний дім»), промисловість (IIoT), сільське господарство та міську інфраструктуру відкриває величезні можливості для автоматизації, оптимізації ресурсів та підвищення якості життя.

Разом з тим, бурхливе зростання ринку породило ключову системну проблему — високу гетерогенність та фрагментацію екосистеми IoT. Більшість виробників створюють власні закриті програмно-апаратні платформи, що використовують несумісні протоколи та пропрієтарні API. Це призводить до того, що кінцевий користувач змушений взаємодіяти з десятками окремих мобільних додатків, а створення комплексних сценаріїв взаємодії між пристроями різних брендів стає нетривіальною, а іноді й неможливою задачею.

Аналіз існуючих рішень показав, що на ринку домінують два типи платформ, які не задовольняють потреби масового користувача. З одного боку, це локальні open-source системи для ентузіастів (напр., Home Assistant), які є надзвичайно гнучкими, але вимагають глибоких технічних знань для налаштування та підтримки. З іншого боку, це потужні корпоративні хмарні платформи (напр., AWS IoT Core), які є надлишковими, дорогими та складними для індивідуального використання. Таким чином, існує обґрунтована необхідність у розробці системи, що поєднувала б простоту використання хмарних SaaS-рішень з доступністю та функціоналом, орієнтованим на кінцевого користувача.

У зв’язку з цим, метою даної кваліфікаційної магістерської роботи є розробка та дослідження гнучкої мікросервісної архітектури для універсальної веб-системи моніторингу та керування IoT-пристроями, яка вирішує проблему фрагментації шляхом надання єдиного, інтуїтивно зрозумілого інтерфейсу.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

- провести аналіз предметної області, сучасних архітектур, протоколів та підходів до безпеки в IoT-системах;
- спроєктувати гнучку мікросервісну архітектуру системи;
- розробити та програмно реалізувати ключові компоненти системи, включаючи сервіси автентифікації, управління пристроями та збору статистики;
- створити клієнтський веб-додаток з дашбордом для візуалізації даних;
- провести експериментальну перевірку працездатності розробленого прототипу.

Об’єктом дослідження є процес управління та моніторингу гетерогенними IoT-пристроями.

Предметом дослідження є методи, моделі та програмні засоби для інтеграції різнорідних IoT-пристроїв в єдину хмарну веб-платформу.

Для досягнення поставленої мети було використано комплекс наукових методів дослідження. До теоретичних методів належать: аналіз науково-технічної літератури для визначення стану проблеми; порівняльний аналіз для оцінки переваг та недоліків існуючих технологій та платформ; системний аналіз для декомпозиції проблеми на складові. До емпіричних методів належать: методи об'єктно-орієнтованого проєктування для створення логічної структури системи; моделювання для візуалізації архітектури та потоків даних; програмна реалізація та експериментальна перевірка для підтвердження працездатності розроблених рішень.

Наукова новизна одержаних результатів полягає у тому, що отримала подальший розвиток архітектура хмарних IoT-платформ, яка, на відміну від існуючих промислових рішень, орієнтована на кінцевого користувача з низьким порогом входу, та, на відміну від локальних систем для ентузіастів, надає переваги централізованого SaaS-рішення. Новизна полягає у комплексній інтеграції мікросервісного підходу, асинхронної взаємодії на базі брокера повідомлень (RabbitMQ) та використанні високоефективного протоколу gRPC для внутрішньосервісної комунікації з метою створення гнучкої, відмовостійкої, масштабованої та простої у використанні системи.

Практичне значення одержаних результатів полягає у створенні програмного прототипу універсальної веб-системи для моніторингу та керування IoT-пристроями. Розроблена архітектура та програмні компоненти можуть бути використані як основа для створення комерційних та некомерційних продуктів у таких сферах, як «розумний дім», моніторинг невеликих об'єктів (напр., теплиці, склади, сонячні електростанції), а також можуть слугувати навчальною платформою для вивчення принципів побудови сучасних мікросервісних систем у сфері Інтернету речей.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА СУЧАСНИХ РІШЕНЬ У СФЕРІ ІНТЕРНЕТУ РЕЧЕЙ

1.1 Аналіз предметної області та постановка задачі дослідження.

Сучасний технологічний ландшафт характеризується стрімким поширенням пристроїв Інтернету речей (IoT), які інтегруються у всі сфери людської діяльності, від побуту до промисловості. Згідно з аналітичними прогнозами, це зростання має експоненційний характер. Так, аналітики Statista прогнозують, що до 2030 року кількість підключених до мережі IoT-пристроїв перевищить 29 мільярдів одиниць [1]. Водночас економічний вплив цього тренду є не менш значущим: за оцінками Grand View Research, глобальний ринок IoT може досягти вартості у 2,4 трильйона доларів США до 2028 року [2].

Разом із цим зростанням виникає комплексна проблема фрагментації управління та моніторингу, зумовлена значною гетерогенністю IoT-екосистеми. Технічна суть цієї проблеми полягає у відсутності єдиних стандартів: пристрої від різних виробників часто використовують власні, несумісні протоколи зв’язку (Wi-Fi, Zigbee, Bluetooth LE), мають закриті пропрієтарні програмні інтерфейси (API) та передають дані у різних форматах. Це призводить до створення замкнених, ізольованих екосистем, де кожен тип пристроїв вимагає використання окремого фірмового додатку.

Слід зазначити, що поточна проблема фрагментації не є новою, а є результатом тривалої еволюції ринку автоматизації та підключених пристроїв. На ранніх етапах, в епоху до масового поширення Інтернету, системи автоматизації будівель (BMS) та промислового контролю (SCADA) були виключно пропрієтарними, дорогими екосистемами від великих виробників. З появою першої хвилі споживчих «розумних» пристроїв у 2010-х роках ця бізнес-модель була перенесена на масовий ринок: кожен виробник прагнув створити власний закритий "сад" (walled garden), щоб прив'язати користувача до своїх продуктів. Подальше здешевлення мікроелектроніки та хмарних технологій призвело до «кембрійського вибуху» нових IoT-гаджетів, що лише посилило цю тенденцію.

Таким чином, сучасний стан гетерогенності є наслідком ринкових сил та відсутності єдиних стандартів на ранніх етапах розвитку, що робить задачу створення універсальної інтеграційної платформи особливо складною та актуальною [21].

Яскравим прикладом цієї проблеми є типовий сценарій «розумного будинку». Користувач може одночасно експлуатувати систему освітлення Philips Hue, що працює через власний шлюз по протоколу Zigbee, «розумний» термостат Google Nest, підключений через Wi-Fi та інтегрований у власну хмарну екосистему Google, та декілька розеток від TP-Link, що також використовують Wi-Fi, але керуються через окремий мобільний додаток та хмару виробника. У результаті для управління трьома різними системами користувач змушений встановлювати та перемикатися між трьома різними додатками, що є неефективним та незручним. Більш того, створення комплексного сценарію, наприклад, «вимкнути все світло та знизити температуру при виході з дому», стає нетривіальною задачею, що вимагає використання сторонніх сервісів-агрегаторів, які не завжди працюють надійно [3].

Ця проблема не обмежується лише побутовим сегментом. У промисловості (IIoT) та міській інфраструктурі («розумні міста») вона набуває ще більших масштабів. Наприклад, на сучасному підприємстві можуть одночасно функціонувати система моніторингу енергоспоживання на базі датчиків, що передають дані по протоколу MQTT, система відеоспостереження з власним пропрієтарним API та система контролю доступу від іншого виробника. Агрегація даних з цих різнорідних систем для отримання цілісної картини стану об'єкта стає серйозним інженерним викликом, що вимагає значних витрат на розробку та підтримку проміжних програмних шлюзів [4].

Таким чином, існує чітко виражена науково-технічна потреба у створенні універсальних, гнучких програмних платформ, здатних інтегрувати різнорідні пристрої в єдиний інтерфейс управління та моніторингу.

Актуальність вирішення цієї проблеми підтверджується ключовими ринковими та технологічними трендами, описаними вище, а також зростаючим попитом на централізований збір та аналіз даних для оптимізації споживання ресурсів (електроенергії, води), що є важливим економічним та екологічним фактором.

Виходячи з вищеописаної проблеми та її актуальності, метою даної дипломної роботи є розробка та дослідження гнучкої мікросервісної архітектури універсальної веб-системи моніторингу та керування IoT-пристроями, яка вирішує проблему фрагментації шляхом надання єдиного, інтуїтивно зрозумілого інтерфейсу.

Об’єктом дослідження є процес управління та моніторингу гетерогенними IoT-пристроями.

Предметом дослідження є методи та програмні засоби для інтеграції IoT-пристроїв в єдину веб-платформу.

Для досягнення поставленої мети необхідно вирішити наступні задачі:

- проаналізувати існуючі архітектури та протоколи, що використовуються в IoT-системах;
- розробити архітектуру системи, що забезпечує реєстрацію та автентифікацію користувачів;
- реалізувати функціонал для додавання, налаштування та видалення IoT-пристроїв у системі;
- створити інтерфейс для візуалізації даних, що надходять з пристроїв, у режимі реального часу;
- забезпечити можливість збору, зберігання та відображення історичних даних для аналізу статистики роботи пристроїв.

Для демонстрації та валідації функціональних можливостей системи буде розглянуто прикладний сценарій (use-case) її застосування для управління

інфраструктурою «розумного будинку». Практичне значення одержаних результатів полягає у створенні детально описаної архітектури та програмного прототипу, що її валідує. Розроблена архітектура може слугувати шаблоном для створення комерційних та некомерційних продуктів у сферах «розумного дому», моніторингу невеликих об'єктів (напр., теплиці, склади), а також як навчальна платформа для вивчення принципів побудови сучасних мікросервісних систем у сфері Інтернету речей.

1.2 Огляд концепцій та технологій Інтернету речей

Для проєктування та розробки ефективної системи моніторингу та керування IoT-пристроями необхідно провести аналіз фундаментальних концепцій, архітектурних моделей та технологій, що лежать в основі предметної області. Цей підрозділ присвячено огляду узагальненої архітектури IoT-систем, ключових протоколів передачі даних, форматів їх представлення та моделей розгортання платформ[5].

1.2.1 Порівняльний аналіз архітектурних моделей IoT

Для опису та проєктування складних IoT-систем в індустрії використовуються різні архітектурні моделі, що дозволяють декомпозирувати систему на логічні рівні. Найбільш поширеними є трьох- та п'ятирівневі моделі[38].

1.2.1.1 Трьохрівнева архітектурна модель

Трьохрівнева модель є найбільш узагальненою та простою для розуміння. Вона ідеально підходить для високорівневого опису системи та виділяє три основні логічні шари, як показано на рис. 1.1.

1. Рівень сприйняття (Perception Layer): Також відомий як фізичний рівень. Він включає кінцеві пристрої — сенсори та виконавчі механізми (актуатори), які безпосередньо взаємодіють з фізичним середовищем для збору даних або виконання дій.

2. Мережевий рівень (Network Layer): Відповідає за надійну передачу даних, зібраних на першому рівні, до платформи обробки. Цей рівень охоплює всі комунікаційні технології та протоколи, такі як Wi-Fi, Ethernet, Zigbee, а також глобальну мережу Інтернет.
3. Рівень додатків (Application Layer): Верхній рівень архітектури, де відбувається обробка, зберігання, аналіз та візуалізація даних. Саме на цьому рівні функціонують кінцеві додатки, з якими взаємодіє користувач.

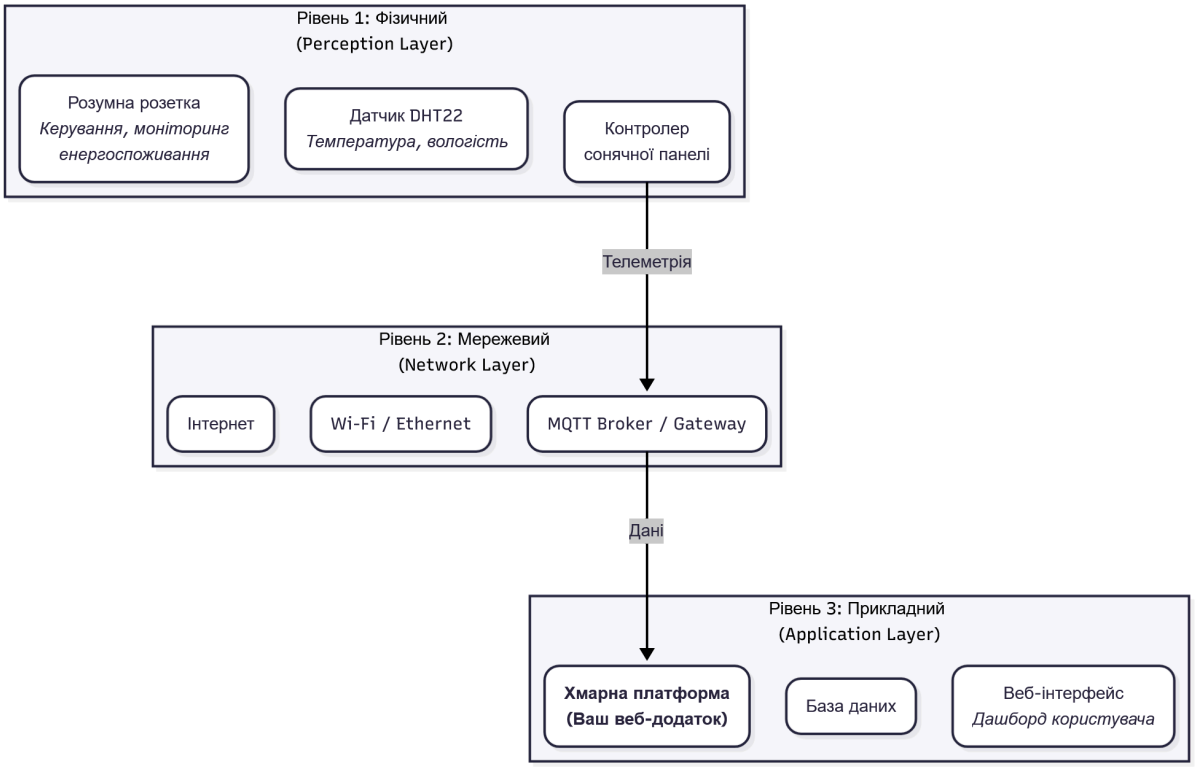


Рисунок 1.1 – Узагальнена трьохрівнева архітектура IoT-системи

Ця модель є корисною для загального розуміння потоку даних, однак її недоліком є недостатня деталізація процесів, що відбуваються на рівні додатків, особливо у складних, розподілених системах [5].

1.2.1.2 П'ятирівнева архітектурна модель

Для більш глибокого аналізу та проєктування складних IoT-рішень часто використовують п'ятирівневу модель(рис 1.2). Вона розширює трьохрівневу, додаючи два проміжні рівні, що дозволяє краще відокремити логіку обробки даних від бізнес-логіки.

1. Рівень об'єктів (Object/Perception Layer): Аналогічний рівню сприйняття у трьохрівневій моделі.
2. Рівень транспортування (Transport Layer): Аналогічний мережевому рівню.
3. Рівень обробки (Processing/Middleware Layer): Це ключовий проміжний рівень. Він відповідає за отримання "сирих" даних з мережевого рівня, їх зберігання, фільтрацію, агрегацію та базову обробку. Тут функціонують бази даних, системи кешування та проміжне програмне забезпечення (middleware).
4. Рівень додатків (Application Layer): Цей рівень використовує вже оброблені дані з попереднього рівня для надання конкретних сервісів користувачеві: візуалізація даних на дашбордах, надсилання сповіщень, виконання команд.
5. Бізнес-рівень (Business Layer): Найвищий рівень, що визначає загальну бізнес-логіку та цінність системи. Він керує правилами, аналітикою, генерує звіти та приймає рішення на основі даних, отриманих з рівня додатків [14].

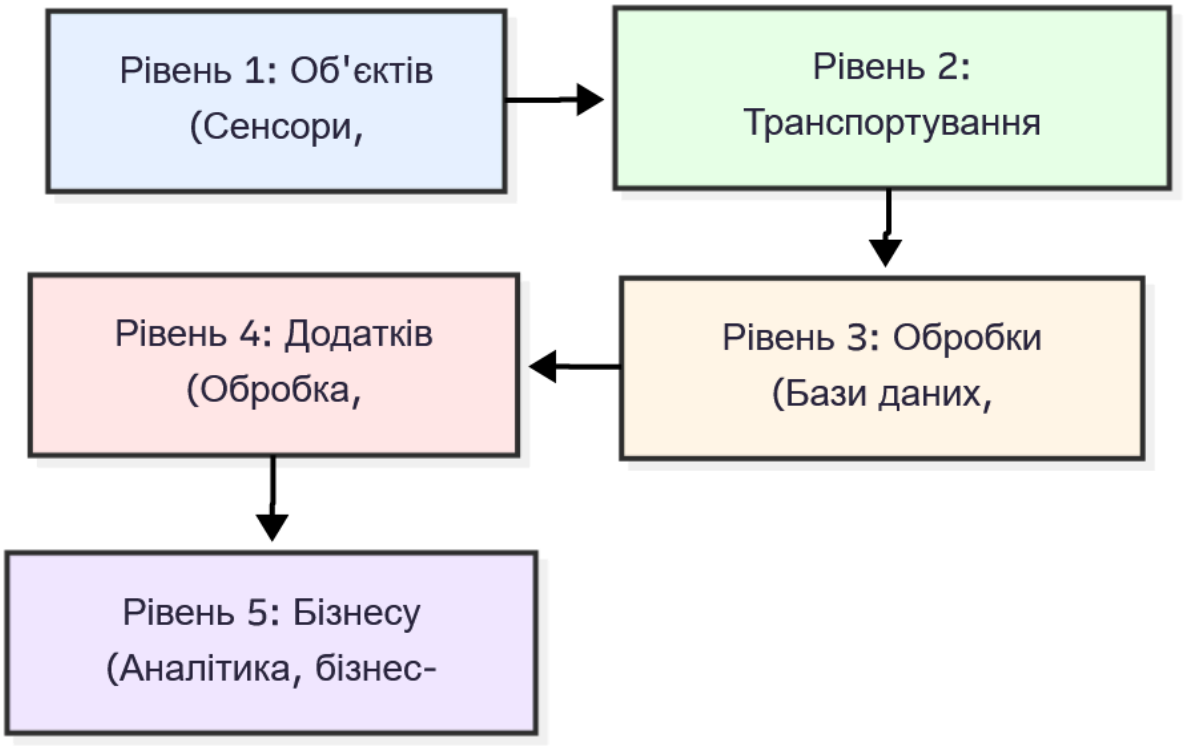


Рисунок 1.2 – Узагальнена п'ятирівнева архітектура IoT-системи

1.2.1.3 Висновки та вибір моделі для аналізу

Трьохрівнева модель є ефективною для загального опису, зрозумілого широкому загалу. Водночас п'ятирівнева модель надає значно більшу деталізацію, що є критично важливим для проєктування складних, масштабованих систем. Вона дозволяє чітко розмежувати відповідальність між компонентами: наприклад, логіку зберігання "сирих" даних (Рівень обробки) та логіку їх представлення користувачеві (Рівень додатків).

Хоча розроблювана в рамках даної роботи система є єдиним веб-додатком, для її теоретичного аналізу та обґрунтування архітектурних рішень у подальших розділах доцільно спиратися саме на п'ятирівневу модель, оскільки вона найбільш точно відображає внутрішню структуру сучасних хмарних IoT-платформ.

1.2.2 Детальний огляд протоколів комунікації

Ефективність, надійність та масштабованість будь-якої IoT-системи безпосередньо залежать від обраних протоколів комунікації. У сучасних розподілених системах, подібних до тієї, що є предметом дослідження, використовуються різні протоколи, кожен з яких оптимізований для вирішення специфічних завдань: комунікації з пристроями, взаємодії між серверними компонентами та доставки даних кінцевому користувачеві.

1.2.2.1 Протокол MQTT для комунікації з пристроями

MQTT (Message Queuing Telemetry Transport) є де-факто стандартом для передачі даних від IoT-пристроїв до сервера. Це легковий бінарний протокол, що функціонує поверх TCP/IP та працює за асинхронною моделлю "видавець-підписник" (publish-subscribe). Ключові особливості MQTT:

Архітектура "видавець-підписник": Пристрої ("видавці") та серверні компоненти ("підписники") не з'єднуються один з одним напряму. Вся комунікація відбувається через центральний компонент — брокер повідомлень. Видавець

відправляє повідомлення у певний "топік", а брокер доставляє це повідомлення всім клієнтам, які підписані на цей топік. Це забезпечує низьку зв'язаність компонентів системи.

Ієрархічна структура топіків: Топіки в MQTT мають ієрархічну структуру, що нагадує файлову систему (напр., sensors/living_room/temperature). Це дозволяє організовувати гнучку маршрутизацію повідомлень. Наприклад, клієнт може підписатися на конкретний топік для отримання лише температури, або на sensors/living_room/#, щоб отримувати дані з усіх датчиків у вітальні.

Рівні якості обслуговування (QoS): MQTT надає три рівні гарантії доставки повідомлень, що дозволяє розробникам знаходити баланс між надійністю та накладними витратами.

