НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
В. о. завідувача кафедри
_____ Михайло НОВОТАРСЬКИЙ
(підпис)

“__” _____2025 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

на тему: Сервіс для віддаленого моніторингу стану та керування пристроями

Виконав : студент __4__ курсу, групи ІМ-13
(шифр групи)

_____Землянський Олександр Валентинович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____асистент Гордієнко Н.Ю._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант _____асистент Міщенко Л.Д._____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент _____асистент Соколовський В.В._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2025 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
В. о. завідувача кафедри
_____ Михайло НОВОТАРСЬКИЙ
(підпис)

“__” _____2025 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Землянського Олександра Валентиновича

1. Тема проєкту *Сервіс для віддаленого моніторингу стану та керування пристроями*
керівник проєкту Гордієнко Нікіта Юрійович, асистент
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 23 травня 2025 року №1705-с

2. Термін здачі студентом закінченого проєкту _____

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

Розділ 1 Огляд існуючих систем

Розділ 2 Огляд технологій для розробки системи

Розділ 3 Деталі розробки системи

Розділ 4 Дослідження та аналіз розробленої системи

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структурна схема системи, функціональна схема (діаграма класів), алгоритм дій програмного забезпечення.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Міщенко Л.Д.		

7. Дата видачі завдання: 25 вересня 2024___

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	10.01.2025 – 17.01.2025	
2.	*Вивчення та аналіз завдання*	24.01.2025 – 16.03.2025	
3.	*Розробка архітектури та загальної структури системи*	16.03.2025 – 01.04.2025	
4.	*Розробка структур окремих підсистем*	01.04.2025 – 11.04.2025	
5.	*Програмна реалізація системи*	11.04.2025 – 01.05.2025	
6.	*Оформлення пояснювальної записки*	01.05.2025 – 25.05.2025	
7.	*Захист програмного продукту*	29.05.2025	
8.	*Передзахист*	30.05.2025	
9.	*Захист*	19.06.2025	

Студент-дипломник _____ Олександр ЗЕМЛЯНСЬКИЙ
(підпис)

Керівник проєкту _____ Нікіта ГОРДІЄНКО
(підпис)

АНОТАЦІЯ

У ході виконання дипломної роботи було реалізовано сервіс для віддаленого моніторингу та керування пристроями у реальному часу. Архітектура проєкту поєднує REST та WebSocket протоколи для забезпечення двосторонньої комунікації між сервером і клієнтськими застосунками. Серверна частина реалізована з використанням NestJS, PostgreSQL з розширенням для роботи з часовими рядами, а також кешуючої системи на базі Redis, що забезпечує високу швидкодію. У фокусі реалізації перебуває уніфікована архітектура, що забезпечує взаємодію між клієнтами та сервером за допомогою протоколу gRPC — сучасного високопродуктивного рішення для обміну повідомленнями. Розроблено набір інтерфейсів для збирання, обробки та зберігання телеметричних даних у режимі реального часу, включаючи підтримку різних класів пристроїв.

Ключові слова: адміністрування проектів, моніторинг пристроїв, gRPC, IoT, телеметрія, аутентифікація, кешування, рольовий доступ, WebSocket, REST, NestJS, NodeJS

ANNOTATION

During the thesis, a service for remote monitoring and control of Пристрої with real-time support was implemented. The project architecture integrates the REST and WebSocket protocols, thereby facilitating bidirectional communication between the server and client applications. The server side employs NestJS, PostgreSQL with an extension for working with time series, and a Redis-based caching system that provides high performance. The implementation is centered on a unified architecture that facilitates client-server interaction through the utilization of the gRPC protocol, a contemporary high-performance messaging solution. A set of interfaces has been developed for the collection, processing and storage of telemetry data in real time, including support for various classes of devices.

Key words: project administration, device monitoring, gRPC, IoT, telemetry, authentication, caching, role-based access, WebSocket, REST, NestJS, NodeJS

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Сервіс для віддаленого	4		
			моніторингу стану та			
			керування пристроями			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Сервіс для віддаленого	75		
			моніторингу стану та			
			керування пристроями			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Структурна схема системи	1		
	А4	***ІАЛЦ.467200.005 Д2***	Діаграма сутностей	1		
	А4	***ІАЛЦ.467200.006 Д3***	Алгоритм дій програмного	1		
			забезпечення			
	А4	***ІАЛЦ.467200.007 Д4***	Текст програмного коду	79		

					# *ІАЛЦ.467200.001 ОА*		
Зм	*Лист*	*№ докум.*	*Підп*	*Дата*			
Розроб		Землянський О.В.			*Сервіс для віддаленого моніторингу стану та керування пристроями* Опис альбому	Літ.	Аркуш 1 / Аркушів 1
Перев		Гордієнко Н.Ю.					
						НТУУ "КПІ" ФІОТ ІМ-13	

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Сервіс для віддаленого моніторингу стану та керування пристроями*»

Київ – 2025

ЗМІСТ

НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ 2

ПІДСТАВИ ДЛЯ РОЗРОБКИ 2

МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ 2

ДЖЕРЕЛА РОЗРОБКИ 2

ТЕХНІЧНІ ВИМОГИ 3

- Вимоги до розробленого продукту 3
- Вимоги до програмного забезпечення 3
- Вимоги до апаратної частини 3

ЕТАПИ РОЗРОБКИ 4

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку сервісу для віддаленого моніторингу стану та керування пристроями та його подальшу підтримку й вдосконалення системи.

Областю застосування цієї системи є адміністрування та моніторинг роботи систем інтернету речей в різних доменах, а саме: розумних будинках, промислових або складських об’єктах, міській інфраструктурі та системах охорони.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є унифікація та спрощення адміністрування і моніторингу систем інтернет речей шляхом створення стандартизованої платформи, яка буде слугувати універсальним «шаром абстракції» між фізичним пристроєм та кінцевим користувачем.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки даного дипломного проекту є офіційні документації, публікації та статті в мережі Інтернет на дану тему, науково-технічна література.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Простий і інтуїтивно-зрозумілий серверний інтерфейс для взаємодії.
- Надати можливість користувачу підключення фізичних пристроїв з системи інтернет речей.
- Створити універсальний інтерфейс для підключення пристроїв та можливістю легкої інтеграції нових незалежно від протоколу взаємодії.
- Надати вичерпну та зрозумілу документацію до розрозбленого продукту.

5.2. Вимоги до програмного забезпечення

- ОС Windows, Mac чи Linux.
- Node.js версії 20.x або вище
- Yarn версії 4.9.x або вище
- PostgreSQL версії 16.x або вище
- Redis версії 7.x або вище

5.3. Вимоги до апаратної частини

- ЦП на архітектурі AMD64 (x86_64) або ARM.
- RAM не менше ніж 4 ГБ.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	10.01.2025 – 17.01.2025
Вивчення та аналіз завдання	24.01.2025 – 16.03.2025
Розробка архітектури та загальної структури системи	16.03.2025 – 01.04.2025
Розробка структур окремих частин системи	01.04.2025 – 11.04.2025
Програмна реалізація системи	11.04.2025 – 01.05.2025
Виправлення помилок	01.05.2025 – 25.05.2025
Оформлення пояснювальної записки	29.05.2025

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Сервіс для віддаленого моніторингу стану та керування пристроями*»

Київ – 2025

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ 4
ВСТУП 5
РОЗДІЛ 1 ОГЛЯД ІСНУЮЧИХ СИСТЕМ 7
1.1 Основні поняття та вимоги 7
1.2 Приклади існуючих рішень 8
1.2.1 Ядро розумних пристроїв Amazon 8
1.2.2 Хаб розумних пристроїв Azure 10
1.2.3 ThingsBoard 12
1.3 Порівняльний аналіз платформ 14
ВИСНОВОК ДО РОЗДІЛУ 1 16
РОЗДІЛ 2 ОГЛЯД ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ СИСТЕМИ 18
2.1 Мови розробки 18
2.1.1 Python 18
2.1.2 Java 19
2.1.3 JavaScript 19
2.1.4 TypeScript 20
2.1.5 Висновок 20
2.2 Середовище виконання 21
2.2.1 Node.js 22
2.2.2 Bun 24
2.2.3 Deno 24
2.2.4 Висновок 25
2.3 Бібліотеки прикладного інтерфейсу 26
2.3.1 ExpressJS 26
2.3.2 FastifyJS 27
2.3.3 NestJS 28
2.3.4 Висновок 29

2.4 Прикладні протоколи 30
2.4.1 Передача репрезентативного стану 30
2.4.2 Веб шлюз 31
2.4.3 Виклик віддаленої процедури 31
2.4.4 Графова мова запитів 32
2.4.5 Висновок 32
2.5 СУБД 33
2.5.1 MySQL 33
2.5.2 PostgreSQL 34
2.5.3 MongoDB 34
2.6 Кешування даних 35
2.6.1 Кешування у пам’яті додатку 36
2.6.2 Тимчасове сховище кешу 37
2.6.3 Висновок 39
ВИСНОВОК ДО РОЗДІЛУ 2 40
РОЗДІЛ 3 ДЕТАЛІ РОЗРОБКИ СИСТЕМИ 43
3.1 Основні компоненти системи 43
3.2 Основний сервіс 43
3.2.1 Архітектура сервісу 43
3.2.2 Модуль пристроїв 45
3.2.3 Модуль підключень 47
3.2.4 Модуль користувачів 48
3.3 Адаптери підключення 49
3.3.1 Інтерфейс взаємодії з основним сервісом 50
3.3.2 Підключення та симуляція пристроїв в цілях демонстрації та тестування 51
3.4 Клієнтська частина 53
3.4.1 Сторінка пристроїв 54
3.4.2 Сторінка подій 55

3.4.3 Сторінка тригерів 55
3.4.4 Сторінка адаптерів 56
3.4.5 Сторінка автентифікації 56
3.5 Безпека системи 57
3.5.1 Модуль автентифікації 57
3.5.2 Захист обміну даних між основним сервісом та адаптерами 58
ВИСНОВОК ДО РОЗДІЛУ 3 59
РОЗДІЛ 4 ДОСЛІДЖЕННЯ ТА АНАЛІЗ РОЗРОБЛЕНОЇ СИСТЕМИ 61
4.1 Встановлення цілей дослідження та основних вимог 61
4.2 Перевірка базової роботи прикладного інтрефейсу з використанням уніфікованого програмного забезпечення 61
4.2.1 Потік телеметрії з адаптера 61
4.2.2 Двосторонній шлюз основного сервісу 66
4.2.3 Кінцеві точки авторизації та автентифікації 66
4.3 Клієнтська частина 70
4.3.1 Авторизація, автентифікація 71
4.3.2 Управління пристроями 73
4.3.3 Журнал подій 76
4.3.4 Налаштування тригерів 78
4.3.5 Налаштування адаптерів 80
ВИСНОВОК ДО РОЗДІЛУ 4 81
ВИСНОВКИ 82
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 83
ДОДАТОК 1 3
ДОДАТОК 2 5
ДОДАТОК 3 7
ДОДАТОК 4 9

ПЕРЕЛІК СКОРОЧЕНЬ

IoT	(Internet of Things) Інтернет речей
API	(Application Portable Interface) Прикладний інтерфейс додатку
JSON	(JavaScript Object Notation) Об’єкт запису JavaScript
RPC	(Remote Procedure Call) Віддалений виклик процедури
REST	(Representational State Transfer) Передача репрезентативного стану
СУБД	Система Управління Базою Даних
SQL	(Simple Query Language) Проста мова запитів
AWS	(Amazon Web Services) Мережеві сервіси Amazon
MQTT	(Message Queue Telemetry Transport) Транспорт телеметрії черги повідомлень
HTTP	(Hyper Text Transport Protocol) Транспортний протокол гіпер-тексту
TLS	(Transport Layer Security) Безпека транспортного рівня
TTL	(Time To Live) Час життя
SDK	(Standard Developer Kit) Стандартний набір розробника
gRPC	(google Remote Procedure Call) Віддалений виклик процедури за стандартом Google
JS	JavaScript
UUID	(Universally Unique Identifier) Універсально-унікальний ідентифікатор

ВСТУП

Технології стрімко змінюються та розвиваються. Побутові прилади, конвеєрні лінії, транспортні засоби й навіть ліхтарі міського освітлення дедалі частіше набувають здатності автономно збирати дані, передавати їх у мережу та реагувати на віддалені команди. Цей зсув від локальної автоматизації до пов'язаних між собою «розумних» об'єктів є основоположним для виникнення такого поняття, як «Інтернет речей». Це створює як і великі можливості, так і нові виклики одним із яких може бути складність інтеграції різних пристроїв у єдину систему.

Однією з головних проблем при створенні систем інтернету речей являє собою питання сумісності окремих приладів від різних виробників. Кожен з них просуває власну шину зв'язку, власний формат повідомлень і нерідко — власну інфраструктуру для зберігання даних. Інженер, який намагається побудувати комплексну систему моніторингу, змушений підтримувати велику кількість драйверів, шлюзів та конверторів. Це не лише ускладнює масштабування й обслуговування, а й може підвищити фінансові витрати.

Саме тому необхідно створити сервіс який дозволить унифікувати підключення пристроїв та спростить їх контроль. Сервіс для віддаленого моніторингу стану та керування пристроями, має дозволити підключити декілька пристроїв без прив'язки до драйверу або протоколів з'єднання. Основна ідея полягає в розділенні системи на два чіткі шари. Перший – універсальний сервіс, який збирає й зберігає події та телеметрію з пристроїв, відсилає сповіщення за встановленими користувачем умовами та забезпечує безпеку за допомогою чіткої рольової моделі доступу. Другий – це адаптер який уніфікує інформацію отриману з пристроїв до стандарту, яке підтримує ядро, та відправляє її на збереження в ядро. Він має бути простим та виконувати лише посередницьку функцію.

Такий підхід може значно спростити подальше масштабування проєкту. Наприклад, поява нового класу пристроїв може потребувати розширення програмної бази без перебудови всієї архітектури. Централізований доступ спрощує адміністрування та моніторинг, а узгоджений формат телеметрії полегшує її подальшу обробку та вивчення, так як зберігається в одній базі даних. Тому, з написаного вище, можна зробити висновок, що питання стандартизації підключень пристроїв та спрощення адміністрування й моніторингу систем інтернет речей є важливим.

РОЗДІЛ 1 ОГЛЯД ІСНУЮЧИХ СИСТЕМ

1.1 Основні поняття та вимоги

Інтернет речей (IoT) це сукупність фізичних пристроїв, оснащених засобами мережевого зв’язку, вбудованою логіки та ідентифікації. Ці пристрої здатні самостійно створювати, приймати да обмінюватися даними. Кожен такий приклад має свій набор сенсорів або акторів та працює за власним протоколом [1].

Набори сенсорів збирають дані, які потім потоково передаються пристроєм. Ці дані називають телеметрією. Телеметричне повідомлення має бути швидким, структурованим та семантичним. Також, такі пристрої здатні обробляти команди від акторів і відповідати на них підтверджуючи чи відхиляючи таку команду, якщо вона не правильна, чи щось пішло не так. Команди можуть відповідати за увімкнення чи вимкнення пристрою, калібрування сенсорів чи просто отримування поточного статусу.

Основними вимогами до сервісу буде підтримка найпоширеніших протоколів з’єднання, та легка можливість для її розширення, яка не буде потребувати великих змін у архітектурі. Сервіс має бути стабільним та бути готовим до стрімкого збільшення кількості розумних пристроїв та кінцевих користувачів. Для цього система має мати гарну здібність до масштабування, як вертикального, так і горизонтального.

Крім вищезазначеного, важливо також подбати й про безпеку. Отримання телеметричних даних та інша взаємодія з пристроями має відбуватися в ізольованій мережі. Кінцевий користувач має доступ лише до прикладного рівня API який є захищеним та регулює доступ до внутрішніх даних на основі чіткої рольової моделі «користувач/адміністратор». Комунікація на рівні між сервером і пристроями має йти по захищеному каналу.

Ще однією вимогою може бути зберігання телеметричних даних та можливість їх запиту для подальшого аналізу. А також, функціонал створення автоматичних сповіщень за встановленими умовами.

1.2 Приклади існуючих рішень

На ринку існує чимала кількість рішень для IoT систем. Кожне має свої переваги та недоліки. Розглянемо наявні платформи, аби визначити на які елементи варто звернути увагу, а які написати самостійно. Було розглянуто 3 загальнодоступних рішення, з яких 2 пропрієтарних від корпорацій Amazon та Microsoft, й одне з відкритим вихідним кодом.

1.2.1 Ядро розумних пристроїв Amazon

AWS IoT Core позиціонується як керована хмарна платформа, здатна одночасно обслуговувати мільярди пристроїв та маршрутизувати трильйони повідомлень без потреби в адмініструванні серверної інфраструктури. Сервіс надає набір високорівневих API й SDK, що спрощують інтеграцію вбудованих систем, веб та мобільних застосунків з екосистемою AWS, залишаючи за Amazon відповідальність за автоматичне масштабування, оновлення та моніторинг ядра платформи [2].

Ключовими ланками архітектури є шлюз пристроїв та брокер повідомлень. Шлюз виконує роль точки входу й підтримує довготривалі двобічні з’єднання через протокол телеметрії MQTT, його варіант поверх веб-сокетів і захищений протокол передавання гіпертексту. Компонент автоматично масштабується зі зростанням навантаження, забезпечуючи низьку затримку обміну даними, спрощена схема наведена на Рис. 1.1.

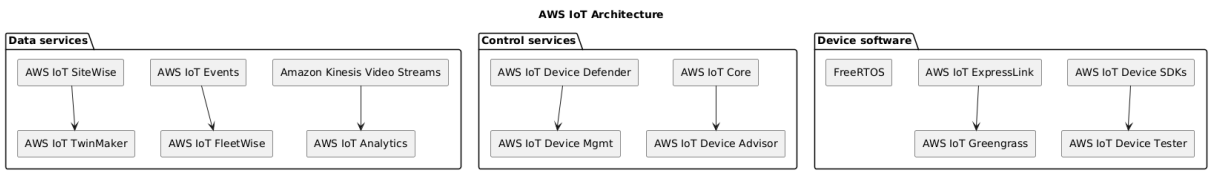


Рисунок 1.1 – Заявлена архітектура AWS IoT

Брокер повідомлень, своєю чергою, реалізує шаблон «опублікувати / підписатися» у межах специфікації MQTT 5.0, дозволяючи детально розмежовувати доступ до тем за політиками та гарантуючи високу пропускну здатність під час широкомовних розсилок.

У цьому контексті варто згадати механізм правил: він збагачує повідомлення, застосовує фільтри з БД-подібним синтаксисом і надалі переспрямовує результати до інших служб хмарної платформи або до сервісів кінцевого користувача. Безпека вибудувана за принципом наскрізного шифрування і взаємної автентифікації. Хмарний сервіс інтернету речей приймає сертифікати X.509, підписані власним чи стороннім центром сертифікації, підтримує підпис запиту четвертої версії для веб-клієнтів і дозволяє впроваджувати користувацькі механізми авторизації для токенових схем [3]. Поєднання рукостискання TLS і політик інтернету речей надає змогу детально регламентувати права аж до окремої теми чи конкретної дії (опублікувати, підписатися, під'єднатися). Повідомлення, що надходять зі шлюзу, можуть одразу оброблятися механізмом правил. Правила записуються у формі псевдо-SQL-виразів і дозволяють фільтрувати, трансформувати та маршрутизувати дані до понад десяти служб хмарної екосистеми (безсерверні функції, нереляційна база, об'єктне сховище, потоковий сервіс, сервіс сповіщень тощо), а також ініціювати бізнес-процеси чи запускати висновки машинного навчання — без розгортання власних вузлів обробки подій. Опція «базове завантаження» допомагає зменшити витрати, оминаючи брокер під час прямого запису даних. Модель ціноутворення дотримується принципу оплати за фактичне споживання: окремо тарифікуються хвилини активного з'єднання, повідомлення (інкременти по 5 КБ), операції реєстру / тіньового стану пристрою та виклики механізму правил, причому обов'язкових мінімальних платежів немає.

У підсумку, розглянутий хмарний сервіс інтернету речей демонструє високий рівень зрілості. Модульна архітектура, багатопротокольна підтримка,

гнучкі механізми безпеки та інтеграцій, а також прозора система білінгу роблять рішення привабливим для стартапів і великих підприємств, що прагнуть швидко вивести продукт на ринок без значних капітальних витрат на інфраструктуру пропрієтарних API і політик AWS може вимагати ретельного планування при міграції або побудові мультихмарних рішень, особливо у випадку суворих регуляторних обмежень щодо зберігання даних.

1.2.2 Хаб розумних пристроїв Azure

Є підстави вважати, що Azure IoT Hub посилив конкуренцію між хмарними екосистемами Інтернету речей, бо він пропонує єдиний керований шлюз для приймання телеметрії та виконання команд [4].

Хаб інтернету речей виконує подвійне призначення: з одного боку, слугує центральним реєстром ідентичностей пристроїв, з іншого — забезпечує наскрізний канал обміну даними, гарантуючи доставку телеметрії та команд незалежно від обраного протоколу сполучення. Спрощений приклад розгортання наведено на Рис. 1.2.

Архітектурно система поділяється на дві площини. Перша — «пристроєорієнтована» — об’єднує вхідні точкові з’єднання за допомогою протоколу телеметрії, протоколу черг повідомлень і захищеного протоколу передавання гіпертексту (усі в класичному та веб-сокетному варіантах). Друга — сервісна — надає потоковий журнал подій, сумісний зі шлюзом подій хмарної платформи, призначений для споживачів даних. Журнал подій розбивається на частини (до тридцяти двох), що мінімізує навантаження за умов паралельних операцій читання-запису й дає змогу горизонтально нарощувати пропускну спроможність без простоїв. Базова конфігурація S1 гарантує чотириста тисяч повідомлень на добу для одного екземпляра, тоді як S3 масштабується до трьохсот мільйонів [5].

Підтримка згаданих протоколів охоплює як стандартні, так і варіанти поверх веб-сокетів: це суттєво полегшує роботу через корпоративні проксі. Протокол телеметрії й протокол черг повідомлень забезпечують мінімальну

затримку, тоді як захищений гіпертекстовий протокол призначено для малоресурсних або епізодично під’єднаних пристроїв. Рекомендації щодо вибору беруть до уваги розмір клієнтських бібліотек, необхідність мультиплексування та вимоги до портів.

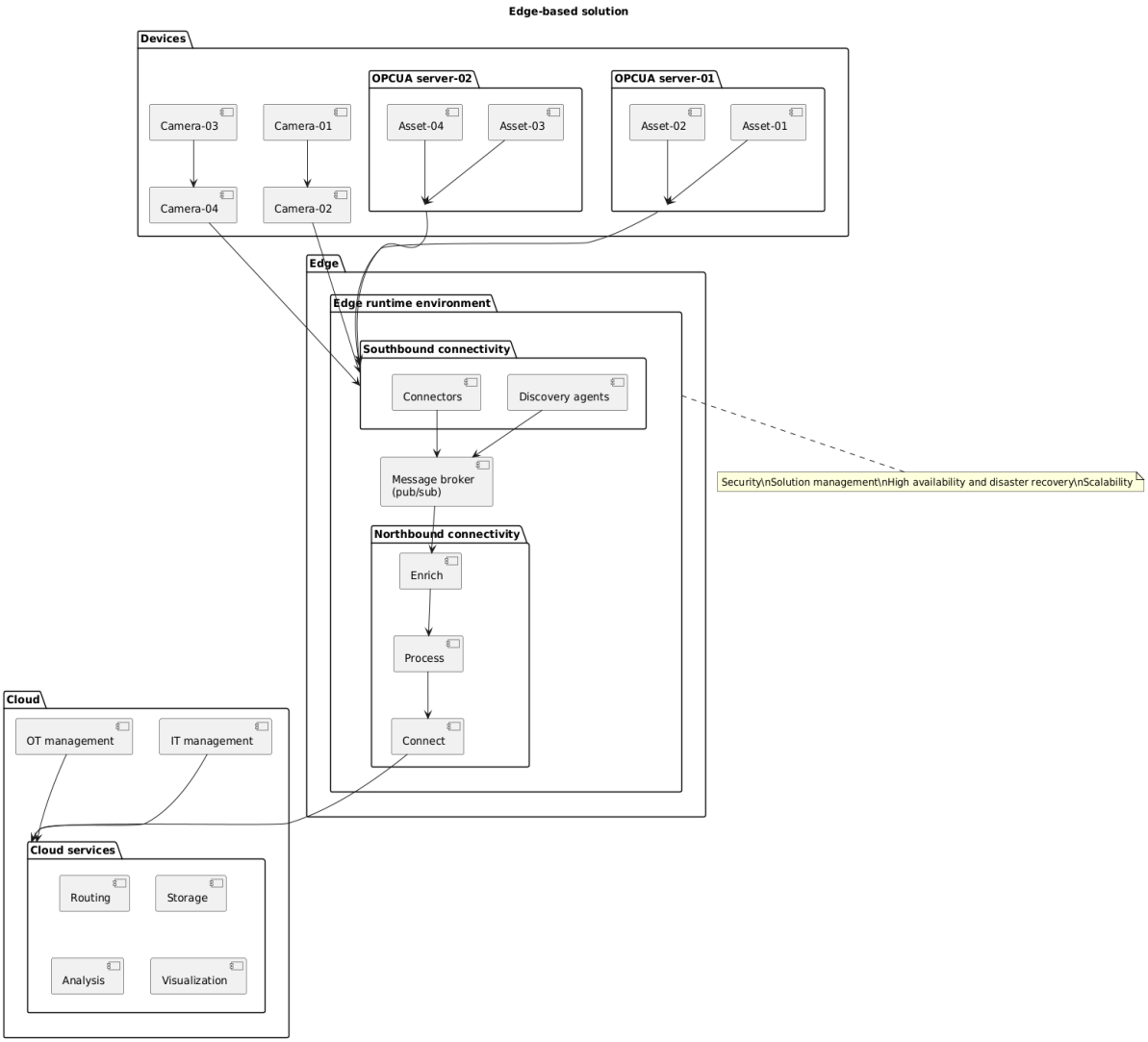


Рисунок 1.2 – Приклад архітектури з використанням Azure IoT Hub

Модель безпеки вибудувано на поєднанні симетричних токенів спільного доступу та сертифікатів формату X.509. Для промислових сценаріїв корпорація рекомендує схему з підписом центром сертифікації, що спрощує ротацію ключів і централізований аудит. Політики доступу реалізуються через рольове керування ресурсами й деталізовані політики самого хаба, які обмежують дії (опублікувати, отримати, викликати метод) на рівні окремої сутності.

Компонент маршрутизації повідомлень дає змогу за допомогою SQL-подібних виразів відфільтровувати телеметрію, події життєвого циклу або зміни цифрових двійників і спрямовувати їх до стандартного потоку чи визначених кінцевих точок — зокрема об’єктного сховища, сервісу черг, розподіленої бази даних або шлюзу подій. Підтримується збагачення заголовків, резервні маршрути та інтеграція з мережею подій для реактивної автоматизації.

Отже, можна дійти висновку, що хмарний хаб інтернету речей формує цілісне середовище, де захист, маршрути, цифрові двійники, оновлення й аналітика взаємодіють узгоджено. Порівняно з пропозицією іншого великого постачальника, платформа робить особливий акцент на гнучкості механізмів автентифікації та насиченій інтеграції із сервісами «великих даних» власної екосистеми.

1.2.3 ThingsBoard

Розглянута відкрита й розширювана платформа для «інтернету речей» побудована на мікросервісних засадах і орієнтована на багатоновий режим обслуговування пристроїв, активів і користувачів. Її ядро написано для виконання на віртуальній машині та мови яка підтримує як монолітний, так і кластерний способи розгортання: у кластері головні компоненти взаємодіють через службу координації та спільні черги подій, що забезпечує лінійне горизонтальне масштабування без зупинок.

З погляду під’єднання обладнання платформа охоплює широкий набір протоколів: MQTT (зокрема профіль «Спаркплаг»), гіпертекстовий, протокол обмежених застосунків, протокол керування ознаками та мережеве керування пристроями. Для мереж із великою кількістю вузлів передбачено шлюзовий механізм, який агрегує телеметрію від підлеглих сенсорів і передає її одним з’єднанням, зменшуючи витрати на автентифікацію. Повна підтримка протоколу керування ознаками, зокрема сервера початкового налаштування й захищеного транспортного рівня, робить рішення привабливим для

енергоощадних стільникових систем вузької смуги. Модуль обробки правил ґрунтується на ланцюгах правил — графі вузлів, де кожен елемент виконує атомарну дію фільтрації, збагачення чи виклику зовнішньої служби. Інженер формує такий ланцюг у графічному редакторі, а система одразу застосовує зміни без перезапуску контейнерів. Безпека організована багаторівнево. Кожному пристрою можуть бути призначені токени доступу, симетричні ключі, X.509-сертифікати або облікові записи з JWT для взаємної TLS аутентифікації. Додатково реалізовано контроль прав на рівні клієнтських застосунків і окремих операцій [6].

Рольова модель ґрунтується на матриці дозволів, а механізм суб-орендарів делегує частину прав партнерським організаціям, покращуючи прозорість відповідальності. Архітектуру платформи можна побачити на Рис. 1.3.

Для сценаріїв офлайн-роботи розроблено компонент ThingsBoard Edge. Він розгортається на периферії (наприклад, у цеху чи на віддаленій підстанції) і містить локальну копію рушія правил та візуалізації. синхронізація із хмарним кластером відбувається асинхронно, що знижує затримку і зменшує трафік між сегментами мережі. Можна зробити висновок, що ThingsBoard поєднує транспортні шлюзи, цифрові двійники, аналітику та спостережливість у єдиному відкритому стеку [7]. Платформа забезпечує горизонтальне масштабування, гнучку модель безпеки та підтримку різноманітних протоколів.

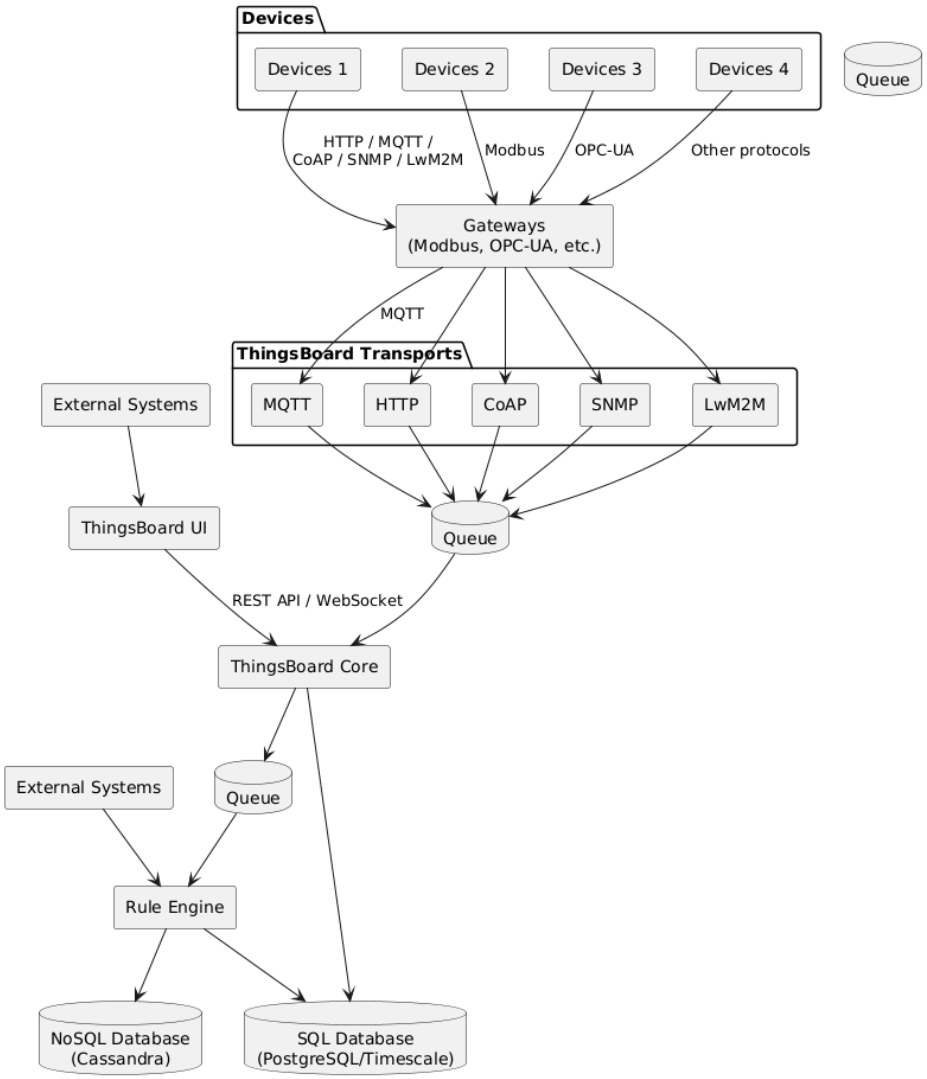


Рисунок 1.3 – Заявлена архітектура ThingsBoard

1.3 Порівняльний аналіз платформ

Порівняння трьох платформ — AWS IoT Core, Azure IoT Hub і ThingsBoard — дає змогу глибше усвідомити, яка інфраструктура найліпше підтримує сервіс віддаленого моніторингу та керування пристроями. Є підстави вважати, що кожна з них закриває базові потреби підключення, зберігання та оброблення телеметрії, однак робить це різними архітектурними шляхами, тому вибір безпосередньо впливатиме на витрати, швидкість прототипування й рівень контролю над даними.

Можна також зазначити, що AWS IoT Core та Azure IoT Hub надають повністю керовані брокери, отже потенційний клієнт фактично делегує масштабування та збереження даних хмарному провайдерові. ThingsBoard натомість пропонує відкритий код, що розгортається у приватному сервері або на публічних хмарних ресурсах. З цього виходить, що, якщо, підприємство має високі вимоги до конфіденційності даних або бажає адаптувати ядро під нестандартний протокол, то вони більш охоче виберуть self-hosted сценарій.

Щоб зібрати ключові характеристики у стислій формі, доцільно представити їх у табличному вигляді:

Таблиця 1.1 – Порівняння ключових характеристик кожного з сервісів

Критерій	**AWS IoT Core**	**Azure IoT Hub**	**ThingsBoard**
Модель розгортання	Повністю керований сервіс	Повністю керований сервіс	Відкритий код
Підтримувані протоколи	MQTT, MQTT WSS, HTTPS	MQTT, MQTT WSS, HTTPS	MQTT, HTTP
Масштабування	Автоматичне, брокер партиціює топіки	Додавання елементів	Додавання нод t
Безпека	TLS, X.509,	TLS або X.509	TLS, токени або X.509,
Вартість	Оплата за повідомлення, трафік	Оплата за повідомлення,	Витрати на хостинг

З цього виходить, що швидкий прототип доцільно будувати на AWS або Azure, бо хмарний брокер дозволяє зосередитися на бізнес-логіці та інтерфейсі користувача. Коли ж система переходить у фазу промислової експлуатації й стикається з тонкими вимогами щодо локалізації даних, розширюваності правил рушія і власного брендування, ThingsBoard дає ширший простір для кастомізації.

ВИСНОВОК ДО РОЗДІЛУ 1

Проведене дослідження дозволило зібрати цілісну картину ландшафту IoT-платформ, окресливши ключові поняття, без яких неможливо розмовляти про сучасні системи віддаленого моніторингу. Було з’ясовано, що вимоги до таких рішень формуються на перетині безпеки, масштабованості, гнучкості протоколів та розвиненої екосистеми інструментів. Саме ці критерії слугували «лакмусовим папірцем» під час порівняння платформ.

Аналіз AWS IoT Core продемонстрував, що хмарні сервіси Amazon фактично задають індустріальний стандарт. Автоматизоване керування сертифікатами, підтримка багатьох протоколів, а також глибока інтеграція з іншими сервісами AWS забезпечують швидкий старт і майже необмежену горизонтальну масштабованість. Azure IoT Hub, своєю чергою, робить акцент на корпоративних сценаріях, пропонуючи «Digital Twins» та потужний набір аналітичних інструментів. водночас уявна легкість початку роботи компенсується доволі складною моделлю білінгу. ThingsBoard вирізняється відкритим кодом і можливістю самостійного хостингу, що відкриває шлях до використання платформи в середовищах із суворими вимогами до локального контролю даних, проте накладає відповідальність за підтримку інфраструктури на власника рішення.

Порівняльний аналіз підтвердив відсутність так званої «срібної кулі». AWS показав найкращі результати у швидкості розгортання та ширині сервісної екосистеми, проте виявився чутливим до вартості при великих обсягах телеметрії. Azure переконливо виглядає у проектах, де потрібна тісна інтеграція з корпоративним стеком Microsoft, але поступається конкурентам у простоті конфігурації. ThingsBoard демонструє високу адаптивність завдяки відкритому коду та модульності, проте потребує додаткових зусиль для досягнення рівня відмовостійкості, який «із коробки» пропонують хмарні провайдери.

Таким чином, зроблено висновок, що вибір платформи повинен ґрунтуватися не тільки на технічних характеристиках, а й на стратегічних обмеженнях проєкту, а саме бюджеті, вимогах до приватності даних та очікуваному навантаженні. Отримані результати створюють основу для обґрунтованого рішення щодо архітектури майбутньої системи та логічно переходять до наступного розділу, де увагу буде зосереджено на детальному аналізі технологій, необхідних для реалізації поставлених цілей.

РОЗДІЛ 2 ОГЛЯД ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ СИСТЕМИ

2.1 Мови розробки

Перш ніж занурюватися у конкретику архітектурних рішень, варто окреслити мовний ландшафт, на який спиратиметься подальша розробка. У цьому розділі розглянуто чотири провідні мови — Python, Java, JavaScript та TypeScript — крізь призму їхніх парадигм, еволюційних тенденцій та екосистем. Такий підхід дозволяє оцінити не лише «придатність» до певного проєкту, а й глибинні особливості, що формують мислення розробника та визначають довгострокові витрати на підтримку коду.

2.1.1 Python

З моменту офіційного релізу 1991 року Python утримує репутацію «філософії зрозумілого коду», у якій читабельність ставиться вище за синтаксичну економію [8]. Формально мова належить до мультипарадигмальних. Вона підтримує імперативний, об’єктно-орієнтований, функціональний та процедурний стилі. Динамічна система типів у сполученні з автоматичним збиранням сміття спрощує прототипування, проте водночас ускладнює статичний аналіз продуктивності.

Стандартна бібліотека мови програмування Python традиційно описується як «постачається з усім необхідним», оскільки охоплює широкий спектр модулів для роботи з мережею, базами даних, текстовими форматами, файловими системами, операційною системою та навіть базовими криптографічними операціями. Такий підхід істотно спрощує початковий етап розробки та зменшує потребу в сторонніх залежностях, особливо для типових задач прикладного рівня. Разом із тим, інтерпретатор CPython — основна й водночас референсна реалізація Python — обмежений однопотоковим виконанням через механізм глобального інтерпретаторного замка. Цей замок (англ. Global Interpreter Lock) дозволяє одночасне виконання лише одного

потоку байткоду, навіть на багатоядерних системах, що стримує ефективне використання апаратних ресурсів для задач із високим ступенем паралелізму [8]. Для часткового обходу цієї архітектурної особливості Python підтримує створення підпроцесів, асинхронне програмування на основі бібліотеки asyncio, а також має альтернативні реалізації, зокрема PyPy (з JIT-компіляцією), Jython (для інтеграції з віртуальною машиною Java) та IronPython (орієнтовану на .NET-платформу). Проте жодна з них не усуває обмеження повністю у загальному випадку — кожна має свої компроміси щодо сумісності, продуктивності чи стабільності. Таким чином, Python демонструє прагнення до досягнення балансу між зручністю розробки, швидкістю прототипування та обчислювальною ефективністю. Саме завдяки цьому балансу мова стала надзвичайно популярною в галузях, де важлива не стільки пікова продуктивність, скільки гнучкість і наявність готових інструментів — таких як наукові обчислення, обробка даних і машинне навчання.

2.1.2 Java

Java позиціонується Oracle як «загального призначення, суворо типізована, класична ООП-мова», що компілюється у байт-код під специфікацію віртуальної машини Java (JVM) [9]. Статична типізація та збирач сміття водночас формують стабільний рівень продуктивності й безпеки пам’яті, що робить Java привабливою на ринку рішень. Паралелізм реалізується через багатопоточність JVM які відчутно спрощують конкурентні сценарії. Проте суворіший синтаксис і громіздка конфігурація часто потребують додаткових інструментів (Maven, Gradle), що впливають на поріг входження новачків.

2.1.3 JavaScript

JavaScript народився як браузерний скрипт, однак еволюціонував у багатопарадигмовий інструмент, який демонструє прототипне успадкування, першу класність функцій та подієвий цикл із неблокувальним ввід-виводом [10]. Є підстави вважати, що стандартизація через «ECMAScript» щороку

підживлює мову новими конструкціями, забезпечуючи поступову гармонізацію синтаксису з сучасними вимогами розробників.

Універсальність JS зумовлена можливістю запуску як у браузері, так і на сервері через платформу Node.js. Проте відсутність статичної типізації часто викликає «крадіжку часу» на етапі дебагу, зокрема у великих командах. Універсальний пакетний менеджер та система модулів ECMAScript компенсують цей недолік значною екосистемою, яка охоплює практично всі сфери — від інтернету речей до штучного інтелекту.