- QoS 0 (At most once): "Доставка не більше одного разу". Найшвидший рівень без підтвердження доставки. Оптимальний для некритичних, часто оновлюваних даних телеметрії (наприклад, показники температури).
- QoS 1 (At least once): "Доставка щонайменше один раз". Гарантує, що повідомлення буде доставлено, але допускає можливість дублювання.
- QoS 2 (Exactly once): "Доставка рівно один раз". Найнадійніший, але й найповільніший рівень, що використовує чотирьохетапне рукостискання. Оптимальний для критично важливих команд, де втрата або дублювання є неприпустимими (наприклад, команда на вимкнення обладнання) [6].

1.2.2.2 Протокол HTTP та архітектура REST для веб-сервісів

HTTP (HyperText Transfer Protocol) є основою для передачі даних у Всесвітній павутині. У контексті IoT-платформ він найчастіше використовується для реалізації API (Application Programming Interface) за архітектурним стилем REST (Representational State Transfer). REST API є стандартним способом взаємодії між клієнтськими додатками (напр., веб-інтерфейсом) та серверною частиною.

Чітке розділення відповідальності між клієнтом (інтерфейс користувача) та сервером (зберігання та обробка даних). Це дозволяє розробляти та масштабувати їх незалежно.

Кожен запит від клієнта до сервера повинен містити всю необхідну для його обробки інформацію. Сервер не зберігає стан клієнта між запитами, що підвищує надійність та спрощує масштабування. Для ідентифікації користувача зазвичай використовуються токени доступу (напр., JWT), що передаються у заголовку кожного запиту [7].

REST розглядає будь-які дані як "ресурси" (напр., пристрої, користувачі), кожен з яких має унікальний ідентифікатор (URI). Для маніпуляції цими ресурсами використовуються стандартні методи HTTP: GET (отримання даних), POST (створення нового ресурсу), PUT/PATCH (оновлення ресурсу), DELETE (видалення ресурсу).

1.2.2.3 Протокол gRPC для внутрішньосервісної взаємодії

gRPC (gRPC Remote Procedure Calls) — це сучасний, високоефективний фреймворк для віддаленого виклику процедур, розроблений компанією Google. Він є популярним вибором для побудови комунікації між мікросервісами у розподілених системах. На відміну від REST, що базується на текстовому форматі JSON, gRPC використовує бінарний формат Protocol Buffers (Protobuf) та функціонує поверх протоколу HTTP/2.

Бінарна серіалізація Protobuf є значно швидшою та компактнішою за текстовий JSON, що зменшує затримки та навантаження на мережу. Використання HTTP/2 дозволяє здійснювати мультиплексування запитів через одне TCP-з'єднання.

Структура даних та сервісних методів визначається у .proto файлах. Це створює строгий "контракт" між сервісами. На основі цих файлів можна

автоматично генерувати код клієнтів та серверів для різних мов програмування, що виключає помилки інтеграції [15].

gRPC нативно підтримує двонаправлену потокову передачу даних, що дозволяє реалізовувати складні сценарії взаємодії.

1.2.2.4 Протокол WebSocket для взаємодії в реальному часі

WebSocket — це протокол, що забезпечує встановлення постійного, двонаправленого (full-duplex) з'єднання між клієнтом (зазвичай, веб-браузером) та сервером. Це є його ключовою відмінністю від HTTP, що працює за моделлю "запит-відповідь".

Основна перевага WebSocket — можливість сервера відправляти дані клієнту з власної ініціативи, без необхідності очікувати на запит від нього. Це робить його ідеальним інструментом для оновлення даних в реальному часі, наприклад, для миттєвого відображення на дашборді нових показників, що надійшли від IoT-пристрою. Альтернативний підхід, відомий як HTTP-polling (періодичне опитування сервера), є значно менш ефективним, оскільки створює велике навантаження на мережу та сервер через постійне встановлення нових з'єднань та передачу надлишкових HTTP-заголовків [16].

Для узагальнення та наочного порівняння розглянутих протоколів комунікації, зведемо їх ключові характеристики у таблицю (Таблиця 1.1).

Таблиця 1.1 – Порівняльний аналіз протоколів комунікації

Критерій	MQTT	HTTP/REST	gRPC	WebSocket
Модель комунікації	Видавець-підписник (асинхронна)	Клієнт-сервер (синхронна)	Клієнт-сервер (RPC, стрімінг)	Повнодуплексна (асинхронна)
Формат даних	Бінарний	Текстовий (JSON, XML)	Бінарний (Protobuf)	Бінарний / Текстовий

Накладні витрати	Дуже низькі	Високі (заголовки)	Низькі	Низькі (після handshake)
Основне призначення	Комунікація "пристрій-сервер"	API для веб- та мобільних клієнтів	Внутрішньосервісна комунікація	Взаємодія в реальному часі "сервер-клієнт"
Стан з'єднання	Постійне	Короткочасне	Постійне (HTTP/2)	Постійне

Аналіз таблиці показує, що не існує єдиного "найкращого" протоколу. Вибір конкретної технології залежить від задачі: MQTT є оптимальним для передачі телеметрії з ресурс-обмежених пристроїв, REST API є стандартом для клієнт-серверної взаємодії, gRPC — для швидкої комунікації у мікросервісній архітектурі, а WebSocket — для організації миттєвої доставки даних на користувацькі інтерфейси.

1.2.3 Огляд технологій асинхронної взаємодії на базі брокерів повідомлень

У сучасних розподілених системах, що складаються з великої кількості незалежних мікросервісів, гостро постає проблема організації ефективної та надійної комунікації між ними. Пряма синхронна взаємодія (наприклад, через REST або gRPC), коли один сервіс очікує на відповідь від іншого, може призводити до каскадних збоїв та сильної зв'язаності (coupling) компонентів. Якщо один сервіс виходить з ладу, це може паралізувати роботу всіх сервісів, що від нього залежать.

Для вирішення цієї проблеми широко застосовується підхід на основі асинхронної взаємодії через брокерів повідомлень (Message Brokers).

1.2.3.1 Концепція брокера повідомлень

Брокер повідомлень — це проміжне програмне забезпечення, що функціонує за моделлю "видавець-підписник". Сервіси-продюсери ("видавці")

відправляють повідомлення (події) у брокер, не знаючи, хто і коли їх отримає. Інші сервіси-консюмери ("підписники") підписуються на певні типи повідомлень і обробляють їх по мірі надходження.

Такий підхід забезпечує слабку зв'язаність (loose coupling) та часову незалежність (temporal decoupling). Сервісам не потрібно знати про існування один одного; вони взаємодіють лише з брокером. Продюсер може відправити повідомлення навіть тоді, коли консюмер тимчасово недоступний. Брокер збереже повідомлення у черзі та доставить його, коли консюмер відновить роботу. Це значно підвищує відмовостійкість всієї системи [8].

1.2.3.2 Огляд RabbitMQ як реалізації протоколу AMQP

Одним з найпопулярніших брокерів повідомлень є RabbitMQ. Це надійне та гнучке рішення з відкритим вихідним кодом, яке реалізує протокол AMQP (Advanced Message Queuing Protocol).

Ключові компоненти архітектури RabbitMQ:

- Producer: Додаток, що відправляє повідомлення.
- Exchange (Обмінник): Отримує повідомлення від продюсерів та направляє їх у відповідні черги на основі правил маршрутизації. Існують різні типи обмінників (direct, topic, fanout), що дозволяє реалізовувати складні сценарії доставки.
- Queue (Черга): Буфер, що зберігає повідомлення до моменту їх обробки консюмером.
- Consumer: Додаток, що підписується на чергу, отримує з неї повідомлення та обробляє їх.

У контексті IoT-платформи, така архітектура є надзвичайно ефективною. Наприклад, сервіс, що приймає дані з пристроїв, може опублікувати подію "отримано нові дані телеметрії" в RabbitMQ. На цю подію можуть бути підписані одразу декілька незалежних сервісів: один для збереження даних у базу, другий

для аналізу на аномалії, третій для оновлення статусу в реальному часі. Це дозволяє легко додавати нову функціональність, не змінюючи код існуючих сервісів — достатньо лише додати нового підписника [16].

1.2.4 Фундаментальні аспекти безпеки в IoT-системах

Безпека є одним з найважливіших аспектів при проєктуванні будь-якої IoT-системи. Велика кількість підключених пристроїв створює численні вектори для атак.

Фундаментальним механізмом захисту даних "в дорозі" (data-in-transit) є використання протоколу шифрування TLS (Transport Layer Security), який є наступником SSL (Secure Sockets Layer). TLS працює на транспортному рівні моделі OSI і забезпечує встановлення захищеного, зашифрованого каналу зв'язку між двома сторонами (наприклад, між IoT-пристроєм та сервером, або між браузером та веб-сервером). Він гарантує три ключові аспекти безпеки: конфіденційність - дані шифруються і не можуть бути прочитані третьою стороною, цілісність - захист від модифікації даних під час передачі та автентифікацію - перевірка справжності сервера, а іноді й клієнта, за допомогою цифрових сертифікатів. Використання захищених версій протоколів, таких як MQTTS, HTTPS та WSS, передбачає інкапсуляцію їх трафіку всередину TLS-з'єднання, що є обов'язковою вимогою для будь-якої сучасної та безпечної системи [22].

Ключовими механізмами захисту на рівні додатків є автентифікація та авторизація як користувачів, так і пристроїв.

1.2.4.1 Автентифікація та авторизація на базі JWT

У сучасних веб-сервісах, особливо в системах з розділеною клієнт-серверною або мікросервісною архітектурою, класичний підхід на базі сесій є не завжди ефективним. На зміну йому прийшов підхід на базі токенів доступу, найпопулярнішим стандартом для яких є JWT (JSON Web Token).

JWT — це відкритий стандарт (RFC 7519) для створення компактних та самодостатніх токенів, що використовуються для передачі інформації між сторонами як JSON-об'єкт. Ця інформація може бути перевірена та є довіреною, оскільки вона має цифровий підпис. Токен складається з трьох частин:

1. Заголовок (Header): Містить інформацію про тип токена та алгоритм підпису.
2. Корисне навантаження (Payload): Містить "твердження" (claims) — інформацію про користувача (наприклад, userId, role) та метадані токена (наприклад, термін дії exp).
3. Підпис (Signature): Створюється на основі заголовка, корисного навантаження та секретного ключа, що зберігається на сервері. Підпис гарантує, що токен не було змінено після видачі.

Схема автентифікації з JWT виглядає наступним чином:

1. Користувач надсилає свої облікові дані (логін/пароль) на сервер автентифікації.
2. Сервер перевіряє дані і, у разі успіху, генерує JWT та відправляє його клієнту.
3. Клієнт зберігає токен (наприклад, у localStorage або cookies) і додає його до заголовка Authorization (зазвичай як Bearer <token>) у кожному наступному запиті до захищених ресурсів API.
4. Сервер, отримуючи запит, перевіряє валідність підпису токена і, якщо все гаразд, надає доступ до ресурсу.

Цей підхід є "stateless", тобто, не вимагає зберігання сесій на сервері, що ідеально підходить для масштабованих мікросервісних систем [9].

1.2.4.2 Протокол OAuth 2.0 та інтеграція зі сторонніми провайдерами

Часто користувачам зручніше не створювати новий акаунт, а використовувати вже існуючий в одного з великих провайдерів, наприклад,

Google, Facebook та GitHub. Для безпечної реалізації такої функціональності використовується протокол OAuth 2.0.

OAuth 2.0 — це відкритий стандарт для делегованої авторизації. Він дозволяє додатку (у нашому випадку, IoT-платформі) отримати обмежений доступ до акаунту користувача на сторонньому HTTP-сервісі без отримання його паролю.

Стандартний потік "Authorization Code Grant" виглядає так:

1. Додаток перенаправляє користувача на сторінку провайдера ідентифікації (напр., Google).
2. Користувач вводить свої логін та пароль на сайті провайдера і надає згоду на доступ додатку до певної інформації (напр., email та ім'я).
3. Провайдер перенаправляє користувача назад до додатку, передаючи тимчасовий "код авторизації".
4. Додаток на своєму бекенді обмінює цей код на "токен доступу" (access token), звернувшись безпосередньо до сервера провайдера.
5. Використовуючи цей токен, додаток може отримати інформацію про користувача (напр., його email) та або створити новий акаунт у своїй системі, або знайти існуючий.

Такий підхід значно підвищує безпеку, оскільки додаток ніколи не бачить і не зберігає пароль користувача від стороннього сервісу. Крім того, це значно спрощує процес реєстрації та входу для кінцевого користувача [18].

1.2.5 Огляд технологій зберігання даних в IoT-системах

IoT-системи генерують величезні обсяги різнорідних даних, що висуває особливі вимоги до систем їх зберігання. Дані можна умовно поділити на три категорії: метадані пристроїв та користувачів, дані часових рядів (телеметрія) та тимчасові дані (кеш, сесії). Для кожної з цих категорій оптимальними є різні технології баз даних.

1.2.5.1 Реляційні та NoSQL бази даних для зберігання метаданих

Метадані — це структурована інформація про сутності системи, такі як користувачі, пристрої, їх налаштування та групи. Ці дані характеризуються чітко визначеними зв'язками (наприклад, "користувач володіє багатьма пристроями", "пристрій належить до однієї групи").

Реляційні бази даних, такі як PostgreSQL, MySQL. Вони є класичним вибором для зберігання метаданих завдяки строгій схемі, підтримці транзакцій (ACID) та потужним можливостям для виконання складних запитів зі зв'язуванням таблиць. Вони гарантують високу цілісність та консистентність даних.

NoSQL бази даних, наприклад, MongoDB (документо-орієнтована), Cassandra (колонко-орієнтована). Вони пропонують більшу гнучкість схеми даних та кращу горизонтальну масштабованість, що може бути перевагою у дуже великих системах. Однак вони зазвичай надають слабші гарантії консистентності даних порівняно з SQL-базами [9].

Вибір між SQL та NoSQL для метаданих залежить від пріоритетів проєкту: гарантована цілісність даних проти гнучкості та масштабованості.

1.2.5.2 Часо-рядні бази даних (Time-Series DB) для телеметрії

Дані телеметрії (показники сенсорів, статистика роботи) є часовими рядами — послідовністю точок даних, індексованих за часом. Вони мають специфічні характеристики: переважна операція — запис, дані рідко оновлюються, запити зазвичай агрегують дані за певний проміжок часу.

Для таких навантажень традиційні реляційні БД є неефективними. Спеціалізовані часо-рядні бази даних (TSDB), такі як InfluxDB, TimescaleDB, Prometheus, розроблені спеціально для вирішення цих завдань.

До ключових переваг TSDB передусім відносять високу продуктивність запису та компресію даних. Використання спеціалізованих алгоритмів дозволяє

ефективно записувати великі потоки інформації та суттєво економити дисковий простір.

Іншою важливою перевагою є наявність оптимізованих функцій для роботи з часом. Вони забезпечують швидку агрегацію даних за визначеними інтервалами, дозволяють виконувати проріджування (downsampling), а також керувати життєвим циклом даних через політики зберігання (retention policies) [19].

1.2.5.3 Системи кешування в пам'яті

Для підвищення швидкодії та зменшення навантаження на основні бази даних широко використовуються системи кешування в пам'яті (in-memory data stores), такі як Redis. Redis — це надзвичайно швидке сховище типу "ключ-значення", яке зберігає дані в оперативній пам'яті.

Серед типових сценаріїв використання Redis в IoT-платформах передусім виділяють кешування «гарячих» даних, наприклад, поточних статусів пристроїв. Це дозволяє мінімізувати кількість звернень до повільної дискової бази даних. Також система забезпечує ефективну роботу з сесіями та токенами, гарантуючи швидке збереження та верифікацію даних користувачів або JWT.

Крім функцій сховища, Redis часто застосовується як легковагий брокер повідомлень. Завдяки вбудованим механізмам Pub/Sub, його доцільно використовувати для організації простої асинхронної комунікації між окремими сервісами системи [20].

1.3 Аналіз існуючих платформ та програмних рішень

Для обґрунтування доцільності розробки та визначення ринкової ніші нової системи необхідно провести детальний аналіз існуючих на ринку програмних рішень. Цей аналіз дозволить виявити їхні архітектурні особливості, сильні та слабкі сторони. Для всебічного огляду було обрано три репрезентативні

платформи, що представляють різні сегменти ринку: Home Assistan - локальна open-source система для ентузіастів, AWS IoT Core - глобальна хмарна платформа корпоративного рівня та ThingsBoard - enterprise-ready open-source платформа.

1.3.1 Огляд платформи Home Assistant

Home Assistant — це безкоштовна платформа з відкритим вихідним кодом, що розповсюджується за моделлю локального розгортання (self-hosted). Вона є лідером у сегменті "розумного будинку" для технічних ентузіастів завдяки своїй гнучкості та широким можливостям кастомізації.

Платформа має монолітну архітектуру, написану на Python. Ядро системи відповідає за управління пристроями, автоматизаціями та користувацьким інтерфейсом. Ключовою особливістю є система інтеграцій, яка дозволяє підключати тисячі різноманітних пристроїв та сервісів [10].

Головною перевагою Home Assistant є повна незалежність від сторонніх хмар та максимальний рівень приватності даних, оскільки вся система функціонує в межах локальної мережі користувача. Величезна та активна спільнота забезпечує постійний розвиток та підтримку широкого спектру обладнання.

Основним бар'єром для масового користувача є необхідність мати та самостійно адмініструвати власне серверне обладнання (напр., Raspberry Pi). Процес налаштування та підтримки вимагає технічних знань, а відсутність вбудованих інструментів для багатокористувацького доступу (multi-tenancy) робить її непридатною для створення комерційних сервісів на її основі [11].

Home Assistant ідеально розкриває свій потенціал у руках технічного ентузіаста для створення складних, кастомізованих автоматизацій. Наприклад, користувач може реалізувати сценарій "симуляції присутності" під час відпустки. Система, інтегруючись з онлайн-календарем, автоматично активує цей режим. Ввечері вона за допомогою двигуна автоматизацій хаотично вмикає та вимикає світло в різних кімнатах, використовуючи пристрої різних виробників, та

відтворює звуки на розумних колонках, створюючи ілюзію присутності господарів. Реалізація такого глибоко персоналізованого сценарію є сильною стороною платформи.

1.3.2 Огляд корпоративних хмарних платформ на прикладі AWS IoT Core

AWS IoT Core є флагманським сервісом у портфоліо Amazon Web Services для побудови IoT-рішень. Це керована хмарна платформа, розроблена для вирішення завдань корпоративного та промислового рівнів з мільйонами підключених пристроїв.

AWS IoT Core є набором взаємопов'язаних сервісів, а не монолітним додатком. Основні компоненти включають: брокер повідомлень (Message Broker), що підтримує MQTT та WebSocket; механізм автентифікації та авторизації (Device Gateway); та двигун правил (Rules Engine), що дозволяє маршрутизувати дані з пристроїв в інші сервіси AWS (напр., бази даних, сховища, сервіси машинного навчання) [12].

Ключовою перевагою є практично нескінченна горизонтальна масштабованість та висока надійність, що гарантується інфраструктурою AWS. Глибока інтеграція з десятками інших сервісів екосистеми дозволяє створювати надзвичайно складні та потужні рішення для аналізу даних.

Для індивідуальних розробників та малих проєктів платформа є надлишковою. Складна та гранулярна модель ціноутворення робить фінальні витрати важкопрогнозованими. Високий поріг входу вимагає глибоких знань у сфері хмарних обчислень, а інтерфейси орієнтовані на інженерів, а не на кінцевих користувачів.

AWS IoT Core є оптимальним вибором для великих промислових проєктів, таких як предиктивне обслуговування обладнання. Уявімо виробниче підприємство з тисячами станків, обладнаних датчиками вібрації та температури. Дані з цих датчиків у реальному часі надходять до AWS IoT Core. За допомогою

двигуна правил (Rules Engine) цей потік даних автоматично направляється в сервіс машинного навчання Amazon SageMaker, де заздалегідь навчена модель аналізує патерни та прогнозує ймовірність відмови обладнання. У разі виявлення аномалії система автоматично створює заявку для сервісної служби. Цей сценарій демонструє силу AWS у роботі з великими даними та інтеграції з потужною екосистемою.

1.3.3 Огляд платформи ThingsBoard

ThingsBoard — це open-source IoT-платформа, призначена для збору, обробки, візуалізації та управління даними з IoT-пристроїв. На відміну від Home Assistant, вона з самого початку спроєктована для вирішення завдань бізнесу та може бути розгорнута як локально (on-premise), так і в хмарі.

ThingsBoard має мікросервісну архітектуру, написану на Java. Це дозволяє гнучко масштабувати окремі компоненти системи. Ключовими елементами є потужний двигун правил (Rule Engine), що дозволяє обробляти вхідні дані "на льоту" без написання коду, та гнучкий конструктор дашбордів. Важливою особливістю є вбудована підтримка багатокористувацького доступу, що дозволяє створювати ізольовані середовища для різних клієнтів в рамках однієї інсталяції платформи [13].

Платформа поєднує переваги обох світів. З одного боку, як open-source рішення, вона надає можливість розгортання на власному сервері для повного контролю над даними та інфраструктурою. З іншого боку, наявність enterprise-функцій, таких як multi-tenancy та потужний Rule Engine, робить її готовим рішенням для створення комерційних IoT-додатків.

Хоча ThingsBoard є більш дружнім до користувача, ніж AWS IoT, його налаштування та адміністрування все ще вимагає значних технічних компетенцій, особливо при розгортанні у відмовостійкому кластері. Для індивідуального

користувача, якому не потрібні enterprise-функції, його можливості можуть бути надлишковими.

ThingsBoard чудово підходить для створення комерційних B2B-сервісів. Наприклад, компанія-стартап надає послуги моніторингу парку рефрижераторних вантажівок. Використовуючи ThingsBoard, розгорнутий на власному сервері, компанія створює ізольовані акаунти для кожного свого клієнта завдяки функції multi-tenancy. Дані з GPS-трекерів та датчиків температури надходять на платформу. За допомогою двигуна правил налаштовується логіка: якщо температура у вантажівці виходить за межі норми, система автоматично надсилає SMS-сповіщення водію та email-повідомлення менеджеру. Клієнти, в свою чергу, мають доступ до кастомізованих дашбордів, де бачать стан лише свого автопарку.

1.3.4 Порівняльний аналіз та висновки

Зведемо ключові характеристики розглянутих платформ у порівняльну таблицю (Таблиця 1.2).

Таблиця 1.2 – Порівняльний аналіз IoT-платформ

Критерій	Home Assistant	AWS IoT Core	ThingsBoard
Цільова аудиторія	Технічні ентузіасти	Великі корпорації	Бізнес (від СМБ до enterprise)
Модель розгортання	Локальна (Self-hosted)	Публічна хмара (SaaS/PaaS)	Локальна / Хмарна
Поріг входу	Високий	Дуже високий	Середній-високий
Multi-tenancy	Відсутня	Присутня	Присутня
Основна перевага	Приватність, гнучкість	Масштабованість, екосистема	Enterprise-функції, гнучкість розгортання
Основний недолік	Складність для новачка	Складність, вартість	Надлишковість для простих задач

Проведений аналіз трьох різних типів платформ підтверджує висновок, зроблений у підрозділі 1.2. Існує незайнята ніша для створення публічної хмарної SaaS-платформи, орієнтованої не на корпорації чи технічних ентузіастів, а на масового кінцевого користувача. Таке рішення має взяти найкраще від розглянутих аналогів: простоту використання, властиву хмарним сервісам (як AWS IoT), але з низьким порогом входу, зрозумілою моделлю використання та функціоналом, сфокусованим на потребах індивідуального користувача (як Home Assistant), без надлишкових enterprise-функцій (як у ThingsBoard). Саме цю нішу і покликана зайняти система, що розроблюється.

1.4 Висновки

У даному розділі було проведено комплексний аналіз предметної області Інтернету речей, огляд сучасних технологій та існуючих програмних рішень, що дозволило сформулювати проблему, визначити мету та задачі дослідження, а також обґрунтувати актуальність та доцільність розробки нової системи моніторингу та керування IoT-пристроями.

За результатами аналізу предметної області було встановлено, що ключовою проблемою, яка стримує масове впровадження IoT-технологій, є висока гетерогенність та фрагментація ринку. Відсутність єдиних стандартів призводить до створення замкнених екосистем від різних виробників, що змушує користувачів взаємодіяти з десятками несумісних додатків та унеможливлює створення комплексних сценаріїв автоматизації. Актуальність вирішення цієї проблеми підтверджується стрімким кількісним (прогнозується понад 29 мільярдів пристроїв до 2030 року) та економічним (прогнозований обсяг ринку понад 2 трлн доларів) зростанням ринку IoT.

В ході огляду сучасних технологій було розглянуто фундаментальні концепції, що лежать в основі побудови IoT-систем. Проведено порівняльний аналіз трьох- та п'ятирівневої архітектурних моделей, що дозволило визначити

останню як найбільш релевантну для проєктування складних систем. Детально проаналізовано ключові протоколи комунікації: MQTT як стандарт для взаємодії з пристроями, HTTP/REST та gRPC для побудови API та внутрішньосервісної комунікації, та WebSocket для забезпечення взаємодії в реальному часі. Також було розглянуто сучасні підходи до забезпечення безпеки на базі JWT та OAuth 2.0 та зберігання даних з використанням реляційних, NoSQL та спеціалізованих часо-рядних баз даних.

Аналіз існуючих програмних платформ, таких як, Home Assistant, AWS IoT Core, ThingsBoard показав, що на ринку існує значний розрив між різними класами рішень. Локальні open-source системи, як Home Assistant, надають високий рівень гнучкості та приватності, але є занадто складними для нетехнічних користувачів. Глобальні хмарні платформи, як AWS IoT Core, є потужними та масштабованими, але надлишковими, дорогими та недружніми до індивідуального розробника. Enterprise-ready рішення, як ThingsBoard, хоча й пропонують гнучкість розгортання, все ж орієнтовані на бізнес-користувачів та вимагають значних компетенцій для адміністрування.

Таким чином, головним висновком проведеного аналізу є обґрунтування необхідності створення хмарної SaaS-платформи, яка б поєднувала простоту використання та відсутність потреби в адмініструванні з функціоналом, орієнтованим на потреби масового кінцевого користувача. Сформульовані в ході аналізу вимоги до такої системи (функціональні, архітектурні та технологічні) є вихідними даними для подальшого проєктування та розробки.

2. ОБҐРУНТУВАННЯ ВИБОРУ МЕТОДІВ ТА ЗАСОБІВ ДЛЯ РОЗРОБКИ СИСТЕМИ

2.1 Аналіз можливих методів і підходів до вирішення задачі

Розробка сучасної хмарної IoT-платформи є комплексним завданням, що вимагає прийняття низки фундаментальних архітектурних рішень. Від цих рішень залежить гнучкість, масштабованість, надійність та зручність підтримки майбутньої системи. У даному підрозділі проводиться аналіз ключових альтернативних підходів до проєктування серверної та клієнтської частин системи, а також до організації взаємодії між її компонентами.

2.1.1 Аналіз архітектурних підходів до побудови серверної частини: моноліт проти мікросервісів

При проєктуванні бекенду існують два основні архітектурні стилі: монолітний та мікросервісний.

У монолітному підході весь функціонал програми реалізовано в рамках єдиного, неподільного додатку. Всі компоненти, наприклад, автентифікація, управління пристроями, збір статистики тісно пов'язані між собою та використовують спільну базу коду і, як правило, спільну базу даних.

Основною перевагою моноліту є простота розробки та розгортання на початкових етапах проєкту. Відсутність мережевої взаємодії між компонентами спрощує тестування та налагодження.

З ростом функціоналу моноліт стає складним для розуміння та модифікації. Виникає сильна технологічна зв'язаність — зміна фреймворку чи мови програмування вимагає переписування всього додатку. Ключовим недоліком є складність масштабування. Якщо один компонент (наприклад, збір статистики) зазнає високого навантаження, необхідно розгортати додаткові копії всього додатку, що є неефективним з точки зору використання ресурсів.

Мікросервісний підхід передбачає декомпозицію системи на набір невеликих, незалежних сервісів, кожен з яких відповідає за конкретну бізнес-функцію (наприклад, "Auth Service", "IoT Broker Service"). Кожен сервіс може бути розроблений, розгорнутий та масштабований незалежно від інших.

Головною перевагою є гнучкість масштабування. Наприклад, під час релізу продукту може виникнути значний наплив нових користувачів, що створює навантаження на сервіс автентифікації. У мікросервісній архітектурі можна легко додати додаткові екземпляри лише цього сервісу. Згодом, коли навантаження зміститься на прийом даних від пристроїв, можна аналогічно масштабувати "IoT Broker Service". Це забезпечує оптимальне використання ресурсів. Також цей підхід надає технологічну гнучкість (різні сервіси можуть бути написані на різних мовах) та підвищує відмовостійкість (збій одного сервісу не обов'язково призводить до відмови всієї системи).

Операційна складність значно вища порівняно з монолітом. Виникає необхідність у додаткових інструментах для розгортання, моніторингу та оркестрації сервісів. Мережеві затримки та розподілене тестування також є значними викликами [23].

Для порівняння цих підходів зведемо їх характеристики у таблицю.

Таблиця 2.1 – Порівняння монолітної та мікросервісної архітектур

Критерій	Монолітна архітектура	Мікросервісна архітектура
Масштабованість	Низька (вертикальна, масштабується вся система)	Висока (горизонтальна, масштабуються окремі сервіси)
Відмовостійкість	Низька (збій компонента може зупинити все)	Висока (ізоляція збоїв)
Складність розробки	Низька на старті, висока при зростанні	Висока на старті, керована при зростанні

Гнучкість стеку	Низька (єдиний стек для всієї системи)	Висока (різні технології для різних сервісів)

Для наочної демонстрації фундаментальних відмінностей між розглянутими підходами, наведемо їх узагальнені схеми на рис. 2.1.

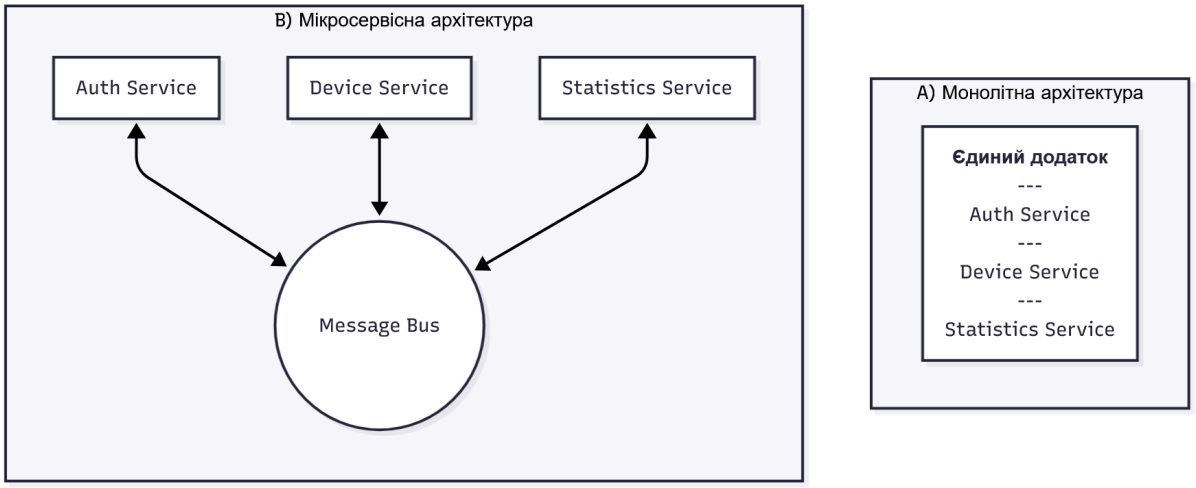


Рисунок 2.1. Порівняння монолітної (A) та мікросервісної (B) архітектур

Як видно зі схеми, у монолітній архітектурі всі компоненти тісно пов'язані, тоді як у мікросервісній вони є незалежними та взаємодіють через спільну шину повідомлень, що забезпечує гнучкість та відмовостійкість.

2.1.2 Аналіз методів внутрішньосервісної взаємодії: синхронний проти асинхронного

У мікросервісній архітектурі вибір способу комунікації між сервісами є критично важливим.

При синхронному підході один сервіс робить прямий мережевий виклик до іншого (наприклад, через REST або gRPC) і блокує свою роботу, очікуючи на відповідь.

До переваг можна віднести, що логіка такої взаємодії є простою та зрозумілою та сервіс-клієнт отримує миттєвий зворотний зв'язок про успіх чи невдачу операції.

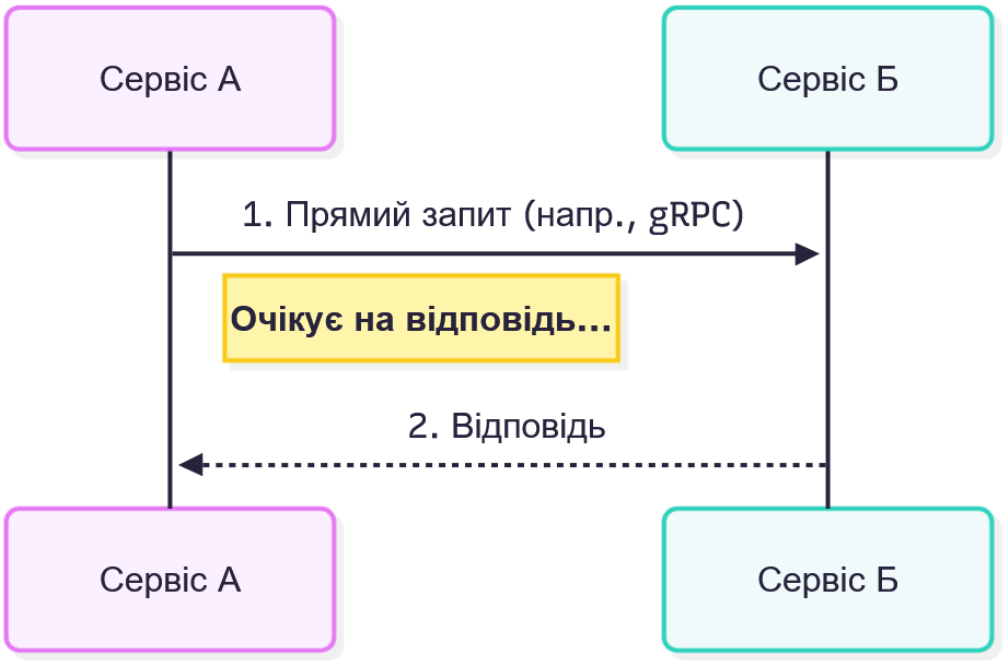
До недоліків, що створюється сильна зв'язаність (tight coupling) між сервісами. Якщо сервіс-виконавець тимчасово недоступний або перевантажений, сервіс-клієнт також буде заблокований, що може призвести до каскадних збоїв у системі. Наприклад, якщо "IoT Broker Service" синхронно викликає "Statistics Service", то при перевантаженні останнього перший перестане приймати дані від пристроїв, що призведе до їх втрати [24].

У асинхронному підході сервіси не взаємодіють напряму, а обмінюються повідомленнями (подіями) через проміжний компонент — брокер повідомлень (напр., RabbitMQ)[39].

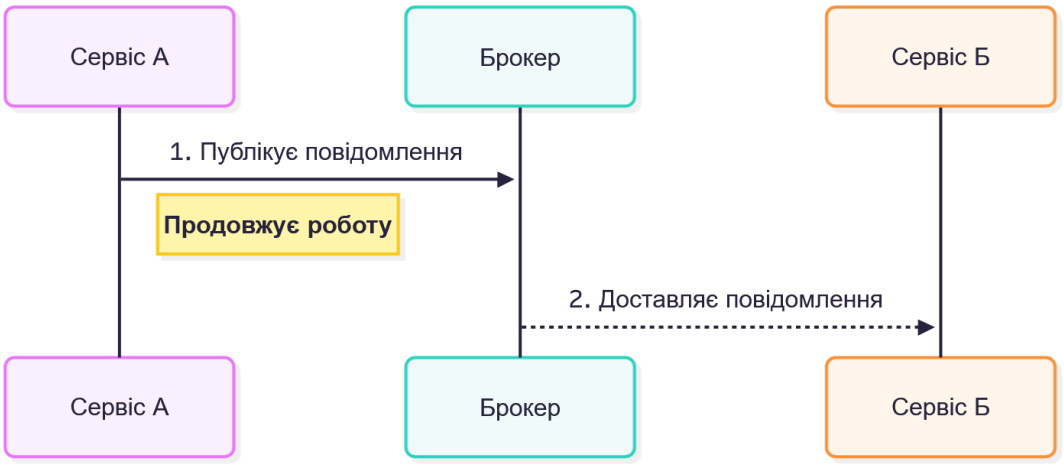
Забезпечується слабка зв'язаність (loose coupling). "IoT Broker Service" просто відправляє повідомлення у брокер і продовжує свою роботу. "Statistics Service" обробляє повідомлення з черги у власному темпі. Брокер виступає як буфер, що робить систему стійкою до тимчасових перевантажень та збоїв окремих сервісів, гарантуючи, що жодне повідомлення не буде втрачено.

Архітектура стає значно складнішою. Виникає проблема "кінцевої консистентності" (eventual consistency), оскільки дані оновлюються не миттєво.

Різницю між синхронним та асинхронним підходами можна продемонструвати за допомогою діаграм послідовності (рис. 2.2).



А) Синхронна взаємодія



В) Асинхронна взаємодія

Рисунок 2.2 – Порівняння синхронної (А) та асинхронної (В) моделей взаємодії

На діаграмі А показано, що Сервіс А блокує свою роботу до отримання відповіді від Сервісу Б. На діаграмі Б Сервіс А не очікує на відповідь, а лише відправляє повідомлення у брокер, що робить систему стійкою до затримок та збоїв Сервісу Б.

2.1.3 Аналіз архітектурних підходів до побудови клієнтської частини: MPA проти SPA

Архітектура фронтенду визначає користувацький досвід та ефективність взаємодії з платформою.

Багатосторінковий додаток (MPA - Multi-Page Application) - це традиційний підхід, де кожна дія користувача, наприклад, перехід на іншу сторінку призводить до повного перезавантаження HTML-сторінки з сервера.

До перевах відносять: простоту реалізації на початкових етапах та хороша індексацію пошуковими системами (SEO).

Головним недоліком для інтерактивних систем, таких як IoT-дашборд, є низька продуктивність та "рваний" користувацький досвід. Необхідність повного перезавантаження сторінки для оновлення одного графіка є неприйнятною для системи моніторингу в реальному часі.

У випадку з односторінковим додатком (SPA - Single-Page Application) весь додаток (HTML, CSS, JavaScript) завантажується один раз. Надалі всі взаємодії з сервером відбуваються динамічно через API без перезавантаження сторінки.

SPA забезпечує швидкий, плавний та безшовний користувацький досвід, що нагадує роботу з нативним десктопним додатком. Це дозволяє миттєво оновлювати окремі компоненти інтерфейсу (віджети, статуси) у реальному часі. Також такий підхід забезпечує чітке розділення відповідальності між фронтендом, який відповідає лише за візуалізацію, та бекендом, що надає дані через API.

Початкова розробка може бути складнішою, а для забезпечення SEO-оптимізації можуть знадобитися додаткові техніки (напр., Server-Side Rendering) [25].

2.2 Обґрунтування вибору методу дослідження

На основі проведеного в попередньому підрозділі аналізу можливих методів та підходів, у даному розділі здійснюється обґрунтований вибір комплексної методики проєктування та розробки системи. Обрана методика базується на комбінації сучасних архітектурних підходів, які, незважаючи на свою початкову складність, є найбільш доцільними для досягнення поставленої мети — створення гнучкої, масштабованої та надійної IoT-платформи.

2.2.1 Обґрунтування вибору мікросервісної архітектури

Для побудови серверної частини системи було обрано мікросервісну архітектуру. Незважаючи на те, що монолітний підхід є простішим на початковому етапі, його недоліки стають критичними при зростанні системи. Обраний напрям розробки є стратегічним: платформа з самого початку проєктується з розрахунком на майбутнє зростання кількості користувачів, пристроїв та функціоналу. У таких умовах гнучкість масштабування є значно більш пріоритетною, ніж швидкість початкової розробки.

Мікросервісний підхід дозволяє ефективно вирішувати низку ключових архітектурних завдань, насамперед забезпечуючи незалежне масштабування компонентів. Це дає можливість гнучко розподіляти ресурси, збільшуючи потужність лише тих сервісів, які зазнають пікового навантаження (наприклад, модуль автентифікації або прийому даних), що є критично важливим для раціонального використання інфраструктури.

Крім того, така архітектура гарантує підвищену відмовостійкість та технологічну гнучкість. Ізоляція процесів означає, що збій в одному сервісі не призводить до зупинки всієї системи, підвищуючи загальну надійність платформи. Водночас розробники отримують свободу вибору найбільш відповідного технологічного стека для кожного окремого модуля, що дозволяє оптимізувати продуктивність та пришвидшити процес розробки.

Таким чином, вибір мікросервісної архітектури є виправданим компромісом між початковою складністю та довгостроковою життєздатністю і гнучкістю проєкту.

2.2.2 Обґрунтування вибору асинхронної взаємодії

Для організації комунікації між мікросервісами було обрано асинхронний, подієво-орієнтований підхід на базі брокера повідомлень. Цей вибір продиктований однією з ключових вимог до IoT-платформи — гарантованою доставкою даних від пристроїв. Втрата телеметрії через тимчасові перевантаження чи збої окремих сервісів є неприпустимою для системи моніторингу.

Використання синхронної взаємодії несе ризик каскадних збоїв, коли відмова одного сервісу блокує роботу інших. Натомість асинхронний підхід дозволяє нівелювати цю проблему, забезпечуючи слабку зв'язаність (loose coupling) системи. Завдяки відсутності прямих залежностей між компонентами значно спрощується процес їх незалежної розробки та подальшої модифікації.

Також асинхронна модель гарантує ефективну буферизацію та стійкість до пікових навантажень. Брокер повідомлень виступає проміжною ланкою, що згладжує нерівномірний потік даних і дозволяє обробникам працювати у власному темпі без ризику втрати інформації. Це також забезпечує надійність доставки: навіть якщо сервіс-споживач тимчасово недоступний, повідомлення зберігаються в черзі й будуть оброблені одразу після відновлення його роботи.

Хоча впровадження брокера повідомлень ускладнює архітектуру, переваги у надійності та відмовостійкості, які він надає, є критично важливими для успішного функціонування системи.