2.1.4 TypeScript

TypeScript позиціонується Microsoft як «строго типізована надбудова над JavaScript», що компілюється у чистий JS і лишається сумісною з будь-яким рушієм JavaScript [11]. Структурна система типів, інтерфейси та «загальні типи» запроваджують елементи, характерні для класичних мов на кшталт Java або C#, і водночас зберігають гнучкість динамічної природи JS. Наявність статичних анотацій не лише покращує автодоповнення й рефакторинґ у сучасних IDE, а й мінімізує помилки. При цьому компілятор «tsc» пропонує суворий режим, який, хоч і подовжує час збірки, забезпечує раннє виявлення потенційних дефектів. Поява декораторів, асинхронних ітераторів та підтримка зовнішніх декларативних файлів свідчить про постійну еволюцію мови у напрямку кращої експресивності без втрати сумісності.

2.1.5 Висновок

Стисло підсумовуючи, можна ствердити, що Python вирізняється експресивністю та багатою стандартною бібліотекою, Java — стабільністю та промисловою зрілістю, JavaScript — універсальністю виконання і реактивною екосистемою, а TypeScript поєднує переваги JS із гарантіями статичної типізації. Усвідомлення цих рис дозволить у подальших розділах обґрунтовано поєднувати мови у багаторівневому рішенні, враховуючи як продуктивність, так і експлуатаційні витрати.

Таблиця 2.1 – Порівняння основних характеристик

Критерій	Python	Java	JavaScript	TypeScript
Типізація	динамічна, duck typing	статична, JVM	динамічна, без вбудованої статичної типізації	статична, надбудова
Парадигми	імперативна, ООП, функц., процедурна	ООП	прототипне успадкування, функціональне, подієва модель, ООП	модульна, ООП
Продуктивність	GIL, обмежений один потік, asyncio	висока, багатопотоковість JVM, Loom-threads	неблокувальний ввід/вивід через Event Loop	додаткові витрати на компіляцію
Стандартна бібліотека	мережа, БД, текст, крипто	базова, enterprise-пакети через екосистему	мінімальна, масив модулів	типові декларації

2.2 Середовище виконання

Під середовищем виконання зазвичай розуміють програмний шар, який відповідає за інтерпретацію або компіляцію вихідного коду в машинні інструкції, керує життєвим циклом виконуваного процесу, взаємодіє з операційною системою, забезпечує виділення ресурсів, управління пам’яттю, планування потоків та обробку системних викликів.

У контексті технологій, орієнтованих на мову JavaScript, середовище виконання пройшло значну еволюцію — від вбудованих рушіїв, призначених виключно для роботи в межах браузера, до самостійних повнофункціональних платформ, здатних ефективно виконувати роль серверної логіки, обробки запитів, асинхронної взаємодії та масштабованої обробки даних. Це наблизило їх до рівня складності й можливостей таких потужних віртуальних машин, як середовище виконання для мови Java або інфраструктура .NET.

На сучасному етапі розвитку JavaScript екосистеми спостерігається конкуренція між декількома ключовими реалізаціями серверного середовища виконання, які втілюють різні архітектурні підходи, пріоритети продуктивності й моделі безпеки. У цьому розділі розглянуто три найпомітніші з них — Node.js,

Bun та Deno. Кожне з цих середовищ демонструє власне бачення розвитку серверних технологій: від класичної сумісності з широкою екосистемою до фокусування на швидкодії, безпеці та вбудованому інструментарії.

2.2.1 Node.js

Node.js — найдавніше з розглянутих середовищ виконання, що базується на рушії V8 і бібліотеці libuv. У його основі — однопоточна модель з подійною петлею, яка делегує блокувальні операції пулу робочих потоків. Така архітектура дає змогу досягти неблокувального введення-виведення й ефективно обслуговувати тисячі паралельних клієнтів без істотного навантаження на пам’ять або процесор. Принцип роботи можна побачити Рис. 2.1.

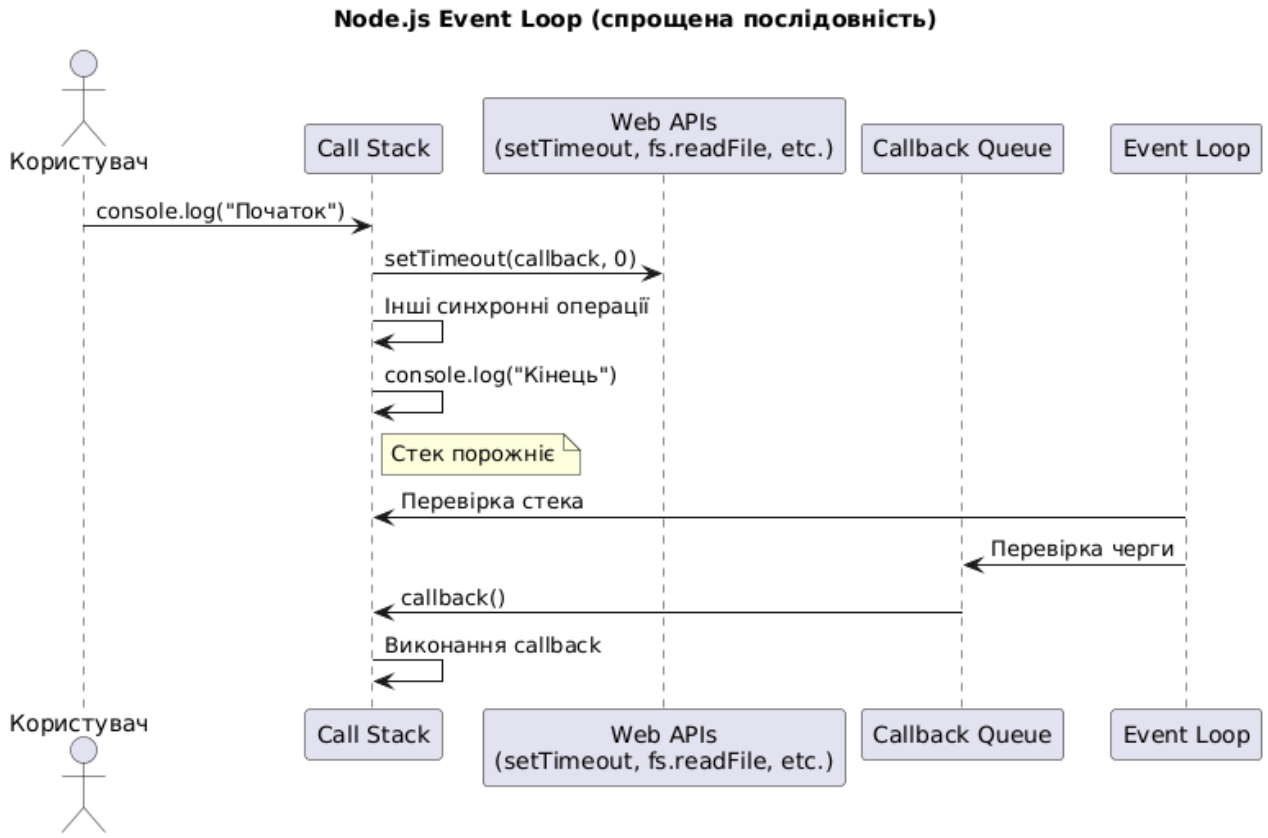


Рисунок 2.1 – Спрощена схема роботи EventLoop у Node.js

Ядро Node.js складається з трьох ключових компонентів: рушія V8, який транслює сценарії в машинний код і управляє збиранням сміття; бібліотеки

libuv, що забезпечує кросплатформену абстракцію асинхронного введення-виведення; та шару зв’язку мовою Сі/Сі++, що формує прикладний інтерфейс Node API [12]. Бібліотека libuv реалізує механізм подійної петлі, пул потоків, черги таймерів та інші низькорівневі абстракції. Основний сценарій користувача виконується в одному потоці, але виклики до файлової системи, мережі, криптографічних функцій автоматично передаються в тред-пул або ядро операційної системи. Це дозволяє уникати блокування головного циклу. Починаючи з дванадцятої версії, Node.js підтримує робочі потоки, що дає змогу виносити ресурсоємні обчислення в окремі потоки без створення зовнішніх процесів. У поєднанні з кластеризацією (механізмом створення окремих процесів) і опцією автоматичного перезапуску, ці функції роблять Node.js придатним для побудови високопродуктивних серверних застосунків, здатних до масштабування. Окремо варто відзначити механізм запуску: розробник не викликає явно подійну петлю — після початкової ініціалізації середовище самостійно переходить до її виконання й завершує процес, коли всі колбеки оброблено. Такий підхід спрощує структуру програми та полегшує відлагодження.

Екосистема Node.js організована навколо менеджера пакунків, що став галузевим стандартом. У сховищі нараховується понад два мільйони модулів, що охоплюють майже всі сфери розробки: від роботи з базами даних і побудови API до генерації документів і управління потоками. Водночас така масштабність потребує суворого контролю: ланцюжки залежностей можуть містити застарілі чи вразливі бібліотеки, конфлікти версій або неочікувані побічні ефекти. Тому при розробці довготривалих рішень критично важливо впроваджувати політику керування залежностями, регулярні перевірки оновлень і аудит безпеки.

2.2.2 Bun

Розглянута платформа позиціонується як «усе в одному»: швидке середовище виконання, засіб збирання, тестувальник і керувач пакунками, створені на компільованій системній мові та побудовані поверх рушія виконання скриптів [13]. Відмова від динамічних бібліотек і збірка коду в єдиний самодостатній файл істотно скорочують час холодного запуску й споживання пам’яті. Окрім виконання сценаріїв, платформа підтримує вбудоване тестування, пакування та формування автономних двійкових файлів, що спрощує конвеєр безперервної інтеграції й доставки. Задекларована сумісність із прикладним інтерфейсом поширеного серверного середовища покликана знизити бар’єр переходу: більшості наявних пакунків достатньо перекомпіляції, аби працювати без змін. Водночас використання альтернативного рушія виконання накладає певні обмеження. Не всі низькорівневі механізми, розраховані на попереднє середовище, реалізовано повною мірою, тож за глибокої інтеграції з власним машинним кодом може знадобитися додаткова адаптація. Попри це, стратегія «рухатися швидко й одночасно стискати все» робить описану платформу привабливою для безсерверних сценаріїв, де критичними залишаються час холодного старту та мінімальний обсяг контейнера [13].

2.2.3 Deno

Deno — проєкт, започаткований одним із авторів вузлового середовища, який прагне усунути недоліки попередника шляхом моделі «безпека за замовчуванням». Будь-яка програма спочатку позбавлена доступу до файлової системи, мережі чи змінних середовища, допоки розробник явно не дозволить потрібні права. Рушій об’єднує високошвидкісний інтерпретатор, мову Раст і асинхронне середовище виконання, що ґрунтується на технології Tokio, з підтримкою сучасних веб-мов — скриптової, надбудови з перевіркою типів і

байт-кодових модулів — без потреби в зовнішніх компіляторах або перетворювачах [14].

Deno постачається з повноцінною стандартною бібліотекою та вбудованими засобами для розробки. Засоби форматування коду, тестування, статичного аналізу й збирання доступні без потреби встановлення сторонніх інструментів, що значно спрощує початкове налаштування. Імпорт модулів реалізовано через прямі посилання на веб-ресурси, а кешування обробляється автоматично без прив’язки до централізованого сховища на кшталт традиційного реєстру. Це знижує ризики, пов’язані з атаками на ланцюг постачання, проте водночас потребує переосмислення типових шаблонів автоматизованого розгортання.

Додаткову гнучкість забезпечує підтримка хмарного середовища для виконання програм, яке розміщує код ближче до кінцевого користувача з мінімальною затримкою. Поява бібліотек, сумісних із вузловою екосистемою, засвідчує поступову інтеграцію двох напрямів розвитку скриптових платформ

2.2.4 Висновок

Отже, Node.js залишається еталоном зрілої екосистеми з колосальним набором бібліотек і високою сумісністю з існуючими засобами автоматизації. Bun обирає шлях радикальної оптимізації швидкості та спрощення ланцюга збірки. Deno робить акцент на безпеці, сучасних веб-стандартах і вбудованому інструментарії. Усвідомлення цих відмінностей дозволить у наступних розділах коректно корелювати вибір ран-тайму з архітектурними вимогами, показниками продуктивності та політикою безпеки майбутньої системи.

Таблиця 2.2 - Порівняння основних характеристик

Критерій	Node.js	Bun	Deno
Ядро	однопоточний	написано на Zig, «усе в одному»	V8 + Tokio (Rust), secure-by-default
Швидкий старт	середній, через V8	дуже швидкий, один бінарник, мінімальні залежності	середній, перший cold start з кешем модулів
Інструментарій «із коробки»	обмежений, потребує npm-пакетів	вбудовані bundler, тест-ранер, пакувальник	форматер, лінтер, тест-ранер, URL-імпорт без npm
Безпека	відкритий доступ до FS/мережі	як у Node.js	за замовчуванням без доступу до FS/мережі
Сумісність API	велика (npm-модулі)	сумісність із Node.js API часткова	обмежена

2.3 Бібліотеки прикладного інтерфейсу

Сучасна розробка серверних застосунків на JavaScript грунтується накаркас ПЗах, що інкапсулюють низькорівневі HTTP-операції, пропонують життєвий цикл запиту-відповіді та формують конвенції для масштабування проєкту. Попри спільну платформу Node.js, ExpressJS, FastifyJS і NestJS вирішують одну й ту саму задачу різними шляхами, що відображається на продуктивності, структурі коду й експлуатаційних витратах.

2.3.1 ExpressJS

ExpressJS, представлений ще у 2010 році, реалізує мінімалістичну концепцію «фреймворку без нав’язаної думки» та надає розробнику лише тонкий абстрактний шар над прикладним інтерфейсом роботи з HTTP у вузловому середовищі [15]. Основна функціональність зосереджена в механізмі проміжних обробників (так званих посередників) і маршрутизаторі, тоді як більшість необхідних можливостей — зокрема обробка політики

кросдоменного доступу, валідація даних, обробка помилок або авторизація — підключається сторонніми модулями.

Завдяки цьому підходу екосистема доступних модулів у загальному сховищі стала чи не найбільш розвиненою: легко інтегрувати взаємодію з базами даних, платіжними системами чи інструментами моніторингу. Однак така гнучкість має і зворотний бік. Відсутність єдиної рекомендованої структури застосунку призводить до фрагментованості стилю програмування: у командах, особливо великих, може виникати конфлікт підходів до обробки помилок, організації контролерів, тестування та повторного використання логіки. Це знижує передбачуваність і ускладнює супровід проєктів у довгостроковій перспективі. Окрім того, під час масштабування проявляється ще одна проблема — кожен додатковий посередник у ланцюжку маршруту створює накладні витрати. Це особливо помітно в системах з високими вимогами до продуктивності, де накопичення посередників призводить до зростання затримок і середнього часу відгуку. У підсумку, Express-застосунки з великою кількістю логіки, розміщеної в ланцюжках обробників, поступаються сучасним рішенням, що оптимізовані для швидкодії та асинхронної обробки на рівні ядра.

2.3.2 FastifyJS

FastifyJS позиціонується як «швидкий каркас ПЗ» із мінімальними накладними витратами» та демонструє пропускну здатність до 30 тисяч запитів на секунду на одному процесі, що підтверджується як офіційними, так і незалежними вимірюваннями продуктивності [16]. У центрі його концепції — підхід, заснований на схемах даних. Розробник описує очікувану структуру запитів і відповідей за допомогою формалізованих описів, після чого рушій ще до запуску застосунку компілює маршрути й генерує оптимізований код серіалізації. Такий підхід знижує навантаження під час виконання, зменшує час обробки запиту та забезпечує передбачувані показники продуктивності, які

можна оцінити на етапі розробки. Схему відповідної архітектури наведено на Рис. 2.2.

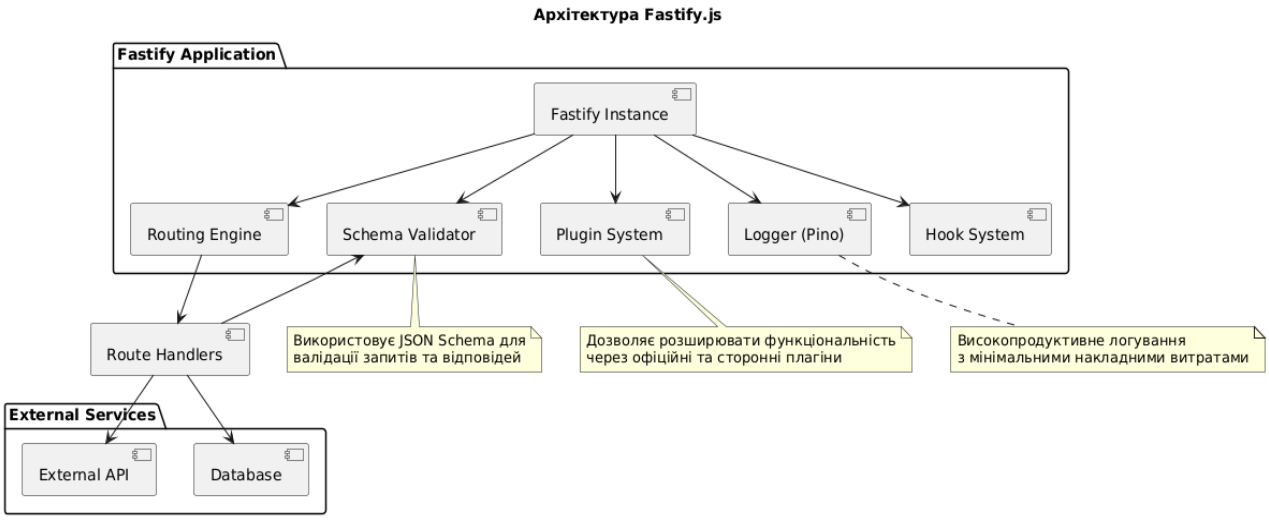


Рисунок 2.2 – Архітектура Fastify.js

Модель розширення через плагіни в Fastify частково наслідує підхід Express, але вводить ізольовані контексти конфігурації, що унеможливлює конфлікти залежностей. Кожен плагін має власний простір імен і обмеження видимості, що сприяє модульності та повторному використанню. Крім того, у Fastify вже вбудовано підтримку високопродуктивного логування, а також протоколу HTTP/2, що зменшує залежність від сторонніх бібліотек для базової інфраструктури. Втім, порівняно молодий вік екосистеми означає, що для малопопулярних або нішевих задач може бракувати перевірених рішень або зрілих плагінів. Це слід враховувати під час планування тривалої підтримки або інтеграції з нестандартними службами. Тим не менш, структурна ясність, висока швидкодія та прозора модель розширення роблять Fastify привабливим вибором для розробки сучасних веб-сервісів, орієнтованих на навантаження та масштабованість.

2.3.3 NestJS

NestJS вирізняється прагненням до архітектурної впорядкованості, поєднуючи досвід Angular та Spring у середовищі Node.js [17]. Приклад типової архітектури можна побачити на Рис. 2.3. Ключові концепти — модулі, контролери, сервіси — реалізують сувору інверсію керування через вбудований контейнер залежностей. Це уможливлює масштабування монолітів до мікросервісів і сприяє тестованості завдяки явним контрактам між шарами.

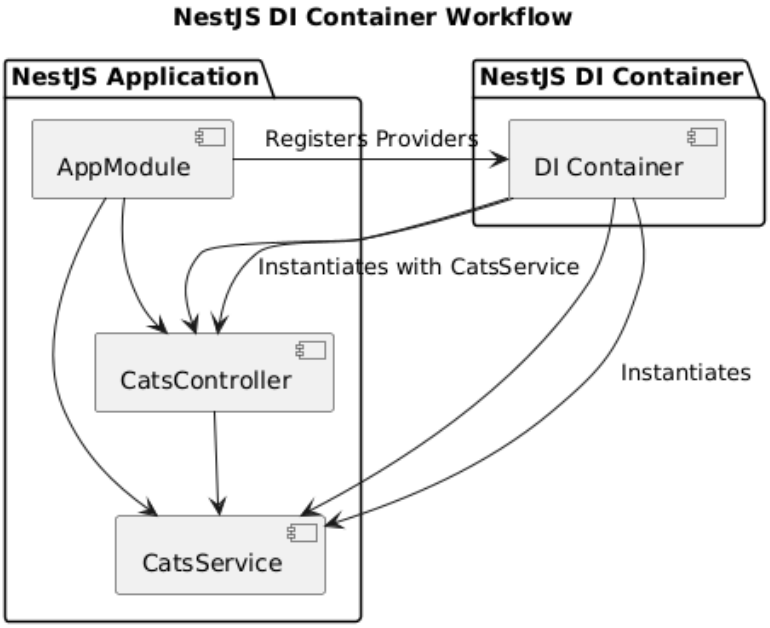


Рисунок 2.3 – Типова архітектура NestJS додатка

Nest компілюється TypeScript-транслятором і надає типову безпечність, недосяжну для чистого Express.каркас ПЗ підтримує дві стратегії запуску HTTP-шару — на базі Express або Fastify, що дозволяє віднайти компроміс між зрілістю екосистеми та продуктивністю. Розширення у вигляді транспортів (gRPC, RabbitMQ, WebSockets) і модулів CQRS позиціонують Nest як enterprise-рішення, однак за це доводиться платити вищим порогом входження та тривалішою фазою складання.

2.3.4 Висновок

Таким чином, ExpressJS залишається класичним вибором для малих сервісів з мінімальними вимогами до формальної структури. FastifyJS демонструє найвищу ефективність і систематизує плагіни навколо схем, що привабливо для чутливих API. NestJS пропонує повноцінний каркас із дієвою системою та статичною типізацією, полегшуючи супровід великих проєктів.

Таблиця 2.3 - Порівняння основних характеристик

Критерій	ExpressJS	FastifyJS	NestJS
Підхід	unopinionated, middleware-based	schema-driven, плагіни із namespacing	модульний, DI-контейнер, архітектура за SOLID
Продуктивність	середня латентність через ланцюги middleware	до 30 000 запитів/с, JSON-schema препросінг маршрутів	залежить від адаптера (Express/Fastify), вище через DI
Структура проєкту	вільна, ризик хаосу в великих командах	структурована навколо схем	сувора: модулі, контролери, сервіси

2.4 Прикладні протоколи

Після вибору мови й середовища виконання надходить черга визначити «мову» взаємодії між клієнтом і сервером. Саме протокол API диктує горизонт масштабування, спосіб еволюції контракту та вимоги до мережевої інфраструктури. Нижче окреслено чотири найпоширеніші підходи — REST, WebSocket, gRPC + буфери протоколів і GraphQL — із наголосом на їхніх архітектурних принципах, сильних і слабких сторонах.

2.4.1 Передача репрезентативного стану

Концепцію Representational State Transfer сформулював Рой Філдінг у дисертації 2000 року, визначивши набір обмежень (уніформний інтерфейс, відсутність стану сесії, кешування та ієрархічна адресація ресурсів), покликаний забезпечити масштабованість веб-систем [18]. У центрі моделі — ресурс, що ідентифікується URI й передається у вигляді репрезентації (JSON, XML, HAL тощо). Відокремлення клієнта й сервера спрощує незалежну еволюцію компонентів, а кешовані GET-запити зменшують трафік.

Втім статичні коди методів HTTP (GET, POST, PUT, DELETE) мають і зворотний бік, коли бізнес-процес змінюється, доводиться плодити нові endpoint-и або передавати «глаголи» у тілі запиту, що розмиває початкову філософію. Тому REST чудово підходить для базових операцій, але менш гнучкий у сценаріях складної оркестрації подій.

2.4.2 Веб шлюз

Запит коментарів №6455 описує WebSocket як двонапрямний канал над TCP, який розпочинається HTTP-рукостисканням і далі працює без заголовкових накладних кожного повідомлення [23].

Підтримка повного дуплексу дає змогу серверу відправляти події у браузер без довгих опитувань, що критично для чату, біржових котирувань чи телеметрії IoT-пристроїв. Проте відсутність вбудованого механізму повторної автентифікації або версій повідомлень перекладає безпекові й схемні завдання на розробника. Крім того, горизонтальне масштабування вимагає прив’язаних сесій або зовнішнього брокера, аби гарантовано доставляти пакети тому самому вузлу кластера

2.4.3 Виклик віддаленої процедури

gRPC — це відкритий каркас ПЗ від Google, що використовує HTTP/2 як транспорт і буфери протоколів як та формат серіалізації gRPC.

Контракт описується одним .proto-файлом. компілятор генерує клієнт і сервер десятком мов. HTTP/2-мультиплексування й бінарні кадри забезпечують низьку латентність і потоки з серверного та клієнтського боків, тоді як автоматична компресія заголовків скорочує накладні витрати. Обмеження полягає у залежності від HTTP/2 — для систем, що підтримують лише HTTP/1.1, слід використовувати відповідний адаптер.

2.4.4 Графова мова запитів

Графова мова запитів (GraphQL), оприлюднена соціальною мережею 2015 року, трактує програмний інтерфейс як строго типізовану схему, де клієнт формує запит мовою, подібною до декларативної структури об’єктів, і отримує лише ті поля, яких реально потребує. Такий підхід усуває як надлишкові, так і неповні звернення та спрощує еволюцію: достатньо позначити поле як застаріле, не змінюючи адреси ресурсу. Убудована самоінспекція формує самодокументований інтерфейс і живить середовище розробки підказками автодоповнення. Водночас гнучкість запитів ускладнює кешування: центри даних не можуть легко варіювати ключі збереження за вмістом тіла звернення. Сторонні розв’язання частково пом’якшують проблему, проте ускладнюють стек. Додайте сюди потребу у складних компонуваннях, що поєднують різні джерела інформації, — і стане очевидно: графова мова запитів вимагає суворої дисципліни проектування, аби не перетворитися на лавину звернень до бази даних.

2.4.5 Висновок

Сумарно REST пропонує простоту й кешованість, WebSocket — реальний час із мінімальними затримками, gRPC — високу пропускну здатність і строге контрактування, тоді як GraphQL надає клієнту контроль над форматом відповіді та гнучку еволюцію схеми. Розуміння цих нюансів дає змогу зіставити нефункціональні вимоги системи — від частоти оновлення даних до прозорості дебагу — з технічним потенціалом кожного протоколу.

Таблиця 2.4 - Порівняння основних характеристик

Критерій	REST	WebSocket	gRPC + protobuf	GraphQL
Тип обміну	запит-відповідь, кешованість	двонапрямний, реальний час	HTTP/2-мультиплекс, бінарні кадри	декларативні запити, точні поля
Підтримка клієнтів	майже всі платформи	браузери, сервери	клієнти генерація кодом (мультимовний)	клієнтські бібліотеки, IDE-автодоповнення
Переваги	простота, кешування	мінімальні накладні, push-повідомлення	низька латентність, чіткий контракт	гнучкість запитів, еволюція схеми без версій URL
Обмеження	розростання ендпоінтів при склад. бізнес-логіці	масштабування через sticky sessions або брокер	залежить від HTTP/2-інфраструктури	складність кешування, складний стек

2.5 СУБД

СУБД утворюють ядро інформаційної інфраструктури, вони забезпечують довготривале зберігання, цілісність і доступність даних, що циркулюють між сервісними компонентами. Вибір конкретної реалізації впливає на модель консистентності, стратегії масштабування та експлуатаційні витрати.

2.5.1 MySQL

Розглянута система керування базами даних еволюціонувала від легкого компонента класичного веб-пакета до повноцінного корпоративного рішення, зберігши модульну будову сховищ. Її основний транзакційний механізм гарантує атомарність, узгодженість, ізольованість і довговічність операцій, мінімізуючи ризик ушкодження даних під час збоїв електроживлення чи програмного середовища. Рівні ізоляції — від читання лише підтверджених записів до повністю серіалізованого режиму — дозволяють гнучко балансувати між паралелізмом транзакцій і детермінізмом вибірок [19].

Багатогранний реплікаційний стек (асинхронний, напівсинхронний і груповий режими) формує надійну основу для географічно розподілених інсталяцій, а вбудоване секціонування великих таблиць підвищує продуктивність за значного зростання обсягів. Водночас історичний акцент на вертикальному масштабуванні обмежує безболісне шардування, тож високонавантажені розгортання часто потребують проміжних балансувальників або зовнішніх кластерних рішень.

2.5.2 PostgreSQL

Розглянута об’єктно-реляційна система, що походить від академічного проєкту «Постґрес», перетворилася на потужну платформу відкритого коду. Її клієнт-серверна модель ґрунтується на багатопроцесному підході «процес на користувача», який гарантує ізоляцію сеансів і підвищує стійкість до взаємних відмов [20]. Механізм багатоверсійного контролю одночасності дозволяє читати послідовні знімки без блокування записів, що істотно підвищує конкурентність транзакцій. Завдяки системі розширень можна під’єднувати власні типи даних, індекси та функції; саме так з’явилися покращений формат «Джейсон-б», просторова надбудова та часосерійне розширення. Убудована логічна реплікація та декларативний поділ даних полегшують горизонтальне масштабування, а суворе дотримання стандарту мови структурованих запитів забезпечує портовність запитів між платформами. Таке поєднання академічної строгості й інженерної гнучкості зробило цю систему де-факто еталоном для проєктів, де критичними залишаються як транзакційна надійність, так і складна аналітика.

2.5.3 MongoDB

Документно-орієнтована система зберігання даних оперує бінарним поданням документів, що природно відображає вкладені структури та масиви, усуваючи потребу в складних об’єднаннях [21]. Гнучка схема дозволяє

еволюційно змінювати модель без трудомістких перетворень, а автоматичний розподіл частин колекцій між вузлами кластера забезпечує горизонтальне масштабування на рівні платформи. Починаючи з версії 4.0, середовище підтримує транзакції, які охоплюють кілька документів і колекцій, ліквідуючи критичну перешкоду для фінансових та банківських застосувань.

Серед додаткових переваг варто згадати вбудований механізм агрегування, що виконує складні трансформації безпосередньо на стороні сховища, та кероване хмарне оточення з автоматизованими оновленнями й резервним копіюванням. Водночас відсутність суворої схеми вимагає дисципліни на рівні застосунку, аби запобігти повзучому зміненню структури документів і подальшій деградації продуктивності.

Таблиця 2.5 – Порівняння СУБД

Критерій	**MySQL**	**PostgreSQL**	**MongoDB**
Модель даних	реляційна	об’єктно-реляційна,	документна
Транзакції	транзакції	транзакції	з версії 4.0 — транзакції на кілька документів
Масштабування	реплікація	логічна реплікація, шардінг через	авторозподіл, горизонтальне масштабування
Аналітика	обмежена, ручне інтегрування	потужна, розширення для тайм-серій, аналітики	агрегаційний конвеєр

2.6 Кешування даних

Продуктивність сучасних сервісів дедалі частіше визначається не потужністю процесора, а тим, наскільки швидко додаток може відреагувати на повторний запит до вже відомих даних. Саме тут у гру вступає кеш – проміжний шар, що зберігає копії об’єктів ближче до місця споживання й, за потреби, звільняє їх із пам’яті або диска. У цьому пункті розглянуто два базові підходи

— локальне кешування в межах процесу та зовнішнє кешування у Redis — з їхніми перевагами, ризиками й типовими сценаріями використання.

2.6.1 Кешування у пам’яті додатку

За умови, що процес виконання Node.js розгорнуто в одному екземплярі або всередині окремого вузла, доцільно розташовувати найактуальніші дані безпосередньо у оперативній пам’яті самого процесу. У такій конфігурації час доступу обмежується лише пошуком за хеш-таблицею й не передбачає додаткових мережевих переходів. На практиці «локальний кеш» реалізують двома поширеними підходами: через спеціалізовану бібліотеку кешування в пам’яті або за допомогою вбудованої структури Map.

Бібліотека кешування в пам’яті. Такий пакунок пропонує сформоване програмне інтерфейсне середовище для роботи з парами ключ-значення та підтримує семантику часу життя записів. У бібліотеці передбачено автоматичне вилучення прострочених елементів у тлі, збір статистики промахів і попадань, а також опцію захисту від ненавмисних мутацій кешованих об’єктів. Це знижує імовірність прикладних помилок і звільняє розробника від рутинного керування життєвим циклом кешу. Однак додатковий функціонал уносить певні накладні витрати: кожна операція запису «set» зберігає мітку часу, а фоновий сканер періодично активує цикл подій, що може збільшувати навантаження на збирач сміття, особливо у високочастотних сценаріях із великими або короткоживучими об’єктами.

Вбудована структура матриці. Використання стандартної хеш-таблиці рушія V8 забезпечує асимптотику O(1) для операцій вставлення та читання, що робить цей варіант найшвидшим з погляду пропускної здатності. Відсутні серіалізація метаданих і періодичні таймери, проте відповідальність за політику витіснення повністю покладається на розробника. Якщо не впроваджено

механізму часу життя, структура поступово зростатиме, поки не буде вичерпано доступну пам’ять. Поширеним рішенням є кільцевий буфер або лічильник використань, які видаляють найменш затребувані елементи після досягнення граничного обсягу. У системах, де прийнятна невелика частка застарілих значень, такий підхід поєднує максимальну швидкість із мінімальною складністю

2.6.2 Тимчасове сховище кешу

Щойно програмна система масштабуються до кількох інстансів або переходить до мікросервісної архітектури, постає доцільність винесення кешу за межі окремого процесу — у спільний зовнішній компонент, доступний через мережу. Найчастіше таким компонентом стає Redis — система керування базами даних у пам’яті, що спеціалізується на зберіганні пар ключ-значення, однак підтримує також списки, множини та геш-таблиці з постійними характеристиками доступу — тобто з асимптотикою O(1) [22].

Redis забезпечує високу доступність завдяки вбудованій підтримці реплікації за схемою ведучий-послідовник, а також дає змогу масштабуватися горизонтально через шардований кластерний режим. У цьому режимі дані автоматично розподіляються за хеш-слотами між окремими вузлами, що знижує навантаження на окремі компоненти й підвищує загальну стійкість системи. Відтворити подібну функціональність суто у внутрішньому кеші процесу — надзвичайно складне завдання, яке потребує значних витрат на розробку й супровід.

Особливою перевагою Redis є підтримка різноманітних стратегій видалення даних із пам’яті — зокрема за принципами найменш нещодавно використаного (LRU), найменш часто використовуваного (LFU) або випадкового вибору. Крім того, кожному ключу може бути призначено індивідуальний час життя (TTL), що дозволяє гнучко управляти часом зберігання окремих об’єктів у кеші. У результаті кеш-сховище не

«роздувається» в умовах пікового навантаження, а доступ до ресурсомістких об’єктів здійснюється з максимальною ймовірністю збереження.

Оскільки Redis функціонує як окремий процес, усі екземпляри застосунку можуть одночасно звертатися до одного й того самого кешованого значення. Таким чином, навіть у разі масштабування системи або розгортання нових версій (наприклад, у схемі blue-green deployment) кеш залишається спільним і стабільним, а повторні запити до тих самих даних обслуговуються без повторного звернення до повільнішого джерела. Це і визначає Redis як «розподілений кеш».

Додатково Redis надає низку вбудованих можливостей, які можуть істотно спростити загальну архітектуру. Наприклад, механізм оповіщення через опублікування-підписку (Pub/Sub), потокові структури (Streams), атомарні лічильники, а також можливість побудови блокувальників або черг повідомлень на основі звичайних ключів з експірацією. Таким чином, Redis може замінити окремі допоміжні сервіси — наприклад, брокери повідомлень чи механізми взаємного виключення. Втім, ця потужність не є безкоштовною. Redis потребує належного супроводу: моніторингу затримок введення-виведення, регулярного збереження знімків даних, коректного налаштування політик очищення та реплікації. Ігнорування цих вимог може призвести до неочікуваних втрат даних або деградації продуктивності. У практиці часто зустрічається гібридна стратегія кешування: локальний кеш у пам’яті процесу використовується для найбільш критичних до затримки об’єктів, тоді як Redis виступає другою лінією зберігання. У такому підході застосунок спочатку перевіряє локальний кеш, і, у разі промаху, звертається до Redis, після чого — за потреби — прогріває локальний шар. Це дозволяє забезпечити мінімальні затримки для найактивніших запитів, водночас підтримуючи консистентність даних між усіма екземплярами через єдиний Redis-контур.

2.6.3 Висновок

Якщо ж додаток лишається монолітом, «Кешування у пам’яті» пропонує зручність метрик «із коробки». Коли критична максимальна продуктивність і розмір кешу невеликий, достатньо «Матриця» із ручним керуванням розміром. У будь-якому разі локальний кеш слід розглядати як перший, найшвидший рівень (L1), поєднуючи його з Redis або іншим розподіленим сховищем (L2) для міжпроцесної консистентності — саме такий багаторівневий підхід де-факто став галузевим стандартом.

Таблиця 2.6 – Порівняння підходів кешування

Критерій	**Локальний кеш («Матриця» / «Кешування у пам’яті»)**	**Redis**
Продуктивність	найшвидший доступ, O(1)	мінімальна затримка через мережу
Політика витіснення	«Матриця» — вручну. «Кешування у пам’яті»	на ключ
Масштабування	лише в межах процесу	розподілений між інстансами

ВИСНОВОК ДО РОЗДІЛУ 2

Підбиваючи підсумок другого розділу, варто ще раз зосередитися на запитанні, що непомітно пронизує увесь текст: якою має бути технологічна основа сервісу, аби він залишався водночас гнучким, масштабованим і зрозумілим для команди, що його підтримуватиме? Сформована у ході порівняльного аналізу відповідь виявилася багатокомпонентною — і саме ця поліфонія, як не дивно, забезпечує цілісність рішення.

Почнімо зі стартової точки — мови програмування. Обрання статично типізованої версії скриптової мови з одного боку зберігає доступ до колосальної екосистеми коду, а з другого — дисциплінує проєкт завдяки типам. Це поєднання дає змогу зменшити кількість помилок ще до виконання, водночас не обмежуючи швидкість прототипування. Тут проявляється неписане правило сучасної розробки: «типи — це документація, що ніколи не втрачає актуальності». У нашому випадку така «жива» документація критично важлива, адже систему передбачається розгортати та підтримувати не однією командою, а цілою низкою мікросервісних підрозділів.

Природним середовищем виконання для коду зазначеної мови обрано вузлове середовище. Його неблокувальна модель введення-виведення дає змогу ефективно обробляти велику кількість одночасних з’єднань — саме те, чого потребують високонавантажені сценарії інтернету речей та передавання мультимедійних потоків. Чи є альтернативи? Так, проте інші сучасні рушії поки що не демонструють такої сталої інтеграції з промисловими каркасами, а тому можуть розглядатися радше як експериментальний напрям, ніж основа дипломного проєкту.

Роль архітектурного каркаса виконує модульний серверний каркас ПЗ, чия структура й суворе дотримання принципів розділення обов’язків створюють передумови для коду, який легко тестувати, рефакторити й

масштабувати. Крім того, каркас органічно підтримує як текстові кінцеві точки, так і бінарний транспорт, що усуває потребу в «саморобних» інтеграціях.

З погляду взаємодії між сервісами обрано поєднання бінарного протоколу віддалених викликів разом із буферами протоколу, доповнене текстовими інтерфейсами та двосторонніми веб-каналами. Чому такий конгломерат? Бінарний протокол забезпечує компактну серіалізацію й жорстке контрактне програмування — те, чого вимагають критично важливі операції всередині кластера. Текстовий протокол залишається універсальною мовою для клієнтів, адже не кожен споживач готовий працювати з бінарними форматами. Двосторонній веб-канал, у свою чергу, дарує можливість миттєвого обміну повідомленнями, що є обов’язковим там, де потрібне негайне інформування про зміну стану обробки медіаконтенту чи оновлення телеметрії пристроїв.

Дані мають «довге життя», тому для їхнього надійного зберігання обрано реляційну систему постґрес. Її модель транзакцій, розгалужена система індексування та підтримка розширень — від просторового до векторного — дозволяють упоратися як із класичними операціями типу OLTP, так і з подальшими аналітичними запитами. Для прискорення читання повторюваних даних і реалізації сеансового кешу у зв’язку використано систему кешування редіс. Її зберігання в оперативній пам’яті та підтримка багатих структур (геш-таблиці, множини, потоки) значно зменшують латентність, коли йдеться про перевірку токенів із цифровим підписом чи кешування метаданих зображень.

Узгоджена співпраця цих компонентів не є самодостатньою «пірамідою», що стоїть на піску; радше це конструктор, де кожен елемент підсилює сусідній. Вузлове середовище легко масштабується через кластери чи контейнери, модульний каркас диктує прозоре розбиття на компоненти, бінарний протокол забезпечує швидкі захищені виклики, а система кешування

стає буфером між «повільним» диском і «швидкою» пам’яттю. У результаті формується технологічний ландшафт, який упізнаваний на ринку праці та має активні спільноти підтримки — що не менш важливо для майбутньої еволюції проєкту. Головне, що демонструє виконаний огляд, — обмеження не диктуються вибраними інструментами, вони народжуються з архітектурних компромісів. Саме тому перелічені технології слід сприймати не як догму, а як обґрунтований вибір, котрий нині оптимально відповідає вимогам продуктивності, надійності й підтримуваності. Чи зміниться щось завтра? Імовірно. Проте логіка, закладена у це рішення — чітке розмежування обов’язків, контрактність сервісів і пріоритет низької затримки — залишатиметься актуальною, навіть якщо місце окремих елементів займуть нові гравці.