2.2.3 Обґрунтування вибору архітектури односторінкового додатку (SPA)

Для побудови клієнтської частини було обрано архітектуру односторінкового додатку (SPA). Основна цінність розроблюваної платформи для кінцевого користувача полягає у можливості комфортно та ефективно моніторити стан системи в режимі реального часу. Традиційний багатосторінковий підхід (MPA) з повним перезавантаженням сторінок при кожній взаємодії створює "рваний" та повільний користувацький досвід, що повністю нівелює цю цінність.

Архітектура SPA дозволяє забезпечити плавний та швидкий інтерфейс, наближений за чутливістю до десктопних додатків. Це досягається завдяки динамічному оновленню лише окремих компонентів сторінки (віджетів, графіків, статусів) без необхідності її повного перезавантаження. Крім того, такий підхід забезпечує ефективну роботу з даними в реальному часі, адже SPA ідеально інтегрується з технологіями на кшталт WebSocket для миттєвого відображення змін, отриманих від сервера.

Іншою суттєвою перевагою є чітке розмежування зон відповідальності. Розробка клієнтської та серверної частин може відбуватися паралельно й незалежно, оскільки їх взаємодія регламентується строго визначеним контрактом, найчастіше реалізованим через REST API.

Таким чином, вибір SPA є безальтернативним для створення сучасного, інтерактивного та конкурентоздатного користувацького інтерфейсу для IoT-платформи.

2.2.4 Обґрунтування вибору комплексної методики як основи наукової новизни

Слід зазначити, що наукова новизна даної роботи полягає не у винаході окремих алгоритмів чи протоколів, а у розробці та обґрунтуванні комплексної методики проєктування, яка синтезує декілька сучасних архітектурних підходів

для вирішення специфічної задачі — створення доступної та простої у використанні IoT-платформи.

Обрана комбінація мікросервісної архітектури, асинхронної взаємодії та SPA не є унікальною для веб-розробки загалом, однак її адаптація та синергетичне застосування саме в контексті IoT-платформи для масового користувача становить основу наукової новизни. На відміну від промислових гігантів, що фокусуються на роботі з "великими даними", та локальних систем для ентузіастів, що часто мають монолітну структуру, запропонований підхід дозволяє досягти унікального балансу. Він забезпечує масштабованість та відмовостійкість (властиві enterprise-рішенням) при збереженні гнучкості та низького порогу входу для кінцевого користувача. Таким чином, новизна полягає у створенні архітектурного шаблону, оптимізованого для специфічної, раніше недостатньо охопленої ринкової ніші.

2.3 Опис програмного забезпечення

Для реалізації спроєктованої мікросервісної архітектури було обрано комплекс сучасних програмних засобів, технологій та фреймворків. Узагальнена архітектурна схема розроблюваної системи, що показує взаємозв'язок основних компонентів, наведена на рис. 2.3.

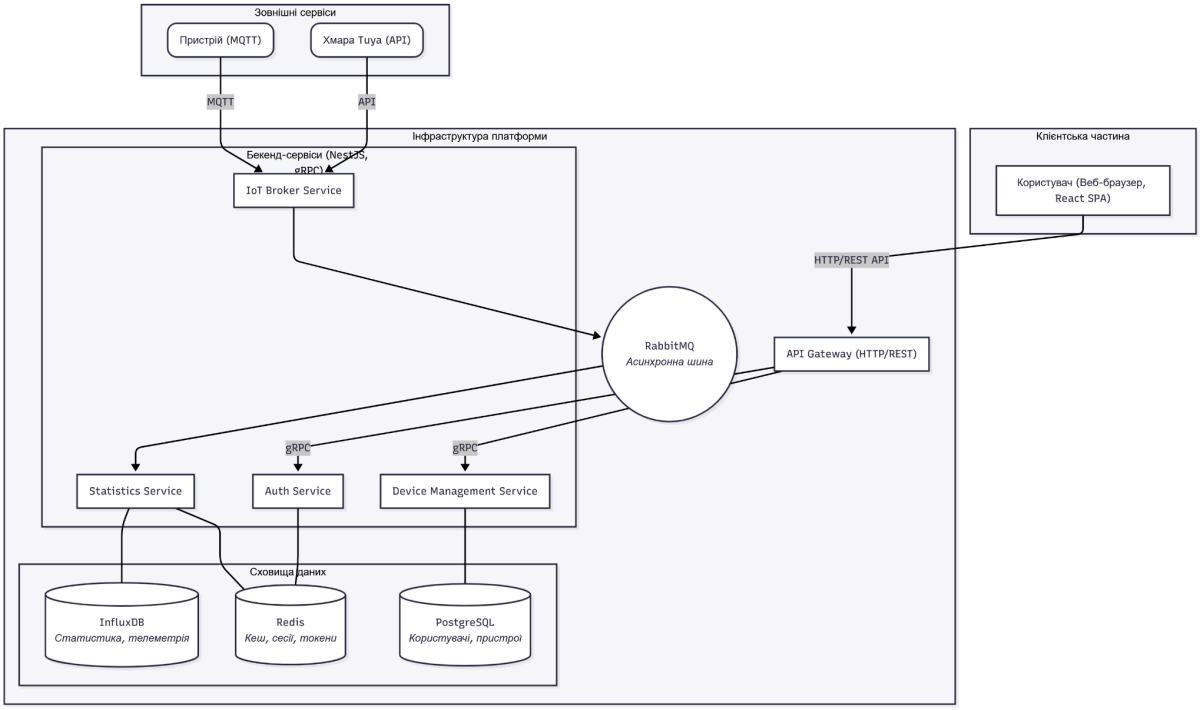


Рисунок 2.3. Узагальнена архітектурна схема розроблюваної системи

Вибір кожного компонента стеку, показаного на схемі, обґрунтовано його відповідністю поставленим задачам, гнучкістю, продуктивністю та наявністю розвиненої екосистеми.

2.3.1 Архітектура та технологічний стек серверної частини

Бекенд системи реалізовано на платформі Node.js з використанням мови програмування TypeScript. Такий вибір дозволяє використовувати єдину мову для розробки як серверної, так і клієнтської частин, що спрощує процес розробки. TypeScript додає до JavaScript строгу типізацію, що значно підвищує надійність та підтримуваність коду у великих проєктах[40].

В якості основного фреймворку для побудови мікросервісів було обрано NestJS. Це прогресивний Node.js фреймворк, який надає потужну архітектурну основу "з коробки", включаючи впровадження залежностей (Dependency Injection), модульну структуру та підтримку різних протоколів комунікації.

При виборі серверного фреймворку розглядалися такі альтернативи, як Express.js та Fastify. Express.js є більш мінімалістичним та гнучким, однак вимагає від розробника самостійної побудови архітектури. Fastify, у свою чергу, робить основний акцент на максимальній швидкодії. Було обрано NestJS, оскільки він надає готову, потужну та масштабовану архітектуру "з коробки", що базується на принципах модульності та впровадження залежностей. Вбудована підтримка TypeScript та чітка структура проєкту значно прискорюють розробку та полегшують підтримку коду в довгостроковій перспективі, що є пріоритетом для даної роботи.

Для організації внутрішньосервісної комунікації обрано високоефективний протокол gRPC. Визначення структури даних та сервісних методів здійснюється у спеціальних .proto файлах, на основі яких за допомогою бібліотеки ts-proto автоматично генерується відповідний TypeScript-код. Такий підхід гарантує строгу типізацію на рівні інтерфейсів та дозволяє фактично виключити ризик виникнення помилок під час інтеграції компонентів.

Паралельно з цим, для реалізації асинхронної взаємодії, забезпечення відмовостійкості та слабкої зв'язаності сервісів використовується брокер повідомлень RabbitMQ. Він виконує роль надійного транспорту в межах подієво-орієнтованої архітектури, дозволяючи компонентам системи ефективно обмінюватися даними без створення прямих взаємних залежностей.

В якості альтернативи RabbitMQ розглядався брокер повідомлень Apache Kafka. Kafka є надзвичайно потужною платформою, оптимізованою для обробки потоків даних у реальному часі (event streaming) та роботи з "великими даними". Однак для завдань даної системи, де пріоритетом є гнучка маршрутизація повідомлень між мікросервісами, а не довгострокове зберігання потоків подій, можливості Kafka є надлишковими, а його розгортання та підтримка — значно складнішими. RabbitMQ, що реалізує протокол AMQP, надає більш гнучкі механізми маршрутизації (через різні типи обмінників) і є простішим в

адмініструванні, що робить його оптимальним вибором для даної мікросервісної архітектури.

2.3.2 Технології зберігання даних

Для організації роботи з даними в системі застосовується підхід «Polyglot Persistence», що передбачає використання найбільш відповідного сховища для кожного типу задач. Так, для зберігання структурованих метаданих (інформація про користувачів, реєстр пристроїв, групи) обрано основну реляційну СУБД PostgreSQL, яка гарантує надійність та цілісність даних завдяки підтримці ACID-транзакцій [26]. Водночас для обробки телеметрії, яка представляє собою часові ряди (статистика енергоспоживання, показники сенсорів), використовується спеціалізована база даних InfluxDB. Її архітектура оптимізована для надшвидкого запису та агрегації інформації, що є критично важливим для аналітичних компонентів IoT-платформи [19].

Окремою ланкою архітектури виступає Redis, що використовується як високошвидкісне сховище в оперативній пам'яті (in-memory). Його основне призначення полягає в кешуванні «гарячих» даних для зниження навантаження на дискові бази даних, а також у забезпеченні швидкого доступу до сесійних даних та JWT-токенів.

2.3.3 Архітектура та технологічний стек клієнтської частини

Фронтенд системи розроблено як односторінковий додаток (SPA) з використанням бібліотеки React. React, розроблена компанією Facebook, є однією з найпопулярніших бібліотек для побудови користувацьких інтерфейсів завдяки своїй компонентній архітектурі, високій продуктивності та великій екосистемі. Компонентний підхід дозволяє створювати перевикористовувані елементи інтерфейсу (віджети, графіки), що значно прискорює та спрощує розробку.

Для організації навігації між різними сторінками (розділами) додатку використовується бібліотека React Router, яка є стандартом де-факто для маршрутизації в React-додатках.

В якості альтернатив React розглядалися фреймворки Angular та Vue. Angular є комплексним MVC-фреймворком, що надає більш жорстку структуру, але є більш громіздким. Vue є схожим на React за своєю філософією, але має іншу екосистему. Вибір було зроблено на користь React через його величезну спільноту, що гарантує наявність великої кількості готових бібліотек та рішень, високу продуктивність завдяки віртуальному DOM, та гнучкість, яка дозволяє інтегрувати його з будь-якими іншими бібліотеками за потреби.

2.3.4 Інструменти розробки та інфраструктура

Для організації ефективної взаємодії із зовнішнім світом ключову роль відіграє API Gateway, який слугує єдиною точкою входу для клієнтських запитів. У межах мікросервісної архітектури цей компонент виконує функцію фасаду, що не лише маршрутизує трафік, а й вирішує низку наскрізних завдань: від перевірки JWT-токенів та захисту системи через обмеження частоти запитів (Rate Limiting) до агрегації даних і трансформації зовнішніх REST-викликів у внутрішні gRPC-запити. Для документування цього інтерфейсу та генерації інтерактивної документації застосовується специфікація OpenAPI разом із інструментом Swagger UI.

Управління кодовою базою та розгортанням реалізовано з використанням сучасних підходів. Проєкт структуровано як монорепозиторій під управлінням Turborepo, що дозволяє централізовано працювати з усіма сервісами та спільними бібліотеками. Для забезпечення ізоляції компонентів застосовується технологія Docker: усі мікросервіси та інфраструктурні елементи пакуються у контейнери, а

їх оркестрація та швидкий запуск у локальному середовищі здійснюються за допомогою інструменту Docker Compose [27].

2.4 Аналіз і узагальнення фактичного матеріалу

На попередніх етапах було обґрунтовано вибір архітектурних підходів та технологічного стеку для розробки системи. Для подальшого проєктування необхідно провести аналіз фактичного матеріалу, що виступає вхідними даними для системи, а також сформулювати науково-технічну гіпотезу, яка буде перевірена в ході експериментальних досліджень.

2.4.1 Аналіз структури вхідних даних

Фактичним матеріалом, який обробляє система, є потоки даних (телеметрія та статуси), що надходять від різнорідних IoT-пристроїв. Незважаючи на різноманітність протоколів, після уніфікації на рівні "IoT Broker Service" ці дані приводяться до стандартизованої структури у форматі JSON. Аналіз типових пристроїв дозволяє визначити очікувані формати повідомлень.

Приклад 1: Дані від розумної розетки (протокол MQTT).

Пристрій, що працює по протоколу MQTT, може надсилати повідомлення з показниками енергоспоживання та поточним станом. Очікувана структура JSON-повідомлення:

```
 {
"deviceId": "mqtt-socket-1a2b3c",
"timestamp": "2024-10-20T14:30:00Z",
"payload": {
  "power_watts": 15.5,
  "voltage": 220.1,
  "state": "ON"
}}
```

Приклад 2: Дані від пристрою, інтегрованого через хмару Tuya.

Пристрої, що працюють через екосистему Tuya, надають свої дані через хмарний API. Відповідь від API зазвичай містить масив об'єктів, що описують різні функції пристрою. Після уніфікації ці дані можуть бути представлені у наступному вигляді:

```
{
  "deviceId": "tuya-bulb-4d5e6f",
  "timestamp": "2024-10-20T14:31:15Z",
  "payload": {
    "switch_led": true,
    "brightness": 850,
    "color_temp": 4000
  }
}
```

Аналіз цих структур показує необхідність проєктування гнучкої моделі даних, здатної зберігати та обробляти різноманітні поля у "корисному навантаженні" (payload), що підтверджує правильність вибору на користь частково неструктурованих сховищ (InfluxDB) для телеметрії.

Приклад 3: Пакетна передача даних від автономного датчика.

Пристрої, що працюють від батареї (наприклад, метеодатчик у полі), часто накопичують вимірювання протягом певного періоду (наприклад, години), а потім відправляють їх одним пакетом для економії заряду. Структура такого повідомлення може виглядати наступним чином:

```
{
  "deviceId": "weather-sensor-7g8h9i",
  "payload": [
    { "timestamp": "2024-10-20T14:00:00Z", "temperature": 15.2, "humidity": 65 },
    { "timestamp": "2024-10-20T14:15:00Z", "temperature": 15.4, "humidity": 64 },
```

{ "timestamp": "2024-10-20T14:30:00Z", "temperature": 15.5, "humidity": 64 },
{ "timestamp": "2024-10-20T14:45:00Z", "temperature": 15.3, "humidity": 66 }
]
}

Цей сценарій висуває додаткові вимоги до сервісу обробки даних, який повинен бути здатним розпарсити та обробити масив вимірювань в рамках одного повідомлення. Це ще раз підтверджує доцільність вибору гнучких інструментів, здатних працювати зі складними структурами даних.

2.4.2 Аналіз ключового користувацького сценарію

Для перевірки доцільності обраних архітектурних рішень було проаналізовано базовий сценарій ідеальної взаємодії з платформою “happy path”. Робота розпочинається зі швидкої та безпечної реєстрації через Google-акаунт OAuth 2.0. Наступним кроком є додавання обладнання: у особистому кабінеті користувач обирає необхідну інтеграцію, наприклад, Tuya та, дотримуючись інструкцій, здійснює прив'язку акаунту, після чого список доступних пристроїв автоматично підтягується в інтерфейс системи.

Основний етап роботи включає моніторинг та аналіз даних. На дашборді через інтерактивні віджети відображається поточний стан пристроїв, який оновлюється миттєво при будь-яких змінах без перезавантаження сторінки. Також користувачеві доступна детальна статистика з історичними даними, наприклад, графіками енергоспоживання. Реалізація саме такого сценарію підтверджує критичну важливість використання архітектури SPA для забезпечення плавності інтерфейсу та асинхронної обробки даних для їх гарантованої доставки в реальному часі.

2.4.3 Формулювання гіпотези дослідження та критеріїв її перевірки

На основі проведеного аналізу та обґрунтування вибору методів формулюється наступна науково-технічна гіпотеза: застосування комплексної

методики, що поєднує мікросервісну архітектуру, асинхронну взаємодію на базі брокера повідомлень та архітектуру односторінкового додатку, дозволить створити IoT-платформу, яка буде одночасно масштабованою, відмовостійкою та забезпечуватиме високу якість користувацького досвіду при моніторингу в реальному часі.

Для експериментального підтвердження висунутої гіпотези у третьому розділі заплановано тестування прототипу за низкою ключових метрик. Зокрема, вимірюватиметься час наскрізної затримки (end-to-end latency) — період від генерації повідомлення пристроєм до його візуалізації в інтерфейсі, який для систем реального часу не повинен перевищувати 1–2 секунд. Паралельно оцінюватиметься пропускна здатність (throughput), що визначає граничну кількість повідомлень, яку система здатна стабільно обробляти за секунду без суттєвої деградації продуктивності.

Крім показників швидкодії, буде проведено моніторинг ефективності використання ресурсів (CPU та оперативної пам'яті) окремими сервісами під час зростання навантаження. Це дозволить перевірити дієвість механізмів незалежного масштабування. Отримані у ході експерименту результати слугуватимуть основою для обґрунтованого висновку щодо ефективності обраного архітектурного підходу.

2.5 Висновки

У даному розділі було проведено глибокий аналіз та обґрунтування вибору методів і програмних засобів для розробки універсальної системи моніторингу та керування IoT-пристроями.

За результатами порівняльного аналізу можливих архітектурних підходів було зроблено обґрунтований вибір на користь комплексної методики, що поєднує мікросервісну архітектуру для побудови серверної частини, асинхронну (подієво-орієнтовану) взаємодію між компонентами та архітектуру

односторінкового додатку (SPA) для клієнтської частини. Такий підхід, незважаючи на свою початкову складність, забезпечує необхідний рівень гнучкості, масштабованості, відмовостійкості та високу якість користувацького досвіду, що є критично важливим для досягнення поставленої в роботі мети.

На основі обраної методики було визначено та описано конкретний стек програмного забезпечення. Для серверної частини обрано платформу Node.js з фреймворком NestJS, протокол gRPC для внутрішніх комунікацій та брокер повідомлень RabbitMQ. Для зберігання даних застосовано підхід "Polyglot Persistence" з використанням PostgreSQL, InfluxDB та Redis. Клієнтська частина реалізується на базі бібліотеки React. Увесь комплекс програмних засобів розгортається у контейнерах за допомогою Docker та Docker Compose.

Результатом роботи, проведеної у даному розділі, є не лише вибір інструментарію, а й формулювання чіткої науково-технічної гіпотези про ефективність обраного комплексного підходу. Було визначено ключові метрики, такі як, час затримки, пропускна здатність, використання ресурсів, за якими ця гіпотеза буде експериментально перевірятися.

Таким чином, у даному розділі було створено повне теоретичне та технологічне підґрунтя для переходу до наступного етапу роботи — безпосереднього проєктування та програмної реалізації системи, що буде детально описано у третьому розділі.

3. ПРОЄКТНО-ПРОГРАМНІ РІШЕННЯ ТА РЕЗУЛЬТАТИ ЇХ ДОСЛІДЖЕННЯ

3.1 Архітектурне проєктування системи

В основі розроблюваної платформи лежить архітектурний шаблон "мікросервіси", вибір якого було обґрунтовано у попередньому розділі. Цей підхід дозволяє декомпозувати складну систему на набір незалежних, слабко зв'язаних сервісів, кожен з яких відповідає за свою бізнес-логіку[37]. Таке архітектурне рішення забезпечує високу гнучкість, масштабованість та відмовостійкість, що є критичними вимогами для сучасних IoT-платформ.

3.1.1 Компонентна модель системи

Загальну статичну структуру системи можна представити у вигляді компонентної моделі, що складається з чотирьох логічних рівнів. Взаємозв'язок компонентів на цих рівнях показано на рис. 3.1.