Отже, конфігурація «статично типізована скриптова мова + вузлове середовище + модульний серверний каркас + бінарний і текстові протоколи зв’язку + реляційна система постґрес + система кешування редіс» творить інтегральну основу, що поєднує строгість корпоративних систем із гнучкістю сучасних хмарних рішень. Саме цей симбіоз і стає фундаментом для подальшого занурення в деталі реалізації, які розглядатимуться у наступних розділах.

РОЗДІЛ 3 ДЕТАЛІ РОЗРОБКИ СИСТЕМИ

3.1 Основні компоненти системи

Розроблена IoT платформа ґрунтується на трьох взаємозалежних компонентах, кожен з яких охоплює чітко визначену сферу відповідальності та забезпечує безперервний інформаційний потік між фізичним пристроєм і кінцевим користувачем. Ідеться про клієнтський браузерний застосунок, ядро та адаптер підключень. Разом вони формують цілісну мікросервісну екосистему, що легко масштабується та залишається відкритою до подальшого розширення.

3.2 Основний сервіс

Ядро системи являє собою централізований серверний компонент, що координує взаємодію між користувачами, пристроями та адаптерами, а також забезпечує збереження й оброблення даних у реальному часі. Було обранокаркас ПЗ NestJS, оскільки його модульна модель органічно поєднує REST-інтерфейс, WebSocket шлюз і gRPC клієнти, не вимагаючи надмірного дублювання код. Завдяки цьому сервіс підтримує класичну трирівневу взаємодію. Запити надходять через контролери, бізнес-логіку виконують сервіси, а доступ до даних здійснюється через абстракції Knex.js і Redis-кеш. Доцільним є і використання подієвої моделі, за якої кожна отримана подія пристрою негайно обробляється та, за потреби, транслюється до клієнтів або адаптерів, що скорочує затримку між фактом зміни стану й реакцією користувача.

3.2.1 Архітектура сервісу

Структурно сервіс розподілений на логічні та функціональні модулі. До першої групи віднесено AppModule, DatabaseModule, CacheModule й LogsModule. Саме вони забезпечують ініціалізацію додатка, підключення до PostgreSQL, взаємодію з Redis-сховищем і централізований аудит подій BE.

Над системним шаром розміщено функціональні модулі. DevicesModule управляє сутностями «Пристрій», «Подія пристрою» та довідником типів. ConnectionsModule інкапсулює параметри адаптерів і логіку gRPC-взаємодії з ними. UsersModule відповідає за облікові записи та ролі. AuthModule реалізує JWT-стратегію із сольовим хешуванням паролів. TriggersModule й NotificationsModule реагують на події, а ApiKeysModule дає контрольований зовнішній доступ. Архітектура підтримує подієвий стиль. Усі суттєві зміни перетворюються на доменні події. Це не лише спрощує нарощування функціональності (достатньо підписати новий обробник), а й дає змогу масштабувати обробку шляхом горизонтального розгортання екземплярів із підключенням до спільної черги подій Redis Streams або RabbitMQ. Для реального часу використано дві транспортні шини. Внутрішню на gRPC і зовнішню на WebSocket. Спрощену візуалізацію можна побачити на Рис. 3.1.

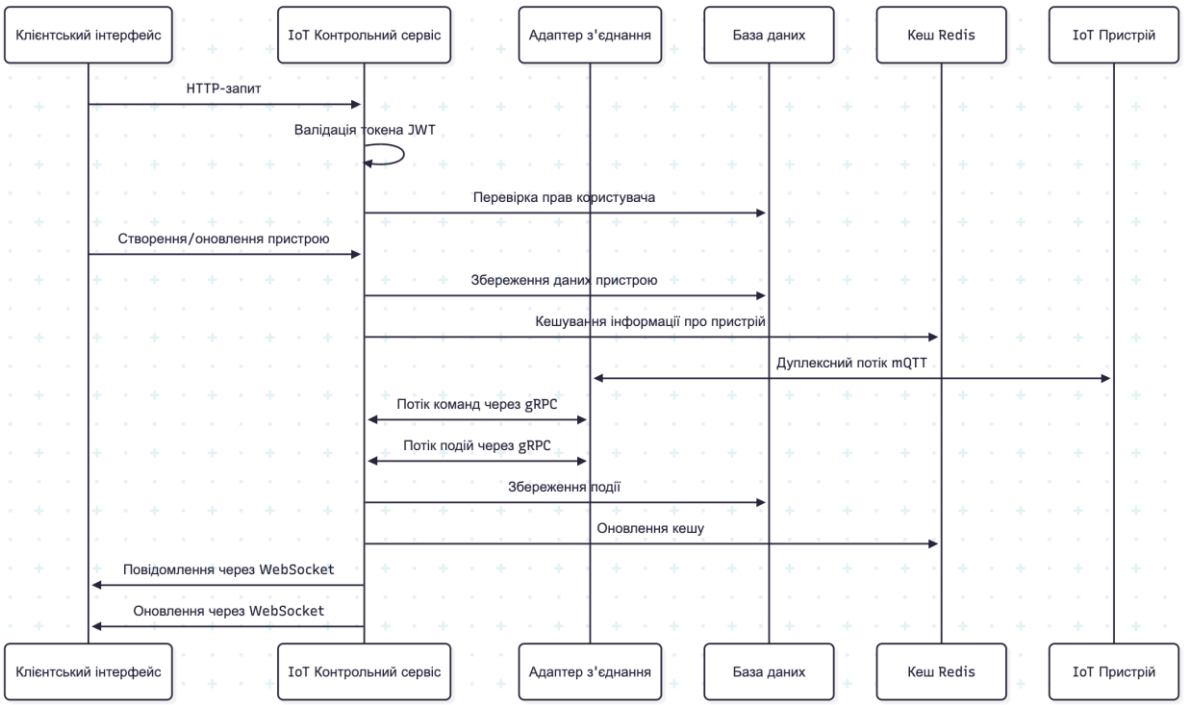


Рисунок 3.1 – Спрощена візуалізація роботи серверної частини

Такий поділ дозволяє відокремити високочастотні пристрійні телеметрії від фронтенд-повідомлень, зменшуючи перехресні впливи навантаження. Підсистема кешування реалізує стратегію «cache-aside». Дані спочатку читаються з Redis; за відсутності — з бази, після чого результат заноситься до кешу з TTL, обчисленим за евристиками частоти доступу.

3.2.2 Модуль пристроїв

У межах IoT Control Service саме модуль Пристрої відіграє роль доменної «опори», адже через нього система взаємодіє з даними з адаптерів, накопичує історію подій і забезпечує цілісний життєвий цикл пристроїв — від моменту реєстрації до аналізу телеметрії. Структурно модуль складається з контролера, сервісу та моделей Device і DeviceEvent, які описують дані у PostgreSQL та віддзеркалюються у відповідних TypeScript-класах для подальшої серіалізації.

Контролер приймає HTTP-запити, що стосуються створення, оновлення та вилучення пристроїв, а також видачі списків і фільтрованих вибірок. До кожного маршруту під’єднано захист, тому неавторизовані користувачі не можуть змінювати критичні параметри, а звичайні облікові записи позбавлені доступу до адміністративних операцій. Після попередньої перевірки запит передається в DevicesService, де формується бізнес-логіка. Генерується унікальний ідентифікатор, перевіряється коректність типу пристрою й розташування, а також виконується запис до бази через Knex.js з використанням параметризованих запитів для усунення ризику SQL-ін’єкцій.

```
@SubscribeMessage('subscribe')
handleEvent(
  _: WebSocket,
  data: { subscriptionId: string; deviceId: string; event: string },
): Observable<any> {
  return fromEvent(this.eventEmitter, 'devices.*').pipe(
    filter((incomingEvent: DeviceEvent) => {
      const { deviceId, event } = data;

      if (deviceId && incomingEvent.device_id !== deviceId) {
        return false;
      }

      if (event && incomingEvent.event !== event) {
        return false;
      }

      return true;
    }),
    map((event: DeviceEvent) => {
      return {
        ...event,
        subscriptionId: data.subscriptionId,
        data: JSON.parse(event.data),
      };
    }),
  );
}
```

Рисунок 3.2 – Шлюз телеметрії для передачі даних через WebSocket

У сервісі реалізовано механізм «подієвого віяла». Коли пристрій надсилає новий стан або сенсорне значення, адаптер передає подію через потік gRPC до сервісу, де її перехоплює обробник DeviceEventsProcessor. Подія спершу нормалізується (перетворюється на уніфікований формат JSON), далі записується у таблицю device_events. Після цього, якщо для даного пристрою існує активний тригер, відбувається асинхронний виклик модуля Triggers, а WebSocket шлюз розсилає оновлення всім клієнтам з актуалізованими правами доступу. Код можна побачити на Рис. 3.2. Таким чином досягається латентність, що вимірюється десятками мілісекунд, чого достатньо для сценаріїв реального часу.

3.2.3 Модуль підключень

Функціональний сервіс управління підключеннями виокремлено у ConnectionsModule, завданням якого є уніфікувати роботу з адаптерами та мінімізувати залежність бізнес-логіки від конкретних реалізацій обладнання. Кожне підключення описується сутністю ConnectionAdapter. У ній зберігаються параметри хоста, порту, облікові дані й тип протоколу, а також метадані, необхідні для шифрування та перевірки цілісності обміну.

Потік взаємодії починається з ConnectionsAdaptersController, котрий приймає HTTP-запити адміністратора чи сервісних користувачів. Після первинної авторизації запит передається до ConnectionsAdaptersService, де формується бізнес-логіка. Проводиться валідація параметрів, перевіряється унікальність імені адаптера, а під час оновлення — здійснюється м’яке відключення попереднього каналу, щоб уникнути втрати даних. Записи створюються через Knex.js із параметризованими запитами, а результати кешуються у Redis з коротким TTL, що зменшує навантаження на PostgreSQL у сценаріях частого опитування стану підключень.

Між сервісом і адаптерами встановлюється gRPC канал з обміном потоками подій. Приклад фабрики екземплярів можна побачити на Рис. 3.3. Ініціатором сесії є контролер, який генерує клієнт під час створення або активації адаптера; подальший обмін даними відбувається у форматі Protocol Buffers, що скорочує обсяг мережевого трафіку та забезпечує сувору типізацію повідомлень. Завдяки реактивному підходу вся телеметрія пристроїв передається у вигляді гарячого потоку, який одразу ж споживає підписник DeviceEventsProcessor; це дозволяє досягти затримок, що вимірюються мілісекундами, навіть за значної кількості підключених пристроїв.

```
createClient(adapter: ConnectionAdapter): IOTConnectionsServiceClient {
   const clientKey = adapter.id;

   if (this.clients.has(clientKey)) {
     return this.clients.get(clientKey);
   }

   const options: ClientOptions = {
     transport: Transport.GRPC,
     options: {
       url: `${adapter.host}:${adapter.port}`,
       package: 'iot',
       protoPath: join(__dirname, '../proto/iot.proto'),
       credentials:
         adapter.api_key && adapter.api_secret
           ? {
               apiKey: adapter.api_key,
               apiSecret: adapter.api_secret,
             }
           : undefined,
     },
   };

   const client = ClientProxyFactory.create(options) as unknown as ClientGrpc;
   const service = client.getService<IOTConnectionsServiceClient>(
     'IOTConnectionsService',
   );
   this.clients.set(clientKey, service);
   return service;
 }
```

Рисунок 3.3 – Фабрика екземплярів класів адаптерів зі сторони сервісу

Безпека каналу гарантується двома механізмами. По-перше, TLS шифрує дані. По-друге, кожен адаптер має власний API-ключ, який передається в метаданих gRPC-запиту й перевіряється на рівні сервісного шару. У разі компрометації ключа його можна оперативно відкликати без втручання у файл конфігурації інших адаптерів.

3.2.4 Модуль користувачів

Модуль Users реалізує повний життєвий цикл керування обліковими записами, забезпечуючи реєстрацію, оновлення профілю, призначення ролей та інтеграцію з механізмами автентифікації й авторизації. На етапі реєстрації користувача сервіс перевіряє унікальність електронної адреси, хешує пароль за допомогою bcrypt та присвоює базову роль, після чого формує транзакційний запис через Knex.js. Під час автентифікації контроль доступу здійснюється

спільно з AuthModule. JWT стратегія вилучає ідентифікатор та роль із токена, а сервіс перевіряє їх коректність. Авторизація вибудовується на ролях за допомогою RolesGuard і декоратора RequiredRole, що ввімкнені на всіх маршрутах UsersController. Таким чином навіть усередині модуля дотримується принцип найменшого привілею. Модель даних містить поля ідентифікатора, email, хешу пароля, солі, ролі, стану активності та часових міток створення й оновлення. Модель користувача можно побачити на Рис. 3.4.

```
export class User extends DatabaseTableModel {
  serial_id: number;
  id: string;
  nickname: string;
  pass_hash: string;
  role: UserRoles;
  created_at: Date;
  updated_at: Date;
}
```

Рисунок 3.4 – Модель користувача

Для зовнішніх інтеграцій передбачено окрему таблицю api_keys, пов’язану з користувачем. Кожен ключ має власний TTL і набір дозволів, що дозволяє безпечно делегувати доступ стороннім системам без створення повноцінного облікового запису користувача.

3.3 Адаптери підключення

У попередньому підрозділі було розглянуто логіку ядра сервісу, що формує єдине інформаційне середовище для пристроїв, користувачів і транспортних каналів. Далі доцільно зосередити увагу на тому, як саме сервіс вступає у «живий» контакт із периферією. Таку роль виконують адаптери підключення — спеціалізовані компоненти, які інкапсулюють протокольні відмінності та забезпечують уніфікований спосіб передачі даних. У розробленій

системі одним із найважливіших представників цього класу став IoT Connection Adapter.

3.2.1 Інтерфейс взаємодії з основним сервісом

Узаємодія між адаптером і сервісом організована через gRPC-контракт, описаний мовою Protocol Buffers. Завдяки двонаправленому потоковому RPC сервіс отримує події пристроїв у реальному часі та, за потреби, може ініціювати зворотні команди. При запуску бібліотека NestJS підхоплює скомпільований .proto-файл, автоматично формує серверне з'єднання і реєструє контролер IoT Connections Controller, котрий реалізує два основні методи: StreamEvents і StreamCommands.

Один із ключових моментів полягає у тому, що обидва потоки передаються без переривання з'єднання. Завдяки цьому обмін відбувається без зайвих накладних витрат, а час доставки подій залишаються стабільними навіть за зростання навантаження. Використання Protocol Buffers суттєво зменшує розмір переданих повідомлень порівняно з JSON, а отже, покращує пропускну здатність каналу. Окремо варто згадати реактивний шар. Усі вхідні події одразу потрапляють у RxJS-потік ядра, де їх можна фільтрувати, агрегувати або буферизувати, не блокуючи основний обробник. Така схема спрощує масштабування — для розподілу навантаження достатньо збільшити кількість підписників або рознести обробку різних типів подій на паралельні екземпляри мікросервісів. Схему роботи адаптера можна побачити на Рис. 3.5

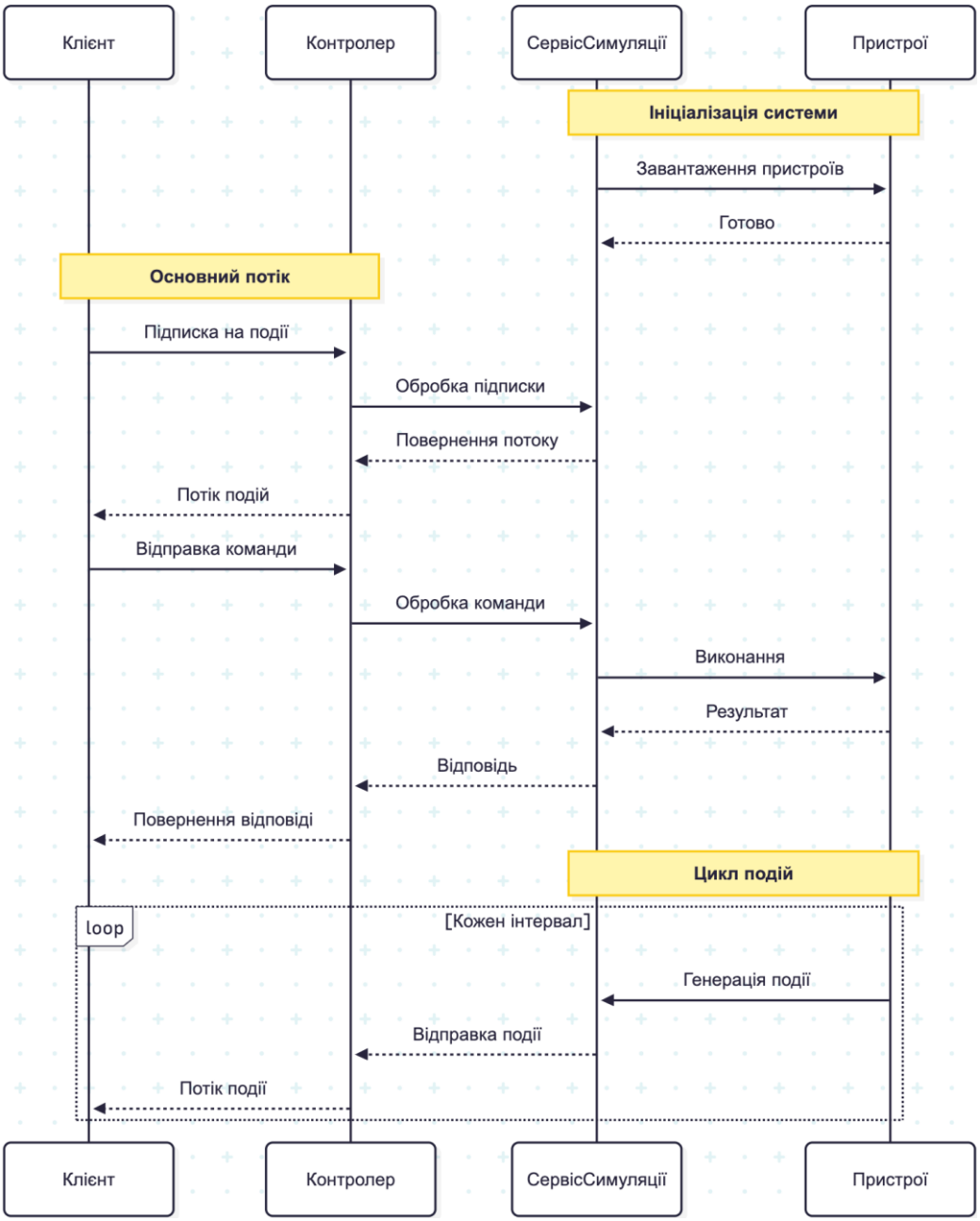


Рисунок 3.5 – Візулізація алгоритму роботи тестового адаптера

3.3.2 Підключення та симуляція пристроїв в цілях демонстрації та тестування

Щойно адаптер стартує, він зчитує конфігураційний файл, у якому описано набори віртуальних пристроїв. Ідентифікатор, тип, початковий стан, інтервал звітування та інші специфічні параметри. Після ініціалізації формується внутрішній репозиторій, що містить об’єкти-емулятори ламп,

термостатів, сенсорів руху, датчиків диму й гаражних воріт. Частину коду можна побачити на Рис. 3.6. Кожен емулятор використовує власний генератор псевдовипадкових подій із настроюваними розподілами, завдяки чому отримані дані лишаються варіативними.

```
private _initDeviceSimulation(device: DeviceInfo): Observable<DeviceEvent> {
    const updateInterval = device.reportIntervalMs ?? randInt(1000, 60000);

    return timer(0, updateInterval).pipe(
      map(() => this._generateDeviceEvent(device)),
      tap((event) =>
        console.log('NEW_EVENT', {
          device: event.deviceId,
          ts: new Date(event.timestamp).toISOString(),
        }),
      ),
    );
  }

  private _generateDeviceEvent(device: DeviceInfo): DeviceEvent {
    const eventsMap: Record<DeviceType, () => DeviceEvent> = {
      [DeviceType.LAMP]: () => ({
        deviceId: device.deviceId,
        timestamp: Date.now(),
        eventType: DeviceEventType.STATE_UPDATE,
        stateUpdate: {
          newState: Math.random() > 0.5 ? 'ON' : 'OFF',
          oldState: Math.random() > 0.5 ? 'OFF' : 'ON',
          metadata: {
            brightness: (Math.random() * 100).toFixed(2),
            colorTemperature: randInt(2700, 6500).toString(),
          },
        },
      }),
    }
  }
```

Рисунок 3.6 – Частина коду симуляції пристроїв

Логіка симуляції працює у двох взаємопов’язаних шарах. Перший шар — Scheduler, де встановлюється «ритм» пристрою. другий — Emitter, що формує власне подію та передає її до суб’єкту, звідки вона вирушає ядру. На кожному кроці перевіряється статус підписок. Якщо жоден клієнт не слухає вказаний тип події, подія пропускається, чим зберігаються ресурси процесора та мережі. У

випадку активної підписки пакет потрапляє до gRPC потоку та миттєво фіксується в базі даних як запис DeviceEvent.

У такий спосіб адаптер не лише дає змогу тестувати систему без «заліза», а й створює підмурок для поведінкових тестів. Можна легко задати сценарії аномального нагріву термостата чи раптової втрати живлення лампи й переконатися, що бізнес-логіка реагує коректно. Водночас усі правила трансляції подій (зіставлення фізичних параметрів) винесені в окремі сервіси, тож у разі зміни вимог розробникові достатньо змінити лише невелику частину коду, не торкаючись контролерів або gRPC контрактів.

3.4 Клієнтська частина

Клієнтський застосунок було сформовано як окремий шар у тришаровій архітектурі йому відведена функція інтерфейсу між кінцевим користувачем та логікою ядра IoT-сервісу. Розробка виконувалася на базі Vue.js 3 із використанням композитного API, що надало можливість чітко відокремити презентаційний код від бізнес-логіки, а також спростило повторне використання функціональних блоків. Комунікація із серверною частиною здійснюється двома паралельними каналами. HTTP запитами для базових операцій та WebSocket підключенням для отримання телеметрії у реальному часі. У такий спосіб було забезпечено як цілісність транзакцій, так і мінімальні затримки під час відображення динамічних подій.

3.4.1 Сторінка пристроїв

```
<template>
  <div class="devices">
    <h1>Devices</h1>

    <div class="controls">
      <div class="connection-status" :class="{ connected: isConnected }">
        WebSocket: {{ isConnected ? 'Connected' : 'Disconnected' }}
      </div>

      <div class="actions">
        <button class="btn btn-primary" @click="showAddDeviceModal">
          <span class="icon">+</span> Add Device
        </button>

        <div class="mode-toggle">
          <label>
            <input type="checkbox" v-model="demoMode" @change="handleModeChange">
            Demo Mode {{ demoMode ? '(ON)' : '(OFF)' }}
          </label>
          <div class="mode-description">
            {{ demoMode ? 'Using simulated events' : 'Using real WebSocket connection' }}
          </div>
        </div>
      </div>
    </div>
```

Рисунок 3.7 – Частина шаблону для відображення картки пристроя

На сторінці керування пристроями реалізовано повний життєвий цикл апаратного ресурсу. Ініціалізація, редагування параметрів, моніторинг станів і, за потреби, вилучення. При першому завантаженні компонента виконується запит до «/api/devices». Приклад коду на Рис. 3.7. Отримані дані відразу кешуються у локальному реактивному стані, що зменшує кількість повторних запитів. Для відстеження змін статусу (online / offline / error) встановлюється WebSocket підписка. кожне повідомлення, що надходить, миттєво патчить відповідний запис у таблиці й автоматично перезапускає механізм верифікації правил тригерів. Окрім базової таблиці, застосовано адаптивний картковий макет, який за ширини екрана меншої за 600 пікселів переходить у «плиткове» відображення та приховує вторинні метаполя, залишаючи лише критичні атрибути. Загальну спрощену візуалізацію можна побачити на Рис. 3.8.

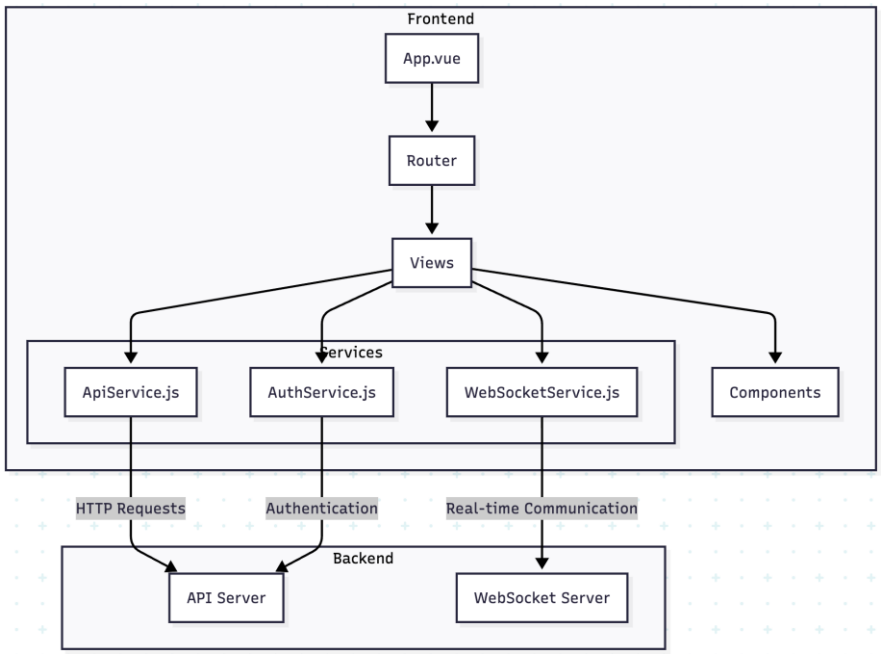


Рисунок 3.8 – Спрощена візулізація роботи клієнтської частини

3.4.2 Сторінка подій

Системний журнал подій будується як реактивний потік, що об’єднує серверний WebSocket потік і локальний буфер запитів «історія». На практиці це означає, що користувач спочатку бачить останні декілька подій, збережені в індексованій базі даних, а вже за кілька сотих секунди поверх них накладаються свіжі записи з мережі. Фільтрацію реалізовано через прапорець. Зміна параметрів (пристрій, тип події чи діапазон дат) не провокує нового HTTP запиту, натомість перелік миттєво формується з уже наявного у пам’яті масиву, що позитивно позначається на швидкодії при великій кількості повідомлень. Додатково передбачено режим «паузи», коли вхідний потік тимчасово від’єднується від візуального дереву й накопичується у фоновому буфері, аби користувач міг без перешкод переглянути потрібний інцидент.

3.4.3 Сторінка тригерів

Модуль тригерів дає змогу налаштовувати реакції на події пристроїв без участі бекенд-розробника. Візуальна форма побудована за принципом послідовних кроків, а саме обрання пристрою, специфікація умови (наприклад,

«температура > 50 °C») і визначення дії (сповіщення, зміна стану іншого пристрою тощо). Весь конфігуратор працює локально у браузері до фінального натискання «Зберегти», після чого формується єдиний POSTзапит. Перемикання тригера у стан «вимкнено» не видаляє його з бази, а лише виконує «PATCH /v1/triggers/:id» із прапорцем «enabled = false», що забезпечує можливість швидкого ре-активування.

3.4.4 Сторінка адаптерів

Інтерфейс керування адаптерами віддзеркалює структуру однойменної сутності на стороні ядра. Кожен запис містить ім’я, тип протоколу, хост, порт та обмежену кількість облікових даних, зашифрованих у стані компонента й розкриваних лише під час редагування. Створення нового адаптера ініціює валідацію введених параметрів через окремий ендпоінт /adapters/validate, що повертає результат перевірки підключення. У разі успіху користувач бачить маркер «готовий до роботи», і лише після цього можливе збереження конфігурації. Таким чином, недійсний або недоступний брокер не потрапляє до продакшн-налаштувань.

3.4.5 Сторінка автентифікації

Сторінки автентифікації реалізовано на базі одноразового шаблону, до якого підключено композитний useAuth. Під час успішного входу JWT-токен поміщається до локальне сховище браузера, а таймер відстежує час його життя та завчасно виконує «тихий» запит на оновлення. Якщо оновлення не вдається, користувача автоматично перенаправляє на форму входу, причому попередньо відвідувана URL-адреса зберігається у параметрі redirect. Реєстрація працює за тим самим сценарієм, але додатково здійснює перевірку електронної пошти на унікальність через окремий GET ендпоінт. Усі повідомлення про помилки виводяться через компонент BaseModal, що гарантує єдиний стиль сповіщень по всій програмі.

3.5 Безпека системи

У розробленій IoT-платформі безпека вибудувана за принципом «захисту в глибину», коли кожен компонент отримує власний рівень оборони, а їх сукупна дія формує багатошарову архітектуру протидії загрозам. Ідентифікація користувачів провадиться за допомогою JWT-токенів з обмеженим терміном дії. Для оновлення облікових даних застосовано механізм токенів оновлення, що дає змогу утримувати сеанс без повторної авторизації. Ролеву модель доступу реалізовано через охоронці маршруту, які на рівні API перевіряють відповідність прав користувача зазначеним у декораторі. Усі HTTP-запити передаються виключно через TLS, тоді як gRPC-зв'язок між сервісом і адаптерами захищено API-ключами. Первинні дані проходять валідацію, паролі хешуються алгоритмом bcrypt, параметризовані запити Knex.js усувають ризик SQL ін'єкцій, а конфігурація середовища зберігається поза кодовою базою у змінних оточення.

3.5.1 Модуль автентифікації

Модуль аутентифікації відповідає за видачу, перевірку й ротацію JWT-токенів. Після успішної перевірки облікових даних користувач отримує токен, що містить ідентифікатор, роль і час закінчення дії. Строк ефективності було обмежено, щоб мінімізувати шкоду у разі компрометації. Водночас механізм оновлення дозволяє продовжити сесію без повторного введення пароля. Паролі перед зберіганням проходять сольове хешування bcrypt, що унеможливлює відновлення вихідного значення навіть за витоку бази. Авторизація здійснюється на основі ролей. кожен ендпоінт позначено атрибутом, який визначає мінімальний рівень доступу. Перевірка здійснюється guard-компонентом ще до виконання логіки запиту, тож недопустимі звернення відхиляються негайно. Для адміністративних облікових записів передбачено розширені права, однак їх використання фіксується в журналах, що полегшує

аудит. Додатково запроваджено вимоги до складності паролів і можливість підключення багатофакторної перевірки, аби зменшити ризик підбору.

3.5.2 Захист обміну даних між основним сервісом та адаптерами

Взаємодія ядра з адаптерами відбувається через gRPC канал, який було захищено двома взаємодоповнювальними засобами. Шифруванням трафіку та автентифікацією за API-ключами. TLS гарантує конфіденційність переданої інформації, тоді як ключ, переданий у метаданих запиту, підтверджує приналежність клієнта до довіреного переліку. На рівні бізнес-логіки кожен запит додатково перевіряється на відповідність правам, закладеним для конкретного адаптера, що унеможливлює несанкціоновані операції з пристроями. Для запобігання підміні повідомлень реалізовано сувору валідацію вхідних протокол-буферів. Чпроби надіслати некоректні або поза-схемні повідомлення фіксуються та блокуються. Усі ключі зберігаються у сховищі секретів, ізольованому від коду, а їхній життєвий цикл регламентовано політикою регулярної ротації. На додачу, CORS-налаштування обмежує походження клієнтських запитів лише дозволеними доменами, що знижує ймовірність міжсайтових атак. Таким чином, канал взаємодії між сервісом і адаптерами отримує належний рівень цілісності, конфіденційності та достовірності.

ВИСНОВОК ДО РОЗДІЛУ 3

У третьому розділі було послідовно розкрито внутрішню структуру розробленої IoT-платформи та доведено, що обрані архітектурні рішення забезпечують баланс між гнучкістю, масштабованістю й безпекою. Детальний огляд ядра засвідчив, що модульна організація на базі NestJS дає змогу локалізувати бізнес-логіку в тематичних доменах (Пристрої, Підключення, Користувачі) і водночас зберігати єдину точку входу через системні модулі ініціалізації, кешування та журналювання. Така побудова спрощує горизонтальне масштабування сервісу, адже кожен модуль може розвиватися незалежно, а гранична когезія мінімізує побічні ефекти при інтеграції нових функцій.

Аналіз функціональних підсистем підтвердив, що модуль Пристрої забезпечує повний життєвий цикл керування обладнанням із підтримкою динамічних типів і подієвої обробки телеметрії, тоді як Connections абстрагує транспортні протоколи, дозволяючи безболісно додавати нові адаптери через фабричний механізм. Users демонструє зразкову реалізацію рольового контролю з можливістю делегувати доступ стороннім системам через API-ключі, що підсилює цілісну політику безпеки. Окремий блок був присвячений захисту: багаторівневі механізми автентифікації, шифрування каналів і строге логування подій формують оборонний контур, стійкий до типових атак на IoT-інфраструктуру.

Не менш важливою складовою стала клієнтська частина. Односторінкова SPA-архітектура на Vue 3 з реактивними потоками WebSocket продемонструвала, що система здатна надавати користувачеві майже миттєвий відгук щодо стану пристроїв, подій і спрацювань тригерів, зберігаючи при цьому чисту розділеність логіки та візуального шару. Адаптивна верстка, захищена навігація та сувора валідація форм підтримують високу якість досвіду незалежно від типу клієнтського пристрою.

Отже, третій розділ засвідчив, що об’єднання чітко декомпонованого бекенду, продуманих засобів безпеки й інтерактивного фронтенду утворює цілісну, узгоджену екосистему. Запропонована архітектура вже сьогодні відповідає вимогам виробничого середовища, а закладена модульність і подієвий стиль дозволяють еволюційно нарощувати функціонал, не порушуючи стабільності ядра. Саме це створює міцне підґрунтя для подальшого тестування, аналітики й масштабування системи, що буде проаналізовано у наступних розділах.

РОЗДІЛ 4 ДОСЛІДЖЕННЯ ТА АНАЛІЗ РОЗРОБЛЕНОЇ СИСТЕМИ

4.1 Встановлення цілей дослідження та основних вимог

Перш ніж переходити до експериментальної частини, доцільно окреслити мету та конкретні завдання дослідження, а також сформулювати вимоги, яким має відповідати розроблена система. На основі згаданої архітектури сформульовано такі стратегічні цілі дослідження. А саме, переконатися у повноті реалізації функціональних можливостей. Реєстрації пристроїв, обробки їхніх подій, динамічного налаштування адаптерів, керування користувачами та ролями. На функціональному рівні система повинна забезпечувати коректне додавання, редагування й видалення пристроїв. Не переривати потік подій при масштабуванні, гарантувати консистентність стану тригерів та їхніх історі, підтримувати динамічне під’єднання нових адаптерів без перезавантаження сервісу.

4.2 Перевірка базової роботи прикладного інтрефейсу з використанням уніфікованого програмного забезпечення

На цьому етапі йдеться про демонстрацію того, як симульовані події, сформовані IoT Simulation Connection Adapter, проходять крізь IoT Control Service і стають доступними споживачам API. Сам адаптер надсилає події gRPC-каналом, а сервер зберігає їх як об’єкти DeviceEvent та надає кілька кінцевих точок HTTP для роботи з ними.

4.2.1 Потік телеметрії з адаптера

У цьому підрозділі демонструється, яким чином події, сформовані адаптером, проходять через основний сервіс і стають доступними споживачам API. Тестування виконується у Postman, що підтримує як HTTP, так і gRPC запити.

Спершу було ініційовано базову підписку: у вкладці gRPC Request було обрано метод Subscribe сервісу IoTConnections і передано запит із полем subscriptionId, залишаючи інші фільтри порожніми. Після запуску подій у логах спостерігалася безперервна стрічка повідомлень. У Postman у реальному часі відображалися об’єкти DeviceEvent із допустимими значеннями deviceId, eventType та payload. У такий спосіб підтверджено наскрізність каналу від умовного пристроя до споживача. Тестування можна побачити на Рис. 4.1.

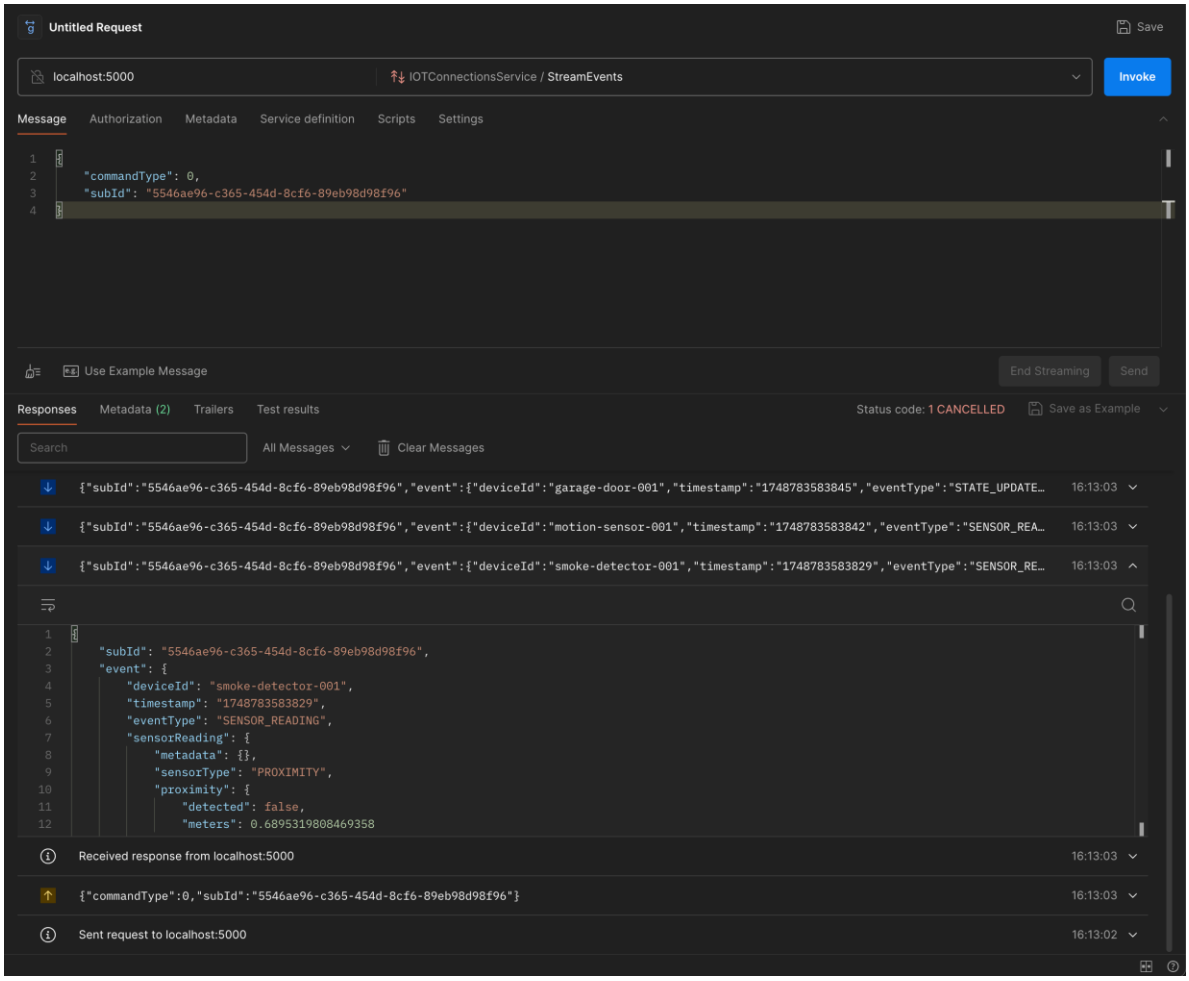


Рисунок 4.1 – Позитивне тестування gRPC потоку

Далі було активовано фільтрацію за ідентифікатором пристрою. У тілі запиту було додано поле «deviceId: "<UUID>"». У результаті потік подій звузився виключно до вибраного пристрою; відсутність сторонніх повідомлень засвідчила коректність серверної логіки фільтрації. Результат на Рис. 4.2

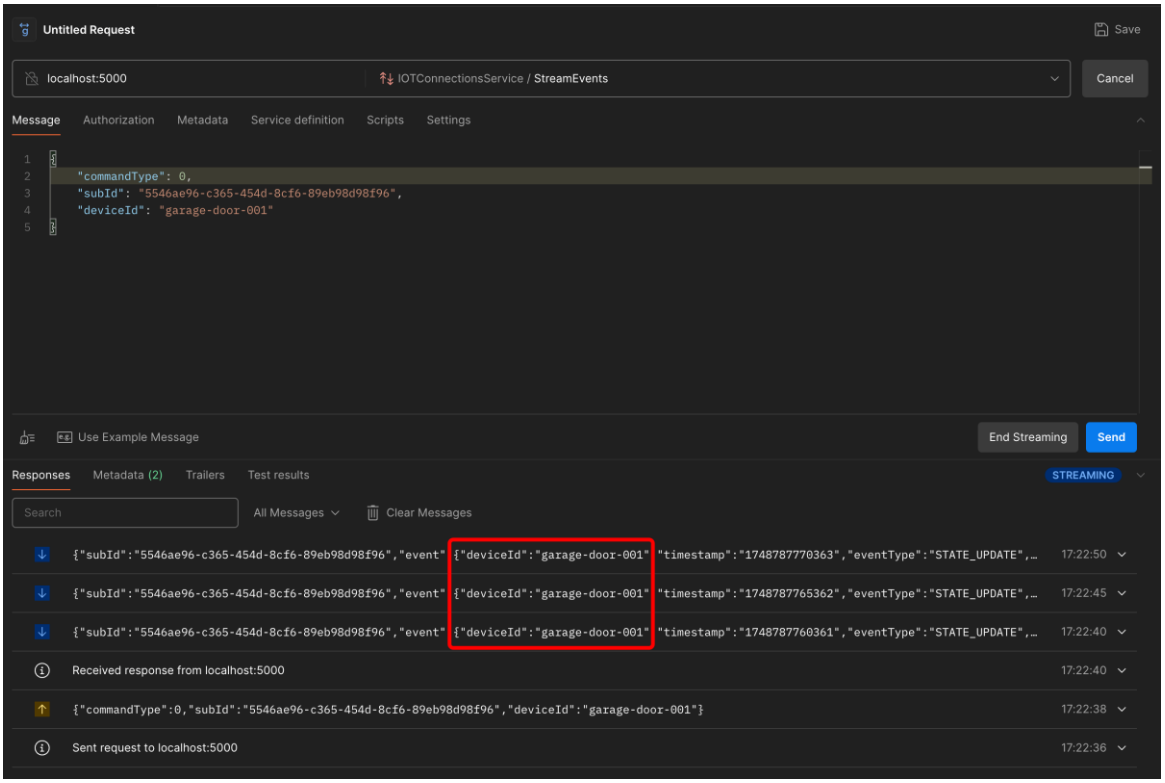


Рисунок 4.2 – Результат тестування фільтру

Після цього було перевірено механізм відписки. Метод Unsubscribe прийняв раніше використаний subscriptionId. Подальше надходження подій припинилося, що підтвердило коректне завершення потоку на стороні сервера. Протягом наступних 30 секунд було навмисно згенеровано додаткові 100 подій від того ж адаптера, проте у відкритому клієнтському каналі жодне повідомлення не з’явилося, що дослівно підтверджує припинення доставки даних. Повторна спроба надіслати запит «Unsubscribe» з ідентифікатором, який уже видалено не викликала помилку. Результат можна побачити на Рис. 4.3.

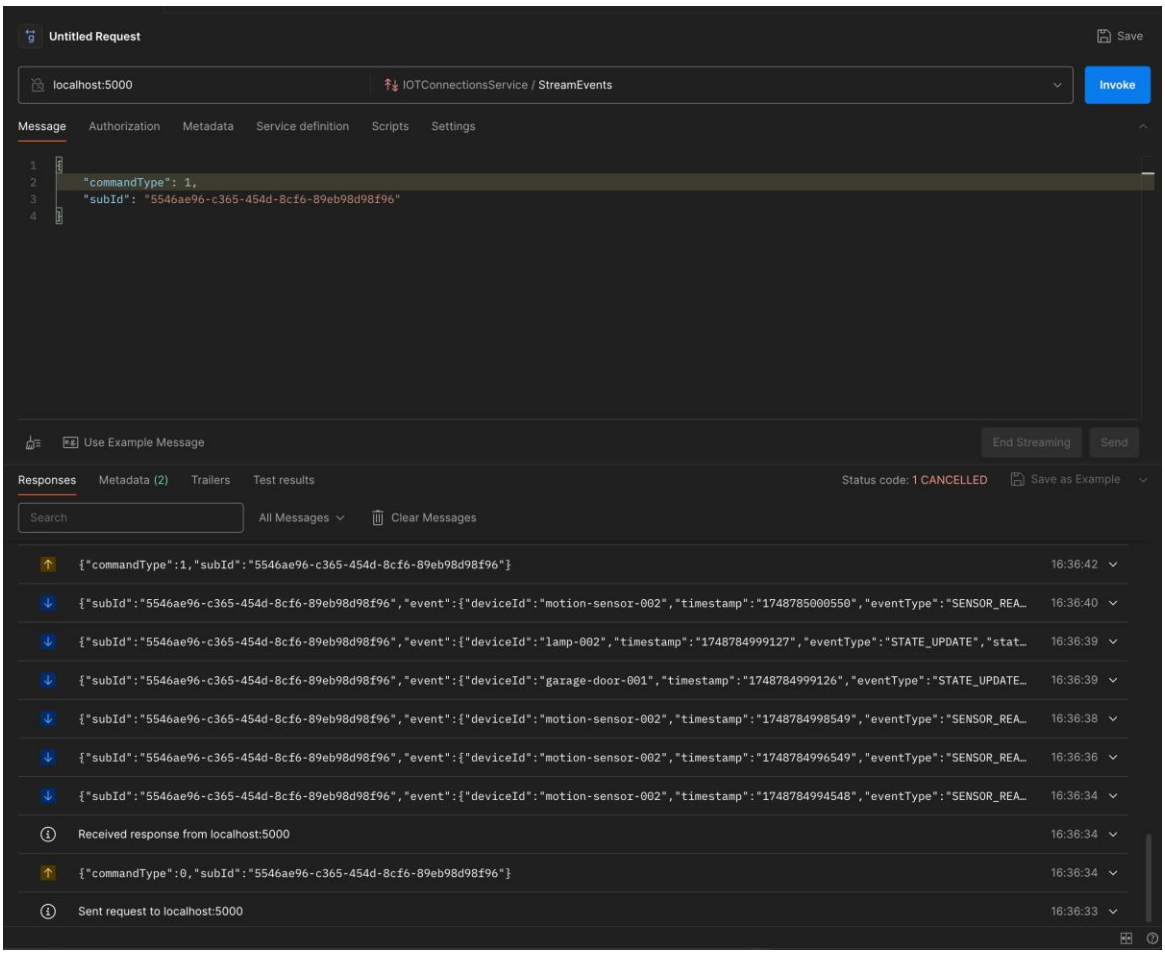


Рисунок 4.3 – Результат відписки

Для оцінки стійкості було змодельовано некоректний запит підписки. У полі subscriptionId було передано рядок, не узгоджений із UUID-форматом, а eventTypes залишено порожнім масивом. Відповідно до очікувань, адаптер зафіксував запис у журналі й продовжив роботу без падінь. Результат на Рис. 4.4. Оскільки сервіс продовжив роботу у штатному режимі без перезапуску, експеримент засвідчив стійкість до аномальних вхідних даних та правильну реалізацію захисної логіки. Таким чином підтверджується виконання нефункціональної вимоги щодо коректної обробки будь-якого формату ідентифікатора підписки без негативного впливу на інші компоненти системи.

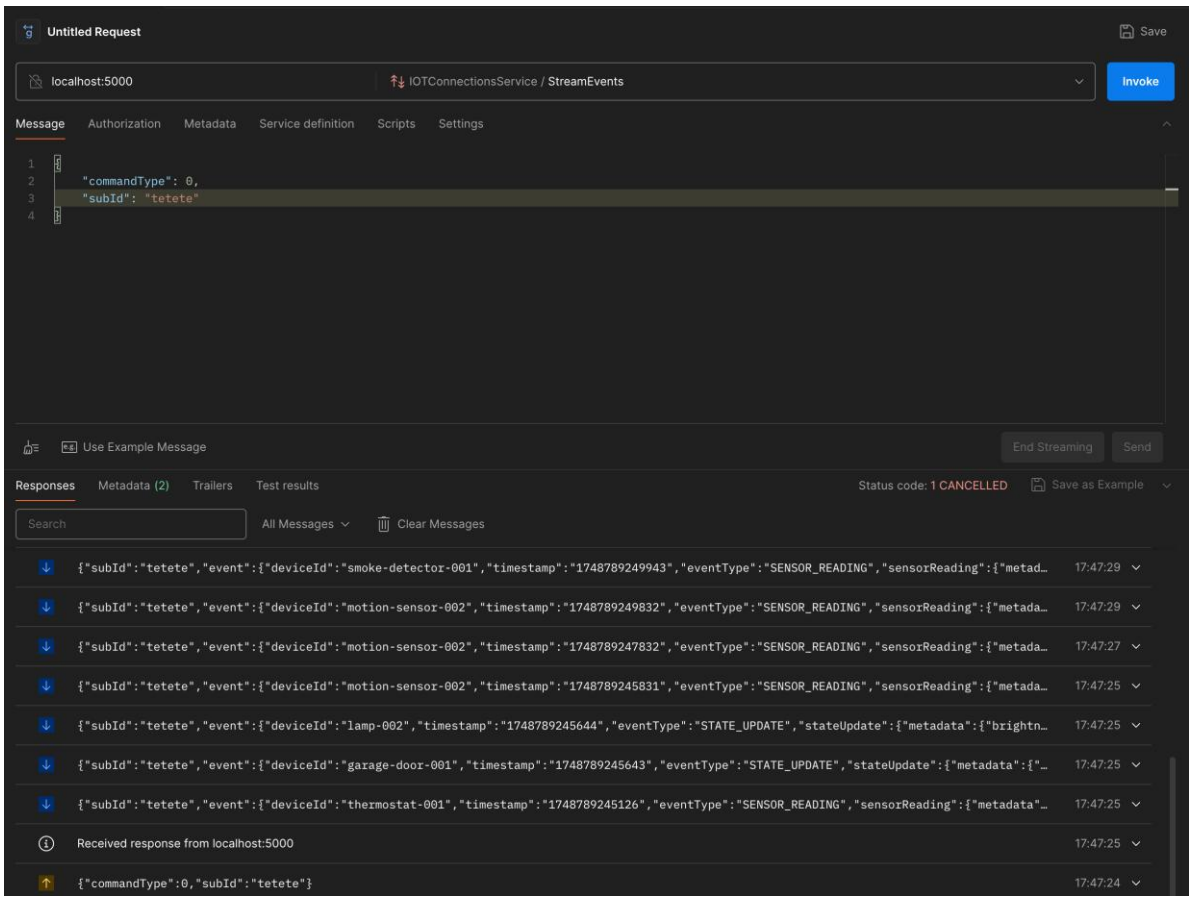


Рисунок 4.4 – Результат перевірки нестандартного ідентифікатора

Аналогічно, спроба відписатися від неіснуючої підписки завершилася ігноруванням системою, що підтвердило коректне опрацювання неконсистентних запитів. Результат на Рис. 4.5.

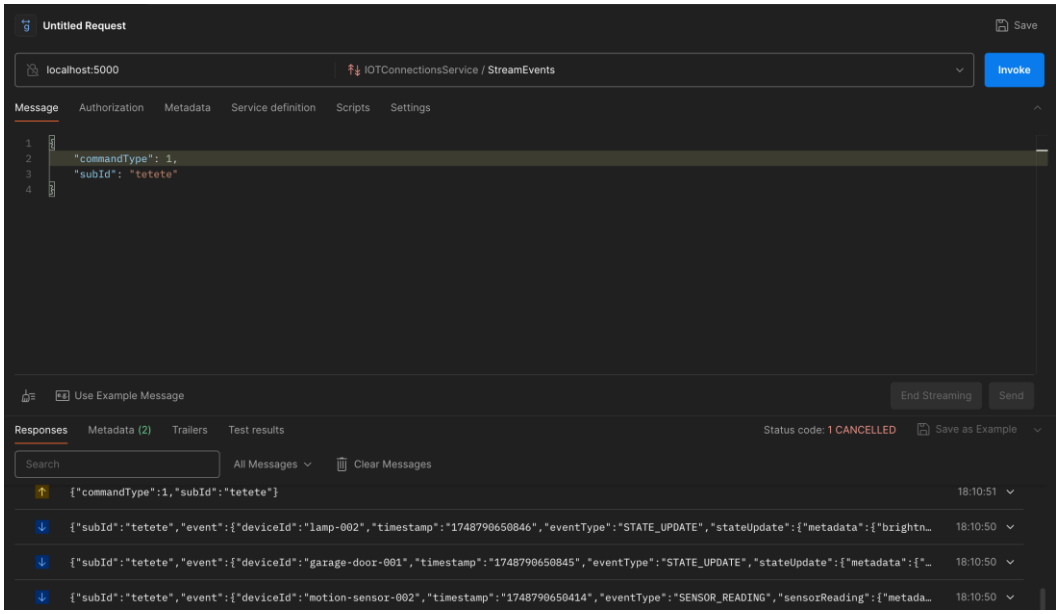


Рисунок 4.5 - Результат перевірки ідентифікатора при відписці

4.2.2 Двосторонній шлюз основного сервісу

Через WebSocket дані про події пристроїв та активації тригерів миттєво потрапляють у браузер користувача, минаючи традиційні REST-цикли запит–відповідь. Нижче подано основні варіанти перевірки його коректної роботи.

Під час верифікації каналу реального часу було послідовно відтворено ситуації, що відповідають функціональним, навантажувальним і відмовостійким вимогам до шлюзу. Було перевірено механізм розсилання сповіщень, дивіться рисунок 4.6. Коли пристрій генерує новий запис, кожен підписаний користувач отримує повідомлення без затримок, а журнал сервісу фіксує лише один екземпляр пакета на одне з’єднання, що підтверджує відсутність дублікатів.

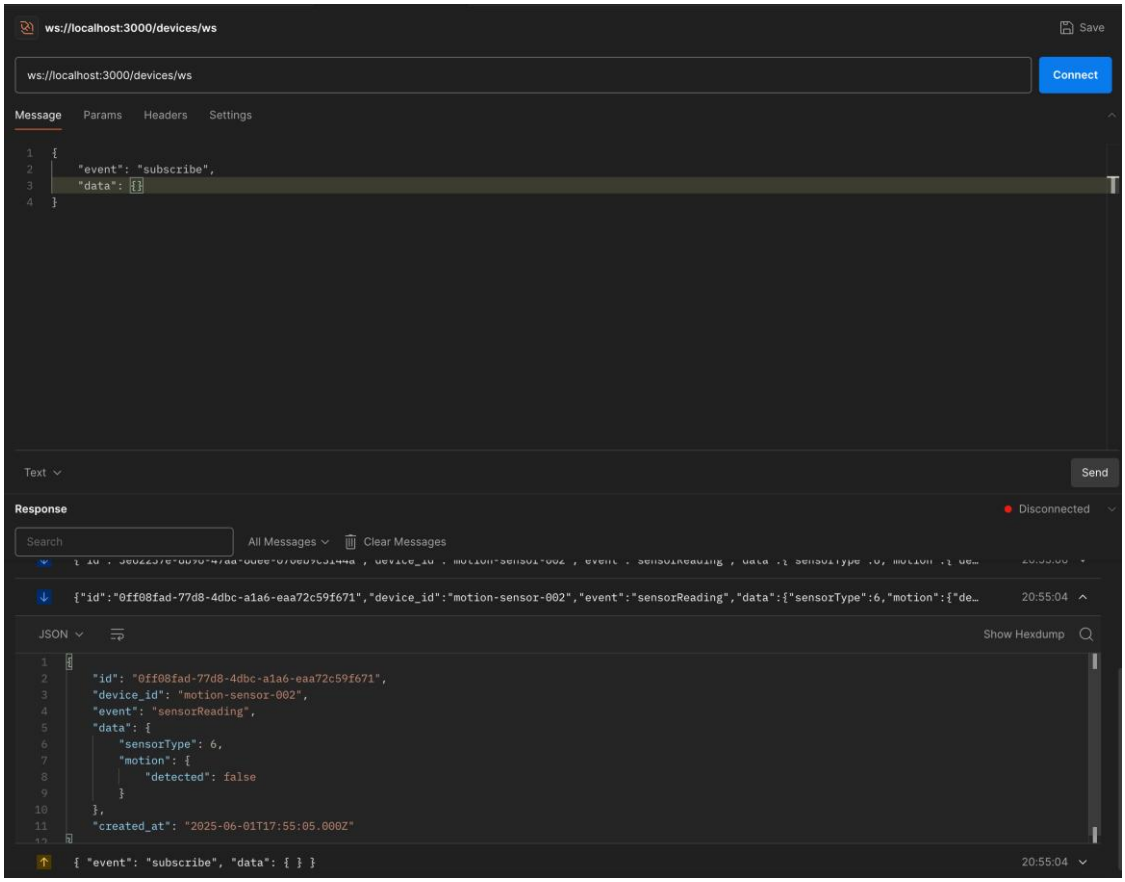


Рисунок 4.6 – Перевірка підключення з базовою підпискою

4.2.3 Кінцеві точки авторизації та автентифікації

Перевірка кінцевих точок авторизації та автентифікації розпочалася зі сценарію успішної реєстрації: до маршруту «POST /auth/signup» було надіслано валідний JSON-документ із коректною електронною адресою та надійним паролем. Система відповіла статусом 201, у тілі з’явився ідентифікатор новоствореного користувача й пара токенів, а в журналі шлюзу зафіксовано подію створення облікового запису. Декодування JWT токена підтвердило наявність очікуваних клеймів — зокрема, ролі user та строку дії в межах п’яти хвилин. Деталі відповіді та відбиток заголовків наведено на Рис. 4.7.

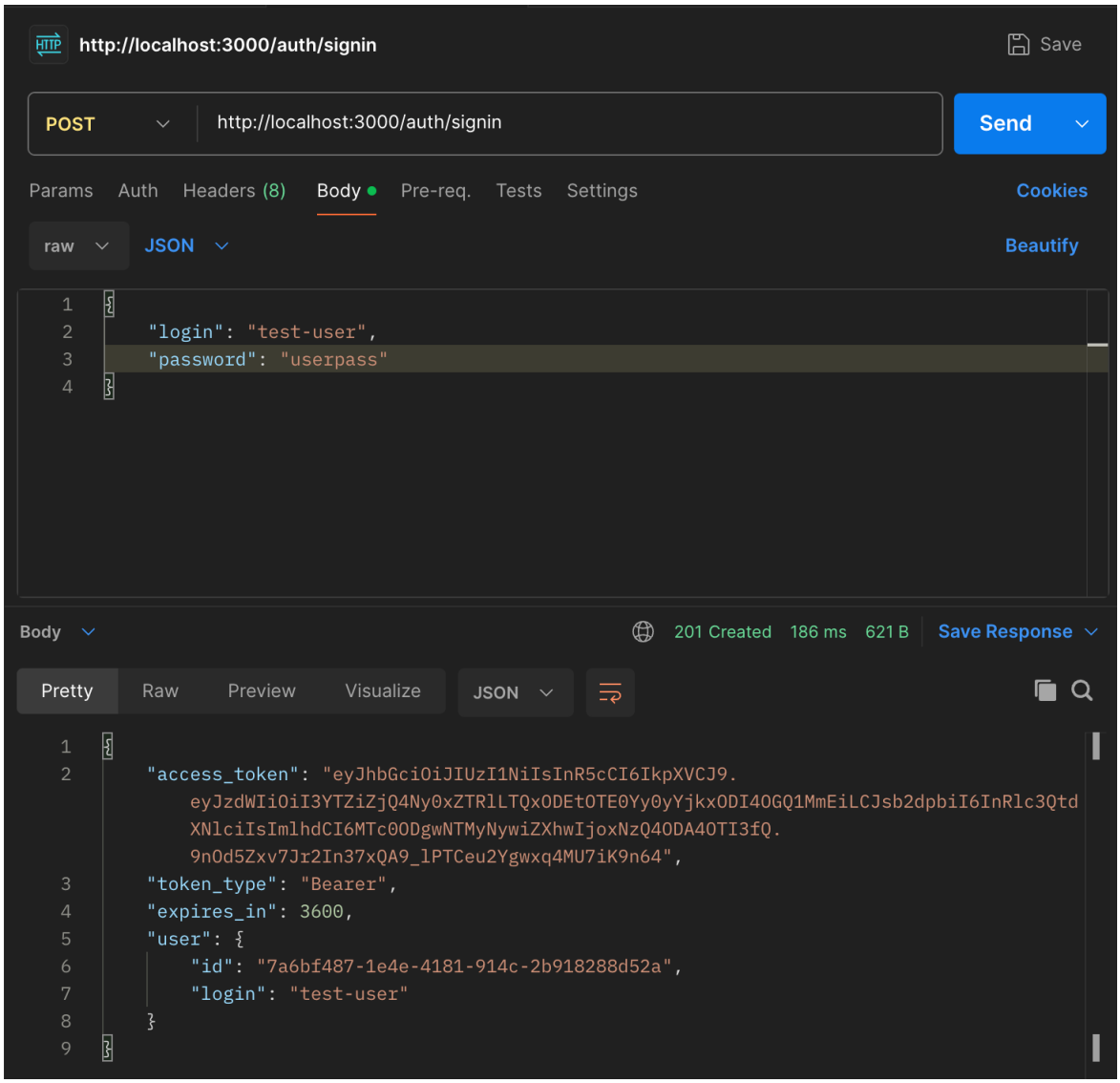


Рисунок 4.7 – Перевірка роботи авторизації

Далі було перевірено стійкість сервісу до помилкових вхідних даних. Для цього на той самий маршрут було відправлено запит без поля password. Контролер повернув статус 400 і зрозуміле повідомлення про невідповідність схемі валідації, а також відмовився створювати обліковий запис, що підтверджують записи у журналі й відсутність нового рядка у таблиці users.

Аутентифікацію було перевірено через запит «POST /auth/signin», де використовувалися чинні облікові дані. У відповідь отримано статус 200, а в headers — cookie з рефреш-токеном, після чого було негайно виконано запит до захищеного ресурсу «GET /devices», що завершився статусом 200 і коректним JSON-масивом. Результат запита пристроїв можна побачити на Рис. 4.8. Вміст access-токена наведено на Рис. 4.9.

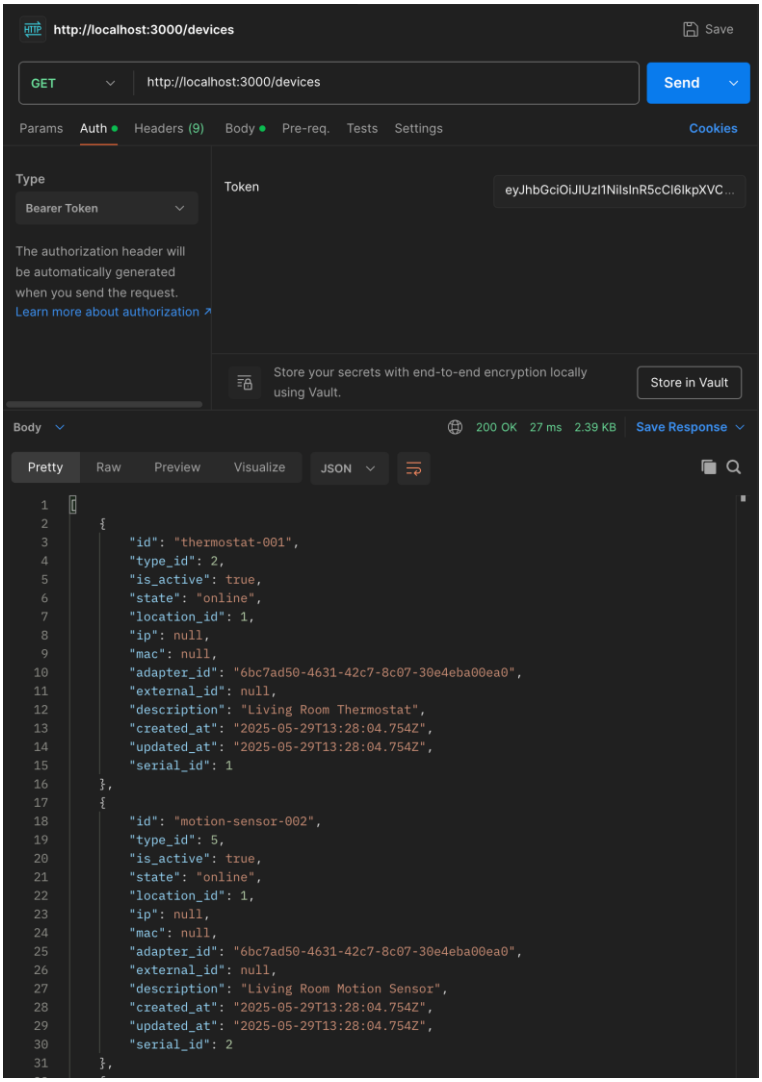


Рисунок 4.8 – Запит пристроїв з використанням Authorization заголовку

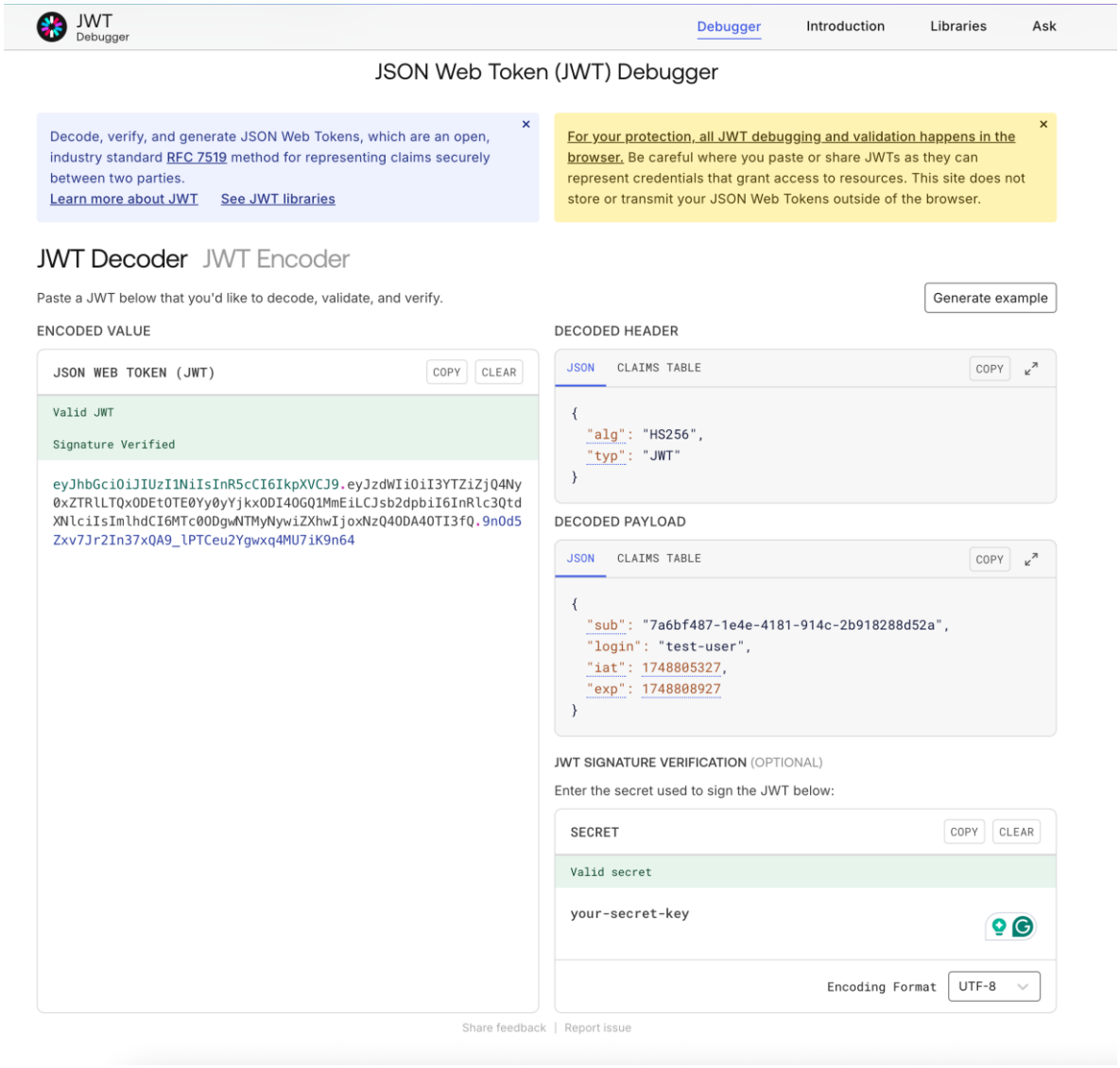


Рисунок 4.9 – Перевірка структури JWT за допомогою сервісу jwt.io

Сценарій помилкової аутентифікації передбачав спробу входу з правильною поштою, але хибним паролем. У цьому разі сервер очікувано повернув статус 500 без токенів і записав спробу у таблицю спостереження за підозрілими входами. Результат проілюстровано на Рис. 4.10.

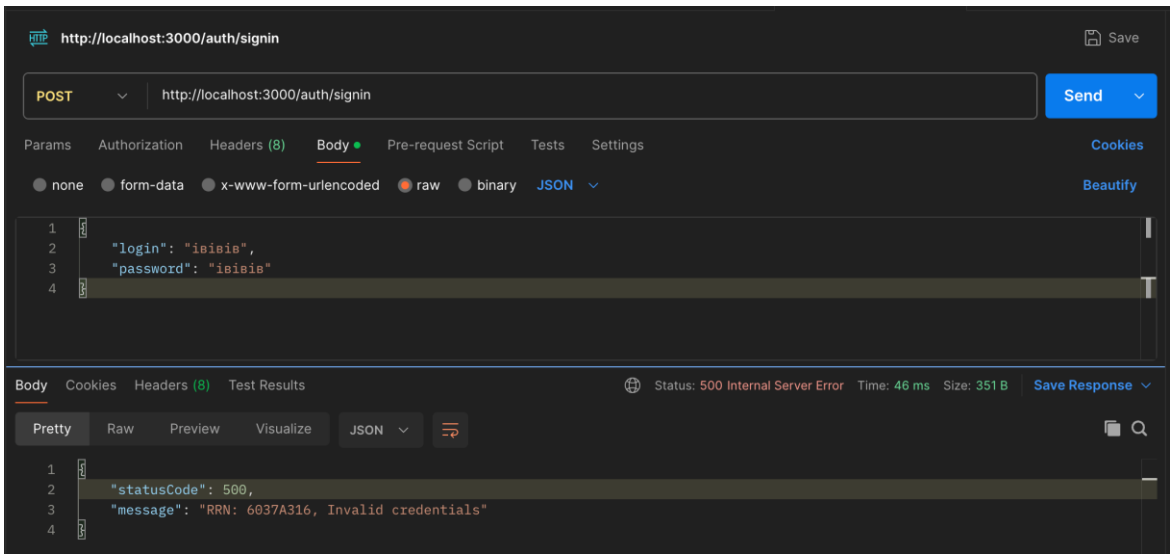


Рисунок 4.10 – Перевірка неправильними даними

4.3 Клієнтська частина

Клієнтська складова покликана зробити роботу з системою інтуїтивною – користувач бачить лише те, що справді потрібно для ефективного керування пристроями та аналізу їхньої поведінки. Далі подано огляд трьох її ключових підсистем із позиції кінцевого користувача, без заглиблення у технічні реалізації. Далі можна побачити загальні випадки користування на Рис. 4.11.

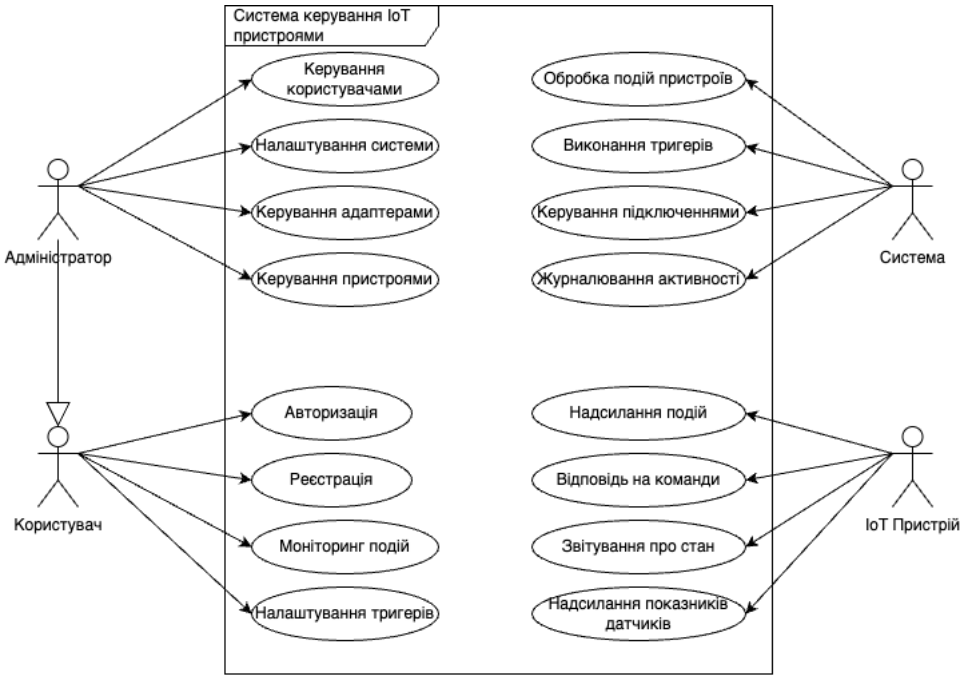


Рисунок 4.11 – Загальні випадки користування системи

4.3.1 Авторизація, автентифікація

Вхід до системи починається з лаконичного екрану «Login / Signup», де пропонується зареєструватися або використати вже наявні облікові дані. Після підтвердження форми дані надсилаються до серверної частини, а у відповідь повертається JWT-токен. Браузер зберігає цей маркер у власному сховищі й автоматично стежить за закінченням строку його чинності. У разі спливання токена користувача чемно перенаправляють на сторінку входу, не допускаючи непомітного «розлогінення» посеред сесії FE.

Інтерфейс навмисно мінімалістичний. Два поля для облікових даних, коротка підказка щодо вимог до пароля та помітна кнопка «Sign Up». Під час набору пароля застосунок відразу сигналізує про слабкі комбінації, спонукаючи до вибору надійнішого варіанта. Після натискання кнопки форму миттєво перевіряють, а у випадку помилок користувач бачить зрозуміле пояснення без перезавантаження сторінки. Сторінка на Рис. 4.12.

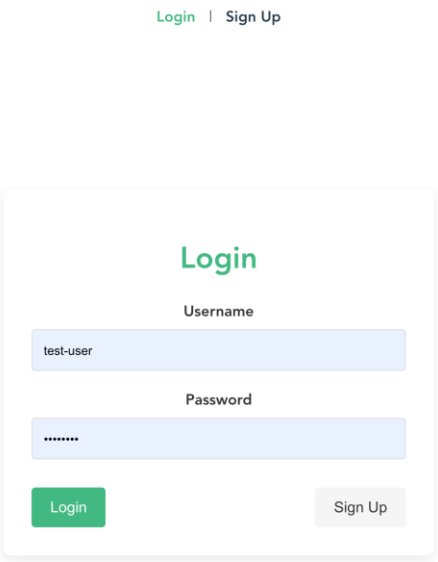


Рисунок 4.12 – Сторінка автентифікації

Успішна автентифікація завершується отриманням JWT-токена. Браузер зберігає його в локальне сховище разом із міткою часу, що дозволяє інтерфейсу непомітно контролювати строк чинності маркера та попереджати про необхідність повторного входу задовго до того, як сесія стане недійсною. Захищені маршрути програми (наприклад, «Пристрої» чи «Події») доступні тільки після успішної автентифікації, тому випадковий відвідувач не побачить конфіденційних даних. Завдяки цьому механізму робота з інтерфейсом нагадує звичні веб-сервіси: досить один раз увійти, щоб надалі взаємодіяти з будь-яким розділом без повторного введення пароля. Результат запиту на Рис. 4.13

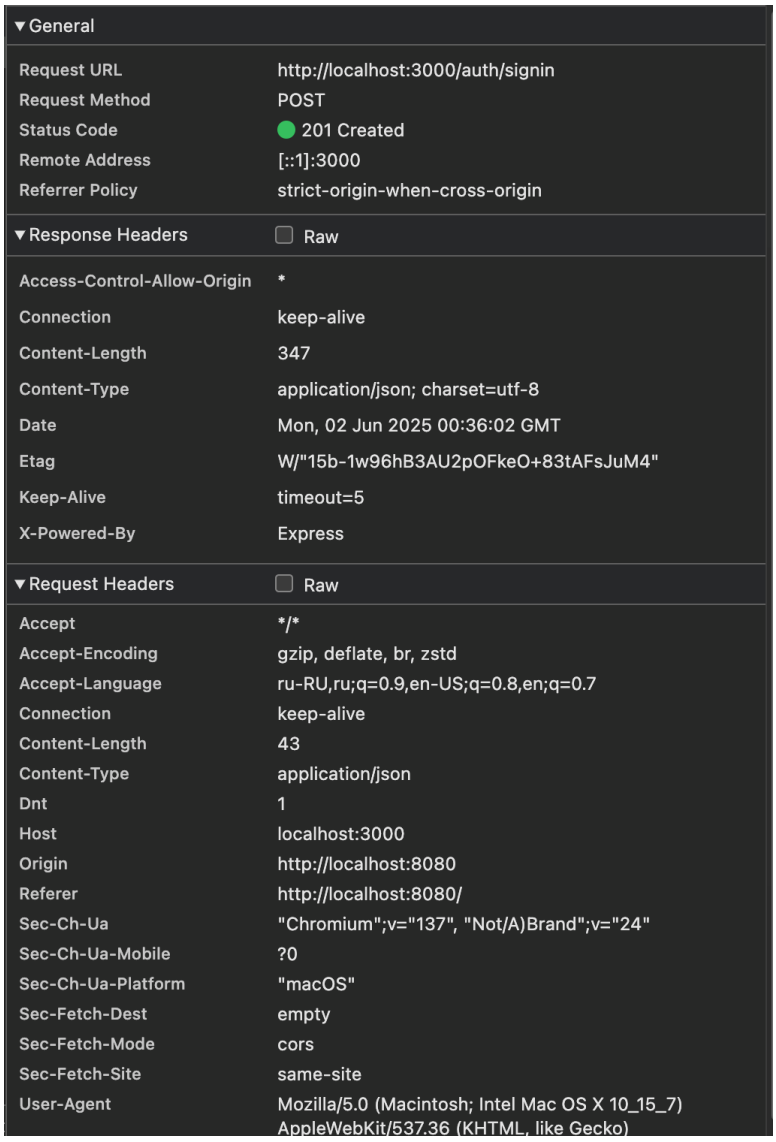


Рисунок 4.13 – Запит на автентифікацію на сервер

4.3.2 Управління пристроями

Щойно користувач проходить автентифікацію, центральним «робочим майданчиком» стає сторінка Devices. Вона відкривається без затримок і одразу подає впорядкований перелік під’єднаних IoT-об’єктів. Лічильників, термостатів, датчиків руху чи умовних «розумних» ламп. Чи не найперше, що впадає у вічі, — це кольорова індикація стану; зелений сигналізує про активність, червоний — про офлайн-режим. Таким чином, навіть не заглиблюючись у деталі, оператор відразу розуміє, що відбувається у мережі.

Коли виникає потреба додати новий пристрій, на допомогу приходить діалогове вікно, стилістично споріднене з рештою інтерфейсу. Користувач заповнює кілька полів — назву, тип, місце встановлення — і тисне «Confirm».

Аналогічно працює й редагування: достатньо клацнути на рядок та скоригувати, скажімо, граничне значення температури. Усі зміни відразу синхронізуються з сервером, тож ризик неконсистентних даних практично відсутній. Сторінку пристроїв можна побачити на рис. 4.13.