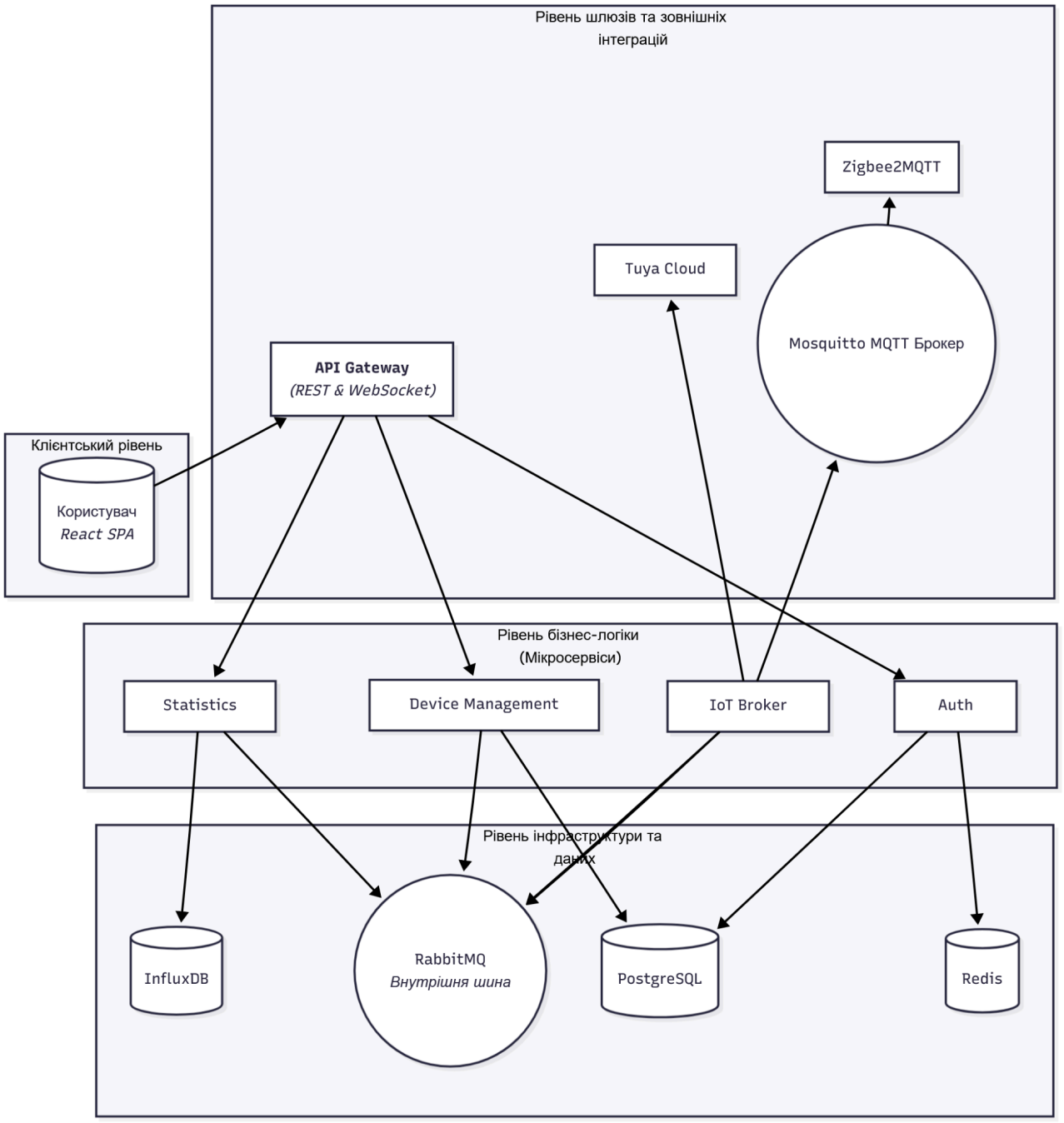


Рисунок 3.1. Компонентна архітектура системи

Клієнтський рівень реалізовано за архітектурним шаблоном "Односторінковий додаток" (SPA). Цей вибір забезпечує високу інтерактивність та плавний користувацький досвід, що є необхідним для систем моніторингу в реальному часі.

Рівень шлюзів та зовнішніх інтеграцій виконує роль "Фасаду" (Facade) для всієї системи. API Gateway інкапсулює складну внутрішню логіку мікросервісів,

надаючи клієнтам єдиний та простий API. Компоненти Mosquitto та Zigbee2MQTT реалізують шаблон "Адаптер" (Adapter), перетворюючи специфічні протоколи (MQTT, Zigbee) у стандартизований потік даних для внутрішньої обробки.

Рівень бізнес-логіки є ядром системи. Кожен мікросервіс спроєктовано за принципом "Єдиної відповідальності" (Single Responsibility Principle), що полегшує його розробку, тестування та підтримку.

Рівень інфраструктури та даних реалізує шаблон "Поліглотне сховище" (Polyglot Persistence), використовуючи різні типи баз даних для різних типів даних. Асинхронна взаємодія між сервісами реалізована за допомогою шаблону "Видавець-Підписник" (Publish-Subscribe) через шину повідомлень RabbitMQ.

3.1.2 Проєктування потоків даних

Динамічна взаємодія між компонентами системи базується на принципах подієво-орієнтованої архітектури (Event-Driven Architecture). Розглянемо ключовий сценарій обробки даних від пристрою до користувача (рис. 3.2), щоб продемонструвати реалізацію цього підходу.

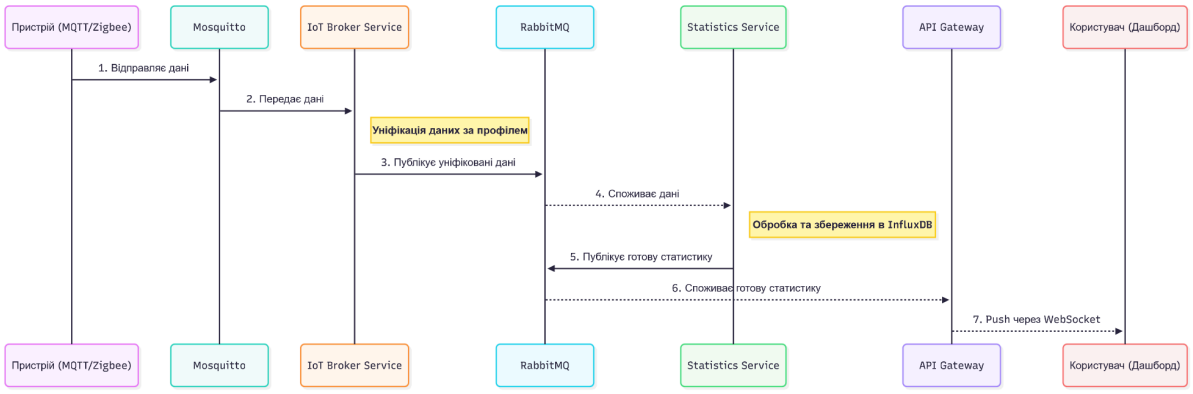


Рисунок 3.2. Схема потоку обробки даних у реальному часі

Процес обробки даних спроєктовано як конвеєр (pipeline), що складається з наступних етапів:

1. Прийом та нормалізація (Ingestion & Normalization): Дані від пристроїв надходять на рівень шлюзів, де IoT Broker Service виконує їх прийом. Цей

сервіс реалізує логіку нормалізації, перетворюючи різнорідні формати даних у єдину, канонічну модель системи. Це ключове проєктне рішення, що дозволяє відокремити логіку інтеграції від логіки обробки.

2. Асинхронна передача (Asynchronous Messaging): Після нормалізації IoT Broker Service публікує подію з уніфікованими даними в шину повідомлень RabbitMQ. Використання шини розриває пряму залежність між сервісом прийому та сервісом обробки, що є основою відмовостійкості системи.
3. Обробка та збагачення (Processing & Enrichment): Statistics Service споживає подію з шини. На цьому етапі дані можуть бути збагачені, наприклад, додано інформацію про власника пристрою та збережені у відповідне сховище (InfluxDB).
4. Сповіщення (Notification): Після успішної обробки Statistics Service генерує нову подію — "статистика_оновлена". Ця подія публікується в інший топік шини повідомлень.
5. Доставка в реальному часі (Real-time Delivery): API Gateway, підписаний на цю подію, отримує її та, використовуючи WebSocket, доставляє фінальні дані на клієнтський додаток. Такий підхід мінімізує затримки та забезпечує ефективну доставку "від сервера до клієнта" (server-push).

Це архітектурне рішення, що базується на подіях та асинхронній комунікації, забезпечує високу гнучкість та масштабованість, дозволяючи в майбутньому легко додавати нові сервіси-обробники без модифікації існуючих компонентів.

3.1.3 Ключові принципи та шаблони проєктування, застосовані в архітектурі

Розроблена архітектура базується не лише на виборі конкретних технологій, але й на застосуванні фундаментальних принципів та шаблонів проєктування програмного забезпечення, що забезпечують її гнучкість, надійність та підтримуваність.

Принцип єдиної відповідальності (Single Responsibility Principle - SRP) - цей принцип, що є частиною SOLID, стверджує, що кожен програмний модуль або клас повинен мати лише одну причину для зміни [28]. У контексті розробленої системи цей принцип застосовано на макрорівні: кожен мікросервіс має чітко визначену та ізольовану зону відповідальності. Наприклад, Auth Service відповідає виключно за ідентифікацію, Device Management Service — за життєвий цикл пристроїв, а Statistics Service — за обробку телеметрії. Такий підхід значно спрощує розробку, тестування та подальшу модифікацію системи, оскільки зміни в одному сервісі мінімально впливають на інші.

Слабка зв'язаність та висока згуртованість (Loose Coupling & High Cohesion) - це два взаємопов'язані принципи, що є основою якісної модульної архітектури. Слабка зв'язаність означає, що компоненти системи мають бути якомога менш залежними один від одного. У розробленій архітектурі цей принцип реалізовано за допомогою асинхронної взаємодії через брокер повідомлень RabbitMQ. Сервіси не викликають один одного напряму, а лише публікують події у шину, що дозволяє їм функціонувати та розвиватися незалежно. Висока згуртованість означає, що функціонал, який логічно пов'язаний, повинен знаходитись разом. Кожен мікросервіс у системі є високозгуртованим, оскільки він інкапсулює всю логіку, необхідну для виконання своєї конкретної бізнес-задачі [29].

У сучасних розподілених системах застосовується принцип проєктування з розрахунком на відмову (Design for Failure). Оскільки часткові збої, такі як проблеми з мережею, програмні помилки або тимчасова недоступність сервісів, є неминучими, архітектура платформи від початку орієнтована не стільки на уникнення інцидентів, скільки на забезпечення стійкості до них. Фундаментом цього підходу виступають ізоляція мікросервісів у Docker-контейнерах та використання асинхронних черг повідомлень, що дозволяє локалізувати проблеми й запобігти їх поширенню на всю систему.

Практична перевага такої моделі стає очевидною в аварійних ситуаціях. Наприклад, у разі виходу з ладу сервісу статистики (Statistics Service), модуль прийому даних (IoT Broker Service) продовжує штатно функціонувати, отримуючи телеметрію від пристроїв та публікуючи її у RabbitMQ. Повідомлення накопичуються в черзі та зберігаються до моменту відновлення роботи сервісу-обробника, що гарантує збереження даних навіть за умов тимчасової недоступності окремих компонентів інфраструктури.

У системі застосовано класичні шаблони проєктування "Банди чотирьох" (GoF) [30]. API Gateway реалізує шаблон "Фасад", надаючи єдину, спрощену точку входу для клієнтського додатку та приховуючи складну внутрішню структуру мікросервісів. Компоненти, що працюють з зовнішніми системами, такі як Zigbee2MQTT або логіка інтеграції з Tuya API всередині IoT Broker Service, реалізують шаблон "Адаптер", перетворюючи зовнішній, несумісний інтерфейс у внутрішній, стандартизований.

Застосування цих загальновизнаних інженерних практик дозволило створити не просто набір програмних компонентів, а цілісну, теоретично обґрунтовану та життєздатну архітектуру.

3.1.4 Проєктування API Gateway як єдиної точки входу

API Gateway є одним з найважливіших компонентів у розробленій мікросервісній архітектурі. Він реалізує архітектурний шаблон "Фасад" (Facade), виступаючи єдиною точкою входу для всіх запитів від клієнтського додатку[33]. Такий підхід надає низку ключових переваг:

- Інкапсуляція внутрішньої структури: Клієнтський додаток не знає про існування та адреси внутрішніх мікросервісів. Він взаємодіє лише з одним, стабільним API, що спрощує розробку фронтенду.
- Централізація наскрізних завдань: API Gateway знімає навантаження з мікросервісів, беручи на себе виконання загальних, "наскрізних" функцій.

- Підвищення безпеки: Внутрішні мікросервіси можуть бути повністю ізольовані у приватній мережі, недоступній ззовні, що зменшує поверхню атаки.

У розробленій системі API Gateway, реалізований на базі NestJS, відіграє ключову роль у керуванні потоками даних. Однією з його основних функцій є маршрутизація запитів, під час якої шлюз аналізує URI вхідного HTTP-виклику та трансформує його у відповідний gRPC-запит до цільового мікросервісу (наприклад, перенаправлення з /api/auth/login до Auth Service). Паралельно на рівні шлюзу реалізовано механізм розвантаження автентифікації (Gateway Offloading): система валідує JWT Access Token у заголовках кожного запиту до захищених ресурсів. Це дозволяє централізувати перевірку доступу та звільнити внутрішні сервіси від рутинних операцій безпеки, залишаючи за ними лише логіку видачі токенів.

Іншим важливим завданням шлюзу є забезпечення цілісності вхідних даних. Завдяки використанню бібліотек class-validator [34] та class-transformer, кожен запит проходить автоматичну перевірку на відповідність затвердженим схемам DTO, що гарантує надходження до мікросервісів лише коректної та очищеної інформації. Для спрощення взаємодії з клієнтською частиною у систему інтегровано інструментарій Swagger (OpenAPI) [35]. Використання спеціальних декораторів у коді контролерів дозволяє автоматично генерувати інтерактивну документацію, яка візуалізує структуру API та надає можливість тестувати ендпоінти безпосередньо через веб-інтерфейс (рис. 3.3).

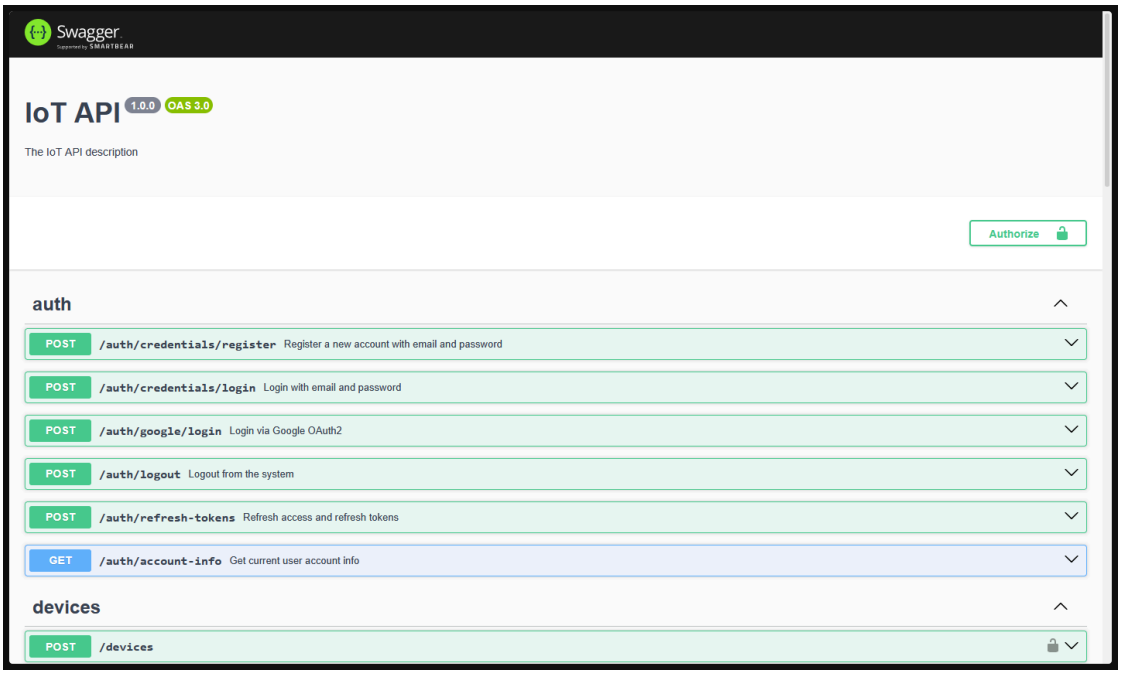


Рисунок 3.3. Приклад згенерованої Swagger UI документації для API Gateway

Таким чином, API Gateway є не просто пасивним проксі, а активним, інтелектуальним компонентом, що значно спрощує взаємодію в розподіленій системі та підвищує її надійність та безпеку.

3.2 Детальне проєктування ключових мікросервісів

У цьому підрозділі детально розглядається архітектура та проєктні рішення для трьох ключових мікросервісів, що складають ядро бізнес-логіки платформи: Auth Service, Device Management Service та IoT Broker Service.

3.2.1 Архітектура та принципи реалізації сервісу автентифікації (Auth Service)

Auth Service є центральним компонентом системи, що відповідає за управління безпекою та ідентифікацією. Його зона відповідальності включає: реєстрацію нових користувачів, автентифікацію за обліковими даними, інтеграцію зі сторонніми провайдерами через OAuth 2.0, а також генерацію, перевірку та оновлення JWT-токенів.

Сервіс спроєктовано за багатошаровою архітектурою, типовою для фреймворку NestJS, що забезпечує чітке розділення відповідальності між компонентами (рис. 3.4).

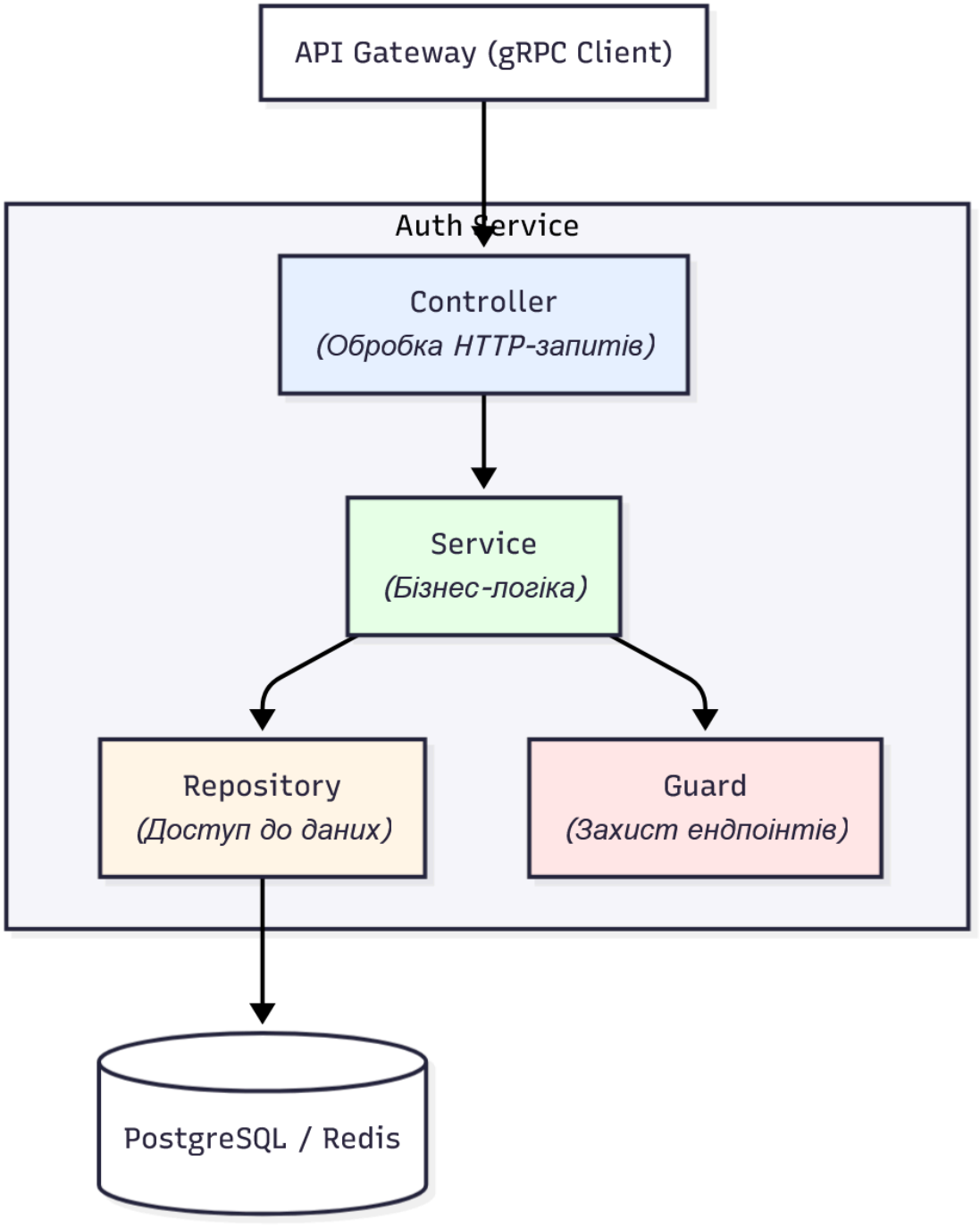


Рисунок 3.4. Внутрішня компонентна архітектура Auth Service

- Controller: Приймає вхідні gRPC-запити від API Gateway, валідує їх та передає на обробку сервісному шару.

- Service: Інкапсулює основну бізнес-логіку: валідацію паролів, генерацію токенів, взаємодію з OAuth-провайдерами.
- Repository: Реалізує шаблон "Репозиторій"[36] для абстрагування роботи з базами даних (PostgreSQL та Redis).
- Guard: Реалізує стратегії захисту, наприклад, перевірку JWT-токенів для доступу до захищених методів.

Для взаємодії з базою даних PostgreSQL використовується ORM (Object-Relational Mapping) TypeORM [31]. Розроблено дві пов'язані сутності: Account для зберігання інформації про користувачів та Token для управління Refresh-токенами сесій.

Таблиця 3.1. Структура сутності Account

Поле	Тип даних (SQL)	Декоратор TypeORM	Опис
id	uuid	@PrimaryGeneratedColumn('uuid')	Первинний ключ, унікальний ідентифікатор акаунту.
fullName	character varying	@Column()	Повне ім'я користувача.
provider	enum	@Column({ type: 'enum' })	Ідентифікатор провайдера (LOCAL, GOOGLE).
email	character varying	@Column({ unique: true })	Унікальний email, використовується як логін.
password	character varying	@Column({ nullable: true })	Хеш паролю (bcrypt). NULL для OAuth-акаунтів.
tokens	- (зв'язок)	@OneToMany(() => Token)	Зв'язок "один-до-багатьох" з сутністю Token для зберігання Refresh-токенів.
createdAt	timestamp	@CreateDateColumn()	Час створення запису.
updatedAt	timestamp	@UpdateDateColumn()	Час останнього оновлення запису.