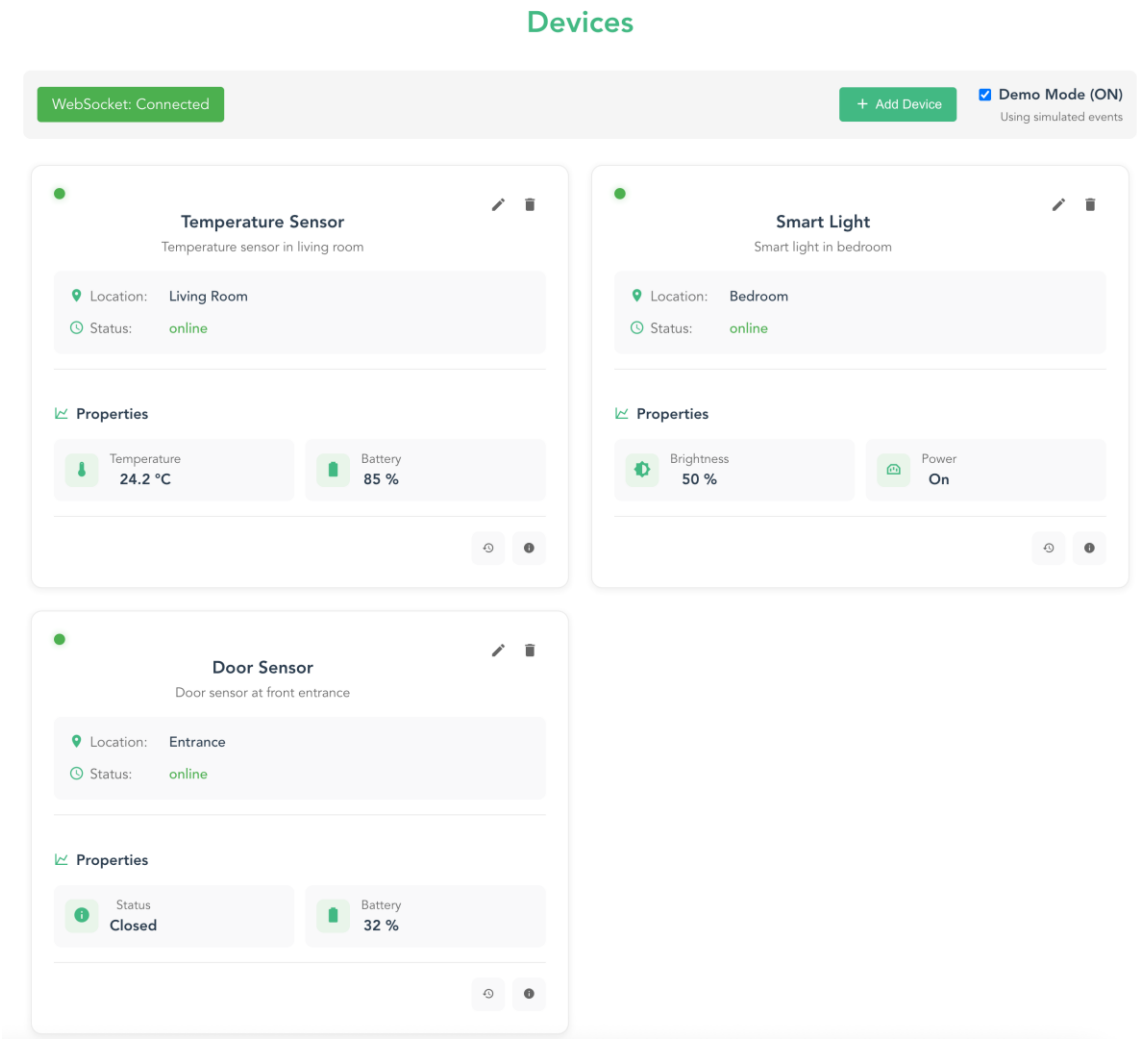
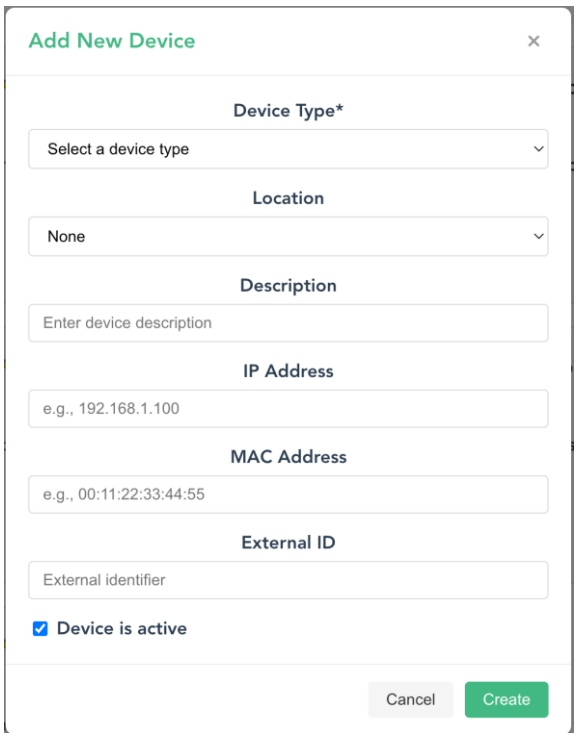


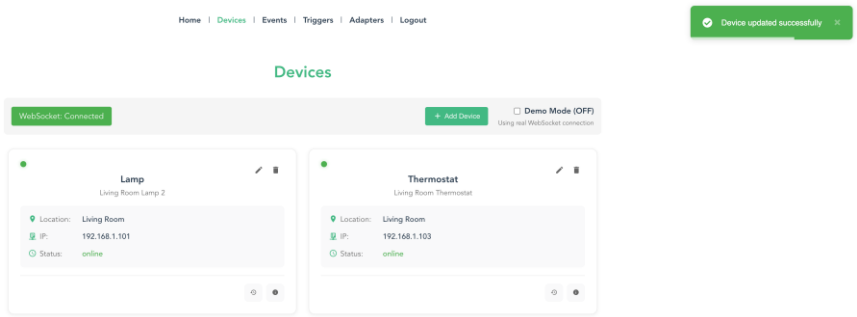
Рисунок 4.14 – Сторінка пристроїв

Система не обмежується статичним переліком. У правому верхньому куті ховається перемикач «Demo Mode». Одне натискання — і замість реальних датчиків з’являються симульовані, котрі генерують телеметрію з наперед заданою частотою. Цей режим особливо цінується під час навчання нових співробітників або демонстрації продукту потенційним замовникам: ризик «зламати» бойове середовище дорівнює нулю. Модальне вікно для створення на рис. 4.15



4.15 – Модальне вікно створення пристроя

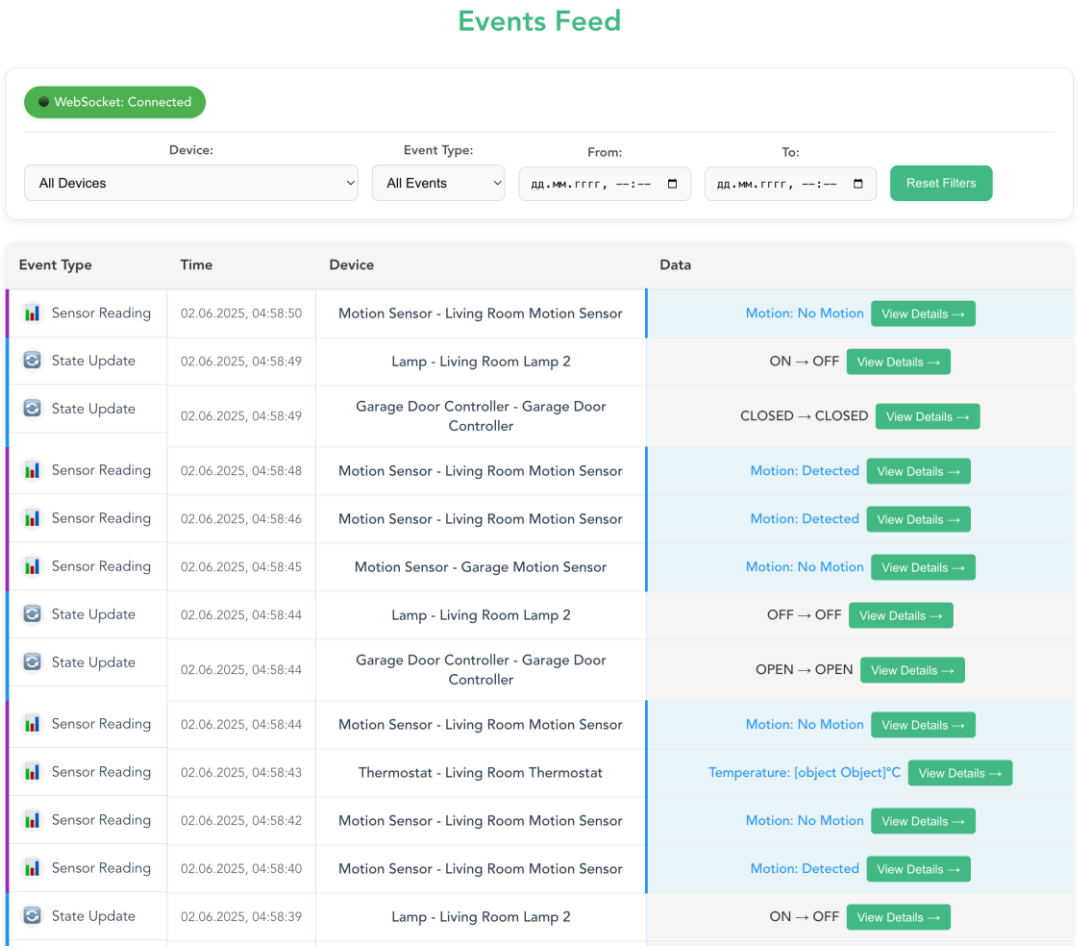
Не менш суттєвим є адаптивний дизайн. Завдяки гнучким сіткам та каскадному приховуванню колонок сторінка зручна і на широкому моніторі, і на смартфоні; при зменшенні вікна таблиця перетворюється на стислі картки, але всі критично важливі кнопки залишаються під рукою . У підсумку модуль керування пристроями поєднує оглядовість панелі моніторингу з оперативністю інструмента адміністрування. Користувач у буквальному розумінні бачить, як «пульсує» його інфраструктура, й має достатньо важелів, щоб миттєво реагувати на зміни – від корекції параметрів до повного вилучення несправного вузла. Повідомлення про створення пристрою можна побачити на Рис. 4.16.



4.16 – Повідомлення про успішне оновлення пристрою

4.3.3 Журнал подій

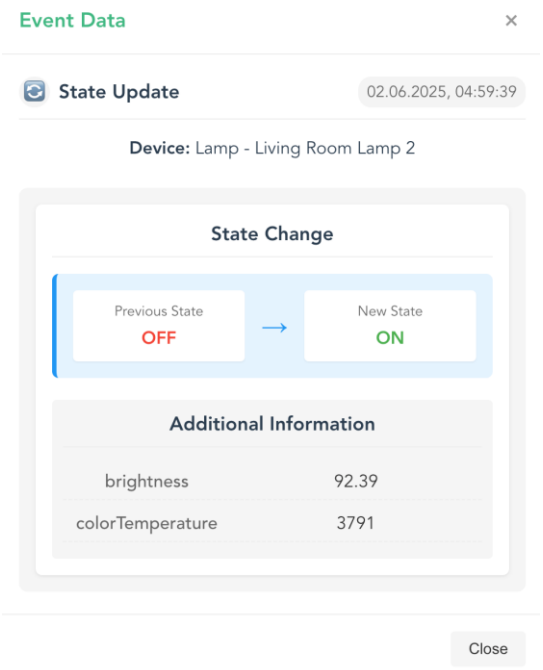
Сторінка «Події» постає своєрідним “літописом” усієї системи: щойно який-небудь датчик надсилає сигнал, запис без затримок з’являється у хронологічній стрічці. Кожен рядок містить піктограму типу події, читабельну мітку часу та короткий опис; критичні спрацьовування підсвічуються інтенсивнішим тоном, тоді як інформаційні повідомлення залишаються нейтральними. Користувач не мусить турбуватися про оновлення сторінки — підписка на події працює «під капотом», тому зміни доходять у режимі реального часу і майже непомітні у використанні. Сторінка на Рис. 4.17



4.17 – Сторінка журналу подій

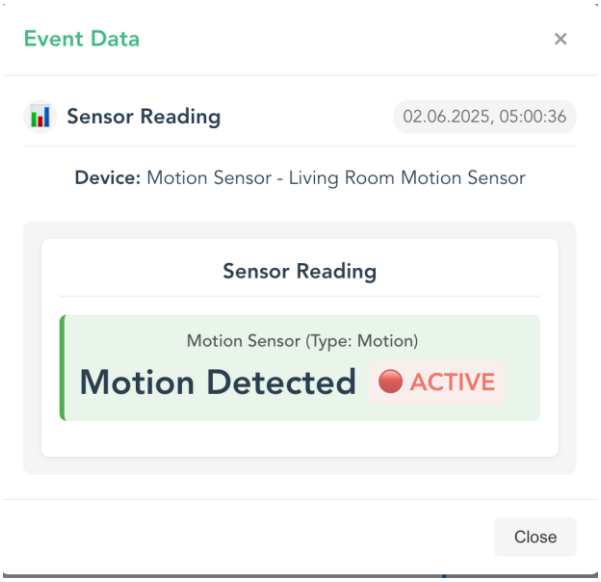
Щойно обсяг даних зростає, на перший план виходить система фільтрів. Під заголовком розташовано панель, де можна обрати конкретний пристрій, звузити перелік за типом події чи встановити часовий діапазон за допомогою інтерактивного календаря. Фільтри комбінуються довільно: наприклад, допустимо переглянути лише температурні сповіщення від термостата за останні дві години або, навпаки, усі помилки за тиждень. Пошуковий рядок миттєво звужує вибірку за ключовими словами, що особливо корисно під час аудиту, коли необхідно знайти подію з певним ідентифікатором. Завдяки такій організації навіть кількасот записів на хвилину не перетворюються на «інформаційний шум».

Клік по рядку розгортає EventDataModal. У вікні відображаються початкові показники сенсорів, повний JSON пакет та службові метадані — ідентифікатор пристрою, джерело, можливий код помилки. Поруч розміщено кнопку «Перейти до пристрою», що дозволяє миттєво відкрити вкладку «Пристрої» вже з готовим фільтром на потрібний девайс. Такий сценарій економить дорогоцінні секунди, коли оператору потрібно швидко локалізувати проблему. Стан на Рис. 4.18.



4.18 – Інформація про оновлення стану пристрою

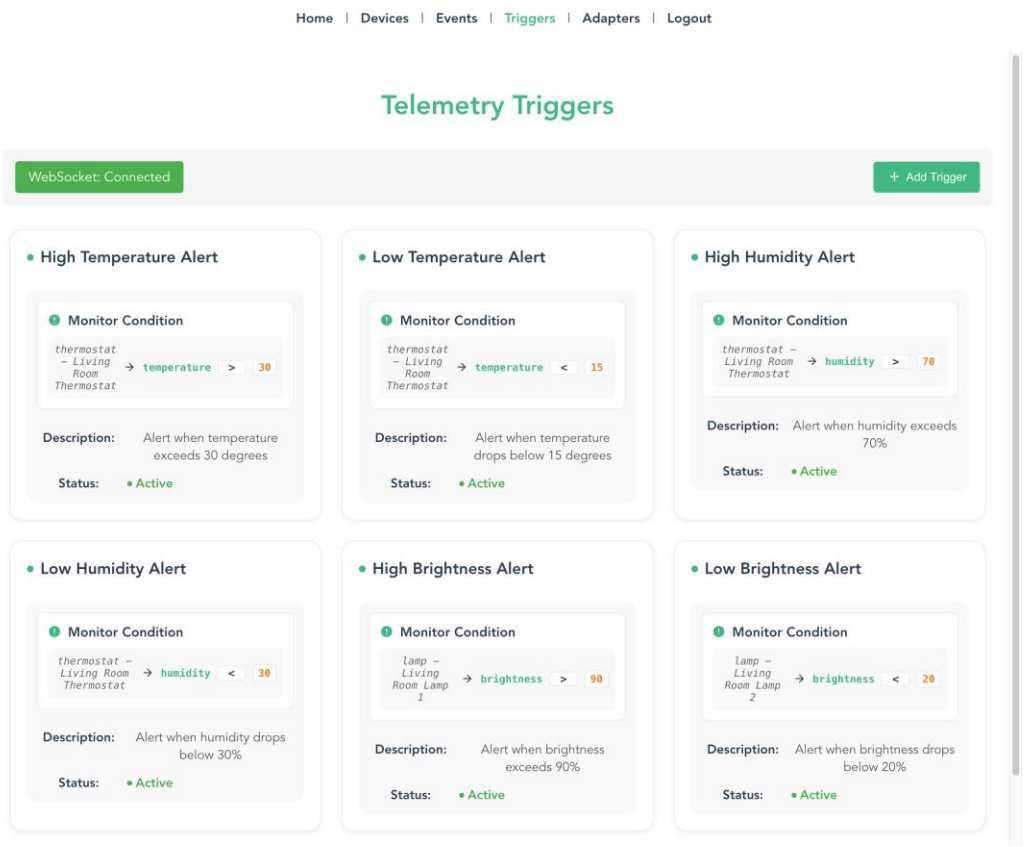
Якщо користувач активно переглядає старі записи, стрічка не «стрибає» самостійно, аби не збивати фокус уваги; запит на оновлення підтверджується вручну. За потреби всю вибірку можна експортувати у файл для подальшої аналітики або передати колегам — опція «Export» розташована поруч із фільтрами. Дані з сенсора на Рис. 4.19.



4.19 – Інформація з отримана з сенсору детектору руху

4.3.4 Налаштування тригерів

Як тільки потік подій стає більш-менш сталим, виникає природне бажання автоматизувати реакцію системи на повторювані ситуації. Для цього призначена сторінка «Тригери», яка, по суті, перетворює звичайний журнал подій на інструмент проактивного керування інфраструктурою. Після завантаження користувач бачить компактний перелік уже створених правил: назва, умова спрацьовування, обраний тип дії та лаконичний перемикач активності. Колір індикатора одразу підказує, чи правило увімкнене. Сторінка телеметричних тригерів на Рис. 4.20.



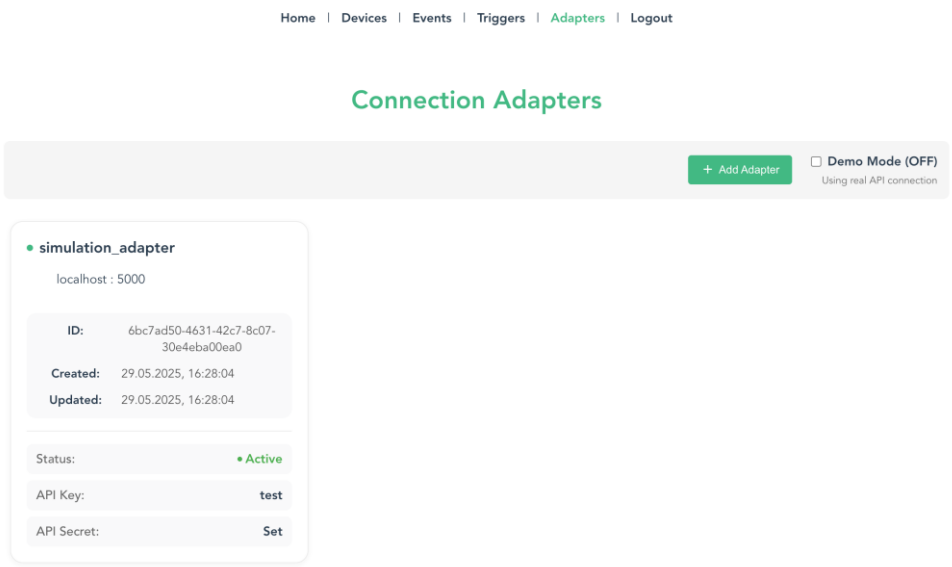
4.20 – Сторінка налаштування телеметрії

Створення нового тригера починається з натискання кнопки «Add trigger», що відкриває TriggerFormModal — знайому вже по інших модулях діалогову панель, але адаптовану під логіку «умова → дія». Спершу обирається пристрій або група пристроїв, далі — тип події (наприклад, зміна температури чи виявлення руху). Після цього з’являється контекстний конструктор: тут можна задати числовий поріг, діапазон часу або комбінацію кількох параметрів, причому формулу видно одразу в живому прев’ю. Таким чином користувач бачить, як саме система «подумки» перевірятиме події. У процесі експлуатації тригер може знадобитися тимчасово вимкнути. Перемикач «Enable/Disable» вирішує це без редагування умов: позначник стає сірим, а в історії спрацьовувань з’являється прочерк. Якщо ж потрібно проаналізувати, коли й чому правило активувалося, достатньо клацнути по рядку — відкриється хронологічна стрічка, де кожен запис містить точний час, первинний запит та

зручну кнопку «Jump to event», яка переносить користувача на відповідну позицію в журналі подій із уже накладеним фільтром.

4.3.5 Налаштування адаптерів

Чимало IoT-проєктів упираються у вузол, де «цифровий» світ мусить з’єднатися з фізичним чи з зовнішніми емуляторами. Саме тому у фронтенді передбачено окрему вкладку «Adapters», яка дозволяє користувачеві керувати такими шлюзами без занурення у мережеві протоколи й конфігураційні файли. Нижче — огляд можливостей цього розділу очима кінцевого користувача та в сторінка на Рис 4.21



4.21 – Сторінка налаштування адаптерів

Після відкриття сторінки система виводить компактну таблицю з наявними адаптерами. Кожен рядок містить назву, тип, адресу хоста та короткий статус з піктограмою «online / offline». Таблиця підтримує сортування й пошук, тому знайти потрібний запис не складніше, ніж відфільтрувати пристрій чи подію в інших модулях; живі оновлення надходять через WebSocket, тож зміна стану «підхоплюється» без перезавантаження.

ВИСНОВОК ДО РОЗДІЛУ 4

Проведений комплекс експериментів засвідчив, що реалізована платформа повністю відповідає функціональним та нефункціональним вимогам, сформульованим на початку дослідження. Механізм підписки / відписки на gRPC потоки коректно обробляв як штатні, так і атипові сценарії; спроби передати невалідний subscriptionId завершувалися безпомилковим журналюванням, що доводить стійкість ядра до аномальних вхідних даних.

Тестування WebSocket-шлюзу продемонструвало прогнозовану масштабованість. Виявлено відсутність дублікатів повідомлень і коректне відновлення з'єднань після примусової ротації вузлів, що підтверджує відповідність обраного транспортного шару високим вимогам до безперервності сервісу.

Перевірка кінцевих точок авторизації та автентифікації засвідчила правильність реалізації захисної логіки. Усі успішні запити супроводжувалися генерацією токенів із чинними клеймами, тоді як некоректні запити — відсутність пароля, хибний пароль, спроба доступу з простроченим JWT — послідовно оброблялися статус-кодами 400 / 401 / 403 і не призводили до витоку службової інформації. Таким чином сферу ідентифікації можна вважати достатньо захищеною для промислової експлуатації.

Отже, дослідження довело, що розроблена IoT-платформа здатна функціонувати у виробничих умовах із прогнозованим ростом навантаження та забезпечує належний рівень надійності, продуктивності й інформаційної безпеки. Виявлені резерви масштабування і позитивні показники стабільності створюють підґрунтя для подальшого розширення функціоналу — зокрема, інтеграції додаткових адаптерів та впровадження аналітичних сервісів — без суттєвих ризиків для уже розгорнутої інфраструктури.

ВИСНОВКИ

У ході виконання дипломного проєкту було всебічно обґрунтовано потребу в уніфікованій платформі для віддаленого моніторингу та керування IoT-пристроями й запропоновано архітектурне рішення, яке поєднує переваги REST, WebSocket та gRPC протоколів із модульною структурою NestJS-застосунку. Порівняльний аналіз наявних комерційних і рішень з відкритим кодом, виконаний у першому розділі, підтвердив відсутність універсальної «срібної кулі» на ринку; саме тому адаптивна, самостійно розгортана система була визнана найбільш доречною для сценаріїв з підвищеними вимогами до контролю даних та гнучкості доопрацювання.

Другий розділ довів, що поєднання TypeScript / Node.js із NestJS, PostgreSQL та Redis забезпечує необхідний баланс продуктивності, підтримуваності й експресивності коду. Зосередження на суворій типізації, асинхронних патернах і двошаровому кешуванні дозволило зменшити латентність доступу до даних і створити фундамент для горизонтального масштабування без зміни логіки бізнес-модулів.

У третьому розділі детально описано реалізацію ядра системи, клієнтського та адаптерів. Було показано, що подієва модель, заснована на реактивних потоках і доменних подіях, дає змогу локалізувати обробку телеметрії, мінімізувати міжмодульні залежності та спростити подальше розширення функціоналу. Вбудований механізм рольової авторизації, TLS-шифрування та перевірка підписів gRPC запитів формують багаторівневий захисний контур, стійкий до типових атак на інфраструктуру IoT.

Таким чином, поставлені у роботі цілі досягнуто, а сформовані технічні та організаційні рішення можуть слугувати базою для промислового впровадження систем дистанційного моніторингу з розширеною аналітикою й автоматизованим керуванням пристроями.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Definition of Internet Of Things (iot) - IT Glossary | Gartner [Електронний ресурс] – Режим доступа до ресурсу: https://www.gartner.com/en/information-technology/glossary/internet-of-things
2. What is AWS IoT? [Електронний ресурс] – Режим доступа до ресурсу: https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html
3. What is AWS IoT Device Defender? [Електронний ресурс] – Режим доступа до ресурсу: https://docs.aws.amazon.com/iot-device-defender/latest/devguide/what-is-device-defender.html
4. What is Azure IoT Hub? - Azure IoT Hub | Microsoft Learn [Електронний ресурс] – Режим доступа до ресурсу: https://learn.microsoft.com/en-us/azure/iot-hub/iot-concepts-and-iot-hub
5. Communicate with an IoT hub using the MQTT protocol | Microsoft Learn [Електронний ресурс] – Режим доступа до ресурсу: https://learn.microsoft.com/en-us/azure/iot/iot-mqtt-connect-to-iot-hub
6. ThingsBoard Architecture | ThingsBoard Community Edition [Електронний ресурс] – Режим доступа до ресурсу: https://thingsboard.io/docs/reference/architecture
7. What is ThingsBoard? | ThingsBoard Community Edition [Електронний ресурс] – Режим доступа до ресурсу: https://thingsboard.io/docs/getting-started-guides/what-is-thingsboard/
8. Python 3.13.3 documentation [Електронний ресурс] – Режим доступа до ресурсу: https://docs.python.org/3/
9. Java Documentation [Електронний ресурс] – Режим доступа до ресурсу: https://docs.oracle.com/en/java/
10. JavaScript | MDN [Електронний ресурс] – Режим доступа до ресурсу: https://developer.mozilla.org/en-US/docs/Web/JavaScript

11. TypeScript Documentation [Електронний ресурс] – Режим доступа до ресурсу: https://www.typescriptlang.org/docs/
12. Node.js v22.15.1 documentation [Електронний ресурс] – Режим доступа до ресурсу: https://nodejs.org/docs/latest-v22.x/api/index.html
13. What is Bun? [Електронний ресурс] – Режим доступа до ресурсу: https://bun.sh/docs
14. Deno Docs [Електронний ресурс] – Режим доступа до ресурсу: https://docs.deno.com/
15. Express Guide [Електронний ресурс] – Режим доступа до ресурсу: https://expressjs.com/en/guide/routing.html
16. Index | Fastify [Електронний ресурс] – Режим доступа до ресурсу: https://fastify.dev/docs/v5.2.x/Guides/
17. Documentation | NestJS - A progressive Node.js framework [Електронний ресурс] – Режим доступа до ресурсу: https://docs.nestjs.com/
18. Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST) [Електронний ресурс] – Режим доступа до ресурсу: https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
19. MySQL :: MySQL Documentation [Електронний ресурс] – Режим доступа до ресурсу: https://dev.mysql.com/doc/
20. Documentation [Електронний ресурс] – Режим доступа до ресурсу: https://www.postgresql.org/docs/
21. MongoDB Documentation [Електронний ресурс] – Режим доступа до ресурсу: https://www.mongodb.com/docs/
22. Docs | Redis [Електронний ресурс] – Режим доступа до ресурсу: https://redis.io/docs/latest/
23. RFC 6455 - The WebSocket Protocol [Електронний ресурс] – Режим доступа до ресурсу: https://datatracker.ietf.org/doc/html/rfc6455

ДОДАТОК 1

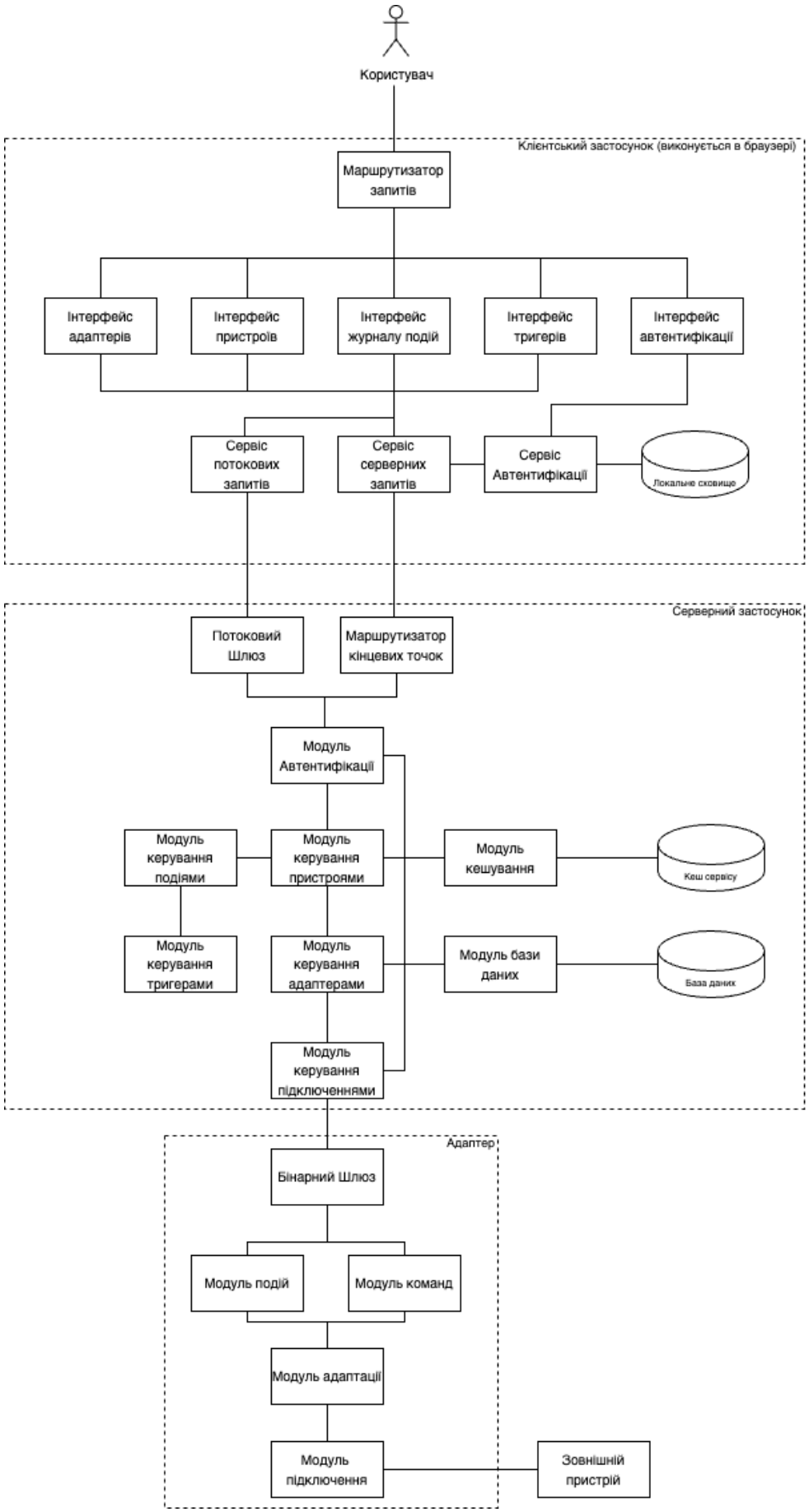
Еволюційні алгоритми глобальної пошукової оптимізації

Структурна схема системи

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2025 р



					ІАЛЦ.467200.004 Д1	
		№ докум.	Підпис	Дата		
Розробив		Землянський О.В.			**Сервіс для віддаленого моніторингу стану та керування пристроями** Структурна схема системи	Літ. / Аркуш: 1 / Аркушів: 1
Перевірив		Гордієнко Н.Ю.				
Реценз.		Соколовський В.В.				КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-13
Н. Контр.		Міщенко Л.Д.				
Затвердив						

ДОДАТОК 2

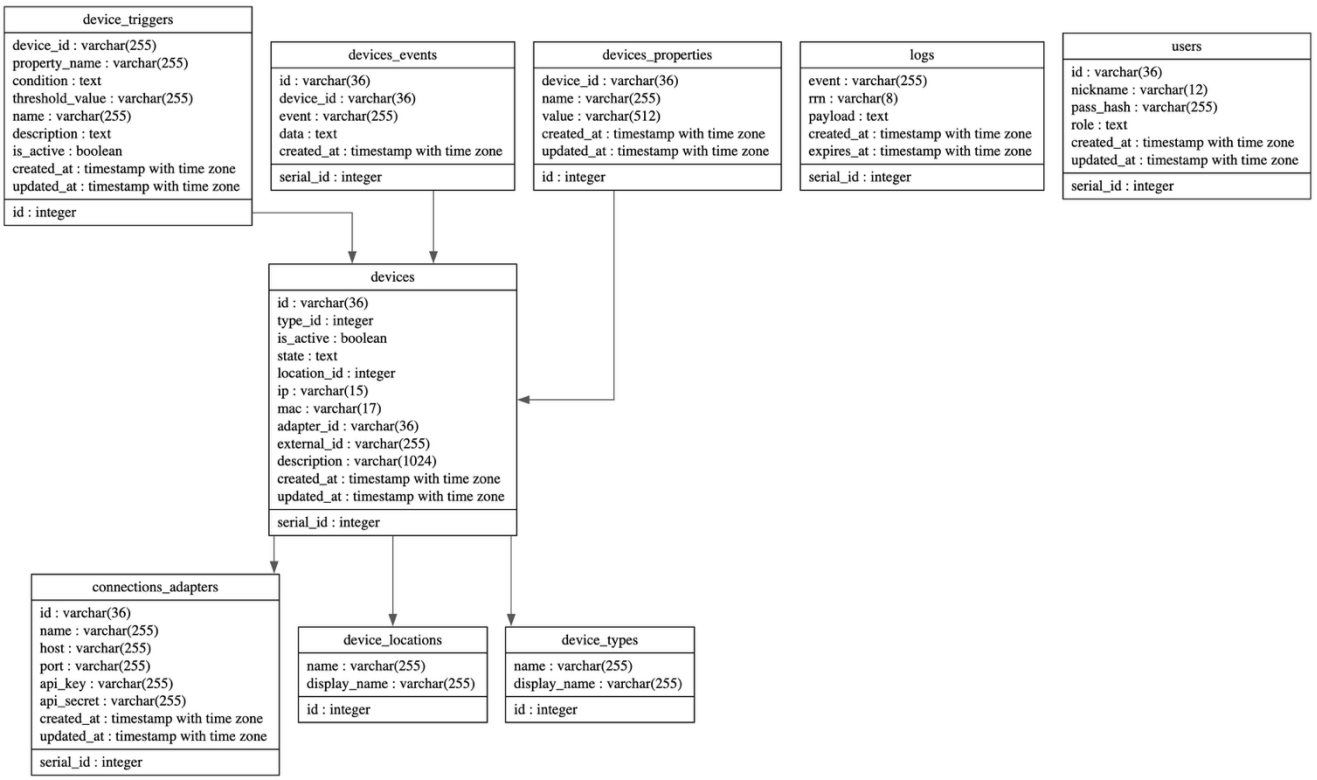
Еволюційні алгоритми глобальної пошукової оптимізації

Діаграма даних

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2025 р



					ІАЛЦ.467200.005 Д2	
		№ докум.	Підпис	Дата		
Розробив		Землянський О.В.			**Сервіс для віддаленого моніторингу стану та керування пристроями** Діаграма даних	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Гордієнко Н.Ю.				
Реценз.		Соколовський В.В.				КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-13
Н. Контр.		Міщенко Л.Д.				
Затвердив						

ДОДАТОК З

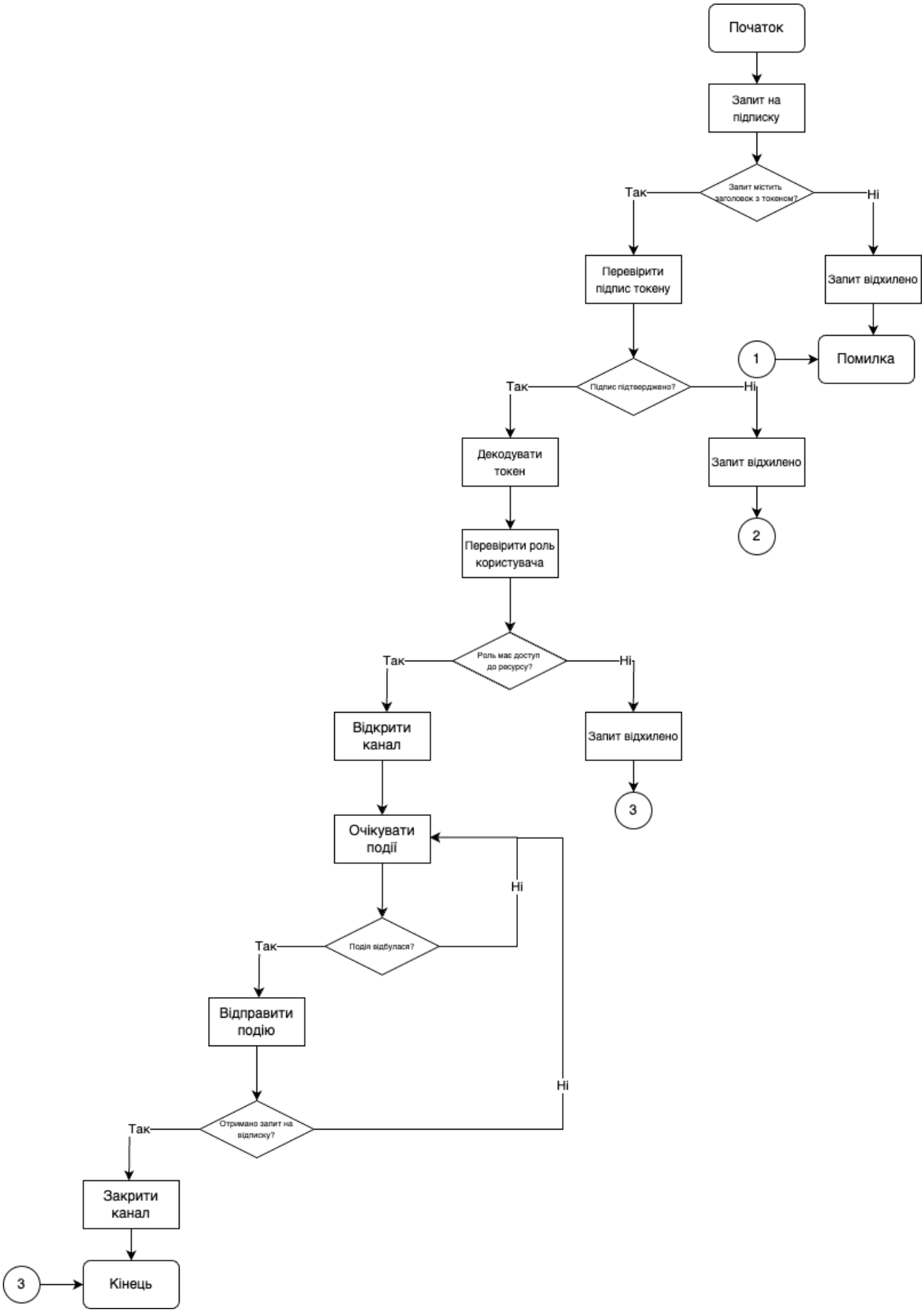
Еволюційні алгоритми глобальної пошукової оптимізації

Алгоритм дій програмного забезпечення

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2025



					ІАЛЦ.467200.006 ДЗ	
		№ докум.	Підпис	Дата		
Розробив		Землянський О.В.			**Сервіс для віддаленого моніторингу стану та керування пристроями** Алгоритм дій програмного забезпечення	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Гордієнко Н.Ю.				
Реценз.		Соколовський В.В.				КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-13
Н. Контр.		Міщенко Л.Д				
Затвердив						

ДОДАТОК 4

Еволюційні алгоритми глобальної пошукової оптимізації

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 1

Київ 2025

Серверна частина

src/database/database.service.ts

```
import { Inject, Injectable, OnModuleInit } from '@nestjs/common';
import { Knex } from 'knex';
import { DB_CONNECTION } from './constants/injection-keys';


@Injectable()
export class DatabaseService implements OnModuleInit {
 constructor(
  @Inject(DB_CONNECTION)
  private readonly db: Knex,
 ) {}

 private _migrateDb(): Promise<void> {
  return this.db.migrate.latest({
   directory: `${__dirname}/migrations`,
   loadExtensions: ['.js'],
  });
 }

 private _seedDb(): Promise<string[][]> {
  return this.db.seed.run({
   directory: `${__dirname}/seeds`,
   loadExtensions: ['.js'],
  });
 }

 async onModuleInit(): Promise<void> {
  await this._migrateDb();
 }
}
```

src/database/database.module.ts

```
import { DynamicModule } from '@nestjs/common';
import { DB_CONNECTION } from './constants/injection-keys';
import knex, { Knex } from 'knex';
import { DatabaseService } from './database.service';
import { envConfig } from '../config';


export class DatabaseModule {
 static forRoot(connection: Knex.StaticConnectionConfig): DynamicModule {
  return {
   providers: [
    {
     provide: DB_CONNECTION,
     useFactory: () => {
      return knex({
       client: 'pg',
       asyncStackTraces: true,
       compileSqlOnError: true,
       connection,
      });
     },
    },
    DatabaseService,
   ],
   exports: [DB_CONNECTION],
   module: DatabaseModule,
   global: true,
  };
 }

 static forRootFromEnv(): DynamicModule {
```