Таблиця 3.2. Структура сутності Token

Поле	Тип даних (SQL)	Декоратор TypeORM	Опис
id	uuid	@PrimaryGeneratedColumn('uuid')	Первинний ключ.
token	uuid	@Column()	Значення Refresh-токена.
expires	timestamp	@Column()	Час завершення терміну дії токена.
account	uuid (зовн. ключ)	@ManyToOne(() => Account)	Зв'язок "багато-до-одного" з акаунтом, якому належить токен.
accountId	uuid	-	Поле зовнішнього ключа.
createdAt	timestamp	@CreateDateColumn()	Час створення запису.

Нижче наведено фрагменти коду, що описують реалізацію цих сутностей.

```
// account.entity.ts

@Entity()
export class Account {
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @Column()
  fullName: string;

  @Column({ type: 'enum', enum: AccountProvider })
  provider: AccountProvider;

  @Column({ unique: true })
  email: string;

  @Column({ nullable: true, default: null })
  password?: string;
```

```
  @OneToMany(() => Token, (token) => token.account)
  tokens: Token[];

  @CreateDateColumn()
  createdAt: Date;

  @UpdateDateColumn()
  updatedAt: Date;
}
// token.entity.ts
@Entity()
export class Token {
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @Column()
  token: string;

  @Column()
  expires: Date;

  @ManyToOne(() => Account, (account) => account.tokens)
  account: Account;

  @Column()
  accountId: string;

  @CreateDateColumn()
  createdAt: Date;
```

}

Такий підхід, що базується на ORM та декораторах, забезпечує строгу типізацію, спрощує міграції схеми бази даних та абстрагує розробника від написання "сирих" SQL-запитів, що підвищує надійність та швидкість розробки.

Таблиця 3.3. Приклад gRPC запитів мікросервісу авторизації

Метод	URI	DTO (Тіло запиту)	Опис
POST	/auth/register	RegisterDto {fullName, email, password}	Реєстрація нового користувача за email/паролем.
POST	/auth/login	LoginDto {email, password}	Автентифікація користувача, повертає пару Access/Refresh токенів.
POST	/auth/refresh	RefreshDto {refreshToken}	Оновлення пари токенів за допомогою Refresh Token.
GET	/auth/google	-	Ініціація процесу автентифікації через Google (перенаправлення).
POST	/auth/logout	RefreshDto {refreshToken}	Вихід з системи, видаляє Refresh-токен з бази даних.
GET	/auth/profile	-	Отримання інформації про поточного автентифікованого користувача.
PUT	/auth/password /reset	ResetPasswordDto {email}	Запит на скидання пароля, відправляє посилання на email.

Нижче наведено спрощений приклад коду контролера Auth Service на NestJS, що демонструє реалізацію gRPC-методу для логіну.

```
// auth.controller.ts (у мікросервісі Auth)

@Controller()
export class AuthController {
```

```
  constructor(private readonly authService: AuthService) {}

  @GrpcMethod('AuthService', 'Login')
  async login(data: LoginRequest): Promise<TokenPair> {
    return this.authService.login(data.email, data.password);
  }
  // ... інші gRPC-методи ...
}
```

3.2.2 Архітектура та принципи реалізації сервісу управління пристроями (Device Management Service)

Device Management Service є центральним сховищем інформації про всі пристрої, зареєстровані в системі. Його зона відповідальності включає: надання CRUD-операцій (Create, Read, Update, Delete) для пристроїв та їх груп, валідацію прав доступу користувача до пристроїв, а також управління "профілями пристроїв", які є ключовим елементом для уніфікації даних.

Аналогічно до Auth Service, цей мікросервіс побудовано за багатошаровою архітектурою на базі NestJS. Він складається з контролера, що обробляє gRPC-запити, сервісного шару з бізнес-логікою, та шару репозиторіїв для доступу до бази даних PostgreSQL.

Сервіс оперує трьома взаємопов'язаними сутностями, що зберігаються в PostgreSQL: DeviceEntity (пристрої), GroupEntity (групи пристроїв) та DeviceProfileEntity (профілі пристроїв). Взаємодія з базою даних реалізована за допомогою TypeORM.

Таблиця 3.4. Структура сутності DeviceEntity

Поле	Тип даних (SQL)	Опис
id	uuid	Первинний ключ, унікальний

		ідентифікатор пристрою в системі.
userId	uuid	Ідентифікатор користувача-власника.
name	character varying	Назва пристрою, що задається користувачем.
externalId	character varying	Унікальний ID пристрою у зовнішній системі (напр., MAC-адреса).
protocol	enum	Протокол, що використовується (MQTT, ZIGBEE, TUYA).
groupId	uuid	Зовнішній ключ, що посилається на GroupEntity (може бути NULL).
profileId	character varying	Зовнішній ключ, що посилається на DeviceProfileEntity.
credentials	jsonb	JSON-об'єкт для зберігання специфічних даних підключення (напр., localKey для Tuya).

Сутність DeviceProfileEntity (Таблиця 3.5) реалізує ключову концепцію "декларативного драйвера".

Таблиця 3.5. Структура сутності DeviceProfileEntity

Поле	Тип даних (SQL)	Опис
id	character varying	Первинний ключ (текстовий, напр., "prof_zigbee_adeo_ldsenk08").
name	character varying	Унікальна назва профілю.
vendor	character varying	Виробник пристрою.
description	text	Опис профілю (опціонально).
protocol	enum	Протокол, що підтримується профілем.
mappings	jsonb	JSON-об'єкт, що містить правила перетворення "сирих" даних.

Сутність GroupEntity (Таблиця 3.6) дозволяє користувачам логічно об'єднувати пристрої (напр., "Освітлення вітальні").

Таблиця 3.6. Структура сутності GroupEntity

Поле	Тип даних (SQL)	Опис
id	uuid	Первинний ключ.

userId	uuid	Ідентифікатор користувача-власника групи.
name	character varying	Назва групи.
description	text	Опис групи (опціонально).

Зв'язки між сутностями реалізовано за допомогою декораторів @ManyToOne та @OneToMany. DeviceEntity має зв'язки "багато-до-одного" з GroupEntity та DeviceProfileEntity. Така реляційна модель забезпечує цілісність даних та дозволяє виконувати ефективні запити, наприклад, "отримати всі пристрої для даного користувача" або "знайти всі пристрої, що використовують певний профіль".

Сервіс надає свій функціонал через gRPC, що забезпечує високу продуктивність внутрішньосервісної комунікації. Контракт API визначається у .proto файлі та логічно розділений на дві частини: методи для управління пристроями та методи для управління групами.

Таблиця 3.7 – Опис gRPC-методів для управління пристроями

Метод	Вхідне повідомлення	Вихідне повідомлення	Опис
CreateDevice	CreateDeviceRequest	Device	Створює новий пристрій (id, userId, type, name, status).
GetDeviceById	GetDeviceByIdRequest	Device	Повертає детальну інформацію про конкретний пристрій.
GetDevicesByUser	GetDevicesRequest	DeviceList	Повертає список усіх пристроїв для вказаного користувача.
UpdateDevice	UpdateDeviceRequest	Device	Оновлює назву або статус (name, status) існуючого пристрою.
DeleteDevice	DeleteDeviceRequest	Empty	Видаляє пристрій.

Таблиця 3.8. Опис gRPC-методів для управління групами

Метод	Вхідне повідомлення	Вихідне повідомлення	Опис
CreateGroup	CreateGroupRequest	Group	Створює нову групу (id, userId, name, description).
GetGroupById	GetGroupByIdRequest	Group	Повертає детальну інформацію про конкретну групу.
GetGroupsByUser	GetGroupsRequest	GroupList	Повертає список усіх груп для вказаного користувача.
UpdateGroup	UpdateGroupRequest	Group	Оновлює назву або опис (name, description) існуючої групи.
DeleteGroup	DeleteGroupRequest	Empty	Видаляє групу.

Нижче наведено спрощений приклад визначення сервісу та повідомлень для управління групами у .proto файлі, що відображає наданий Вами код.

```
  // device_management.proto


syntax = "proto3";

import "google/protobuf/timestamp.proto";


package device_management;


service DeviceManagementService {

 // ... методи для пристроїв ...
```

```
  rpc CreateGroup (CreateGroupRequest) returns (Group);
  rpc GetGroup (GetGroupRequest) returns (Group);
  rpc FindGroups (FindGroupsRequest) returns (GroupList);
  rpc UpdateGroup (UpdateGroupRequest) returns (Group);
  rpc DeleteGroup (DeleteGroupRequest) returns (DeleteResponse);
}

message CreateGroupRequest {
  string userId = 1;
  string name = 2;
  optional string description = 3;
}

message GetGroupRequest {
  string id = 1;
  string userId = 2;
}

message Group {
```

```
  string id = 1;
  string userId = 2;
  string name = 3;
  optional string description = 4;
  int32 devicesCount = 5;
  google.protobuf.Timestamp createdAt = 6;
}

message GroupList {
  repeated Group groups = 1;
  int32 total = 2;
}

// ... інші повідомлення …
```

Як видно з коду реалізації та .proto файлу, сервіс надає повний набір CRUD-операцій для управління групами, що є важливою частиною користувацького досвіду, дозволяючи логічно структурувати велику кількість пристроїв.

3.2.3 Архітектура та принципи реалізації сервісу-брокера (IoT Broker Service)

IoT Broker Service є ключовим інтеграційним компонентом, що виконує роль "мосту" між зовнішніми, різнорідними джерелами даних та внутрішньою уніфікованою екосистемою платформи. Його головне завдання — прийом,

валідація та нормалізація (перетворення) даних від IoT-пристроїв. Цей сервіс реалізує архітектурний шаблон Anti-Corruption Layer, захищаючи ядро системи від складнощів та специфіки зовнішніх протоколів та форматів даних.

Сервіс має модульну структуру, де кожен модуль відповідає за інтеграцію з певним типом джерела даних. Це дозволяє легко додавати підтримку нових протоколів у майбутньому.

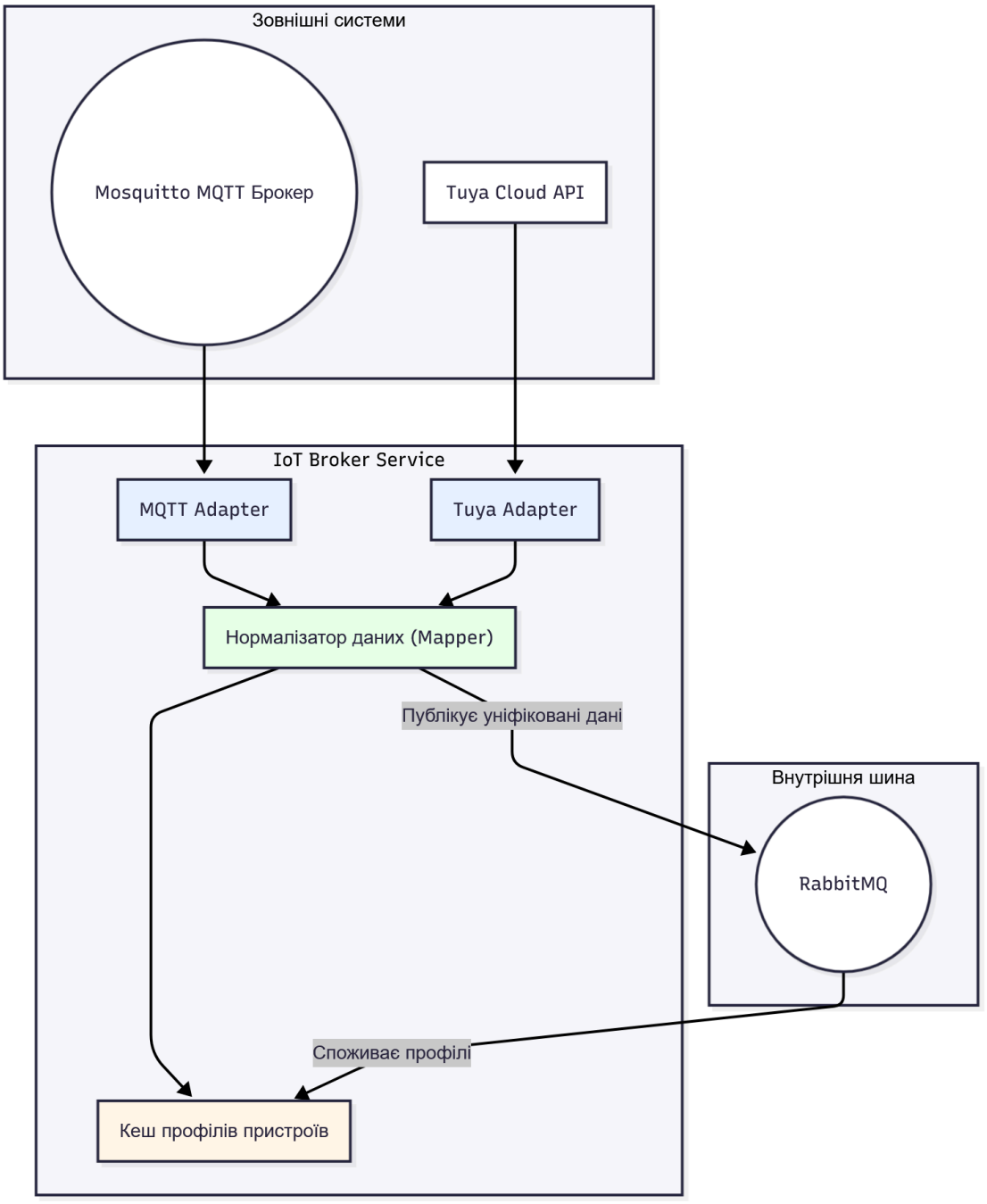


Рисунок 3.5. Внутрішня компонентна архітектура IoT Broker Service

- Адаптери (MQTT Adapter, Tuya Adapter): Кожен адаптер відповідає за технічну реалізацію підключення до певного джерела даних. MQTT Adapter є клієнтом Mosquitto, Tuya Adapter взаємодіє з хмарним REST API. Їхнє завдання — отримати "сирі" дані та передати їх на обробку.
- Кеш профілів (Profile Cache): Для уникнення постійних запитів до бази даних, сервіс підписується на події device.profile.updated з RabbitMQ та зберігає актуальні копії всіх профілів пристроїв у локальному кеші в пам'яті.
- Нормалізатор даних (Mapper): Це ядро сервісу. Він отримує "сирі" дані від адаптерів, завантажує відповідний профіль з кешу та, використовуючи правила з mappings, перетворює дані у єдиний внутрішній формат.

Проєктування внутрішнього конвеєра обробки. Основна логіка сервісу реалізована як конвеєр (pipeline), що виконується для кожного вхідного повідомлення. Цей процес візуалізовано на блок-схемі (рис. 3.6).

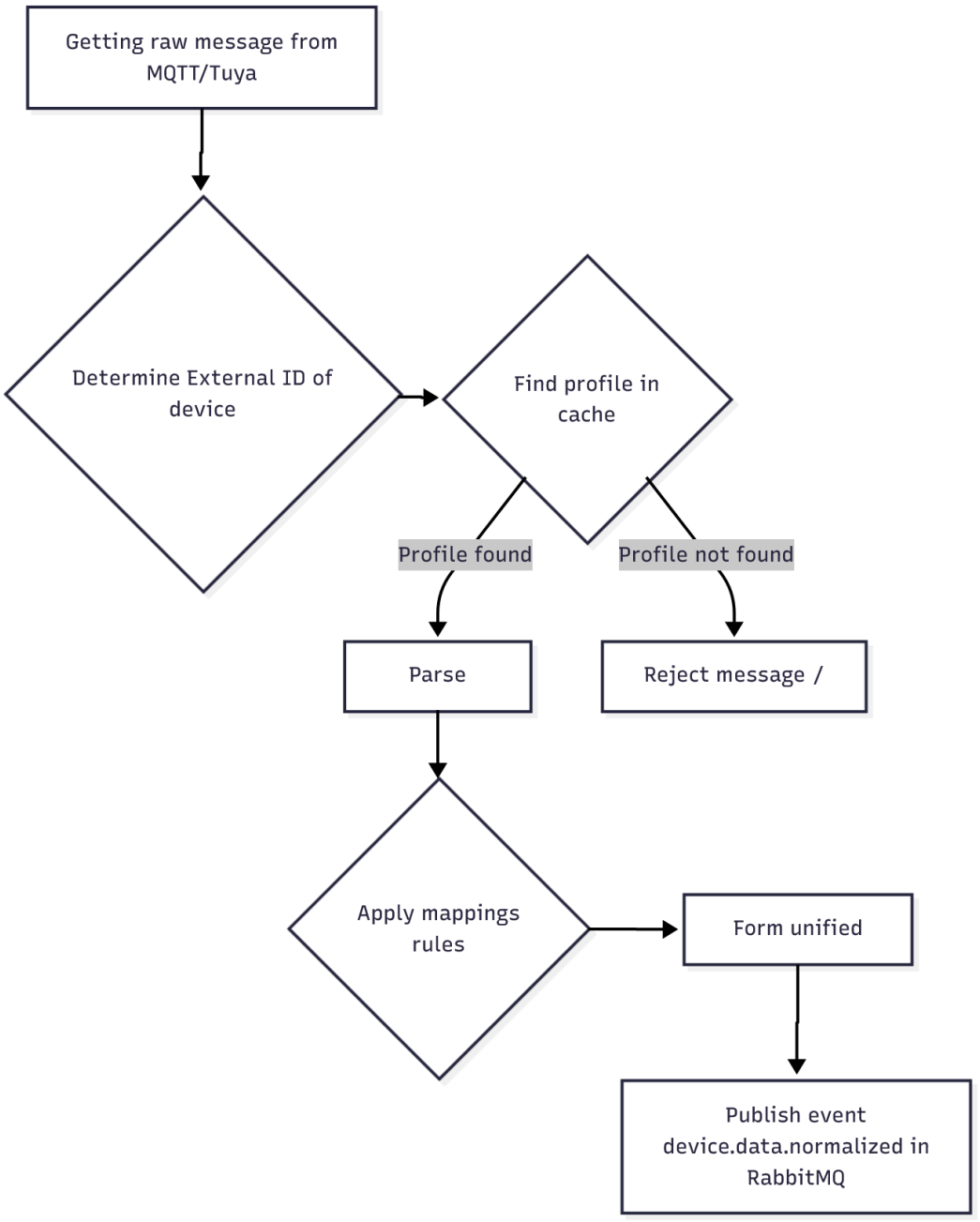


Рисунок 3.6. Блок-схема конвеєра обробки даних в IoT Broker Service

Нижче наведено псевдокод, що ілюструє реалізацію цієї логіки.

```
// Псевдокод логіки обробки в IoT Broker Service

async function onRawMessageReceived(source: string, data: any) {
  // 1. Визначити ID пристрою
  const externalId = parseExternalId(source, data);
```

```
// 2. Завантажити профіль з кешу
const profile = profileCache.get(externalId);
if (!profile) {
  log.error(`Профіль для пристрою ${externalId} не знайдено`);
  return;
}

// 3. Розпарсити "сирі" дані
const parsedData = parsePayload(data, profile.protocol);

// 4. Застосувати правила мапінгу
const unifiedPayload = {};
for (const sourceKey in profile.mappings) {
  const mappingRule = profile.mappings[sourceKey];
  const targetKey = mappingRule.targetMetric;
  const value = getValueFromPath(parsedData, sourceKey);

  if (value !== undefined) {
    unifiedPayload[targetKey] = castToType(value, mappingRule.type);
  }
}

// 5. Сформувати та опублікувати уніфіковану подію
const unifiedEvent = {
  deviceId: profile.internalDeviceId, // Внутрішній UUID
  timestamp: new Date().toISOString(),
  payload: unifiedPayload,
};

messageBus.publish('device.data.normalized', unifiedEvent);
```

}

Найважливішим рішенням є відокремлення логіки інтеграції від логіки обробки. IoT Broker Service не займається аналізом чи збереженням даних. Його єдина, але критично важлива функція — виступати "перекладачем", що дозволяє решті системи працювати з простим та стандартизованим потоком даних, незважаючи на хаос і різноманітність зовнішнього світу IoT-пристроїв. Використання локального кешу для профілів мінімізує затримки та залежність від доступності інших сервісів під час обробки повідомлень.

3.3 Проєктування клієнтської частини (фронтенду)

Клієнтська частина системи є єдиною точкою взаємодії користувача з платформою. Вона спроєктована як односторінковий додаток (SPA), що забезпечує високу продуктивність та інтерактивність, необхідну для ефективного моніторингу IoT-пристроїв.

3.3.1 Вибір технологічного стеку та компонентна архітектура

В якості основної бібліотеки для побудови користувацького інтерфейсу було обрано React [32]. Цей вибір обґрунтований його компонентною архітектурою, яка дозволяє розбивати складний інтерфейс на невеликі, незалежні та перевикористовувані частини (компоненти). Для забезпечення надійності та масштабованості кодової бази проєкт розроблено з використанням TypeScript.

Архітектура фронтенд-додатку базується на компонентному підході. Уся кодова база організована за функціональною ознакою (Feature-Sliced Design), де кожен логічний блок є окремим модулем. Це спрощує навігацію по проєкту та полегшує його підтримку. Навігація між основними розділами додатку реалізована за допомогою бібліотеки React Router.