					ІАЛЦ.467200.007 Д4		
		№ докум.	Підпис	Дата			
Розробив		Землянський О.В.			**Сервіс для віддаленого моніторингу стану та керування пристроями** Текст програмного коду	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Гордієнко Н.Ю.					
Реценз.		Соколовський В.В.				КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-13	
Н. Контр.		Міщенко Л.Д.					
Затвердив							

```
    return DatabaseModule.forRoot({
      host: envConfig.database.host,
      user: envConfig.database.user,
      password: envConfig.database.password,
      port: envConfig.database.port,
      database: envConfig.database.database,
    });
  }
}
```

src/main.ts

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { envConfig } from './config';
import { ValidationPipe } from '@nestjs/common';
import { LogsService } from './logs/logs.service';
import { AppInterceptor } from './app-interceptor.service';
import { SwaggerModule, DocumentBuilder } from '@nestjs/swagger';
import { WsAdapter } from '@nestjs/platform-ws';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  app.useWebSocketAdapter(new WsAdapter(app));

  app.useGlobalPipes(new ValidationPipe());

  const logsService = app.get(LogsService);

  app.useGlobalInterceptors(new AppInterceptor(logsService));

  app.enableCors();

  const swaggerConfig = new DocumentBuilder()
    .setTitle('IoT Control Service')
    .setDescription('API for managing IoT devices')
    .setVersion('1.0')
    .addTag('devices', 'Device management endpoints')
    .addTag('device-types', 'Device type management endpoints')
    .addTag('device-properties', 'Device property management endpoints')
    .addBearerAuth()
    .build();

  const document = SwaggerModule.createDocument(app, swaggerConfig);
  SwaggerModule.setup('docs', app, document);

  await app.listen(envConfig.port);
}

void bootstrap();
```

src/auth/auth.controller.ts

```
import {
  Controller,
  Post,
  Body,
  UseGuards,
  Get,
  Request,
} from '@nestjs/common';
import {
  ApiTags,
  ApiOperation,
```

```
  ApiResponse,
  ApiBearerAuth,
} from '@nestjs/swagger';
import { AuthService } from './auth.service';
import { AuthDto } from './dto/auth.dto';
import { AuthResult } from './results/auth.result';
import { JwtAuthGuard } from './guards/jwt-auth.guard';


@ApiTags('auth')
@Controller('auth')
export class AuthController {
  constructor(private readonly authService: AuthService) {}


  @Post('signin')
  @ApiOperation({ summary: 'Sign in user' })
  @ApiResponse({
    status: 200,
    description: 'User successfully signed in',
    type: AuthResult,
  })
  @ApiResponse({ status: 401, description: 'Invalid credentials' })
  async signIn(@Body() authDto: AuthDto): Promise<AuthResult> {
    return this.authService.signIn(authDto);
  }


  @Post('signup')
  @ApiOperation({ summary: 'Sign up new user' })
  @ApiResponse({
    status: 201,
    description: 'User successfully created',
    type: AuthResult,
  })
  @ApiResponse({ status: 401, description: 'User already exists' })
  async signUp(@Body() authDto: AuthDto): Promise<AuthResult> {
    return this.authService.signUp(authDto);
  }


  @Post('refresh')
  @UseGuards(JwtAuthGuard)
  @ApiBearerAuth()
  @ApiOperation({ summary: 'Refresh access token' })
  @ApiResponse({
    status: 200,
    description: 'Token successfully refreshed',
    type: AuthResult,
  })
  @ApiResponse({ status: 401, description: 'Invalid token' })
  async refreshToken(@Request() req): Promise<AuthResult> {
    return this.authService.refreshToken(req.user.id);
  }


  @Get('profile')
  @UseGuards(JwtAuthGuard)
  @ApiBearerAuth()
  @ApiOperation({ summary: 'Get current user profile' })
  @ApiResponse({ status: 200, description: 'User profile' })
  @ApiResponse({ status: 401, description: 'Unauthorized' })
  getProfile(@Request() req) {
    return req.user;
  }
```

```
}
```

src/auth/auth.service.ts

```
import { Injectable, UnauthorizedException, Logger } from '@nestjs/common';
import { JwtService } from '@nestjs/jwt';
import { UsersService } from '../users/users.service';
import { AuthDto } from './dto/auth.dto';
import { AuthResult } from './results/auth.result';
import * as bcrypt from 'bcrypt';
import { User } from '../users/models/user.model';
import { envConfig } from '../config';
import { randomUUID } from 'node:crypto';

@Injectable()
export class AuthService {
  private readonly logger = new Logger(AuthService.name);

  constructor(
    private readonly usersService: UsersService,
    private readonly jwtService: JwtService,
  ) {}

  async validateUser(login: string, password: string): Promise<any> {
    try {
      const user = await this.usersService.findOne({
        nickname: login,
      });

      if (!user) {
        this.logger.debug(`User with login ${login} not found`);
        return null;
      }

      const passwordMatch = await bcrypt.compare(password, user.pass_hash);

      if (passwordMatch) {
        return {
          id: user.id,
          nickname: user.nickname,
          role: user.role,
        };
      }

      this.logger.debug(`Invalid password for user ${login}`);
      return null;
    } catch (error) {
      this.logger.error(`Error validating user: ${error.message}`, error.stack);
      return null;
    }
  }

  async signIn(authDto: AuthDto): Promise<AuthResult> {
    const user = await this.validateUser(authDto.login, authDto.password);

    if (!user) {
      throw new UnauthorizedException('Invalid credentials');
    }

    const payload = {
      sub: user.id,
      login: user.nickname,
```

```
      role: user.role,
      iat: Math.floor(Date.now() / 1000),
    };

    const access_token = await this.jwtService.signAsync(payload);
    const expires_in = this.getExpiresInSeconds();

    return {
      access_token,
      token_type: 'Bearer',
      expires_in,
      user: {
        id: user.id,
        login: user.nickname,
      },
    };
  }

  async signUp(authDto: AuthDto): Promise<AuthResult> {
    try {
      const existingUser = await this.usersService.findOne({
        nickname: authDto.login,
      });
      if (existingUser) {
        throw new UnauthorizedException('User already exists');
      }

      const saltRounds = 10;
      const hashedPassword = await bcrypt.hash(authDto.password, saltRounds);

      const newUser = new User();
      newUser.populate({
        id: randomUUID(),
        nickname: authDto.login,
        pass_hash: hashedPassword,
      });

      await this.usersService.createOne(newUser);
      const payload = {
        sub: newUser.id,
        login: newUser.nickname,
        role: newUser.role,
        iat: Math.floor(Date.now() / 1000),
      };

      const access_token = await this.jwtService.signAsync(payload);
      const expires_in = this.getExpiresInSeconds();

      return {
        access_token,
        token_type: 'Bearer',
        expires_in,
        user: {
          id: newUser.id,
          login: newUser.nickname,
        },
      };
    } catch (error) {
      this.logger.error(`Error creating user: ${error.message}`, error.stack);
      if (error instanceof UnauthorizedException) {
        throw error;
      }
```

```
      throw new UnauthorizedException('Failed to create user');
    }
  }

  async refreshToken(userId: string): Promise<AuthResult> {
    try {
      const user = await this.usersService.findOneById(userId);

      if (!user) {
        this.logger.warn(`User not found for refresh token: ${userId}`);
        throw new UnauthorizedException('User not found');
      }

      const payload = {
        sub: user.id,
        login: user.nickname,
        role: user.role,
        iat: Math.floor(Date.now() / 1000),
      };

      const access_token = await this.jwtService.signAsync(payload);
      const expires_in = this.getExpiresInSeconds();

      return {
        access_token,
        token_type: 'Bearer',
        expires_in,
        user: {
          id: user.id,
          login: user.nickname,
        },
      };
    } catch (error) {
      this.logger.error(
        `Error refreshing token: ${error.message}`,
        error.stack,
      );
      if (error instanceof UnauthorizedException) {
        throw error;
      }
      throw new UnauthorizedException('Failed to refresh token');
    }
  }

  async verifyToken(token: string): Promise<any> {
    try {
      const payload = await this.jwtService.verifyAsync(token);
      return payload;
    } catch (error) {
      this.logger.error(`Error verifying token: ${error.message}`, error.stack);
      throw new UnauthorizedException('Invalid token');
    }
  }

  private getExpiresInSeconds(): number {
    const expiresIn = envConfig.jwt.expiresIn;

    if (!isNaN(Number(expiresIn))) {
      return Number(expiresIn);
    }

    const match = expiresIn.match(/^(\d+)([smhdw])$/);
```

```
    if (!match) {
      this.logger.warn(
        `Invalid JWT expiresIn format: ${expiresIn}, using default 3600 seconds`,
      );
      return 3600;
    }

    const value = parseInt(match[1], 10);
    const unit = match[2];
    switch (unit) {
      case 's':
        return value;
      case 'm':
        return value * 60;
      case 'h':
        return value * 60 * 60;
      case 'd':
        return value * 24 * 60 * 60;
      case 'w':
        return value * 7 * 24 * 60 * 60;
      default:
        return 3600;
    }
  }
}
```

src/auth/decorators/required-role.decorator.ts

```
import { UserRoles } from '../../users/enums/user-roles.enum';
import { SetMetadata } from '@nestjs/common';

export const ROLES_KEY = Symbol('ROLES');
export function RequiredRole(role: UserRoles): MethodDecorator {
  return SetMetadata(ROLES_KEY, role);
}
```

src/auth/strategies/jwt.strategy.ts

```
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { PassportStrategy } from '@nestjs/passport';
import { ExtractJwt, Strategy } from 'passport-jwt';
import { UsersService } from '../../users/users.service';
import { envConfig } from '../../config';

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor(private readonly usersService: UsersService) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      secretOrKey: envConfig.jwt.secret,
    });
  }

  async validate(payload: any) {
    try {
      const user = await this.usersService.findOne({
        id: payload.sub,
      });
      return { id: user.id, nickname: user.nickname, role: user.role };
    } catch (err) {
      throw new UnauthorizedException('Invalid token');
    }
```

```
  }
}
```

src/auth/results/auth.result.ts

```
import { ApiProperty } from '@nestjs/swagger';

export class AuthResult {
  @ApiProperty({
    description: 'JWT access token',
    example:
'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...',
  })
  access_token: string;

  @ApiProperty({
    description: 'Token type',
    example: 'Bearer',
  })
  token_type: string;

  @ApiProperty({
    description: 'Token expiration time in seconds',
    example: 3600,
  })
  expires_in: number;

  @ApiProperty({
    description: 'User information',
  })
  user: {
    id: string;
    login: string;
  };
}
```

src/auth/auth.module.ts

```
import { Module } from '@nestjs/common';
import { JwtModule } from '@nestjs/jwt';
import { PassportModule } from '@nestjs/passport';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { JwtStrategy } from './strategies/jwt.strategy';
import { UsersModule } from '../users/users.module';
import { envConfig } from '../config';

@Module({
  imports: [
    UsersModule,
    PassportModule,
    JwtModule.register({
      secret: envConfig.jwt.secret,
      signOptions: { expiresIn: envConfig.jwt.expiresIn },
    }),
  ],
  controllers: [AuthController],
  providers: [AuthService, JwtStrategy],
  exports: [AuthService],
})
export class AuthModule {}
```

src/auth/guards/roles.guard.ts

```
import { CanActivate, ExecutionContext, Injectable }
from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { UserRoles } from '../../users/enums/user-
roles.enum';
import { ROLES_KEY } from '../decorators/required-
role.decorator';
```

```
@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private readonly reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const request = context.switchToHttp().getRequest();

    const role = this.reflector.getAllAndOverride<UserRoles>(ROLES_KEY, [
      context.getClass(),
      context.getHandler(),
    ]);

    if (!role) {
      return true;
    }
    const user = request.user || request.meta?.user;

    if (user.role === UserRoles.admin) {
      return true;
    }

    return user?.role === role;
  }
}
```

src/auth/guards/jwt-auth.guard.ts

```
import { Injectable } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {}
```

src/triggers/triggers.controller.ts

```
import {
  Controller,
  Get,
  Post,
  Body,
  Patch,
  Param,
  Delete,
  UseGuards,
  Query,
} from '@nestjs/common';
import { TriggersService } from './triggers.service';
import { CreateDeviceTriggerDto } from './dto/create-device-trigger.dto';
import { UpdateDeviceTriggerDto } from './dto/update-device-trigger.dto';
import { JwtAuthGuard } from '../auth/guards/jwt-auth.guard';
import { RolesGuard } from '../auth/guards/roles.guard';
import { RequiredRole } from '../auth/decorators/required-role.decorator';
import { UserRoles } from '../users/enums/user-roles.enum';
import { ApiOperation, ApiResponse, ApiTags } from '@nestjs/swagger';
import { DeviceTrigger } from './models/device-trigger.model';

@ApiTags('triggers')
@Controller('triggers')
@UseGuards(JwtAuthGuard, RolesGuard)
export class TriggersController {
  constructor(private readonly triggersService: TriggersService) {}

  @Post()
```

```
  @RequiredRole(UserRoles.admin)
  @ApiOperation({ summary: 'Create a new device trigger' })
  @ApiResponse({
    status: 201,
    description: 'The trigger has been successfully created.',
    type: DeviceTrigger,
  })
  create(@Body() createTriggerDto: CreateDeviceTriggerDto) {
    return this.triggersService.create(createTriggerDto);
  }

  @Get()
  @RequiredRole(UserRoles.user)
  @ApiOperation({ summary: 'Get all triggers or filter by device ID' })
  @ApiResponse({
    status: 200,
    description: 'Return all triggers or filtered by device ID',
    type: [DeviceTrigger],
  })
  findAll(@Query('deviceId') deviceId?: string) {
    if (deviceId) {
      return this.triggersService.findByDeviceId(deviceId);
    }
    return this.triggersService.findAll();
  }

  @Get(':id')
  @RequiredRole(UserRoles.user)
  @ApiOperation({ summary: 'Get a trigger by ID' })
  @ApiResponse({
    status: 200,
    description: 'Return the trigger with the specified ID',
    type: DeviceTrigger,
  })
  findOne(@Param('id') id: string) {
    return this.triggersService.findOneById(+id);
  }

  @Patch(':id')
  @RequiredRole(UserRoles.admin)
  @ApiOperation({ summary: 'Update a trigger' })
  @ApiResponse({
    status: 200,
    description: 'The trigger has been successfully updated.',
    type: DeviceTrigger,
  })
  update(
    @Param('id') id: string,
    @Body() updateTriggerDto: UpdateDeviceTriggerDto,
  ) {
    return this.triggersService.update(+id, updateTriggerDto);
  }

  @Delete(':id')
  @RequiredRole(UserRoles.admin)
  @ApiOperation({ summary: 'Delete a trigger' })
  @ApiResponse({
    status: 200,
    description: 'The trigger has been successfully deleted.',
  })
  remove(@Param('id') id: string) {
```

```
    return this.triggersService.remove(+id);
  }
}
```

src/triggers/models/device-trigger.model.ts

```
import { ApiProperty } from '@nestjs/swagger';
import { DatabaseTableModel } from '../../common/models/database-table.model';

export enum TriggerCondition {
  GREATER_THAN = 'greater_than',
  LESS_THAN = 'less_than',
}

export class DeviceTrigger extends DatabaseTableModel {
  @ApiProperty({
    description: 'Unique identifier of the device trigger',
    example: 1,
    type: 'number',
  })
  id: number = undefined;

  @ApiProperty({
    description: 'ID of the device this trigger is for',
    example: 'thermostat-001',
    type: 'string',
  })
  device_id: string = undefined;

  @ApiProperty({
    description: 'Name of the property to monitor',
    example: 'temperature',
    type: 'string',
  })
  property_name: string = undefined;

  @ApiProperty({
    description: 'Condition for the trigger',
    example: 'greater_than',
    enum: TriggerCondition,
  })
  condition: TriggerCondition = undefined;

  @ApiProperty({
    description: 'Threshold value for the trigger condition',
    example: '30',
    type: 'string',
  })
  threshold_value: string = undefined;

  @ApiProperty({
    description: 'Name of the trigger',
    example: 'High Temperature Alert',
    type: 'string',
  })
  name: string = undefined;

  @ApiProperty({
    description: 'Description of the trigger',
    example: 'Alert when temperature exceeds 30 degrees',
    type: 'string',
    required: false,
  })
  description?: string = undefined;

  @ApiProperty({
```

```
    description: 'Whether the trigger is active',
    example: true,
    type: 'boolean',
  })
  is_active: boolean = undefined;

  @ApiProperty({
    description: 'Creation timestamp',
    example: '2024-05-12T12:00:00Z',
    type: 'string',
    format: 'date-time',
  })
  created_at: string = undefined;

  @ApiProperty({
    description: 'Last update timestamp',
    example: '2024-05-12T12:00:00Z',
    type: 'string',
    format: 'date-time',
  })
  updated_at: string = undefined;
}
```

src/triggers/triggers.service.ts

```
import { Inject, Injectable, NotFoundException } from '@nestjs/common';
import { Knex } from 'knex';
import { BaseCrudService } from '../common/services/base-crud.service';
import { DeviceTrigger } from './models/device-trigger.model';
import { CacheService } from '../cache/cache.service';
import { DB_CONNECTION } from '../database/constants/injection-keys';
import { CACHE_SERVICE } from '../cache/constants/injection-keys';
import { CreateDeviceTriggerDto } from './dto/create-device-trigger.dto';
import { UpdateDeviceTriggerDto } from './dto/update-device-trigger.dto';
import { DevicePropertiesService } from '../devices/device-properties.service';
import { EventEmitter2 } from '@nestjs/event-emitter';

@Injectable()
export class TriggersService extends BaseCrudService<DeviceTrigger> {
  constructor(
    @Inject(DB_CONNECTION)
    protected readonly db: Knex,
    @Inject(CACHE_SERVICE)
    protected readonly cacheManager: CacheService,
    private readonly devicePropertiesService: DevicePropertiesService,
    private readonly eventEmitter: EventEmitter2,
  ) {
    super(db, cacheManager, {
      table: 'device_triggers',
      pk: ['id'],
      class: DeviceTrigger,
    });
  }

  async create(
    createTriggerDto: CreateDeviceTriggerDto,
  ): Promise<DeviceTrigger> {
    await this.devicePropertiesService.findByDeviceIdAndName(
      createTriggerDto.device_id,
      createTriggerDto.property_name,
    );
```

```
    const trigger = new DeviceTrigger();
    Object.assign(trigger, createTriggerDto);
    await this.createOne(trigger);
    return trigger;
  }

  async findAll(): Promise<DeviceTrigger[]> {
    return this.findMany({});
  }

  async findByDeviceId(deviceId: string): Promise<DeviceTrigger[]> {
    return this.findMany({ device_id: deviceId });
  }

  async findOneById(id: number): Promise<DeviceTrigger> {
    const trigger = await this.findOne({ id });
    if (!trigger) {
      throw new NotFoundException(`Trigger with ID ${id} not found`);
    }
    return trigger;
  }

  async update(
    id: number,
    updateTriggerDto: UpdateDeviceTriggerDto,
  ): Promise<DeviceTrigger> {
    const trigger = await this.findOneById(id);

    if (
      updateTriggerDto.property_name &&
      updateTriggerDto.property_name !== trigger.property_name
    ) {
      await this.devicePropertiesService.findByDeviceIdAndName(
        trigger.device_id,
        updateTriggerDto.property_name,
      );
    }

    Object.assign(trigger, updateTriggerDto);
    await this.updateOne({ id, ...updateTriggerDto });
    return trigger;
  }

  async remove(id: number): Promise<void> {
    await this.findOneById(id);
    await this.deleteOne({ id });
  }

  async checkTrigger(
    deviceId: string,
    propertyName: string,
    propertyValue: string,
  ): Promise<boolean> {
    const triggers = await this.findMany({
      device_id: deviceId,
      property_name: propertyName,
      is_active: true,
    });

    if (triggers.length === 0) {
      return false;
    }
```

```
    let triggered = false;
    const numericValue = parseFloat(propertyValue);

    for (const trigger of triggers) {
      const thresholdValue = parseFloat(trigger.threshold_value);

      if (isNaN(numericValue) || isNaN(thresholdValue)) {
        continue;
      }

      let isTriggered = false;
      if (trigger.condition === 'greater_than' && numericValue > thresholdValue) {
        isTriggered = true;
      } else if (trigger.condition === 'less_than' && numericValue < thresholdValue) {
        isTriggered = true;
      }

      if (isTriggered) {
        triggered = true;
        this.eventEmitter.emit('triggers.triggered', {
          trigger,
          currentValue: propertyValue,
        });
      }
    }

    return triggered;
  }
}
```

src/triggers/triggers.gateway.ts

```
import {
  WebSocketGateway,
  OnGatewayConnection,
  OnGatewayDisconnect,
  SubscribeMessage,
} from '@nestjs/websockets';
import * as WebSocket from 'ws';
import { randomUUID } from 'node:crypto';
import { filter, fromEvent, map, Observable } from 'rxjs';
import { EventEmitter2 } from '@nestjs/event-emitter';
import { DeviceTrigger } from './models/device-trigger.model';

@WebSocketGateway({
  path: 'triggers/ws',
})
export class TriggersGateway
  implements OnGatewayConnection, OnGatewayDisconnect
{
  private readonly connections: Map<string, WebSocket> = new Map();

  constructor(private readonly eventEmitter: EventEmitter2) {}

  handleConnection(client: WebSocket) {
    const connectionId = randomUUID();
    client['clientId'] = connectionId;
    this.connections.set(connectionId, client);
    console.log(`Triggers client connected: ${connectionId}`);
  }
```

```
  handleDisconnect(client: WebSocket) {
    const connectionId = client['clientId'];
    if (connectionId) {
      this.connections.delete(connectionId);
      console.log(`Triggers client disconnected: ${connectionId}`);
    } else {
      console.warn('Triggers client disconnected without a valid ID');
    }
  }

  @SubscribeMessage('subscribe')
  handleEvent(
    _: WebSocket,
    data: { subscriptionId: string; deviceId?: string },
  ): Observable<any> {
    return fromEvent(this.eventEmitter, 'triggers.triggered').pipe(
      filter((event: { trigger: DeviceTrigger; currentValue: string }) => {
        const { deviceId } = data;

        if (deviceId && event.trigger.device_id !== deviceId) {
          return false;
        }

        return true;
      }),
      map((event: { trigger: DeviceTrigger; currentValue: string }) => {
        return {
          ...event,
          subscriptionId: data.subscriptionId,
          timestamp: new Date().toISOString(),
          message: `${event.trigger.name}: ${event.trigger.property_name} is ${
            event.trigger.condition === 'greater_than' ? 'above' : 'below'
          } threshold of ${event.trigger.threshold_value} (current value: ${
            event.currentValue
          })`,
        };
      }),
    );
  }
}
```

src/triggers/triggers.module.ts

```
import { Module } from '@nestjs/common';
import { TriggersService } from './triggers.service';
import { TriggersController } from './triggers.controller';
import { TriggersGateway } from './triggers.gateway';
import { DevicesModule } from '../devices/devices.module';

@Module({
  imports: [DevicesModule],
  controllers: [TriggersController],
  providers: [TriggersService, TriggersGateway],
  exports: [TriggersService],
})
export class TriggersModule {}
```

src/app.service.ts

```
import { Injectable } from '@nestjs/common';

@Injectable()
export class AppService {
  getHello(): string {
```

```
    return 'Hello World!';
  }
}
```

src/api-keys/api-keys.module.ts

```
import { Module } from '@nestjs/common';

@Module({})
export class ApiKeysModule {}
```

src/app.module.ts

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { DatabaseModule } from './database/database.module';
import { LogsModule } from './logs/logs.module';
import { CacheModule } from './cache/cache.module';
import { ScheduleModule } from '@nestjs/schedule';
import { DevicesModule } from './devices/devices.module';
import { UsersModule } from './users/users.module';
import { AuthModule } from './auth/auth.module';
import { NotificationsModule } from './notifications/notifications.module';
import { ApiKeysModule } from './api-keys/api-keys.module';
import { ConnectionsModule } from './connections/connections.module';
import { EventEmitterModule } from '@nestjs/event-emitter';
import { DeviceEventsProcessor } from './devices/processors/device-events.processor';
import { TriggersModule } from './triggers/triggers.module';

@Module({
  imports: [
    ScheduleModule.forRoot(),
    EventEmitterModule.forRoot({
      wildcard: true,
      global: true,
      ignoreErrors: true,
      maxListeners: 100,
      verboseMemoryLeak: true,
    }),

    DatabaseModule.forRootFromEnv(),
    CacheModule.forRootFromEnv(),
    LogsModule,
    CacheModule,
    DevicesModule,
    UsersModule,
    AuthModule,
    NotificationsModule,
    ApiKeysModule,
    ConnectionsModule,
    TriggersModule,
  ],
  controllers: [AppController],
  providers: [AppService, DeviceEventsProcessor],
})
export class AppModule {}
```

src/connections/models/connection-adapter.model.ts

```
import { ApiProperty } from '@nestjs/swagger';
import { DatabaseTableModel } from '../../common/models/database-table.model';
import { randomUUID } from 'node:crypto';
```

```
export class ConnectionAdapter extends DatabaseTableModel {
  @ApiProperty({
    description: 'Unique identifier of the connection adapter',
    example: randomUUID(),
    type: 'string',
    format: 'uuid',
  })
  id: string = undefined;

  @ApiProperty({
    description: 'Name of the connection adapter',
    example: 'zigbee_adapter',
  })
  name: string = undefined;

  @ApiProperty({
    description: 'URL of the connection adapter',
    example: 'localhost',
    required: false,
  })
  host: string = undefined;

  @ApiProperty({
    description: 'Port of the connection adapter',
    example: '8080',
    required: false,
  })
  port: string = undefined;

  @ApiProperty({
    description: 'API key for the connection adapter',
    example: 'your-api-key',
    required: false,
  })
  api_key: string = undefined;

  @ApiProperty({
    description: 'API secret for the connection adapter',
    example: 'your-api-secret',
    required: false,
  })
  api_secret: string = undefined;

  @ApiProperty({
    description: 'Creation timestamp',
    example: '2024-03-20T12:00:00Z',
    type: 'string',
    format: 'date-time',
  })
  created_at: string = undefined;

  @ApiProperty({
    description: 'Last update timestamp',
    example: '2024-03-20T12:00:00Z',
    type: 'string',
    format: 'date-time',
  })
  updated_at: string = undefined;
}
```

src/connections/connections-adapters.service.ts

```
import { Injectable, Inject, OnModuleInit } from '@nestjs/common';
import { Knex } from 'knex';
import { BaseCrudService } from '../common/services/base-crud.service';
```

```
import { ConnectionAdapter } from './models/connection-adapter.model';
import { CacheService } from '../cache/cache.service';
import { DB_CONNECTION } from '../database/constants/injection-keys';
import { CACHE_SERVICE } from '../cache/constants/injection-keys';
import { GrpcClientFactory } from './factories/grpc-client.factory';
import { from, merge, Observable } from 'rxjs';
import { plainToInstance } from 'class-transformer';
import { randomUUID } from 'crypto';
import { CreateConnectionAdapterDto } from './dto/create-connection-adapter.dto';

import {
  StreamCommandType,
  SubscribeQuery,
  SubscriptionStream,
} from './proto/iot';

@Injectable()
export class ConnectionsAdaptersService
  extends BaseCrudService<ConnectionAdapter>
  implements OnModuleInit
{
  private _mainThread$: Observable<SubscriptionStream>;

  constructor(
    @Inject(DB_CONNECTION)
    protected readonly db: Knex,
    @Inject(CACHE_SERVICE)
    protected readonly cacheManager: CacheService,
    private readonly grpcClientFactory: GrpcClientFactory,
  ) {
    super(db, cacheManager, {
      table: 'connections_adapters',
      pk: ['id'],
      class: ConnectionAdapter,
    });
  }

  async onModuleInit() {
    await this._initSubscriptions();
  }

  private async _initSubscriptions() {
    const adapters = await this.findAll();

    const streams = adapters.map((adapter) => {
      const client = this.grpcClientFactory.createClient(adapter);

      const query: SubscribeQuery = {
        subId: `sub-${adapter.id}-${Date.now()}`,
        commandType: StreamCommandType.SUB_DEVICE_EVENTS,
        eventTypes: null,
      };

      return client.streamEvents(from([query]));
    });

    this._mainThread$ = merge(...streams);
  }

  get eventsStream$(): Observable<SubscriptionStream> {
    return this._mainThread$;
```

```
  }


  async create(args: CreateConnectionAdapterDto): Promise<ConnectionAdapter> {
    const adapter = plainToInstance(ConnectionAdapter, args);
    adapter.id = randomUUID();
    await this.createOne(adapter);
    return adapter;
  }
}
```

src/connections/connections.module.ts

```
import { Module } from '@nestjs/common';
import { ConnectionsAdaptersService } from './connections-adapters.service';
import { GrpcClientFactory } from './factories/grpc-client.factory';
import { ConnectionsAdaptersController } from './connections-adapters.controller';

@Module({
  imports: [],
  controllers: [ConnectionsAdaptersController],
  providers: [ConnectionsAdaptersService, GrpcClientFactory],
  exports: [ConnectionsAdaptersService, GrpcClientFactory],
})
export class ConnectionsModule {}
```

src/connections/factories/grpc-client.factory.ts

```
import { Injectable } from '@nestjs/common';
import { Transport } from '@nestjs/microservices';
import { ConnectionAdapter } from '../models/connection-adapter.model';
import { IOTConnectionsServiceClient } from '../proto/iot';
import { ClientOptions } from '@nestjs/microservices';
import { ClientProxyFactory } from '@nestjs/microservices';
import { ClientGrpc } from '@nestjs/microservices';
import { join } from 'path';

@Injectable()
export class GrpcClientFactory {
  private clients: Map<string, IOTConnectionsServiceClient> = new Map();

  createClient(adapter: ConnectionAdapter): IOTConnectionsServiceClient {
    const clientKey = adapter.id;

    if (this.clients.has(clientKey)) {
      return this.clients.get(clientKey);
    }

    const options: ClientOptions = {
      transport: Transport.GRPC,
      options: {
        url: `${adapter.host}:${adapter.port}`,
        package: 'iot',
        protoPath: join(__dirname, '../proto/iot.proto'),
        credentials:
          adapter.api_key && adapter.api_secret
            ? {
                apiKey: adapter.api_key,
                apiSecret: adapter.api_secret,
              }
            : undefined,
      },
```

```
    };

    const client = ClientProxyFactory.create(options) as unknown as ClientGrpc;
    const service = client.getService<IOTConnectionsServiceClient>(
      'IOTConnectionsService',
    );
    this.clients.set(clientKey, service);
    return service;
  }

  getService(adapter: ConnectionAdapter): IOTConnectionsServiceClient {
    return this.createClient(adapter);
  }

  removeClient(adapterId: string): void {
    this.clients.delete(adapterId);
  }
}
```

src/connections/connections-adapters.controller.ts

```
import {
  Controller,
  Get,
  Post,
  Body,
  Patch,
  Param,
  Delete,
  UseGuards,
  ParseUUIDPipe,
} from '@nestjs/common';
import {
  ApiTags,
  ApiOperation,
  ApiResponse,
  ApiParam,
  ApiBearerAuth,
} from '@nestjs/swagger';
import { ConnectionsAdaptersService } from './connections-adapters.service';
import { CreateConnectionAdapterDto } from './dto/create-connection-adapter.dto';
import { UpdateConnectionAdapterDto } from './dto/update-connection-adapter.dto';
import { ConnectionAdapter } from './models/connection-adapter.model';
import { JwtAuthGuard } from '../auth/guards/jwt-auth.guard';
import { RolesGuard } from '../auth/guards/roles.guard';
import { RequiredRole } from '../auth/decorators/required-role.decorator';
import { UserRoles } from '../users/enums/user-roles.enum';

@ApiTags('connections-adapters')
@ApiBearerAuth()
@Controller('connections-adapters')
@UseGuards(JwtAuthGuard, RolesGuard)
export class ConnectionsAdaptersController {
  constructor(
    private readonly connectionsAdaptersService: ConnectionsAdaptersService,
  ) {}

  @Post()
  @RequiredRole(UserRoles.admin)
  @ApiOperation({ summary: 'Create a new connection adapter' })
  @ApiResponse({
```

```
    status: 201,
    description: 'Connection adapter created',
    type: ConnectionAdapter,
  })
  create(@Body() createConnectionAdapterDto: CreateConnectionAdapterDto) {
    return this.connectionsAdaptersService.create(createConnectionAdapterDto);
  }

  @Get()
  @RequiredRole(UserRoles.user)
  @ApiOperation({ summary: 'Get all connection adapters' })
  @ApiResponse({
    status: 200,
    description: 'Return all connection adapters',
    type: [ConnectionAdapter],
  })
  findAll() {
    return this.connectionsAdaptersService.findAllThruCache();
  }

  @Get(':id')
  @RequiredRole(UserRoles.user)
  @ApiOperation({ summary: 'Get a connection adapter by ID' })
  @ApiParam({ name: 'id', description: 'Connection adapter ID' })
  @ApiResponse({
    status: 200,
    description: 'Return the connection adapter',
    type: ConnectionAdapter,
  })
  @ApiResponse({ status: 404, description: 'Connection adapter not found' })
  findOne(@Param('id', ParseUUIDPipe) id: string) {
    return this.connectionsAdaptersService.findOneThruCache({ id });
  }

  @Patch(':id')
  @RequiredRole(UserRoles.admin)
  @ApiOperation({ summary: 'Update a connection adapter' })
  @ApiParam({ name: 'id', description: 'Connection adapter ID' })
  @ApiResponse({
    status: 200,
    description: 'Connection adapter updated',
    type: ConnectionAdapter,
  })
  @ApiResponse({ status: 404, description: 'Connection adapter not found' })
  async update(
    @Param('id', ParseUUIDPipe) id: string,
    @Body() updateConnectionAdapterDto: UpdateConnectionAdapterDto,
  ) {
    await this.connectionsAdaptersService.updateOne({
      id,
      ...updateConnectionAdapterDto,
    });
    return this.connectionsAdaptersService.findOneThruCache({ id });
  }

  @Delete(':id')
  @RequiredRole(UserRoles.admin)
```

```
  @ApiOperation({ summary: 'Delete a connection adapter' })
  @ApiParam({ name: 'id', description: 'Connection adapter ID' })
  @ApiResponse({
    status: 200,
    description: 'Connection adapter successfully deleted',
  })
  @ApiResponse({ status: 404, description: 'Connection adapter not found' })
  remove(@Param('id', ParseUUIDPipe) id: string) {
    return this.connectionsAdaptersService.deleteOne({ id });
  }
}
```

src/common/models/database-table.model.ts

```
import { pick } from '../helpers';

export class DatabaseTableModel {
  populate(
    fields: { [key: string]: any },
    options?: { include?: string[]; exclude?: string[] },
  ) {
    let keys = Object.keys(this);

    if (options?.include) {
      keys = keys.filter((key) => options.include?.includes(key));
    }

    if (options?.exclude) {
      keys = keys.filter((key) => !options.exclude?.includes(key));
    }

    Object.assign(this, pick(fields, keys));
  }
}
```

src/common/errors/app.error.ts

```
import { HttpStatus } from '@nestjs/common';

export interface TriggerErrorArgs {
  message: string;
  statusCode?: HttpStatus;
  userFriendlyMessage?: string;
  class?: string;
  method?: string;
  ctx?: Record<string, unknown>;
}

export class AppError extends Error {
  userFriendlyMessage?: string;
  class?: string;
  method?: string;
  ctx?: Record<string, unknown>;
  statusCode?: HttpStatus = HttpStatus.INTERNAL_SERVER_ERROR;

  constructor(args: TriggerErrorArgs) {
    super(args.message);
    Object.assign(this, args);
  }
}
```

src/common/helpers/md5.ts

```
import { createHash } from 'crypto';
```

```
export function md5(thing: any) {
  return createHash('md5').update(JSON.stringify(thing)).digest('hex');
}
```

src/common/helpers/wait-for.ts

```
export function waitFor(n: number): Promise<void> {
  return new Promise((resolve) => {
    setTimeout(resolve, n);
  });
}
```

src/common/helpers/pick.ts

```
export function pick<T, K extends keyof T>(obj: T, keys: K[]): Pick<T, K> {
  return keys.reduce(
    (acc, key) => {
      acc[key] = obj[key];
      return acc;
    },
    {} as Pick<T, K>,
  );
}
```

src/common/helpers/index.ts

```
import { pick } from './pick';
import { waitFor } from './wait-for';

export { pick, waitFor };
```

src/common/helpers/db-iso-date.ts

```
import * as dayjs from 'dayjs';

export function dbISODatetime(args?: dayjs.ConfigType) {
  return dayjs(args).toISOString();
}
```

src/common/helpers/omit.ts

```
export function omit<T, K extends keyof T>(obj: T, keys: K[]): Omit<T, K> {
  return Object.fromEntries(
    Object.entries(obj).filter(([key]) => !keys.includes(key as K)),
  ) as Omit<T, K>;
}
```

src/common/services/base-crud.service.ts

```
import { ClassConstructor, plainToInstance } from 'class-transformer';
import { Knex } from 'knex';
import { CacheService } from '../../cache/cache.service';
import ColumnInfo = Knex.ColumnInfo;
import { md5 } from '../helpers/md5';
import { AppError } from '../errors/app.error';

export class BaseCrudService<Model> {
  constructor(
    protected db: Knex,
    protected cacheManager: CacheService,
    protected CRUD: {
      table: string;
      pk: string[];
      class: ClassConstructor<Model>;
    },
  ) {}

  get pk(): string[] {
```

```
    return this.CRUD.pk;
  }

  get tableName(): string {
    return this.CRUD.table;
  }

  startTransaction(): Promise<Knex.Transaction> {
    return this.db.transaction();
  }

  queryBuilder(): Knex.QueryBuilder<Model> {
    return this.db<Model>(this.CRUD.table);
  }

  async createOne(record: Model, trx?: Knex.Transaction): Promise<number[]> {
    const query = this.queryBuilder();
    if (trx) {
      query.transacting(trx);
    }
    const createResult = await query.insert(record as any);
    await this.flushCrudCache();
    return createResult;
  }

  async listFields(): Promise<{ [column: string]: ColumnInfo }> {
    return this.queryBuilder().columnInfo();
  }

  async createMany(
    records: Model[],
    trx?: Knex.Transaction,
  ): Promise<number[]> {
    if (records.length === 0) return [];

    const query = this.queryBuilder();
    if (trx) {
      query.transacting(trx);
    }

    const createResults = await query.insert(records as any);
    await this.flushCrudCache();
    return createResults;
  }

  async deleteOne(
    record: Partial<Model>,
    trx?: Knex.Transaction,
  ): Promise<number> {
    const query = this.queryBuilder();
    if (trx) {
      query.transacting(trx);
    }

    this.validatePrimaryKeys(record, 'deleteOne');
    this.CRUD.pk.forEach((key) => {
      query.where(key, record[key]);
    });

    const affectedRows = await query.delete();
    await this.flushCrudCache();
    return affectedRows;
  }

  async updateOne(
```

```
    record: Partial<Model>,
    trx?: Knex.Transaction,
  ): Promise<number> {
    const query = this.queryBuilder();
    if (trx) {
      query.transacting(trx);
    }

    this.validatePrimaryKeys(record, 'updateOne');
    this.CRUD.pk.forEach((key) => {
      query.where(key, record[key]);
    });

    const affectedRows = await query.update(record as any);
    await this.flushCrudCache();
    return affectedRows;
  }

  async findAll(trx?: Knex.Transaction): Promise<Model[]> {
    const query = this.queryBuilder();
    if (trx) query.transacting(trx);
    const results = (await query) as Model[];
    return plainToInstance(this.CRUD.class, results);
  }

  async findMany(
    args: Partial<Model>,
    trx?: Knex.Transaction,
  ): Promise<Model[]> {
    const query = this.queryBuilder();
    if (trx) query.transacting(trx);
    const results = (await query.where(args)) as Model[];
    return plainToInstance(this.CRUD.class, results);
  }

  async findOne(
    args: Partial<Model>,
    trx?: Knex.Transaction,
  ): Promise<Model | undefined> {
    const query = this.queryBuilder();
    if (trx) query.transacting(trx);
    const result = await query.where(args).first();
    return result ? plainToInstance(this.CRUD.class, result) : undefined;
  }

  async findOneOrFail(
    args: Partial<Model>,
    trx?: Knex.Transaction,
  ): Promise<Model> {
    const record = await this.findOne(args, trx);
    if (!record) {
      throw new AppError({
        message: 'FIND_ONE_FAILED',
        class: BaseCrudService.name,
        method: BaseCrudService.prototype.findOneOrFail.name,
        ctx: {
          table: this.CRUD.table,
          args,
        },
      });
    }
    return record;
  }
```

```
  private getCacheKey(type: 'all' | 'one', args?: Partial<Model>): string {
    const baseKey = `crud:${this.CRUD.table}:v1`;
    if (type === 'all') {
      return `${baseKey}:all`;
    }
    return `${baseKey}:one:${md5(JSON.stringify(args)).slice(0, 8)}`;
  }

  async findAllThruCache(ttl = 60): Promise<Model[]> {
    return await this.cacheManager.wrap(
      this.getCacheKey('all'),
      () => this.findAll(),
      ttl,
    );
  }

  async findOneThruCache(
    args: Partial<Model>,
    ttl = 60,
  ): Promise<Model | undefined> {
    return await this.cacheManager.wrap(
      this.getCacheKey('one', args),
      () => this.findOne(args),
      ttl,
    );
  }

  async flushCrudCache(): Promise<void> {
    try {
      await this.cacheManager.flashKeysByPattern(
        `crud:${this.CRUD.table}:v1:*`,
      );
    } catch (e) {
      console.error({
        message: 'Flush cache error ' + e.message,
        table: this.CRUD.table,
      });
    }
  }

  private validatePrimaryKeys(
    record: Partial<Model>,
    methodName: string,
  ): void {
    this.CRUD.pk.forEach((key) => {
      if (!record[key]) {
        throw new AppError({
          message: 'PRIMARY_KEY_REQUIRED',
          class: BaseCrudService.name,
          method: methodName,
          ctx: {
            table: this.CRUD.table,
            key,
            record,
          },
        });
      }
    });
  }
}
```

src/logs/logs.module.ts

```
import { Global, Module } from '@nestjs/common';
import { LogsService } from './logs.service';
```

```
@Module({
  providers: [LogsService],
  exports: [LogsService],
})
@Global()
export class LogsModule {}
```

src/logs/enums/log-ttl.enum.ts

```
export enum LogTtl {
  'min_1' = 60,
  'min_5' = 60 * 5,
  'min_10' = 60 * 10,
  'min_15' = 60 * 15,
  'min_30' = 60 * 30,
  'min_45' = 60 * 45,
  'hour_1' = 3600,
  'hour_2' = 3600 * 2,
  'hour_4' = 3600 * 4,
  'hour_8' = 3600 * 8,
  'day_1' = 86400,
  'day_3' = 86400 * 3,
  'week_1' = 86400 * 7,
  'week_2' = 86400 * 7 * 2,
  'month_1' = 86400 * 7 * 4,
  'year_1' = 86400 * 360,
}
```

src/logs/models/log.model.ts

```
import { DatabaseTableModel } from '../../common/models/database-table.model';

export class Log extends DatabaseTableModel {
  serial_id?: number = undefined;

  event: string = undefined;

  rrn?: string = undefined;

  payload?: string = undefined;

  created_at: string = undefined;

  expires_at: string = undefined;
}
```

src/logs/logs.service.ts

```
import { LogTtl } from './enums/log-ttl.enum';
import {
  Inject,
  Injectable,
  OnApplicationBootstrap,
  Logger,
} from '@nestjs/common';
import { BaseCrudService } from '../common/services/base-crud.service';
import { Log } from './models/log.model';
import * as NodeCache from 'node-cache';
import { DB_CONNECTION } from '../database/constants/injection-keys';
import { Knex } from 'knex';
import { CACHE_SERVICE } from '../cache/constants/injection-keys';
import { CacheService } from '../cache/cache.service';
import { md5 } from '../common/helpers/md5';
import * as dayjs from 'dayjs';
import { Interval } from '@nestjs/schedule';
import { dbISODatetime } from '../common/helpers/db-iso-date';
```

```
interface LogPayload extends Record<string, unknown> {
  query?: string;
  rrn?: string;
}

@Injectable()
export class LogsService
  extends BaseCrudService<Log>
  implements OnApplicationBootstrap
{
  private readonly logger = new Logger(LogsService.name);
  private readonly DEDUP_TIMEOUT = 300;
  private readonly LOGS_RETENTION_TIME = LogTtl.week_2;
  private readonly PAYLOAD_MAX_LENGTH = 65536;
  private readonly CHUNK_SIZE = 50;
  private readonly MAX_NESTING_LEVEL = 10;

  private readonly _logsStack: Log[] = [];
  private readonly _dedupCache = new NodeCache({ stdTTL: this.DEDUP_TIMEOUT });

  private readonly restrictedFields: string[] = [
    'password',
    'token',
    'secret',
    'apiKey',
    'authorization',
  ];

  constructor(
    @Inject(DB_CONNECTION)
    protected readonly db: Knex,
    @Inject(CACHE_SERVICE)
    protected readonly cacheService: CacheService,
  ) {
    super(db, cacheService, {
      class: Log,
      table: 'logs',
      pk: ['serial_id'],
    });
  }

  onApplicationBootstrap() {
    void this._publishLogsLoop();
  }

  register(event: string, payload?: LogPayload, ttl?: LogTtl): void {
    try {
      const processedPayload = this._processPayload(payload);
      const dedupKey = this._generateDedupKey(event, processedPayload);

      if (this._isDuplicate(dedupKey)) {
        return;
      }

      const log = this._createLogEntry(event, processedPayload, ttl);
      this._logsStack.push(log);
    } catch (error) {
      this.logger.error(
        `Failed to register log: ${error.message}`,
        error.stack,
      );
    }
```

```
  }

  @Interval(2_000)
  private async _publishLogsLoop(): Promise<void> {
    if (this._logsStack.length === 0) {
      return;
    }

    try {
      const chunk = this._logsStack.splice(0, this.CHUNK_SIZE);
      await this.queryBuilder().insert(chunk).onConflict().ignore();
    } catch (error) {
      this.logger.error(
        `Failed to publish logs: ${error.message}`,
        error.stack,
      );
    }
  }

  async cleanExpiredLogs(): Promise<void> {
    try {
      const result = await this.queryBuilder()
        .where('expires_at', '<', dbISODatetime())
        .delete();

      this.logger.log(`Cleaned ${result} expired logs`);
    } catch (error) {
      this.logger.error(
        `Failed to clean expired logs: ${error.message}`,
        error.stack,
      );
    }
  }

  private _processPayload(payload?: LogPayload): LogPayload {
    if (!payload) {
      return {};
    }

    const copy = structuredClone(payload);
    this._securePayload(copy);

    if (copy.query) {
      copy.query = this._normalizeQuery(copy.query);
    }

    return copy;
  }

  private _normalizeQuery(query: string): string {
    return query
      .replaceAll('\n', ' ')
      .replace(/ +(?= )/g, '')
      .trim();
  }

  private _generateDedupKey(event: string, payload: LogPayload): string {
    const rawPayload = JSON.stringify(payload);
    return md5(`${event}:${rawPayload}`).slice(0, 16);
  }

  private _isDuplicate(dedupKey: string): boolean {
    const isDuplicate = this._dedupCache.get(dedupKey);
    if (!isDuplicate) {
```

```
      this._dedupCache.set(dedupKey, true);
    }
    return !!isDuplicate;
  }

  private _createLogEntry(
    event: string,
    payload: LogPayload,
    ttl?: LogTtl,
  ): Log {
    const rawPayload = JSON.stringify(payload);

    if (rawPayload.length > this.PAYLOAD_MAX_LENGTH) {
      throw new Error(
        `Payload exceeds maximum length of ${this.PAYLOAD_MAX_LENGTH} bytes`,
      );
    }

    const now = dayjs();
    const expiresAt = now
      .clone()
      .add(ttl || this.LOGS_RETENTION_TIME, 'seconds');

    const log = new Log();
    log.event = event;
    log.rrn = payload.rrn;
    log.payload = rawPayload;
    log.created_at = now.toISOString();
    log.expires_at = expiresAt.toISOString();

    return log;
  }

  private _securePayload(payload?: Record<string, unknown>, level = 0): void {
    if (!payload || level > this.MAX_NESTING_LEVEL) {
      return;
    }

    for (const key in payload) {
      if (typeof payload[key] === 'object' && payload[key] !== null) {
        this._securePayload(payload[key] as Record<string, unknown>, level + 1);
      }

      if (this.restrictedFields.includes(key)) {
        payload[key] = '***';
      }
    }
  }
}
```

src/logs/schedulers/logs.scheduler.ts

```
import { Injectable } from '@nestjs/common';
import { Cron, CronExpression } from '@nestjs/schedule';
import { LogsService } from '../logs.service';

@Injectable()
export class LogsScheduler {
  constructor(private readonly logsService: LogsService) {}

  @Cron(CronExpression.EVERY_MINUTE)
  logRetention() {
    void this.logsService.cleanExpiredLogs();
```

```
  }
}
```

src/users/enums/user-roles.enum.ts

```
export enum UserRoles {
  admin = 'admin',
  user = 'user',
}
```

src/users/users.service.ts

```
import { Injectable, Inject } from '@nestjs/common';
import { Knex } from 'knex';
import { BaseCrudService } from '../common/services/base-crud.service';
import { User } from './models/user.model';
import { CacheService } from '../cache/cache.service';
import { DB_CONNECTION } from '../database/constants/injection-keys';
import { CACHE_SERVICE } from '../cache/constants/injection-keys';

@Injectable()
export class UsersService extends BaseCrudService<User> {
  constructor(
    @Inject(DB_CONNECTION)
    protected readonly db: Knex,
    @Inject(CACHE_SERVICE)
    protected readonly cacheManager: CacheService,
  ) {
    super(db, cacheManager, {
      table: 'users',
      pk: ['id'],
      class: User,
    });
  }

  async findOneById(id: string): Promise<User | undefined> {
    return this.findOne({ id });
  }

  async findOneByNickname(nickname: string): Promise<User | undefined> {
    return this.findOne({ nickname });
  }
}
```

src/users/users.module.ts

```
import { Module } from '@nestjs/common';
import { UsersService } from './users.service';

@Module({
  providers: [UsersService],
  exports: [UsersService],
})
export class UsersModule {}
```

src/users/models/user.model.ts

```
import { UserRoles } from '../enums/user-roles.enum';
import { DatabaseTableModel } from '../../common/models/database-table.model';

export class User extends DatabaseTableModel {
  serial_id: number;
  id: string;
  nickname: string;
  pass_hash: string;
  role: UserRoles;
  created_at: Date;
```

```
  updated_at: Date;
}
```

src/app.controller.ts

```
import { Controller, Get } from '@nestjs/common';
import { AppService } from './app.service';

@Controller()
export class AppController {
  constructor(private readonly appService: AppService) {}

  @Get()
  getHello(): string {
    return this.appService.getHello();
  }
}
```

src/config.ts

```
import * as process from 'node:process';
import { config } from 'dotenv';

export enum Environments {
  DEV = 'development',
  PROD = 'production',
}

if ((process.env.NODE_ENV as Environments) !== Environments.PROD) {
  config();
}

export const envConfig = {
  port: process.env.PORT || 3000,
  env: (process.env.NODE_ENV as Environments) || Environments.DEV,
  database: {
    host: process.env.DATABASE_HOST || 'localhost',
    port: parseInt(process.env.DATABASE_PORT || '5432'),
    user: process.env.DATABASE_USER || 'postgres',
    password: process.env.DATABASE_PASSWORD || 'postgres',
    database: process.env.DATABASE_NAME || 'iot_db',
  },
  redis: {
    url: process.env.REDIS_URL || '',
  },
  jwt: {
    secret: process.env.JWT_SECRET || 'your-secret-key',
    expiresIn: process.env.JWT_EXPIRES_IN || '1h',
  },

  defaultCacheTTL: 300,
};

export type TConfig = typeof envConfig;
```

src/app-interceptor.service.ts

```
import {
  CallHandler,
  ExecutionContext,
  HttpException,
  HttpStatus,
  Injectable,
  NestInterceptor,
} from '@nestjs/common';
import { LogsService } from './logs/logs.service';
```

```
import { catchError, Observable, throwError } from 'rxjs';
import { randomUUID } from 'node:crypto';
import { AppError } from './common/errors/app.error';
import knex from 'knex';

@Injectable()
export class AppInterceptor implements NestInterceptor {
 constructor(private readonly logsService: LogsService) {}

 intercept(
  context: ExecutionContext,
  next: CallHandler<unknown>,
 ): Observable<unknown> {
  return next.handle().pipe(
   catchError((err: Error) => {
    const request = context.switchToHttp().getRequest();

    const rrn = randomUUID().slice(0, 8).toUpperCase();

    const payload: Record<string, unknown> = {
     rrn,
     stack: err.stack,
     body: request.body,
     url: request.url,
    };
    let message = err.message;
    let statusCode = HttpStatus.INTERNAL_SERVER_ERROR;

    let event = message ?? 'GeneralException';

    if (err instanceof AppError) {
     payload.class = err.class;
     payload.method = err.method;
     payload.ctx = err.ctx;
     payload.error_message = err.message;

     message = err.userFriendlyMessage ?? err.message;
     statusCode = err.statusCode;
    }
    const pgErrorCodes = [
     '23', // constraint violations
     '22', // data exceptions
     '28', // invalid authorization
     '42', // syntax errors
     '53', // insufficient resources
     '54', // program limit exceeded
     '55', // object not in prerequisite state
     '57', // operator intervention
     '58', // system error
     'F0', // configuration file error
     'HV', // foreign data wrapper error
     'P0', // PL/pgSQL error
     'XX', // internal error
    ];

    const errorCode = (err as { code?: string }).code;
    const isDbError =
     err instanceof knex.KnexTimeoutError ||
     pgErrorCodes.some((code) => errorCode?.startsWith(code));

    if (isDbError) {
     event = 'UNEXPECTED_DB_ERROR';
```

```
        message = 'Internal database error, please try again later';
        payload.pg_error_code = errorCode;
      }

      message = `RRN: ${rrn}, ${message}`;

      this.logsService.register(event, {
        ...payload,
        error_message: err.message,
      });

      return throwError(() => new HttpException(message, statusCode));
    }),
  );
 }
}
```

src/notifications/notifications.module.ts

```
import { Module } from '@nestjs/common';

@Module({})
export class NotificationsModule {}
```

src/devices/devices-properties.controller.ts

```
import {
 Controller,
 Get,
 Post,
 Body,
 Patch,
 Param,
 Delete,
 Query,
} from '@nestjs/common';
import {
 ApiTags,
 ApiOperation,
 ApiResponse,
 ApiParam,
 ApiQuery,
} from '@nestjs/swagger';
import { CreateDevicePropertyDto } from './dto/create-device-property.dto';
import { UpdateDevicePropertyDto } from './dto/update-device-property.dto';
import { DeviceProperty } from './models';
import { DevicePropertiesService } from './device-properties.service';

@ApiTags('device-properties')
@Controller('devices/properties')
export class DevicesPropertiesController {
 constructor(
  private readonly devicePropertiesService: DevicePropertiesService,
 ) {}

 @Post()
 @ApiOperation({ summary: 'Create a new device property' })
 @ApiResponse({
  status: 201,
  description: 'Device property created',
  type: DeviceProperty,
 })
 create(@Body() createDevicePropertyDto: CreateDevicePropertyDto) {
```

```
    return
this.devicePropertiesService.create(createDeviceProper
tyDto);
  }


  @Get()
  @ApiOperation({ summary: 'Get all device properties'
})
  @ApiResponse({
    status: 200,
    description: 'Return all device properties',
    type: [DeviceProperty],
  })
  @ApiQuery({
    name: 'device_id',
    required: false,
    description: 'Filter properties by device ID',
  })
  findAll(@Query('device_id') deviceId?: string) {
    if (deviceId) {
      return
this.devicePropertiesService.findByDeviceId(deviceId)
;
    }
    return new Error('Not implemented');
  }


  @Get(':id')
  @ApiOperation({ summary: 'Get a device property by
ID' })
  @ApiParam({ name: 'id', description: 'Device property
ID' })
  @ApiResponse({
    status: 200,
    description: 'Return the device property',
    type: DeviceProperty,
  })
  @ApiResponse({ status: 404, description: 'Device
property not found' })
  findOne(@Param('id') id: string) {
    return
this.devicePropertiesService.findOneById(Number(id))
;
  }


  @Patch(':id')
  @ApiOperation({ summary: 'Update a device
property' })
  @ApiParam({ name: 'id', description: 'Device property
ID' })
  @ApiResponse({
    status: 200,
    description: 'Device property updated',
    type: DeviceProperty,
  })
  @ApiResponse({ status: 404, description: 'Device
property not found' })
  update(
    @Param('id') id: string,
    @Body() updateDevicePropertyDto:
UpdateDevicePropertyDto,
  ) {
    return this.devicePropertiesService.update(
      Number(id),
      updateDevicePropertyDto,
    );
  }


  @Delete(':id')
  @ApiOperation({ summary: 'Delete a device property'
})
  @ApiParam({ name: 'id', description: 'Device property
ID' })
```

```
  @ApiResponse({
    status: 200,
    description: 'Device property successfully deleted',
  })
  @ApiResponse({ status: 404, description: 'Device property not found' })
  remove(@Param('id') id: string) {
    return this.devicePropertiesService.remove(Number(id));
  }
}
```

src/devices/device-events.gateway.ts

```
import {
  WebSocketGateway,
  OnGatewayConnection,
  OnGatewayDisconnect,
  SubscribeMessage,
} from '@nestjs/websockets';
import * as WebSocket from 'ws';
import { randomUUID } from 'node:crypto';
import { filter, from, fromEvent, map, Observable } from 'rxjs';
import { EventEmitter2 } from '@nestjs/event-emitter';
import { DeviceEvent } from './models/device-event.model';

@WebSocketGateway({
  path: 'devices/ws',
})
export class DeviceEventsGateway
  implements OnGatewayConnection, OnGatewayDisconnect
{
  private readonly connections: Map<string, WebSocket> = new Map();

  constructor(private readonly eventEmitter: EventEmitter2) {}

  handleConnection(client: WebSocket) {
    const connectionId = randomUUID();
    client['clientId'] = connectionId;
    this.connections.set(connectionId, client);
    console.log(`Client connected: ${connectionId}`);
  }

  handleDisconnect(client: WebSocket) {
    const connectionId = client['clientId'];
    if (connectionId) {
      this.connections.delete(connectionId);
      console.log(`Client disconnected: ${connectionId}`);
    } else {
      console.warn('Client disconnected without a valid ID');
    }
  }

  @SubscribeMessage('subscribe')
  handleEvent(
    _: WebSocket,
    data: { subscriptionId: string; deviceId: string; event: string },
  ): Observable<any> {
    return fromEvent(this.eventEmitter, 'devices.*').pipe(
      filter((incomingEvent: DeviceEvent) => {
        const { deviceId, event } = data;

        if (deviceId && incomingEvent.device_id !== deviceId) {
          return false;
```

```
      }

      if (event && incomingEvent.event !== event) {
        return false;
      }

      return true;
    }),
    map((event: DeviceEvent) => {
      return {
        ...event,
        subscriptionId: data.subscriptionId,
        data: JSON.parse(event.data),
      };
    }),
  );
 }
}
```

src/devices/enums/device-events-types.enum.ts

```
import { DeviceEventType } from '../../connections/proto/iot';

export enum DeviceEventsTypes {
 raw = 'raw',
 online = 'online',
 offline = 'offline',
 stateUpdate = 'stateUpdate',
 errorAlert = 'errorAlert',
 heartbeat = 'heartbeat',
 sensorReading = 'sensorReading',
}

export const mapProtoToEvents: Record<DeviceEventType, DeviceEventsTypes> = {
 [DeviceEventType.ON]: DeviceEventsTypes.online,
 [DeviceEventType.OFF]: DeviceEventsTypes.offline,
 [DeviceEventType.STATE_UPDATE]: DeviceEventsTypes.stateUpdate,
 [DeviceEventType.ERROR_ALERT]: DeviceEventsTypes.errorAlert,
 [DeviceEventType.HEARTBEAT]: DeviceEventsTypes.heartbeat,
 [DeviceEventType.SENSOR_READING]: DeviceEventsTypes.sensorReading,
 [DeviceEventType.UNRECOGNIZED]: DeviceEventsTypes.raw,
};
```

src/devices/enums/device-state.enum.ts

```
export enum DeviceState {
 online = 'online',
 offline = 'offline',
 maintenance = 'maintenance',
}
```

src/devices/devices.service.ts

```
import { Injectable, Inject, NotFoundException } from '@nestjs/common';
import { Knex } from 'knex';
import { BaseCrudService } from '../common/services/base-crud.service';
import { Device } from './models';
import { CacheService } from '../cache/cache.service';
import { DB_CONNECTION } from '../database/constants/injection-keys';
import { CACHE_SERVICE } from '../cache/constants/injection-keys';
import { FindDevicesArgs } from './args/find-devices.dto';
import { plainToInstance } from 'class-transformer';
```

```
import { omit } from '../common/helpers/omit';
import { UpdateDeviceDto } from './dto/update-device.dto';
import { CreateDeviceDto } from './dto/create-device.dto';
import { randomUUID } from 'node:crypto';

@Injectable()
export class DevicesService extends BaseCrudService<Device> {
  constructor(
    @Inject(DB_CONNECTION)
    protected readonly db: Knex,
    @Inject(CACHE_SERVICE)
    protected readonly cacheManager: CacheService,
  ) {
    super(db, cacheManager, {
      table: 'devices',
      pk: ['id'],
      class: Device,
    });
  }

  async complexSearch(args: FindDevicesArgs) {
    const query = this.queryBuilder()
      .limit(args?.limit ?? 30)
      .offset(args?.offset ?? 0);

    if (args?.search) {
      query.where((qb) => {
        qb.where('id', 'like', `%${args.search}%`).orWhere(
          'external_id',
          'like',
          `%${args.search}%`,
        );
      });
    }

    const searchArgs = omit(args, ['limit', 'offset', 'search']);

    for (const [key, value] of Object.entries(searchArgs)) {
      if (value) {
        query.where(key, value);
      }
    }

    const result = await query;

    return plainToInstance(Device, result);
  }

  async create(args: CreateDeviceDto): Promise<Device> {
    const device = plainToInstance(Device, args);

    device.id = randomUUID();

    await this.createOne(device);

    return device;
  }

  async update(id: string, args: UpdateDeviceDto): Promise<Device> {
    const device = await this.findOne({
      id,
    });
```

```
    if (!device) {
      throw new NotFoundException(`Device with ID ${id} not found`);
    }

    device.populate(args);

    await this.updateOne({ id, ...args });

    return device;
  }

  async delete(id: string): Promise<void> {
    const device = await this.findOne({ id });

    if (!device) {
      throw new NotFoundException(`Device with ID ${id} not found`);
    }

    await this.deleteOne(device);
  }
}
```

src/devices/devices-events.service.ts

```
import { Inject, Injectable } from '@nestjs/common';
import { BaseCrudService } from '../common/services/base-crud.service';
import { DeviceEvent } from './models/device-event.model';
import { DB_CONNECTION } from '../database/constants/injection-keys';
import { Knex } from 'knex';
import { CACHE_SERVICE } from '../cache/constants/injection-keys';
import { CacheService } from '../cache/cache.service';
import { FindDevicesEventsArgs } from './args/find-devices-events.dto';
import { omit } from '../common/helpers/omit';
import { plainToInstance } from 'class-transformer';
import { Device } from './models';

@Injectable()
export class DevicesEventsService extends BaseCrudService<DeviceEvent> {
  constructor(
    @Inject(DB_CONNECTION)
    protected readonly db: Knex,
    @Inject(CACHE_SERVICE)
    protected readonly cacheManager: CacheService,
  ) {
    super(db, cacheManager, {
      table: 'devices_events',
      pk: ['id'],
      class: DeviceEvent,
    });
  }

  async complexSearch(args: FindDevicesEventsArgs) {
    const query = this.queryBuilder()
      .limit(args?.limit ?? 30)
      .offset(args?.offset ?? 0);

    if (args?.search) {
      query.where((qb) => {
        qb.where('id', 'like', `%${args.search}%`)
          .orWhere('event', 'like', `%${args.search}%`)
          .orWhere('device_id', 'like', `%${args.search}%`);
      });
```

```
    }

    if (args?.from) {
      query.where('created_at', '>=', args.from);
    }
    if (args?.to) {
      query.where('created_at', '<=', args.to);
    }

    const searchArgs = omit(args, ['limit', 'offset', 'search', 'from', 'to']);

    for (const [key, value] of Object.entries(searchArgs)) {
      if (value) {
        query.where(key, value);
      }
    }

    const result = await query;

    return plainToInstance(DeviceEvent, result);
  }
}
```

src/devices/device-types.controller.ts

```
import {
  Controller,
  Get,
  Post,
  Body,
  Patch,
  Param,
  Delete,
  ParseIntPipe,
  UseGuards,
} from '@nestjs/common';
import { ApiTags, ApiOperation, ApiResponse, ApiParam, ApiBearerAuth } from '@nestjs/swagger';
import { CreateDeviceTypeDto } from './dto/create-device-type.dto';
import { UpdateDeviceTypeDto } from './dto/update-device-type.dto';
import { DeviceType } from './models';
import { DeviceTypesService } from './device-types.service';
import { JwtAuthGuard } from '../auth/guards/jwt-auth.guard';
import { RolesGuard } from '../auth/guards/roles.guard';
import { RequiredRole } from '../auth/decorators/required-role.decorator';
import { UserRoles } from '../users/enums/user-roles.enum';

@ApiTags('device-types')
@ApiBearerAuth()
@Controller('devices-types')
@UseGuards(JwtAuthGuard, RolesGuard)
export class DevicesTypesController {
  constructor(private readonly deviceTypesService: DeviceTypesService) {}

  @Post()
  @RequiredRole(UserRoles.admin)
  @ApiOperation({ summary: 'Create a new device type' })
  @ApiResponse({
    status: 201,
    description: 'Device type created',
    type: DeviceType,
  })
```

```
  create(@Body() createDeviceTypeDto: CreateDeviceTypeDto) {
    return this.deviceTypesService.create(createDeviceTypeDto);
  }

  @Get('/')
  @RequiredRole(UserRoles.user)
  @ApiOperation({ summary: 'Get all device types' })
  @ApiResponse({
    status: 200,
    description: 'Return all device types',
    type: [DeviceType],
  })
  findAll() {
    return this.deviceTypesService.findAll();
  }

  @Get(':id')
  @RequiredRole(UserRoles.user)
  @ApiOperation({ summary: 'Get a device type by ID' })
  @ApiParam({ name: 'id', description: 'Device type ID' })
  @ApiResponse({
    status: 200,
    description: 'Return the device type',
    type: DeviceType,
  })
  @ApiResponse({ status: 404, description: 'Device type not found' })
  findOne(@Param('id', ParseIntPipe) id: number) {
    return this.deviceTypesService.findOneById(id);
  }

  @Patch(':id')
  @RequiredRole(UserRoles.admin)
  @ApiOperation({ summary: 'Update a device type' })
  @ApiParam({ name: 'id', description: 'Device type ID' })
  @ApiResponse({
    status: 200,
    description: 'Device type updated',
    type: DeviceType,
  })
  @ApiResponse({ status: 404, description: 'Device type not found' })
  update(
    @Param('id', ParseIntPipe) id: number,
    @Body() updateDeviceTypeDto: UpdateDeviceTypeDto,
  ) {
    return this.deviceTypesService.update(id, updateDeviceTypeDto);
  }

  @Delete(':id')
  @RequiredRole(UserRoles.admin)
  @ApiOperation({ summary: 'Delete a device type' })
  @ApiParam({ name: 'id', description: 'Device type ID' })
  @ApiResponse({ status: 200, description: 'Device type successfully deleted' })
  @ApiResponse({ status: 404, description: 'Device type not found' })
  remove(@Param('id', ParseIntPipe) id: number) {
    return this.deviceTypesService.remove(id);
  }
}
```

src/devices/models/device-event.model.ts

```
import { DatabaseTableModel } from '../../common/models/database-table.model';
```

```
import { ApiProperty } from '@nestjs/swagger';

export class DeviceEvent extends DatabaseTableModel
{
  @ApiProperty({
    description: 'Unique identifier of the device event',
    example: '123e4567-e89b-12d3-a456-426614174000',
  })
  id: string = undefined;

  @ApiProperty({
    description: 'ID of the device that generated this event',
    example: '123e4567-e89b-12d3-a456-426614174000',
  })
  device_id: string = undefined;

  @ApiProperty({
    description: 'Type of the event',
    example: 'status_change',
  })
  event: string = undefined;

  @ApiProperty({
    description: 'Event data in JSON string format',
    example: '{"status": "online", "battery": 85}',
  })
  data: string = undefined;

  @ApiProperty({
    description: 'Timestamp when the event was created',
    example: '2024-03-20T15:30:00Z',
  })
  created_at: string = undefined;
}
```

src/devices/models/device-type.model.ts

```
import { ApiProperty } from '@nestjs/swagger';

export class DeviceType {
  @ApiProperty({
    description: 'Unique identifier of the device type',
    example: 1,
    type: 'number',
  })
  id: number = undefined;

  @ApiProperty({
    description: 'Name of the device type',
    example: 'light_bulb',
  })
  name: string = undefined;

  @ApiProperty({
    description: 'Display name of the device type',
    example: 'Smart Light Bulb',
  })
  display_name: string = undefined;

  @ApiProperty({
    description: 'Creation timestamp',
    example: '2024-03-20T12:00:00Z',
    type: 'string',
    format: 'date-time',
  })
  created_at: string = undefined;
}
```

src/devices/models/index.ts

```
export * from './device.model';
export * from './device-property.model';
export * from './device-type.model';
```

src/devices/models/device-location.model.ts

```
import { ApiProperty } from '@nestjs/swagger';

export class DeviceLocation {
  @ApiProperty({
    description: 'Unique identifier of the device location',
    example: 1,
    type: 'number',
  })
  id: number;

  @ApiProperty({
    description: 'Name of the location',
    example: 'living_room',
  })
  name: string;

  @ApiProperty({
    description: 'Display name of the location',
    example: 'Living Room',
  })
  display_name: string;

  @ApiProperty({
    description: 'Creation timestamp',
    example: '2024-03-20T12:00:00Z',
    type: 'string',
    format: 'date-time',
  })
  created_at: string = undefined;
}
```

src/devices/models/device-property.model.ts

```
import { ApiProperty } from '@nestjs/swagger';
import { DatabaseTableModel } from '../../common/models/database-table.model';
import { randomUUID } from 'node:crypto';

export class DeviceProperty extends DatabaseTableModel {
  @ApiProperty({
    description: 'Unique identifier of the device property',
    example: 1,
    type: 'number',
  })
  id: number = undefined;

  @ApiProperty({
    description: 'ID of the device this property belongs to',
    example: randomUUID(),
    type: 'string',
    format: 'uuid',
  })
  device_id: string = undefined;

  @ApiProperty({
    description: 'Name of the property',
    example: 'brightness',
  })
  name: string = undefined;

  @ApiProperty({
```

```
    description: 'Value of the property',
    example: '100',
  })
  value: string = undefined;

  @ApiProperty({
    description: 'Creation timestamp',
    example: '2024-03-20T12:00:00Z',
    type: 'string',
    format: 'date-time',
  })
  created_at: string = undefined;

  @ApiProperty({
    description: 'Last update timestamp',
    example: '2024-03-20T12:00:00Z',
    type: 'string',
    format: 'date-time',
  })
  updated_at: string = undefined;
}
```

src/devices/models/device.model.ts

```
import { ApiProperty } from '@nestjs/swagger';
import { DatabaseTableModel } from '../../common/models/database-table.model';
import { randomUUID } from 'node:crypto';
import { DeviceState } from '../enums/device-state.enum';

export class Device extends DatabaseTableModel {
  @ApiProperty({
    description: 'Unique identifier of the device',
    example: randomUUID(),
    type: 'string',
    format: 'uuid',
  })
  id: string = undefined;

  @ApiProperty({
    description: 'Type ID of the device',
    example: 2,
    type: 'number',
  })
  type_id: number = undefined;

  @ApiProperty({ description: 'Whether the device is active', example: true })
  is_active: boolean = undefined;

  @ApiProperty({
    description: 'Current state of the device',
    enum: DeviceState,
    example: DeviceState.online,
  })
  state: DeviceState = undefined;

  @ApiProperty({
    description: 'Location ID of the device',
    example: 1,
    type: 'number',
    required: false,
  })
  location_id: number = undefined;

  @ApiProperty({
    description: 'IP address of the device',
    example: '192.168.1.100',
```

```
  required: false,
})
ip: string = undefined;

@ApiProperty({
  description: 'MAC address of the device',
  example: '00:11:22:33:44:55',
  required: false,
})
mac: string = undefined;

@ApiProperty({
  description: 'ID of the connection adapter',
  example: randomUUID(),
  type: 'string',
  format: 'uuid',
  required: false,
})
adapter_id: string = undefined;

@ApiProperty({
  description: 'External ID of the device',
  example: 'device_123',
  required: false,
})
external_id: string = undefined;

@ApiProperty({
  description: 'Description of the device',
  example: 'Smart light bulb in living room',
  required: false,
})
description: string = undefined;

@ApiProperty({
  description: 'Creation timestamp',
  example: '2024-03-20T12:00:00Z',
  type: 'string',
  format: 'date-time',
 })
 created_at: string = undefined;

 @ApiProperty({
  description: 'Last update timestamp',
  example: '2024-03-20T12:00:00Z',
  type: 'string',
  format: 'date-time',
 })
 updated_at: string = undefined;
}
```

src/devices/processors/device-events.processor.ts

```
import { Injectable, OnApplicationBootstrap } from '@nestjs/common';
import { ConnectionsAdaptersService } from '../../connections/connections-adapters.service';
import { EventEmitter2 } from '@nestjs/event-emitter';
import {
 DeviceEvent as ExternalDeviceEvent,
 SubscriptionStream,
} from '../../connections/proto/iot';
import { DevicesEventsService } from '../devices-events.service';
import { DeviceEvent } from '../models/device-event.model';
import {
 mapProtoToEvents,
 DeviceEventsTypes,
} from '../enums/device-events-types.enum';
```

```
import { plainToInstance } from 'class-transformer';
import { dbISODatetime } from '../../common/helpers/db-iso-date';
import * as NodeCache from 'node-cache';
import { randomUUID } from 'node:crypto';
import { Interval } from '@nestjs/schedule';
import { unix } from 'dayjs';
import { TriggersService } from '../../triggers/triggers.service';

@Injectable()
export class DeviceEventsProcessor implements OnApplicationBootstrap {
  private readonly MAX_SYNC_CHUNK_SIZE = 50;

  private readonly _eventStack: DeviceEvent[] = [];
  private readonly _dedupCache: NodeCache = new NodeCache({ stdTTL: 300 });

  constructor(
    private readonly eventEmitter: EventEmitter2,
    private readonly deviceEventsService: DevicesEventsService,
    private readonly connectionsAdaptersService: ConnectionsAdaptersService,
    private readonly triggersService: TriggersService,
  ) {}

  onApplicationBootstrap(): void {

this.connectionsAdaptersService.eventsStream$.subscribe(
      (sub: SubscriptionStream) => void this.processEvent(sub.event),
    );
  }

  processEvent(incomingEvent: ExternalDeviceEvent): void {
    const event = mapProtoToEvents[incomingEvent.eventType];

    const eventDataObj = {
      ...incomingEvent.sensorReading,
      ...incomingEvent.stateUpdate,
      ...incomingEvent.errorAlert,
      ...incomingEvent.heartbeat,
    };

    const eventData = plainToInstance(DeviceEvent, {
      id: randomUUID(),
      device_id: incomingEvent.deviceId,
      event,
      data: JSON.stringify(eventDataObj),
      created_at: dbISODatetime(
        unix(Math.ceil(incomingEvent.timestamp / 1000)),
      ),
    });

    const dedupKey = `${incomingEvent.deviceId}-${eventData.event}-${incomingEvent.timestamp}`;

    if (this._dedupCache.has(dedupKey)) return;

    this._dedupCache.set(dedupKey, true);

    this._eventStack.push(eventData);

    void this.eventEmitter.emitAsync(`devices.${event}`, eventData);
```

```
    if (
      event === DeviceEventsTypes.sensorReading &&
      incomingEvent.sensorReading
    ) {
      for (const [propertyName, propertyValue] of Object.entries(
        incomingEvent.sensorReading,
      )) {
        if (propertyValue !== undefined && propertyValue !== null) {
          void this.triggersService.checkTrigger(
            incomingEvent.deviceId,
            propertyName,
            propertyValue.toString(),
          );
        }
      }
    }
  }


  @Interval(2000)
  async syncStackWithDB(): Promise<void> {
    if (this._eventStack.length === 0) return;


    const chunkToPush = this._eventStack.splice(0, this.MAX_SYNC_CHUNK_SIZE);


    try {
      await this.deviceEventsService
        .queryBuilder()
        .insert(chunkToPush)
        .onConflict()
        .ignore();
    } catch (err) {
      console.error('Failed to sync device events with DB:', err);
    }
  }
}
```

src/devices/devices-events.controller.ts

```
import {
  Controller,
  Get,
  Query,
  Param,
  UseGuards,
  ValidationPipe,
} from '@nestjs/common';
import {
  ApiTags,
  ApiOperation,
  ApiResponse,
  ApiParam,
  ApiQuery,
  ApiBearerAuth,
} from '@nestjs/swagger';
import { DevicesEventsService } from './devices-events.service';
import { DeviceEvent } from './models/device-event.model';
import { FindDevicesEventsArgs } from './args/find-devices-events.dto';
import { JwtAuthGuard } from '../auth/guards/jwt-auth.guard';
import { RolesGuard } from '../auth/guards/roles.guard';
import { RequiredRole } from '../auth/decorators/required-role.decorator';
import { UserRoles } from '../users/enums/user-roles.enum';
```

```
@ApiTags('devices-events')
@ApiBearerAuth()
@Controller('devices-events')
@UseGuards(JwtAuthGuard, RolesGuard)
export class DevicesEventsController {
  constructor(private readonly devicesEventsService: DevicesEventsService) {}

  @Get()
  @RequiredRole(UserRoles.user)
  @ApiOperation({ summary: 'Get all device events' })
  @ApiResponse({
    status: 200,
    description: 'Return all device events',
    type: [DeviceEvent],
  })
  @ApiQuery({ type: FindDevicesEventsArgs })
  findAll(
    @Query(new ValidationPipe({ transform: true })) args: FindDevicesEventsArgs,
  ) {
    return this.devicesEventsService.complexSearch(args);
  }

  @Get(':id')
  @RequiredRole(UserRoles.user)
  @ApiOperation({ summary: 'Get a device event by ID' })
  @ApiParam({ name: 'id', description: 'Device Event ID' })
  @ApiResponse({
    status: 200,
    description: 'Return the device event',
    type: DeviceEvent,
  })
  @ApiResponse({ status: 404, description: 'Device event not found' })
  findOne(@Param('id') id: string) {
    return this.devicesEventsService.findOneThruCache({ id });
  }
}
```

src/devices/devices.controller.ts

```
import {
  Controller,
  Get,
  Post,
  Body,
  Patch,
  Param,
  Delete,
  Query,
  ParseUUIDPipe,
  UseGuards,
} from '@nestjs/common';
import {
  ApiTags,
  ApiOperation,
  ApiResponse,
  ApiParam,
  ApiBearerAuth,
} from '@nestjs/swagger';
import { CreateDeviceDto } from './dto/create-device.dto';
import { UpdateDeviceDto } from './dto/update-device.dto';
import { Device } from './models';
import { FindDevicesArgs } from './args/find-devices.dto';
```

```
import { DevicesService } from './devices.service';
import { JwtAuthGuard } from '../auth/guards/jwt-auth.guard';
import { RolesGuard } from '../auth/guards/roles.guard';
import { RequiredRole } from '../auth/decorators/required-role.decorator';
import { UserRoles } from '../users/enums/user-roles.enum';


@ApiTags('devices')
@ApiBearerAuth()
@Controller('devices')
@UseGuards(JwtAuthGuard, RolesGuard)
export class DevicesController {
 constructor(private readonly devicesService: DevicesService) {}


 @Post()
 @RequiredRole(UserRoles.admin)
 @ApiOperation({ summary: 'Create a new device' })
 @ApiResponse({ status: 201, description: 'Device created', type: Device })
 create(@Body() createDeviceDto: CreateDeviceDto) {
  return this.devicesService.create(createDeviceDto);
 }


 @Get()
 @RequiredRole(UserRoles.user)
 @ApiOperation({ summary: 'Get all devices' })
 @ApiResponse({
  status: 200,
  description: 'Return all devices',
  type: [Device],
 })
 findAll(@Query() args: FindDevicesArgs) {
  return this.devicesService.complexSearch(args);
 }


 @Get(':id')
 @RequiredRole(UserRoles.user)
 @ApiOperation({ summary: 'Get a device by ID' })
 @ApiParam({ name: 'id', description: 'Device ID' })
 @ApiResponse({ status: 200, description: 'Return the device', type: Device })
 @ApiResponse({ status: 404, description: 'Device not found' })
 findOne(@Param('id') id: string) {
  return this.devicesService.findOneThruCache({ id });
 }


 @Patch(':id')
 @RequiredRole(UserRoles.admin)
 @ApiOperation({ summary: 'Update a device' })
 @ApiParam({ name: 'id', description: 'Device ID' })
 @ApiResponse({ status: 200, description: 'Device updated', type: Device })
 @ApiResponse({ status: 404, description: 'Device not found' })
 update(@Param('id') id: string, @Body() updateDeviceDto: UpdateDeviceDto) {
  return this.devicesService.update(id, updateDeviceDto);
 }


 @Delete(':id')
 @RequiredRole(UserRoles.admin)
 @ApiOperation({ summary: 'Delete a device' })
 @ApiParam({ name: 'id', description: 'Device ID' })
 @ApiResponse({ status: 200, description: 'Device successfully deleted' })
 @ApiResponse({ status: 404, description: 'Device not found' })
 remove(@Param('id') id: string) {
```

```
    return this.devicesService.delete(id);
  }
}
```