3.3.2 Проєктування ієрархії компонентів

Інтерфейс системи складається з декількох ключових екранів (сторінок), кожен з яких відповідає за певну логічну функцію.

Сторінка входу/реєстрації є точкою входу до системи. Її інтерфейс (рис. 3.7) спроєктовано з акцентом на простоту та зручність. Він надає користувачеві вибір між класичною реєстрацією/входом за email/паролем та швидкою автентифікацією через Google-акаунт, що відповідає сучасним практикам UX-дизайну. Компоненти форми реалізовано з валідацією в реальному часі для покращення взаємодії з користувачем.

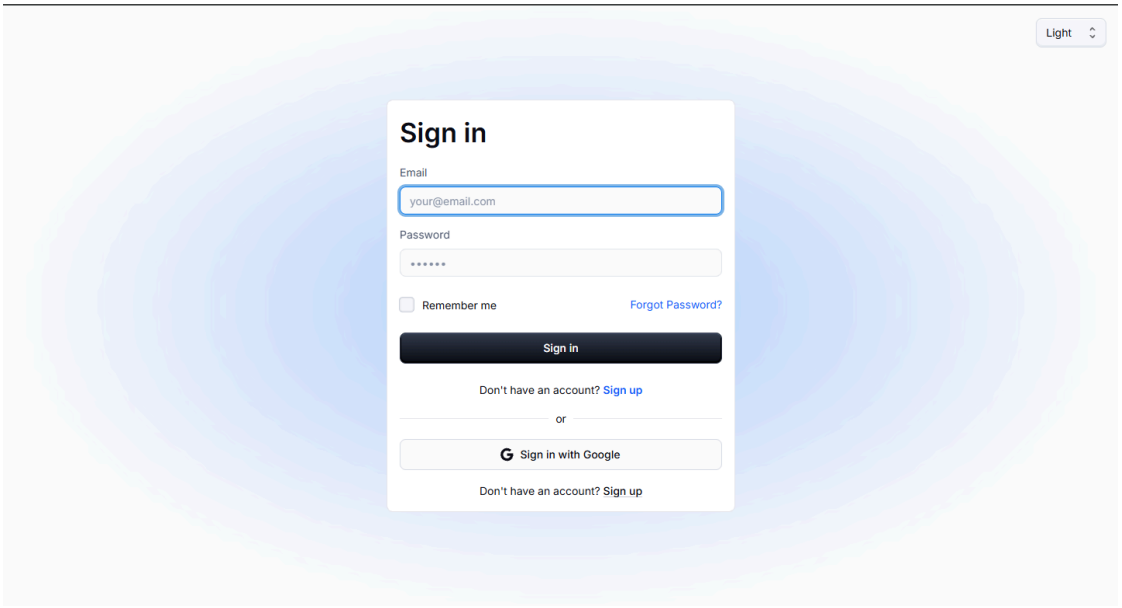


Рисунок 3.7 – Інтерфейс сторінки автентифікації

Центральним елементом інтерфейсу є сторінка "Дашборд". Її архітектура спроєктована як ієрархія вкладених компонентів, що дозволяє інкапсулювати логіку та полегшує перевикористання (рис. 3.8).

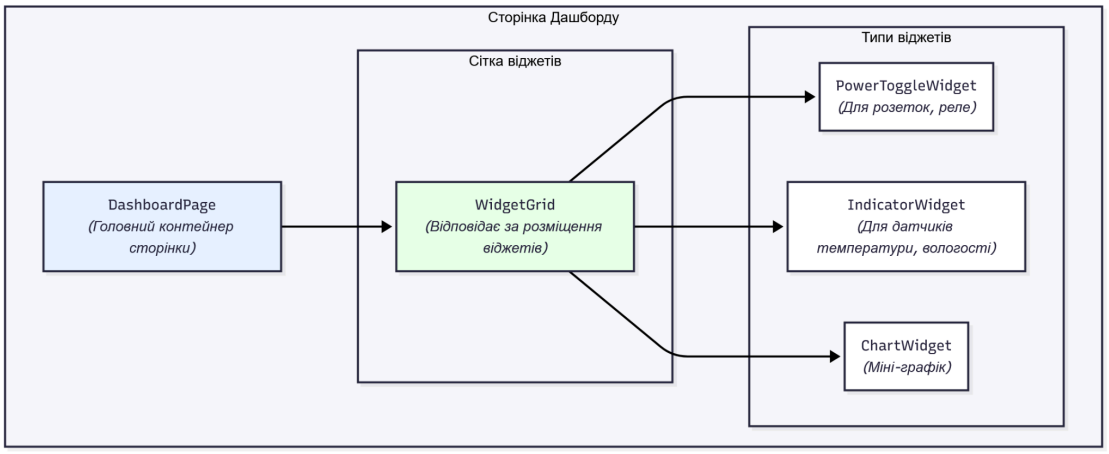


Рисунок 3.8. Ієрархія React-компонентів для сторінки "Дашборд"

- DashboardPage: Компонент-контейнер, що відповідає за отримання загального списку пристроїв та їх стану.
- WidgetGrid: Компонент-макет, що динамічно рендерить сітку та розміщує в ній віджети на основі даних, отриманих від батьківського компонента.
- PowerToggleWidget, IndicatorWidget: Спеціалізовані, "атомарні" компоненти, кожен з яких відповідає за візуалізацію даних від конкретного типу пристрою та обробку взаємодії з ним.

Реалізація цієї компонентної ієрархії у вигляді програмного прототипу дозволила створити загальний макет головного екрану системи (рис. 3.9). На даному етапі розробки він демонструє основні структурні елементи інтерфейсу.

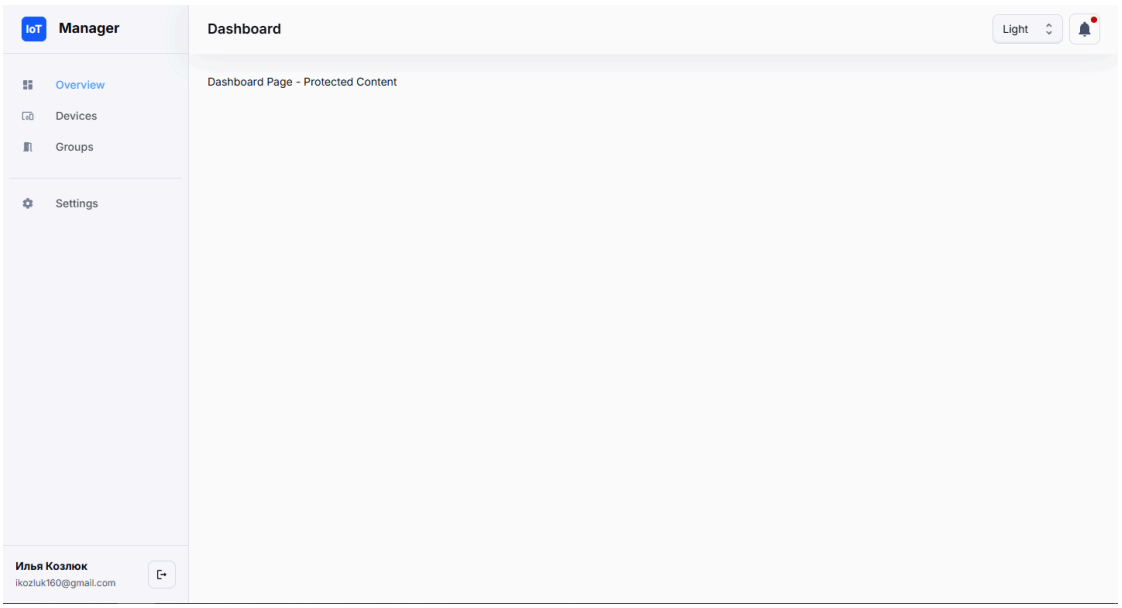


Рисунок 3.9. Програмна реалізація макету головного екрану системи

Як видно з наведеного зображення, макет інтерфейсу спроєктовано з використанням трьох основних функціональних зон. Верхня панель (Header) містить назву системи та елементи керування обліковим записом, тоді як бічна навігаційна панель (Sidebar) забезпечує швидкий доступ до ключових розділів, таких як «Дашборд», «Пристрої» та «Налаштування». Центральним елементом виступає основна робоча область (Content Area) — динамічний простір, призначений для рендерингу компонента WidgetGrid із набором інтерактивних віджетів.

Використання такої компоновки є загальноприйнятим стандартом для сучасних веб-додатків. Вона дозволяє забезпечити інтуїтивно зрозумілу навігацію та логічне групування елементів керування, що створює зручне середовище для взаємодії користувача із системою.

3.3.3 Проєктування взаємодії з бекендом та управління станом

Взаємодія фронтенду з бекендом є чітко розділеною на два канали: REST API для "холодних" даних та WebSocket для "гарячих" даних.

Для управління станом додатку використовуються вбудовані інструменти React: хуки useState, useReducer та Context API. Глобальний стан, що включає інформацію про користувача, список пристроїв та їхні поточні показники, зберігається у React Context, що робить його доступним для всіх компонентів без необхідності "прокидувати" пропси через все дерево.

Оновлення інтерфейсу при отриманні даних через WebSocket відбувається за наступною схемою (рис. 3.10).

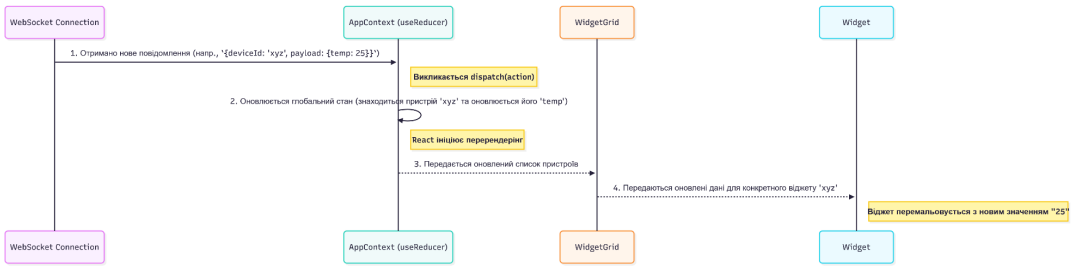


Рисунок 3.10. Схема потоку даних при оновленні через WebSocket

Нижче наведено спрощений приклад коду кастомного хука useWebSocket, що інкапсулює логіку роботи з WebSocket та оновлення глобального стану.

code TypeScript

```
  // hooks/useWebSocket.ts (спрощено)

import { useContext, useEffect } from 'react';
import { AppContext } from '../context/AppContext';
import { io } from 'socket.io-client'; // або нативний WebSocket

export const useWebSocket = (token: string) => {
  const { dispatch } = useContext(AppContext);

  useEffect(() => {
    if (!token) return;

    // 1. Встановлюємо з'єднання, передаючи JWT для автентифікації
    const socket = io('wss://your-api-gateway.com', { auth: { token } });

    // 2. Підписуємось на подію оновлення даних
    socket.on('device:update', (data: DeviceUpdatePayload) => {
      // 3. Викликаємо дію для оновлення глобального стану
      dispatch({ type: 'UPDATE_DEVICE_STATE', payload: data });
    });
```

```
    // Закриваємо з'єднання при розмонтуванні компонента
    return () => {
      socket.disconnect();
    };
  }, [token, dispatch]);
};
```

Такий підхід дозволяє ізолювати складну логіку роботи з мережею у спеціалізованих хуках, залишаючи UI-компоненти простими та сфокусованими виключно на візуалізації даних.

3.4 Валідація архітектурних рішень та демонстрація роботи прототипу

Заключним етапом дослідження в рамках даного розділу є валідація розроблених проєктно-програмних рішень. Метою валідації є не вимірювання граничних показників продуктивності, а експериментальне підтвердження того, що обрана архітектура є життєздатною та коректно вирішує поставлені задачі. Валідація проводилась шляхом наскрізного функціонального тестування ключових сценаріїв роботи системи.

3.4.1 Опис методики функціональної валідації

Тестовий стенд було розгорнуто в локальному середовищі з використанням інструменту Docker Compose, що дозволило повністю відтворити реальну схему взаємодії всіх мікросервісів та інфраструктурних елементів. Для емуляції зовнішніх подій та генерації вхідного потоку даних застосовувався графічний клієнт MQTT Explorer, за допомогою якого здійснювалося ручне відправлення тестових повідомлень на брокер Mosquitto.

Спостереження за коректністю обробки даних на проміжних етапах забезпечувала веб-панель керування RabbitMQ, яка дозволяла в реальному часі моніторити стан черг та обмінників, підтверджуючи проходження повідомлень через асинхронну шину. Для фінальної верифікації результатів на стороні клієнта використовувалися інструменти розробника веб-браузера (зокрема, вкладка «Network»), що надало можливість детально аналізувати REST API запити та відповіді від сервера.

3.4.2 Демонстрація роботи конвеєра обробки даних на бекенді

Цей сценарій валідує ключову частину архітектури — асинхронний конвеєр прийому, уніфікації та обробки даних від пристрою.

Крок 1: Відправка даних. За допомогою MQTT Explorer було сформовано та відправлено тестове JSON-повідомлення в топік, що прослуховується IoT Broker Service. Повідомлення імітувало дані від датчика Zigbee (рис. 3.11).

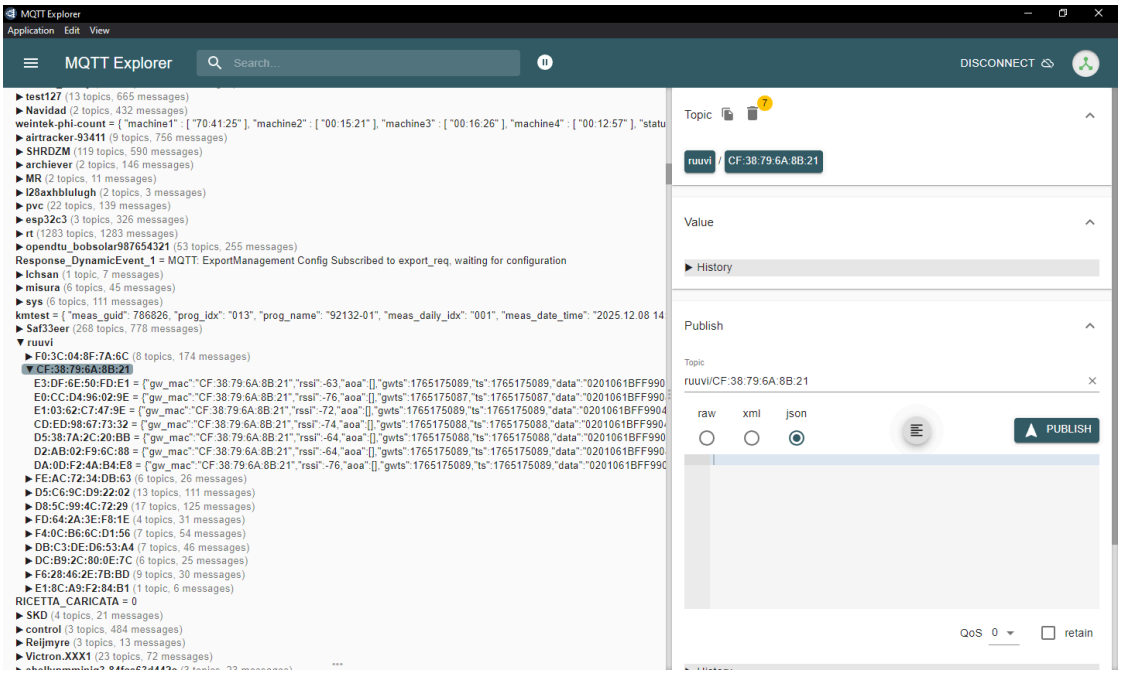


Рисунок 3.11 – Відправка тестового повідомлення через MQTT Explorer

Крок 2: Проходження через RabbitMQ. Одразу після відправки у панелі управління RabbitMQ було зафіксовано проходження повідомлення через систему. На першому етапі повідомлення, опубліковане IoT Broker Service, потрапило в

чергу normalized-data, а потім, після обробки Statistics Service, з'явилося в черзі realtime-updates, призначеній для API Gateway (рис. 3.12).

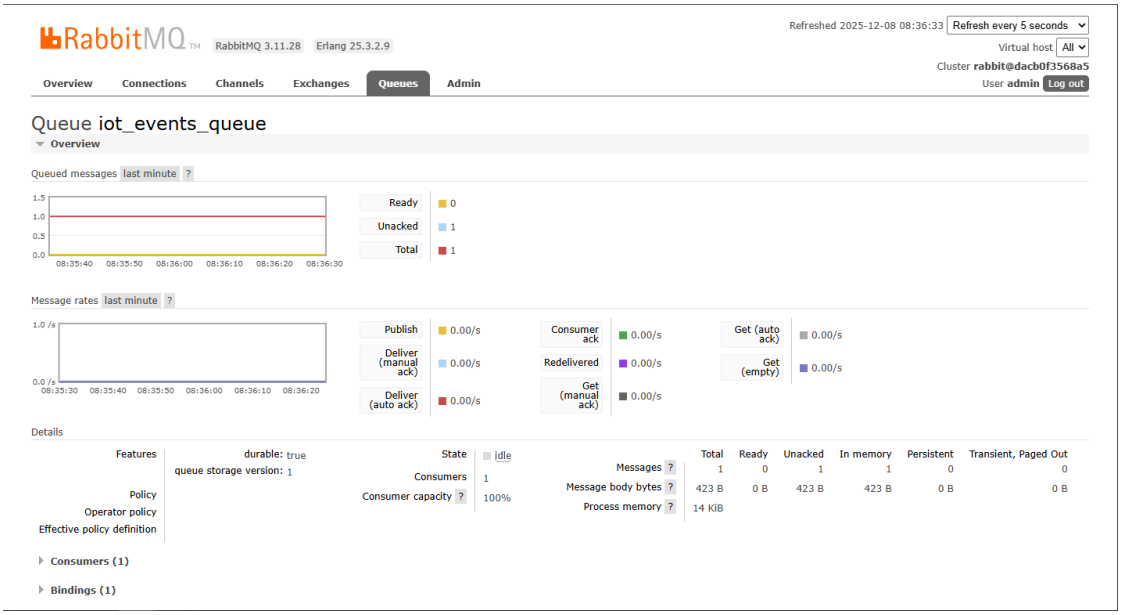


Рисунок 3.12 – Моніторинг проходження повідомлення через черги RabbitMQ

Крок 3: Результат. Успішне проходження повідомлення через RabbitMQ є матеріальним доказом того, що асинхронна, подієво-орієнтована частина архітектури працює коректно. Це підтверджує, що сервіси слабко зв'язані, а дані гарантовано доставляються між ними.

3.4.3 Валідація механізму автентифікації на базі JWT

Цей сценарій валідує роботу Auth Service та взаємодію клієнта з API Gateway для отримання токенів доступу.

Крок 1: Запит на автентифікацію. На сторінці логіну було введено коректні облікові дані користувача та відправлено форму.

Крок 2: Аналіз відповіді сервера. У вкладці "Network" інструментів розробника браузера було зафіксовано POST-запит на ендпоінт /api/auth/login. У відповідь від сервера було отримано статус 201 Created та тіло відповіді, що містить пару JWT-токенів: короткоживучий accessToken та довгоживучий refreshToken (рис. 3.13).

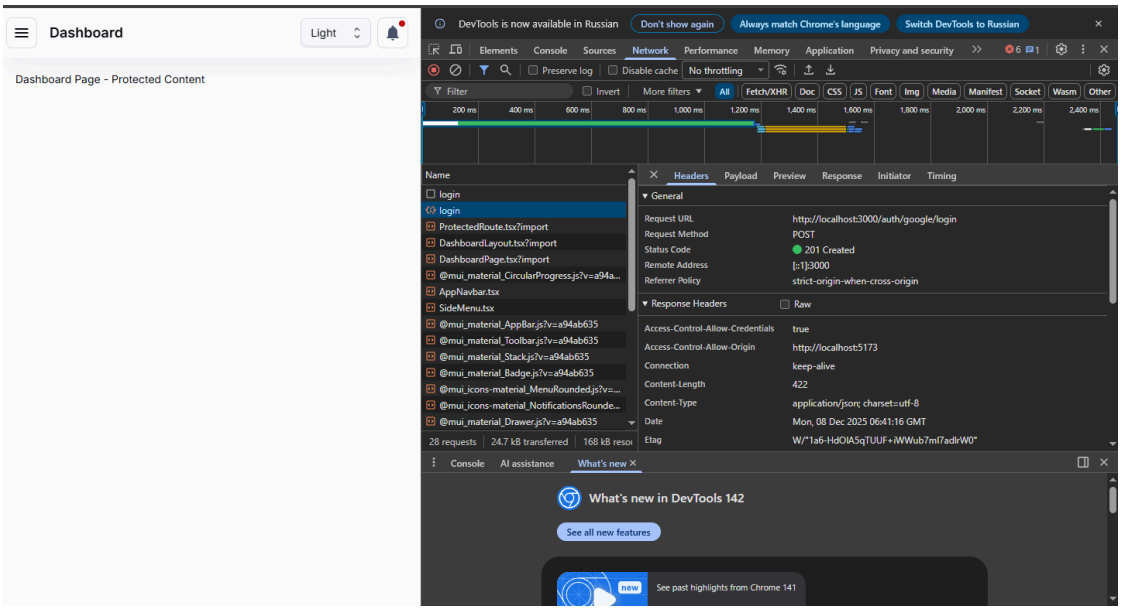


Рисунок 3.13 – Отримання JWT-токенів у відповідь на успішну автентифікацію

Крок 3: Результат. Успішне отримання токенів підтверджує коректну роботу механізму автентифікації, взаємодію API Gateway з Auth Service через gRPC та правильне налаштування публічного REST API. Отриманий accessToken надалі використовувався клієнтом для авторизації доступу до інших захищених ендпоінтів.

3.4.4 Оцінка відповідності архітектурних рішень поставленим вимогам

Проведена функціональна валідація дозволяє здійснити якісну оцінку ефективності закладених проєктних рішень. Зокрема, успішна демонстрація роботи конвеєра даних підтверджує правильність реалізації архітектури IoT Broker Service та концепції «профілів пристроїв», що свідчить про гнучкість та розширюваність системи. Водночас результати моніторингу черг у RabbitMQ доводять дієвість механізмів буферизації, які виступають фундаментом відмовостійкості платформи, забезпечуючи її стабільність відповідно до теоретичних принципів.

Окрему увагу було приділено аспектам безпеки та керованості: успішна перевірка механізму автентифікації засвідчує, що система володіє надійним стандартизованим інструментарієм контролю доступу. Підсумовуючи результати випробувань, можна констатувати, що розроблений програмний прототип повною

мірою валідує ключові переваги запропонованої мікросервісної архітектури та підтверджує її практичну цінність.

3.5 Висновки до розділу

У даному розділі було представлено комплекс проєктно-програмних рішень, розроблених для реалізації універсальної системи моніторингу та керування IoT-пристроями, а також наведено результати валідації ключових архітектурних рішень.

На першому етапі було розроблено та детально описано комплексну мікросервісну архітектуру системи, що базується на загальновизнаних інженерних принципах, таких як Принцип єдиної відповідальності (SRP) та слабка зв'язаність. Було представлено компонентну модель, що ілюструє статичну структуру системи, та діаграму послідовності, що демонструє динаміку асинхронної, подієво-орієнтованої обробки даних. Особливу увагу було приділено проєктуванню API Gateway, який виконує роль "Фасаду" та централізує такі наскрізні задачі, як маршрутизація, автентифікація та валідація вхідних даних.

На другому етапі було проведено глибоке проєктування ключових мікросервісів. Для кожного сервісу (Auth Service, Device Management Service, IoT Broker Service) було представлено його внутрішню компонентну архітектуру, деталізовано моделі даних на рівні сутностей (Entity) для СУБД PostgreSQL, описано API-контракти (REST та gRPC), а також наведено ілюстративні фрагменти коду та псевдокоду. Ключовим проєктним рішенням, детально описаним у цьому розділі, є концепція "профілів пристроїв", яка забезпечує гнучкість та розширюваність системи. Також було детально спроєктовано клієнтську частину, її компонентну ієрархію та механізми взаємодії з бекендом.

На завершальному етапі було проведено функціональну валідацію розробленого програмного прототипу. За допомогою інструментів MQTT Explorer та панелі управління RabbitMQ було продемонстровано та підтверджено коректну роботу наскрізного асинхронного конвеєра обробки даних. Також було валідовано механізм автентифікації на базі JWT. Результати валідації підтвердили, що обрані

архітектурні рішення є життєздатними та відповідають поставленим вимогам щодо гнучкості, надійності та безпеки.

Таким чином, завдання, поставлені на даний розділ, є повністю виконаними. Розроблено та детально задокументовано архітектуру системи, а її ключові аспекти успішно валідовано на програмному прототипі. Це створює міцну інженерну та дослідницьку основу для подальшого розвитку проєкту та підтверджує досягнення мети кваліфікаційної магістерської роботи.

Висновки

У даній кваліфікаційній магістерській роботі було проведено дослідження та розробку архітектури для універсальної системи моніторингу та керування пристроями Інтернету речей. Робота була спрямована на вирішення актуальної проблеми фрагментації ринку IoT, яка створює значні незручності для кінцевих користувачів та ускладнює інтеграцію пристроїв від різних виробників. Метою роботи була розробка та дослідження гнучкої мікросервісної архітектури, здатної подолати цю проблему, та валідація її ключових аспектів на програмному прототипі.

За результатами виконаної роботи було отримано низку важливих аналітичних, проєктних та практичних результатів. Було проведено комплексний аналіз предметної області, сучасних технологій та існуючих рішень, в ході якого було виявлено незайняту ринкову нішу для хмарної SaaS-платформи, орієнтованої на масового кінцевого користувача. На основі аналізу архітектурних підходів, методів комунікації та технологій було обґрунтовано вибір комплексної методики проєктування.

Було розроблено та детально описано гнучку, масштабовану та відмовостійку мікросервісну архітектуру, що становить наукову новизну роботи. Ключовими проєктними рішеннями є декомпозиція системи на незалежні мікросервіси, використання високоефективного протоколу gRPC для внутрішньосервісної комунікації, впровадження асинхронної, подієво-орієнтованої взаємодії на базі брокера повідомлень RabbitMQ, а також застосування концепції "профілів пристроїв" для гнучкої інтеграції різнорідного обладнання.

Було створено програмний прототип, що реалізує запропоновану архітектуру, та проведено його функціональну валідацію, що підтверджує практичне значення роботи. В ході валідації було продемонстровано коректну роботу наскрізного асинхронного конвеєра обробки даних та надійність механізму автентифікації. Це доводить, що розроблена архітектура є життєздатною і може слугувати шаблоном (blueprint) для створення аналогічних систем.

Визначено також можливі напрями подальшої роботи. До них належить функціональний розвиток, зокрема реалізація користувацького двигуна правил для створення сценаріїв автоматизації. Важливим кроком є інфраструктурний розвиток, що передбачає міграцію системи на платформу оркестрації контейнерів Kubernetes для забезпечення високої доступності та автоматичного масштабування. Крім того, перспективним є розширення інтеграцій шляхом додавання підтримки нових протоколів та екосистем «розумного будинку».

Таким чином, усі поставлені у роботі завдання є виконаними, а мета дослідження — досягнутою.

Список використаних джерел

1. Statista. *Internet of Things (IoT) connected devices worldwide from 2019 to 2030*. [Електронний ресурс] – Режим доступу до ресурсу: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/
2. Grand View Research. *Internet of Things (IoT) Market Size, Share & Trends Analysis Report*. (Змінено посилання на більш актуальне) [Електронний ресурс] – Режим доступу до ресурсу: https://www.grandviewresearch.com/industry-analysis/internet-of-things-iot-market
3. Vermesan, O., & Friess, P. *Internet of Things: Converging Technologies for Smart Environments and Integrated Ecosystems*. River Publishers, 2013.
4. Gilchrist, A. *Industry 4.0: The Industrial Internet of Things*. Apress, 2016.
5. Ray, P. P. "A survey on Internet of Things architectures." *Journal of King Saud University-Computer and Information Sciences* 30.3 (2018): 291-319.
6. Hunkeler, U., Truong, H. L., & Stanford-Clark, A. "MQTT-S—A publish/subscribe protocol for Wireless Sensor Networks." *2008 Second International Conference on Sensor Technologies and Applications (sensorcomm 2008)*.
7. Fielding, R. T. "Architectural styles and the design of network-based software architectures." *Dissertation, University of California, Irvine*, 2000.
8. Richards, M. "Software Architecture Patterns: Understanding Common Architecture Patterns and When to Use Them." *O'Reilly Media*, 2015.
9. Sadalage, P., & Fowler, M. "NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence." *Addison-Wesley Professional*, 2012.
10. Official Documentation. Home Assistant. [Електронний ресурс] – Режим доступу до ресурсу: https://www.home-assistant.io/docs/ (дата звернення: 19.10.2024).

11. Installation Guide. Home Assistant. [Електронний ресурс] – Режим доступу до ресурсу: https://www.home-assistant.io/installation/ (дата звернення: 19.10.2024).
12. What Is AWS IoT Core? AWS Documentation. [Електронний ресурс] – Режим доступу до ресурсу: https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html (дата звернення: 19.10.2024).
13. Офіційна документація ThingsBoard. "Rule Engine Overview". [Електронний ресурс] – Режим доступу до ресурсу: https://thingsboard.io/docs/user-guide/rule-engine/
14. Sethi, P., & Sarangi, S. R. "Internet of Things: Architectures, protocols, and applications." *Journal of Electrical and Computer Engineering* 2017 (2017).
15. Офіційна документація gRPC. "Introduction to gRPC". [Електронний ресурс] – Режим доступу до ресурсу: https://grpc.io/docs/what-is-grpc/introduction/
16. Fette, I., & Melnikov, A. "The WebSocket protocol." *RFC 6455, IETF*, 2011.
17. Офіційна документація RabbitMQ. "AMQP 0-9-1 Concepts". [Електронний ресурс] – Режим доступу до ресурсу: https://www.rabbitmq.com/tutorials/amqp-concepts.html
18. Hardt, D. "The OAuth 2.0 authorization framework." *RFC 6749, IETF*, 2012.
19. Офіційна документація InfluxDB. "Key concepts in InfluxDB". [Електронний ресурс] – Режим доступу до ресурсу: https://docs.influxdata.com/influxdb/v2/reference/key-concepts/
20. Офіційна документація Redis. "Introduction to Redis". [Електронний ресурс] – Режим доступу до ресурсу: https://redis.io/docs/about/
21. Kranz, M. "Building the Internet of Things: Implement new business models, disrupt competitors, and transform your industry." *John Wiley & Sons*, 2016.
22. Dierks, T., & Rescorla, E. "The Transport Layer Security (TLS) Protocol Version 1.2." *RFC 5246, IETF*, 2008.
23. Newman, S. "Building Microservices: Designing Fine-Grained Systems." *O'Reilly Media*, 2015.

24. Hohpe, G., & Woolf, B. "Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions." *Addison-Wesley Professional*, 2003.
25. MDN Web Docs. "Single-page application (SPA)". [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Glossary/SPA
26. The PostgreSQL Global Development Group. *PostgreSQL: About*. [Електронний ресурс] – Режим доступу до ресурсу: https://www.postgresql.org/about/ (дата звернення: 20.10.2024).
27. Docker Inc. *What is a container?*. [Електронний ресурс] – Режим доступу до ресурсу: https://docs.docker.com/get-started/overview/#what-is-a-container (дата звернення: 20.10.2024).
28. Martin, R. C. "Agile Software Development, Principles, Patterns, and Practices." *Prentice Hall*, 2002.
29. Fowler, M. "Refactoring: Improving the Design of Existing Code." *Addison-Wesley Professional*, 2018.
30. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. "Design Patterns: Elements of Reusable Object-Oriented Software." *Addison-Wesley Professional*, 1994.
31. TypeORM. *Official Documentation*. [Електронний ресурс] – Режим доступу до ресурсу: https://typeorm.io/ (дата звернення: 21.10.2025).
32. Офіційна документація React. "Thinking in React". [Електронний ресурс] – Режим доступу до ресурсу: https://react.dev/learn/thinking-in-react
33. Richards, M., & Ford, N. "Fundamentals of Software Architecture: An Engineering Approach." *O'Reilly Media*, 2020.
34. *class-validator*. [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/typestack/class-validator (дата звернення: 22.10.2025).
35. SmartBear Software. *Swagger UI*. [Електронний ресурс] – Режим доступу до ресурсу: https://swagger.io/tools/swagger-ui/ (дата звернення: 22.10.2025).
36. Fowler, M. Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2002.

37. Richardson, C. Microservices Patterns: With examples in Java. Manning Publications, 2018. 400 p.
38. ISO/IEC 30141:2018. Internet of Things (IoT) — Reference Architecture. International Organization for Standardization, 2018.
39. Kleppmann, M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media, 2017. 616 p.
40. Cherny, B. Programming TypeScript: Making Your JavaScript Applications Scale. O'Reilly Media, 2019. 324 p.

Додаток А Слайди презентації

Мета та завдання кваліфікаційної магістерської роботи

Мета дослідження

Розробка та дослідження гнучкої мікросервісної архітектури для універсальної веб-системи моніторингу та керування пристроями Інтернету речей.

Об'єкт та предмет дослідження

Об'єкт: Процес управління та моніторингу гетерогенними IoT-пристроями.

Предмет: Методи, моделі та програмні засоби для інтеграції різнорідних IoT-пристроїв в єдину хмарну веб-платформу.

Основні завдання

1. Провести аналіз предметної області, технологій та існуючих рішень.
2. Спроєктувати гнучку мікросервісну архітектуру системи.
3. Розробити програмний прототип, що реалізує ключові компоненти архітектури.
4. Провести функціональну валідацію розроблених проєктних рішень.

Аналіз ринку та визначення незайнятої ніші

Локальні Open-Source системи

Home Assistant

Технічні ентузіасти

\+ Максимальна гнучкість та приватність даних.

− Високий поріг входу: вимагає технічних знань та власного обладнання.

Enterprise Open-Source системи

ThingsBoard

Бізнес (від СМБ до enterprise)

\+ Готові бізнес-функції (multi-tenancy, rule engine).

− Надлишковість: складний в адмініструванні та надлишковий для індивідуальних потреб.

Глобальні хмарні платформи

AWS IoT Core

Великі корпорації

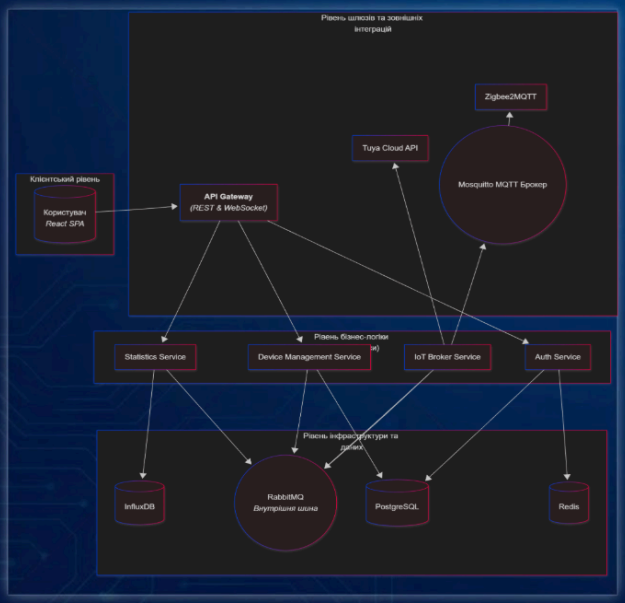
\+ Нескінченна масштабованість та екосистема.

− Дуже висока складність та непередбачувана вартість для малих проєктів.

Висновок: Незайнята ринкова ніша

Існує потреба в публічній хмарній SaaS-платформі, яка б поєднувала простоту використання хмарних сервісів з доступністю та орієнтацією на масового кінцевого користувача.

Розроблена компонентна архітектура системи



Ключові архітектурні рішення:

- **Мікросервісна архітектура**: Система декомпозована на незалежні, спеціалізовані сервіси.
- **Подієво-орієнтована взаємодія**: Сервіси спілкуються асинхронно через шину повідомлень RabbitMQ.
- **Єдина точка входу**: API Gateway виступає як "фасад", що інкапсулює внутрішню складність системи.

Проєктні рішення для серверної частини (Бекенду)

Основа мікросервісів

NestJS Framework

- Обрано як основу для всіх мікросервісів.
- Переваги:
- Надає потужну, структуровану архітектуру "з коробки" (модулі, сервіси, контролери).
- Вбудована підтримка TypeScript забезпечує надійність та масштабованість коду.
- Спрощує реалізацію складних шаблонів (напр., Dependency Injection).

Внутрішньосервісна комунікація

gRPC Protocol

- Обрано для синхронної комунікації між сервісами (напр., API Gateway -> Auth Service).
- Переваги:
- Висока продуктивність завдяки бінарному формату Protobuf та HTTP/2.
- Строгі контракти (.proto файли) виключають помилки інтеграції між сервісами.
- Значно швидший за традиційний REST API для внутрішніх викликів.

Асинхронна взаємодія

RabbitMQ Message Broker

- Обрано як асинхронну шину повідомлень для подієво-орієнтованої взаємодії.
- Переваги:
- Забезпечує слабку зв'язаність: сервіси не залежать один від одного напряму.
- Підвищує відмовостійкість: діє як буфер, що запобігає втраті даних при тимчасових збоях.
- Гнучка маршрутизація повідомлень завдяки реалізації протоколу AMQP.

Підхід "Polyglot Persistence": вибір сховища для кожної задачі

PostgreSQL

- **Призначення:** Зберігання структурованих метаданих.
- **Сутності:** Користувачі (Account), Пристрої (Device), Групи (Group).
- **Чому обрано:** Гарантія цілісності даних (ACID) та реляційні зв'язки.

InfluxDB

- **Призначення:** Зберігання часових рядів (телеметрії).
- **Сутності:** Показники сенсорів, історія змін станів.
- **Чому обрано:** Висока швидкість запису та стиснення часових даних.

Redis

- **Призначення:** Кешування та швидкий доступ.
- **Сутності:** Активні сесії, JWT-токени, "гарячий" стан пристроїв.
- **Чому обрано:** Мінімальні затримки доступу (in-memory).

Проєктні рішення для клієнтської частини (Фронтенду)

Архітектура SPA (Single Page Application)

- **Принцип:** Додаток завантажується один раз, далі працює динамічно.
- **Переваги:** Плавний інтерфейс без перезавантаження сторінок (як у нативних додатків).
- **Структура:** Компонентний підхід (перевикористання віджетів).

Двоканальна взаємодія з сервером

1. REST API (HTTP)
Для "холодних" даних (запит списків, відправка команд, логін).

2. WebSocket (WS)
Для "гарячих" даних (миттєве оновлення дашборду в реальному часі).

Наукова новизна отриманих результатів

Отримала подальший розвиток архітектура хмарних IoT-платформ

Створено рішення, що поєднує промислову надійність з простотою для кінцевого користувача.

Мікросервісна архітектура + gRPC

- Декомпозиція на незалежні сервіси.
- Високоефективна внутрішня комунікація.

Асинхронна взаємодія (RabbitMQ)

- Слабка зв'язаність компонентів.
- Гарантована доставка даних (буферизація).

Уніфікація через "Профілі пристроїв"

- Гнучка інтеграція різнорідного обладнання.
- Декларативний опис правил мапінгу даних.

Схема потоку обробки даних у реальному часі

Генерація

Пристрій надсилає "сирі" дані через MQTT (Mosquitto).

Уніфікація

IoT Broker Service приводить дані до єдиного формату (JSON) згідно з профілем.

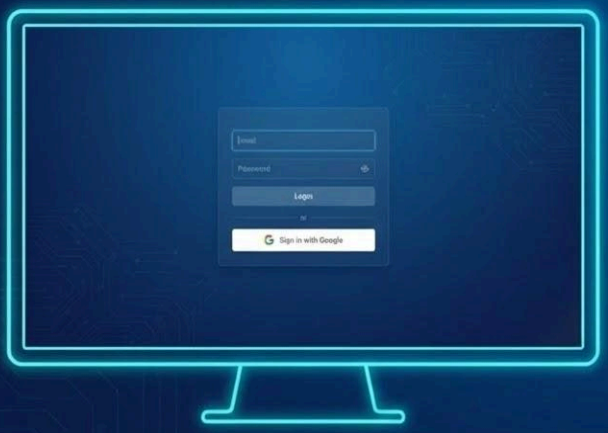
Асинхронна шина

Гарантована доставка повідомлення до сервісів обробки та шлюзу.

Візуалізація

API Gateway миттєво оновлює дашборд через WebSocket.

Демонстрація програмного прототипу



Сторінка авторизації
Реалізовано OAuth 2.0 (Google) та JWT-сесії.

Головний дашборд (SPA)
Компонентна архітектура на базі React.

Результати функціональної валідації архітектури

Асинхронна обробка

ПІДТВЕРДЖЕНО

Успішне проходження повідомлень через черги RabbitMQ від пристрою до шлюзу.

Відмовостійкість

ПІДТВЕРДЖЕНО

Відсутність втрати даних при імітації збою сервісу статистики (буферизація).

Механізм безпеки

ПІДТВЕРДЖЕНО

Коректна генерація та валідація JWT-токенів (Access/Refresh).

Практичне значення отриманих результатів

Архітектурний шаблон

Готовий "blueprint" для створення масштабованих IoT-систем. Економить час на проєктування архітектури для нових стартапів.

Прикладні сфери

- Системи "Розумний будинок".
- Моніторинг малого бізнесу (склади, серверні кімнати, агросектор).

Навчальна платформа

Використання як наочного посібника для вивчення мікросервісів, gRPC та RabbitMQ у навчальному процесі.

Висновки

- Проаналізовано проблему фрагментації ринку IoT та недоліки існуючих платформ.
- Розроблено та обґрунтовано гнучку мікросервісну архітектуру на базі сучасного стеку (NestJS, gRPC, RabbitMQ).
- Створено програмний прототип системи з веб-інтерфейсом (SPA) та підтримкою реального часу.
- Валідовано ключові архітектурні рішення (надійність, безпека, інтеграція) на практиці.

Мета кваліфікаційної магістерської роботи досягнута в повному обсязі.


Дякую за увагу!
Доповідь завершено.
Готовий відповісти на Ваші запитання
Козлюк Ілля Юрійович | 2025