src/devices/devices.module.ts

```
import { Module } from '@nestjs/common';
import { DevicesTypesController } from './device-types.controller';
import { DevicesController } from './devices.controller';
import { DevicesPropertiesController } from './devices-properties.controller';
import { DevicesService } from './devices.service';
import { DeviceTypesService } from './device-types.service';
import { DevicePropertiesService } from './device-properties.service';
import { DeviceLocationsService } from './device-locations.service';
import { DeviceEventsGateway } from './device-events.gateway';
import { DevicesEventsService } from './devices-events.service';
import { DevicesEventsController } from './devices-events.controller';
import { DeviceLocationsController } from './device-locations.controller';

@Module({
  controllers: [
    DevicesController,
    DevicesPropertiesController,
    DevicesTypesController,
    DevicesEventsController,
    DeviceLocationsController,
  ],
  providers: [
    DevicesService,
    DeviceTypesService,
    DevicePropertiesService,
    DeviceLocationsService,
    DeviceEventsGateway,
    DevicesEventsService,
  ],
  exports: [
    DevicesService,
    DeviceTypesService,
    DevicePropertiesService,
    DeviceLocationsService,
    DevicesEventsService,
  ],
})
export class DevicesModule {}
```

src/devices/device-types.service.ts

```
import { Injectable, NotFoundException, Inject } from '@nestjs/common';
import { Knex } from 'knex';
import { BaseCrudService } from '../common/services/base-crud.service';
import { DeviceType } from './models';
import { CacheService } from '../cache/cache.service';
import { CreateDeviceTypeDto } from './dto/create-device-type.dto';
import { UpdateDeviceTypeDto } from './dto/update-device-type.dto';
import { DB_CONNECTION } from '../database/constants/injection-keys';
import { CACHE_SERVICE } from '../cache/constants/injection-keys';

@Injectable()
export class DeviceTypesService extends BaseCrudService<DeviceType> {
```

```
  constructor(
    @Inject(DB_CONNECTION)
    protected readonly db: Knex,
    @Inject(CACHE_SERVICE)
    protected readonly cacheManager: CacheService,
  ) {
    super(db, cacheManager, {
      table: 'device_types',
      pk: ['id'],
      class: DeviceType,
    });
  }


  async create(createDeviceTypeDto: CreateDeviceTypeDto): Promise<DeviceType> {
    const deviceType = new DeviceType();
    Object.assign(deviceType, createDeviceTypeDto);
    await this.createOne(deviceType);
    return deviceType;
  }


  async findOneById(id: number): Promise<DeviceType> {
    const deviceType = await this.findOne({ id });
    if (!deviceType) {
      throw new NotFoundException(`Device type with ID ${id} not found`);
    }
    return deviceType;
  }


  async update(
    id: number,
    updateDeviceTypeDto: UpdateDeviceTypeDto,
  ): Promise<DeviceType> {
    const deviceType = await this.findOneById(id);
    Object.assign(deviceType, updateDeviceTypeDto);
    await this.updateOne({ id, ...updateDeviceTypeDto });
    return deviceType;
  }


  async remove(id: number): Promise<void> {
    await this.deleteOne({ id });
  }
}
```

src/devices/device-locations.service.ts

```
import { Injectable, Inject, NotFoundException } from '@nestjs/common';
import { Knex } from 'knex';
import { BaseCrudService } from '../common/services/base-crud.service';
import { DeviceLocation } from './models/device-location.model';
import { CacheService } from '../cache/cache.service';
import { DB_CONNECTION } from '../database/constants/injection-keys';
import { CACHE_SERVICE } from '../cache/constants/injection-keys';


@Injectable()
export class DeviceLocationsService extends BaseCrudService<DeviceLocation> {
  constructor(
    @Inject(DB_CONNECTION)
    protected readonly db: Knex,
    @Inject(CACHE_SERVICE)
    protected readonly cacheManager: CacheService,
  ) {
    super(db, cacheManager, {
```

```
    table: 'device_locations',
    pk: ['id'],
    class: DeviceLocation,
  });
 }

 async findOneById(id: number): Promise<DeviceLocation> {
   const location = await this.findOne({ id });
   if (!location) {
    throw new NotFoundException(`Device location with ID ${id} not found`);
   }
   return location;
 }

 async findOneByName(name: string): Promise<DeviceLocation | undefined> {
   return this.findOne({ name });
 }
}
```

src/devices/device-locations.controller.ts

```
import {
 Controller,
 Get,
 Param,
 ParseIntPipe,
 UseGuards,
} from '@nestjs/common';
import { ApiTags, ApiOperation, ApiResponse, ApiParam, ApiBearerAuth } from '@nestjs/swagger';
import { DeviceLocation } from './models/device-location.model';
import { DeviceLocationsService } from './device-locations.service';
import { JwtAuthGuard } from '../auth/guards/jwt-auth.guard';
import { RolesGuard } from '../auth/guards/roles.guard';
import { RequiredRole } from '../auth/decorators/required-role.decorator';
import { UserRoles } from '../users/enums/user-roles.enum';

@ApiTags('device-locations')
@ApiBearerAuth()
@Controller('devices-locations')
@UseGuards(JwtAuthGuard, RolesGuard)
export class DeviceLocationsController {
 constructor(private readonly deviceLocationsService: DeviceLocationsService) {}

 @Get()
 @RequiredRole(UserRoles.user)
 @ApiOperation({ summary: 'Get all device locations' })
 @ApiResponse({
  status: 200,
  description: 'Return all device locations',
  type: [DeviceLocation],
 })
 findAll() {
  return this.deviceLocationsService.findAll();
 }

 @Get(':id')
 @RequiredRole(UserRoles.user)
 @ApiOperation({ summary: 'Get a device location by ID' })
 @ApiParam({ name: 'id', description: 'Device location ID' })
 @ApiResponse({
```

```
    status: 200,
    description: 'Return the device location',
    type: DeviceLocation,
  })
  @ApiResponse({ status: 404, description: 'Device location not found' })
  findOne(@Param('id', ParseIntPipe) id: number) {
    return this.deviceLocationsService.findOneById(id);
  }
}
```

src/devices/device-properties.service.ts

```
import { Injectable, NotFoundException, Inject } from '@nestjs/common';
import { Knex } from 'knex';
import { BaseCrudService } from '../common/services/base-crud.service';
import { DeviceProperty } from './models';
import { CacheService } from '../cache/cache.service';
import { CreateDevicePropertyDto } from './dto/create-device-property.dto';
import { UpdateDevicePropertyDto } from './dto/update-device-property.dto';
import { DB_CONNECTION } from '../database/constants/injection-keys';
import { CACHE_SERVICE } from '../cache/constants/injection-keys';

@Injectable()
export class DevicePropertiesService extends BaseCrudService<DeviceProperty> {
  constructor(
    @Inject(DB_CONNECTION)
    protected readonly db: Knex,
    @Inject(CACHE_SERVICE)
    protected readonly cacheManager: CacheService,
  ) {
    super(db, cacheManager, {
      table: 'devices_properties',
      pk: ['id'],
      class: DeviceProperty,
    });
  }

  async create(
    createDevicePropertyDto: CreateDevicePropertyDto,
  ): Promise<DeviceProperty> {
    const deviceProperty = new DeviceProperty();
    Object.assign(deviceProperty, createDevicePropertyDto);
    await this.createOne(deviceProperty);
    return deviceProperty;
  }

  async findByDeviceId(deviceId: string): Promise<DeviceProperty[]> {
    return this.findMany({ device_id: deviceId });
  }

  async findByDeviceIdAndName(
    deviceId: string,
    name: string,
  ): Promise<DeviceProperty> {
    const property = await this.findOne({ device_id: deviceId, name });
    if (!property) {
      throw new NotFoundException(
        `Property ${name} not found for device ${deviceId}`,
      );
    }
    return property;
```

```
  }

  async findOneById(id: number): Promise<DeviceProperty> {
    const deviceProperty = await this.findOne({ id });
    if (!deviceProperty) {
      throw new NotFoundException(`Device property with ID ${id} not found`);
    }
    return deviceProperty;
  }

  async update(
    id: number,
    updateDevicePropertyDto: UpdateDevicePropertyDto,
  ): Promise<DeviceProperty> {
    const deviceProperty = await this.findOneById(id);
    Object.assign(deviceProperty, updateDevicePropertyDto);
    await this.updateOne({ id, ...updateDevicePropertyDto });
    return deviceProperty;
  }

  async remove(id: number): Promise<void> {
    await this.deleteOne({ id });
  }
}
```

Адаптер підключень

src/main.ts

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { MicroserviceOptions, Transport } from '@nestjs/microservices';
import { join } from 'node:path';
import { console } from 'node:inspector';

async function bootstrap() {
  const app = await NestFactory.createMicroservice<MicroserviceOptions>(
    AppModule,
    {
      transport: Transport.GRPC,
      options: {
        package: 'iot',
        protoPath: join(__dirname, 'proto/iot.proto'),
        loader: {
          keepCase: true,
          longs: String,
          defaults: true,
          oneofs: true,
        },
      },
    },
  );
  await app.listen();
}
bootstrap();
```

src/app.service.ts

```
import { Injectable } from '@nestjs/common';

@Injectable()
export class AppService {
  getHello(): string {
    return 'Hello World!';
```

```
  }
}
```

src/app.module.ts

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { IOTConnectionsModule } from './iot-connections/iot-connections.module';

@Module({
  imports: [IOTConnectionsModule],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

src/common/helpers/randInt.ts

```
export function randInt(min: number, max: number): number {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}
```

src/app.controller.ts

```
import { Controller } from '@nestjs/common';
import { AppService } from './app.service';
import { ServerDuplexStream } from '@grpc/grpc-js';
import { GrpcStreamCall } from '@nestjs/microservices';
import { Observable } from 'rxjs';

interface PingRequest {
  message: string;
}

interface PingResponse {
  message: string;
}

@Controller()
export class AppController {
  constructor(private readonly appService: AppService) {}

  @GrpcStreamCall('PingPongService', 'Ping')
  getHello(stream: ServerDuplexStream<PingRequest, PingResponse>) {
    stream.on('data', (request: PingRequest) => {
      console.log('Received ping request:', request);
      const response: PingResponse = {
        message: `Pong: ${request.message}`,
      };
      stream.write(response);
    });

    stream.on('end', () => {
      console.log('Ping stream ended');
    });

    return new Observable();
  }
}
```

src/iot-connections/iot-connections.controller.ts

```
import { Controller } from '@nestjs/common';
import { Observable, Subject } from 'rxjs';
import { map, mergeMap, filter, takeUntil, tap } from 'rxjs/operators';
import {
```

```
  Command,
  CommandResponse,
  CommandType,
  DeviceStatus,
  IOTConnectionsServiceController,
  IOTConnectionsServiceControllerMethods,
  SubscribeQuery,
  SubscriptionStream,
  StreamCommandType,
} from '../proto/iot';
import { IOTSimulationService } from './iot-simulation.service';
import { devicesConfig } from './devices-config';

@Controller()
@IOTConnectionsServiceControllerMethods()
export class IOTConnectionsController
  implements IOTConnectionsServiceController
{
  constructor(private readonly iotSimulationService: IOTSimulationService) {}

  streamEvents(
    request: Observable<SubscribeQuery>,
  ): Observable<SubscriptionStream> {
    return request.pipe(
      mergeMap((query) => this.iotSimulationService.processQuery(query)),
    );
  }

  streamCommands(request: Observable<Command>): Observable<CommandResponse> {
    return request.pipe(
      mergeMap((command) => {
        console.log('Received command:', {
          deviceId: command.deviceId,
          requestId: command.requestId,
          commandType: CommandType[command.commandType],
        });
        return this.iotSimulationService.processCommand(command);
      }),
      tap((response) => {
        console.log('Sending response:', {
          deviceId: response.deviceId,
          requestId: response.requestId,
          success: response.success,
        });
      }),
    );
  }
}
```

src/iot-connections/iot-connections.module.ts

```
import { Module } from '@nestjs/common';
import { IOTConnectionsController } from './iot-connections.controller';
import { IOTSimulationService } from './iot-simulation.service';

@Module({
  controllers: [IOTConnectionsController],
  providers: [IOTSimulationService],
})
export class IOTConnectionsModule {}
```

src/iot-connections/iot-simulation.service.ts

```
import { Injectable, OnModuleInit } from '@nestjs/common';
import { from, Observable, Subject, tap, timer, map, mergeMap } from 'rxjs';
import {
  Command,
  CommandResponse,
  CommandType,
  DeviceEvent,
  DeviceEventType,
  DeviceStatus,
  DeviceType,
  SensorType,
  DeviceInfo,
  SubscribeQuery,
  SubscriptionStream,
  StreamCommandType,
  SensorReading,
} from '../proto/iot';
import { devicesConfig } from './devices-config';
import { randInt } from '../common/helpers/randInt';
import { filter } from 'rxjs/operators';

@Injectable()
export class IOTSimulationService implements OnModuleInit {
  private readonly devicesEvents$: Subject<DeviceEvent> = new Subject();
  private readonly devices: Map<string, DeviceInfo> = new Map();
  private readonly subscriptions: Map<string, Observable<SubscriptionStream>> =
    new Map();

  onModuleInit() {
    this._initSimulation();
  }

  processQuery(query: SubscribeQuery): Observable<SubscriptionStream> {
    if (query.commandType === StreamCommandType.UNSUB_DEVICE_EVENTS) {
      const subStream$ = this.subscriptions.get(query.subId);
      if (!subStream$) {
        console.error('Subscription not found', query.subId);
        return new Observable<SubscriptionStream>();
      }

      this.subscriptions.delete(query.subId);

      return new Observable<SubscriptionStream>();
    }

    return this.initSubscription(query);
  }

  initSubscription(query: SubscribeQuery): Observable<SubscriptionStream> {
    const subStream$ = this.devicesEvents$.asObservable().pipe(
      filter((event: DeviceEvent) => {
        let passesFilter = true;

        if (query.eventTypes) {
          if (query.eventTypes.length === 0) {
            query.eventTypes = Object.values(
              DeviceEventType,
            ) as DeviceEventType[];
          }
```

```
        passesFilter =
          passesFilter && query.eventTypes.includes(event.eventType);
        }

        if (query.deviceId) {
          passesFilter = passesFilter && query.deviceId === event.deviceId;
        }

        return passesFilter;
      }),
      map((event: DeviceEvent) => {
        return {
          event,
          subId: query.subId,
        } as SubscriptionStream;
      }),
    );

    this.subscriptions.set(query.subId, subStream$);

    return subStream$;
  }

  private _initSimulation() {
    from(devicesConfig.devices)
      .pipe(
        tap((deviceInfo: DeviceInfo) => {
          this.devices.set(deviceInfo.deviceId, deviceInfo);
        }),
        mergeMap((device) => this._initDeviceSimulation(device)),
      )
      .subscribe((event: DeviceEvent) => {
        this.devicesEvents$.next(event);
      });
  }

  private _initDeviceSimulation(device: DeviceInfo): Observable<DeviceEvent> {
    const updateInterval = device.reportIntervalMs ?? randInt(1000, 60000);

    return timer(0, updateInterval).pipe(
      map(() => this._generateDeviceEvent(device)),
      tap((event) =>
        console.log('NEW_EVENT', {
          device: event.deviceId,
          ts: new Date(event.timestamp).toISOString(),
        }),
      ),
    );
  }

  processCommand(command: Command): Observable<CommandResponse> {
    const device = this.devices.get(command.deviceId);

    if (!device) {
      return new Observable<CommandResponse>((observer) => {
        observer.next({
          deviceId: command.deviceId,
          requestId: command.requestId,
          success: false,
          timestamp: Date.now(),
          errorReason: `Device with ID ${command.deviceId} not found`,
        });
        observer.complete();
```

```
    });
  }

  switch (command.commandType) {
    case CommandType.ENABLE:
      return this._handleEnableCommand(command, device);
    case CommandType.DISABLE:
      return this._handleDisableCommand(command, device);
    case CommandType.GET_INFO:
      return this._handleGetInfoCommand(command, device);
    case CommandType.GET_CONFIG:
      return this._handleGetConfigCommand(command, device);
    case CommandType.SET_CONFIG:
      return this._handleSetConfigCommand(command, device);
    case CommandType.READ_SENSORS:
      return this._handleReadSensorsCommand(command, device);
    default:
      return new Observable<CommandResponse>((observer) => {
        observer.next({
          deviceId: command.deviceId,
          requestId: command.requestId,
          success: false,
          timestamp: Date.now(),
          errorReason: `Unsupported command type: ${command.commandType}`,
        });
        observer.complete();
      });
  }
 }

 private _handleEnableCommand(
  command: Command,
  device: DeviceInfo,
 ): Observable<CommandResponse> {
  device.deviceStatus = DeviceStatus.ENABLED;

  return new Observable<CommandResponse>((observer) => {
    observer.next({
      deviceId: command.deviceId,
      requestId: command.requestId,
      success: true,
      timestamp: Date.now(),
      deviceStatus: DeviceStatus.ENABLED,
    });
    observer.complete();
  });
 }

 private _handleDisableCommand(
  command: Command,
  device: DeviceInfo,
 ): Observable<CommandResponse> {
  device.deviceStatus = DeviceStatus.DISABLED;

  return new Observable<CommandResponse>((observer) => {
    observer.next({
      deviceId: command.deviceId,
      requestId: command.requestId,
      success: true,
      timestamp: Date.now(),
      deviceStatus: DeviceStatus.DISABLED,
```

```
    });
    observer.complete();
  });
 }

 private _handleGetInfoCommand(
  command: Command,
  device: DeviceInfo,
 ): Observable<CommandResponse> {
  return new Observable<CommandResponse>((observer) => {
    observer.next({
      deviceId: command.deviceId,
      requestId: command.requestId,
      success: true,
      timestamp: Date.now(),
      deviceStatus: device.deviceStatus,
    });
    observer.complete();
  });
 }

 private _handleGetConfigCommand(
  command: Command,
  device: DeviceInfo,
 ): Observable<CommandResponse> {
  return new Observable<CommandResponse>((observer) => {
    const response: CommandResponse = {
      deviceId: command.deviceId,
      requestId: command.requestId,
      success: true,
      timestamp: Date.now(),
    };
    if (device.deviceType === DeviceType.LAMP &&
device.lampConfig) {
      response.lampConfig = device.lampConfig;
    } else if (
      device.deviceType === DeviceType.GARAGE_DOOR &&
      device.garageDoorConfig
    ) {
      response.garageDoorConfig = device.garageDoorConfig;
    } else {
      response.success = false;
      response.errorReason =
        'No configuration available for this device type';
    }

    observer.next(response);
    observer.complete();
  });
 }

 private _handleSetConfigCommand(
  command: Command,
  device: DeviceInfo,
 ): Observable<CommandResponse> {
  if (!command.newConfigData) {
    return new Observable<CommandResponse>((observer) => {
      observer.next({
        deviceId: command.deviceId,
        requestId: command.requestId,
        success: false,
        timestamp: Date.now(),
        errorReason: 'No configuration data provided',
```

```
      });
      observer.complete();
    });
  }

  const response: CommandResponse = {
    deviceId: command.deviceId,
    requestId: command.requestId,
    success: true,
    timestamp: Date.now(),
  };

  if (
    device.deviceType === DeviceType.LAMP &&
    command.newConfigData.lampConfig
  ) {
    device.lampConfig = command.newConfigData.lampConfig;
    response.lampConfig = device.lampConfig;
  } else if (
    device.deviceType === DeviceType.GARAGE_DOOR &&
    command.newConfigData.garageDoorConfig
  ) {
    device.garageDoorConfig = command.newConfigData.garageDoorConfig;
    response.garageDoorConfig = device.garageDoorConfig;
  } else {
    response.success = false;
    response.errorReason = 'Invalid configuration for this device type';
  }

  return new Observable<CommandResponse>((observer) => {
    observer.next(response);
    observer.complete();
  });
}

private _handleReadSensorsCommand(
  command: Command,
  device: DeviceInfo,
): Observable<CommandResponse> {
  if (
    ![
      DeviceType.THERMOSTAT,
      DeviceType.MOTION_SENSOR,
      DeviceType.SMOKE_DETECTOR,
    ].includes(device.deviceType)
  ) {
    return new Observable<CommandResponse>((observer) => {
      observer.next({
        deviceId: command.deviceId,
        requestId: command.requestId,
        success: false,
        timestamp: Date.now(),
        errorReason: 'Device does not have sensors',
      });
      observer.complete();
    });
  }

  const sensorData = this._generateSensorData(device);

  return new Observable<CommandResponse>((observer) => {
    observer.next({
```

```
      deviceId: command.deviceId,
      requestId: command.requestId,
      success: true,
      timestamp: Date.now(),
      sensorData: sensorData as SensorReading,
    });
    observer.complete();
  });
}

private _generateSensorData(device: DeviceInfo) {
  switch (device.deviceType) {
    case DeviceType.THERMOSTAT:
      return {
        metadata: {},
        sensorType: SensorType.TEMPERATURE,
        temperature: {
          celsius: randInt(18, 30) + Math.random(),
        },
      };
    case DeviceType.MOTION_SENSOR:
      return {
        metadata: {},
        sensorType: SensorType.MOTION,
        motion: {
          detected: Math.random() > 0.7,
        },
      };
    case DeviceType.SMOKE_DETECTOR:
      return {
        metadata: {},
        sensorType: SensorType.PROXIMITY,
        proximity: {
          detected: Math.random() > 0.9,
          meters: Math.random() * 10,
        },
      };
    default:
      return null;
  }
}

 private _generateDeviceEvent(device: DeviceInfo): DeviceEvent {
   const eventsMap: Record<DeviceType, () => DeviceEvent> = {
     [DeviceType.LAMP]: () => ({
       deviceId: device.deviceId,
       timestamp: Date.now(),
       eventType: DeviceEventType.STATE_UPDATE,
       stateUpdate: {
         newState: Math.random() > 0.5 ? 'ON' : 'OFF',
         oldState: Math.random() > 0.5 ? 'OFF' : 'ON',
         metadata: {
           brightness: (Math.random() * 100).toFixed(2),
           colorTemperature: randInt(2700, 6500).toString(),
         },
       },
     }),
     [DeviceType.THERMOSTAT]: () => ({
       deviceId: device.deviceId,
       timestamp: Date.now(),
       eventType: DeviceEventType.SENSOR_READING,
       sensorReading: {
         metadata: {},
         sensorType: SensorType.TEMPERATURE,
         temperature: {
```

```
        celsius: randInt(18, 30) + Math.random(),
      },
    },
  }),
  [DeviceType.MOTION_SENSOR]: () => ({
    deviceId: device.deviceId,
    timestamp: Date.now(),
    eventType:
DeviceEventType.SENSOR_READING,
    sensorReading: {
      metadata: {},
      sensorType: SensorType.MOTION,
      motion: {
        detected: Math.random() > 0.7,
      },
    },
  }),
  [DeviceType.SMOKE_DETECTOR]: () => ({
    deviceId: device.deviceId,
    timestamp: Date.now(),
    eventType:
DeviceEventType.SENSOR_READING,
    sensorReading: {
      metadata: {},
      sensorType: SensorType.PROXIMITY,
      proximity: {
        detected: Math.random() > 0.9,
        meters: Math.random() * 10,
      },
    },
  }),
  [DeviceType.GARAGE_DOOR]: () => ({
    deviceId: device.deviceId,
    timestamp: Date.now(),
    eventType: DeviceEventType.STATE_UPDATE,
    stateUpdate: {
      newState: Math.random() > 0.5 ? 'OPEN' :
'CLOSED',
      oldState: Math.random() > 0.5 ? 'CLOSED' :
'OPEN',
      metadata: {
        position: (Math.random() * 100).toFixed(2),
      },
    },
  }),
  [DeviceType.UNRECOGNIZED]: () => ({
    deviceId: device.deviceId,
    timestamp: Date.now(),
    eventType: DeviceEventType.ERROR_ALERT,
    errorAlert: {
      msg: 'Unknown device type',
      code: 'UNRECOGNIZED_DEVICE',
      metadata: {},
    },
  }),
 };

 return eventsMap[device.deviceType]();
 }
}
```

src/iot-connections/devices-config.ts

```
import {
 DeviceInfo,
 DeviceType,
 DeviceState,
 DeviceStatus,
} from 'src/proto/iot';
```

```
type DeviceConfig = {
  devices: DeviceInfo[];
};

export const devicesConfig: DeviceConfig = {
  devices: [
    {
      deviceId: 'lamp-001',
      deviceType: DeviceType.LAMP,
      deviceState: DeviceState.ONLINE,
      deviceStatus: DeviceStatus.ENABLED,
      reportIntervalMs: 20_000,
      lampConfig: {
        brightness: 0.8,
        colorTemperature: 4000,
      },
    },
    {
      deviceId: 'thermostat-001',
      deviceType: DeviceType.THERMOSTAT,
      deviceState: DeviceState.ONLINE,
      deviceStatus: DeviceStatus.ENABLED,
      reportIntervalMs: 15_000,
    },
    {
      deviceId: 'motion-sensor-001',
      deviceType: DeviceType.MOTION_SENSOR,
      deviceState: DeviceState.ONLINE,
      deviceStatus: DeviceStatus.ENABLED,
      reportIntervalMs: 10_000,
    },
    {
      deviceId: 'smoke-detector-001',
      deviceType: DeviceType.SMOKE_DETECTOR,
      deviceState: DeviceState.ONLINE,
      deviceStatus: DeviceStatus.ENABLED,
      reportIntervalMs: 50000,
    },
    {
      deviceId: 'garage-door-001',
      deviceType: DeviceType.GARAGE_DOOR,
      deviceState: DeviceState.ONLINE,
      deviceStatus: DeviceStatus.ENABLED,
      reportIntervalMs: 5000,
      garageDoorConfig: {
        moveDurationMs: 10000,
      },
    },
    {
      deviceId: 'lamp-002',
      deviceType: DeviceType.LAMP,
      deviceState: DeviceState.OFFLINE,
      deviceStatus: DeviceStatus.DISABLED,
      reportIntervalMs: 5000,
      lampConfig: {
        brightness: 0.5,
        colorTemperature: 2700,
      },
    },
    {
      deviceId: 'motion-sensor-002',
      deviceType: DeviceType.MOTION_SENSOR,
      deviceState: DeviceState.MAINTENANCE,
      deviceStatus: DeviceStatus.ENABLED,
      reportIntervalMs: 2000,
    },
  ],
```

```
};
```

Клієнтська частина

src/main.js

```
import { createApp } from 'vue'
import App from './App.vue'
import router from './router'
import Toast from "vue-toastification";
import "vue-toastification/dist/index.css";
import '@mdi/font/css/materialdesignicons.css'

const app = createApp(App)

// Toast notification options
const toastOptions = {
    position: "top-right",
    timeout: 5000,
    closeOnClick: true,
    pauseOnFocusLoss: true,
    pauseOnHover: true,
    draggable: true,
    draggablePercent: 0.6,
    showCloseButtonOnHover: false,
    hideProgressBar: false,
    closeButton: "button",
    icon: true,
    rtl: false
};

app.use(router)
app.use(Toast, toastOptions)
app.mount('#app')
```

src/services/ApiService.js

```
/**
 * API Service for communicating with the IoT Control Service backend
 * This service handles all HTTP requests to the backend API
 */
import authService from './AuthService';

class ApiService {
  constructor() {
    this.baseUrl = process.env.VUE_APP_API_URL || 'http://localhost:3000';
  }

  /**
   * Make a request to the API
   * @param {string} endpoint - API endpoint
   * @param {Object} options - Fetch options
   * @returns {Promise<any>} - Response data
   */
  async request(endpoint, options = {}) {
    const url = `${this.baseUrl}${endpoint}`;

    const headers = {
      'Content-Type': 'application/json',
      ...options.headers
    };

    // Add authorization header if user is authenticated
    if (authService.isAuthenticated()) {
      const authHeader = authService.getAuthHeader();
      Object.assign(headers, authHeader);
    }

    const config = {
```

```
      ...options,
      headers
    };

    try {
      const response = await fetch(url, config);

      if (!response.ok) {
        const error = await response.json();
        throw new Error(error.message || 'API request failed');
      }

      return await response.json();
    } catch (error) {
      console.error('API request error:', error);
      throw error;
    }
  }

  // Device endpoints

  /**
   * Get all devices
   * @param {Object} params - Query parameters
   * @returns {Promise<Array>} - List of devices
   */
  async getDevices(params = {}) {
    const queryParams = new URLSearchParams(params).toString();
    const endpoint = `/devices${queryParams ? `?${queryParams}` : ''}`;
    return this.request(endpoint);
  }

  /**
   * Get a device by ID
   * @param {string} id - Device ID
   * @returns {Promise<Object>} - Device data
   */
  async getDevice(id) {
    return this.request(`/devices/${id}`);
  }

  /**
   * Create a new device
   * @param {Object} deviceData - Device data
   * @returns {Promise<Object>} - Created device
   */
  async createDevice(deviceData) {
    return this.request('/devices', {
      method: 'POST',
      body: JSON.stringify(deviceData)
    });
  }

  /**
   * Update a device
   * @param {string} id - Device ID
   * @param {Object} deviceData - Updated device data
   * @returns {Promise<Object>} - Updated device
   */
  async updateDevice(id, deviceData) {
    return this.request(`/devices/${id}`, {
      method: 'PATCH',
      body: JSON.stringify(deviceData)
    });
  }
```

```
/**
 * Delete a device
 * @param {string} id - Device ID
 * @returns {Promise<Object>} - Response data
 */
async deleteDevice(id) {
  return this.request(`/devices/${id}`, {
    method: 'DELETE'
  });
}

// Device Types endpoints

/**
 * Get all device types
 * @returns {Promise<Array>} - List of device types
 */
async getDeviceTypes() {
  return this.request('/devices-types');
}

/**
 * Get a device type by ID
 * @param {number} id - Device type ID
 * @returns {Promise<Object>} - Device type data
 */
async getDeviceType(id) {
  return this.request(`/devices-types/${id}`);
}

// Device Locations endpoints

/**
 * Get all device locations
 * @returns {Promise<Array>} - List of device locations
 */
async getDeviceLocations() {
  return this.request('/devices-locations');
}

// Device Events endpoints

/**
 * Get device events with optional filtering
 * @param {Object} params - Query parameters for filtering events
 * @param {string} params.search - Search term for filtering events
 * @param {number} params.limit - Maximum number of events to return
 * @param {number} params.offset - Offset for pagination
 * @param {string} params.from - Start date for filtering (ISO format)
 * @param {string} params.to - End date for filtering (ISO format)
 * @param {string} params.device_id - Filter events by device ID
 * @param {string} params.event - Filter events by event type
 * @returns {Promise<Array>} - List of device events
 */
async getDeviceEvents(params = {}) {
  const queryParams = new URLSearchParams(params).toString();
  const endpoint = `/devices-events${queryParams ? `?${queryParams}` : ''}`;
  return this.request(endpoint);
}

/**
```

```
   * Get a device event by ID
   * @param {string} id - Event ID
   * @returns {Promise<Object>} - Device event data
   */
  async getDeviceEvent(id) {
    return this.request(`/devices-events/${id}`);
  }

  // Auth endpoints

  /**
   * Login user
   * @param {Object} credentials - User credentials
   * @param {string} credentials.login - User login
   * @param {string} credentials.password - User password
   * @returns {Promise<Object>} - Authentication data
   */
  async login(credentials) {
    const response = await this.request('/auth/signin', {
      method: 'POST',
      body: JSON.stringify(credentials)
    });

    // Store authentication data
    authService.login(response);

    return response;
  }

  /**
   * Register new user
   * @param {Object} userData - User data
   * @param {string} userData.login - User login
   * @param {string} userData.password - User password
   * @returns {Promise<Object>} - Authentication data
   */
  async signup(userData) {
    const response = await this.request('/auth/signup', {
      method: 'POST',
      body: JSON.stringify(userData)
    });

    // Store authentication data
    authService.login(response);

    return response;
  }

  /**
   * Refresh authentication token
   * @returns {Promise<Object>} - New authentication data
   */
  async refreshToken() {
    if (!authService.isAuthenticated()) {
      throw new Error('Not authenticated');
    }

    const response = await this.request('/auth/refresh', {
      method: 'POST'
    });

    // Update authentication data
    authService.login(response);

    return response;
```

```
  }

  /**
   * Get user profile
   * @returns {Promise<Object>} - User profile data
   */
  async getProfile() {
    return this.request('/auth/profile');
  }

  // Triggers endpoints

  /**
   * Get all triggers
   * @returns {Promise<Array>} - List of triggers
   */
  async getTriggers() {
    return this.request('/triggers');
  }

  /**
   * Get a trigger by ID
   * @param {number} id - Trigger ID
   * @returns {Promise<Object>} - Trigger data
   */
  async getTrigger(id) {
    return this.request(`/triggers/${id}`);
  }

  /**
   * Get triggers for a specific device
   * @param {string} deviceId - Device ID
   * @returns {Promise<Array>} - List of triggers for the device
   */
  async getDeviceTriggers(deviceId) {
    return this.request(`/triggers?device_id=${deviceId}`);
  }

  /**
   * Create a new trigger
   * @param {Object} triggerData - Trigger data
   * @returns {Promise<Object>} - Created trigger
   */
  async createTrigger(triggerData) {
    return this.request('/triggers', {
      method: 'POST',
      body: JSON.stringify(triggerData)
    });
  }

  /**
   * Update a trigger
   * @param {number} id - Trigger ID
   * @param {Object} triggerData - Updated trigger data
   * @returns {Promise<Object>} - Updated trigger
   */
  async updateTrigger(id, triggerData) {
    return this.request(`/triggers/${id}`, {
      method: 'PATCH',
      body: JSON.stringify(triggerData)
    });
  }

  /**
   * Delete a trigger
   * @param {number} id - Trigger ID
```

```
   * @returns {Promise<Object>} - Response data
   */
  async deleteTrigger(id) {
    return this.request(`/triggers/${id}`, {
      method: 'DELETE'
    });
  }

  // Connection Adapters endpoints

  /**
   * Get all connection adapters
   * @returns {Promise<Array>} - List of connection adapters
   */
  async getAdapters() {
    return this.request('/connections-adapters');
  }

  /**
   * Get a connection adapter by ID
   * @param {string} id - Adapter ID
   * @returns {Promise<Object>} - Adapter data
   */
  async getAdapter(id) {
    return this.request(`/connections-adapters/${id}`);
  }

  /**
   * Create a new connection adapter
   * @param {Object} adapterData - Adapter data
   * @returns {Promise<Object>} - Created adapter
   */
  async createAdapter(adapterData) {
    return this.request('/connections-adapters', {
      method: 'POST',
      body: JSON.stringify(adapterData)
    });
  }

  /**
   * Update a connection adapter
   * @param {string} id - Adapter ID
   * @param {Object} adapterData - Updated adapter data
   * @returns {Promise<Object>} - Updated adapter
   */
  async updateAdapter(id, adapterData) {
    return this.request(`/connections-adapters/${id}`, {
      method: 'PATCH',
      body: JSON.stringify(adapterData)
    });
  }

  /**
   * Delete a connection adapter
   * @param {string} id - Adapter ID
   * @returns {Promise<Object>} - Response data
   */
  async deleteAdapter(id) {
    return this.request(`/connections-adapters/${id}`, {
      method: 'DELETE'
    });
  }
}

// Create a singleton instance
const apiService = new ApiService();
```

```
export default apiService;
```

src/services/AuthService.js

```
/**
 * Authentication Service for handling user authentication
 * This service handles login, logout, token storage, and authentication state
 */

class AuthService {
  constructor() {
    this.tokenKey = 'auth_token';
    this.userKey = 'auth_user';
    this.expiresKey = 'auth_expires';
  }

  /**
   * Login user and store authentication data
   * @param {Object} authData - Authentication data from the server
   * @param {string} authData.access_token - JWT token
   * @param {number} authData.expires_in - Token expiration time in seconds
   * @param {Object} authData.user - User data
   */
  login(authData) {
    const { access_token, expires_in, user } = authData;

    // Calculate expiration time
    const expiresAt = new Date().getTime() + expires_in * 1000;

    // Store authentication data in localStorage
    localStorage.setItem(this.tokenKey, access_token);
    localStorage.setItem(this.userKey, JSON.stringify(user));
    localStorage.setItem(this.expiresKey, expiresAt.toString());

    return true;
  }

  /**
   * Logout user and remove authentication data
   */
  logout() {
    localStorage.removeItem(this.tokenKey);
    localStorage.removeItem(this.userKey);
    localStorage.removeItem(this.expiresKey);

    return true;
  }

  /**
   * Check if user is authenticated
   * @returns {boolean} True if user is authenticated, false otherwise
   */
  isAuthenticated() {
    const token = this.getToken();
    const expiresAt = this.getExpiresAt();

    if (!token || !expiresAt) {
      return false;
    }

    // Check if token is expired
    return new Date().getTime() < parseInt(expiresAt);
```

```
  }

  /**
   * Get authentication token
   * @returns {string|null} JWT token or null if not authenticated
   */
  getToken() {
    return localStorage.getItem(this.tokenKey);
  }

  /**
   * Get user data
   * @returns {Object|null} User data or null if not authenticated
   */
  getUser() {
    const userJson = localStorage.getItem(this.userKey);
    return userJson ? JSON.parse(userJson) : null;
  }

  /**
   * Check if the current user is an admin
   * @returns {boolean} True if user is an admin, false otherwise
   */
  isAdmin() {
    const user = this.getUser();
    // Check if the login contains 'admin' or if it's a specific admin account
    return user && (user.login.includes('admin') || user.login === 'test-admin');
  }

  /**
   * Get token expiration time
   * @returns {string|null} Expiration time in milliseconds since epoch or null if not authenticated
   */
  getExpiresAt() {
    return localStorage.getItem(this.expiresKey);
  }

  /**
   * Get authorization header
   * @returns {Object|null} Authorization header or null if not authenticated
   */
  getAuthHeader() {
    const token = this.getToken();
    return token ? { 'Authorization': `Bearer ${token}` } : null;
  }
}

// Create a singleton instance
const authService = new AuthService();

export default authService;
```

src/services/WebSocketService.js

```
/**
 * WebSocket Service for connecting to the IoT Control Service backend
 * This service handles WebSocket connections, reconnection logic, and event handling
 */
import authService from './AuthService';

class WebSocketService {
  constructor() {
    this.socket = null;
    this.triggersSocket = null;
```

```
    this.isConnected = false;
    this.isTriggersConnected = false;
    this.reconnectAttempts = 0;
    this.maxReconnectAttempts = 5;
    this.reconnectTimeout = null;
    this.triggersReconnectTimeout = null;
    this.eventHandlers = new Map();
    this.triggersEventHandlers = new Map();
    this.baseUrl = process.env.VUE_APP_WEBSOCKET_URL || 'ws://localhost:3000/devices/ws';
    this.triggersUrl = process.env.VUE_APP_TRIGGERS_WEBSOCKET_URL || 'ws://localhost:3000/triggers/ws';
  }

  /**
   * Connect to the WebSocket server
   * @returns {Promise} Resolves when connected, rejects on error
   */
  connect() {
    return new Promise((resolve, reject) => {
      if (this.socket && this.isConnected) {
        resolve();
        return;
      }

      // Add authentication token to WebSocket URL if user is authenticated
      let url = this.baseUrl;
      if (authService.isAuthenticated()) {
        const token = authService.getToken();
        // Add token as query parameter
        url = `${url}?token=${token}`;
      }

      this.socket = new WebSocket(url);

      this.socket.onopen = () => {
        console.log('WebSocket connected');
        this.isConnected = true;
        this.reconnectAttempts = 0;
        resolve();
      };

      this.socket.onclose = (event) => {
        console.log(`WebSocket disconnected: ${event.code} ${event.reason}`);
        this.isConnected = false;
        this._attemptReconnect();
      };

      this.socket.onerror = (error) => {
        console.error('WebSocket error:', error);
        reject(error);
      };

      this.socket.onmessage = (event) => {
        try {
          const data = JSON.parse(event.data);
          this._handleMessage(data);
        } catch (error) {
          console.error('Error parsing WebSocket message:', error);
        }
      };
    });
  }
```

```
  /**
   * Disconnect from the WebSocket server
   */
  disconnect() {
    if (this.socket) {
      this.socket.close();
      this.socket = null;
      this.isConnected = false;
    }

    if (this.reconnectTimeout) {
      clearTimeout(this.reconnectTimeout);
      this.reconnectTimeout = null;
    }
  }

  /**
   * Subscribe to device events
   * @param {Object} options Subscription options
   * @param {string} options.deviceId Optional device ID to filter events
   * @param {string} options.event Optional event type to filter events
   * @returns {string} Subscription ID
   */
  subscribeToEvents(options = {}) {
    if (!this.isConnected) {
      throw new Error('WebSocket not connected');
    }

    const subscriptionId = this._generateId();
    const message = {
      event: 'subscribe',
      data: {
        subscriptionId,
        deviceId: options.deviceId || null,
        event: options.event || null,
      }
    };

    this.socket.send(JSON.stringify(message));
    return subscriptionId;
  }

  /**
   * Register an event handler
   * @param {string} eventType Event type to handle
   * @param {Function} handler Handler function
   */
  on(eventType, handler) {
    if (!this.eventHandlers.has(eventType)) {
      this.eventHandlers.set(eventType, []);
    }
    this.eventHandlers.get(eventType).push(handler);
  }

  /**
   * Remove an event handler
   * @param {string} eventType Event type
   * @param {Function} handler Handler function to remove
   */
  off(eventType, handler) {
    if (!this.eventHandlers.has(eventType)) return;

    const handlers = this.eventHandlers.get(eventType);
    const index = handlers.indexOf(handler);
```

```
    if (index !== -1) {
      handlers.splice(index, 1);
    }
  }

  /**
   * Handle incoming WebSocket messages
   * @private
   * @param {Object} data Message data
   */
  _handleMessage(data) {
    // Handle both standard message format and NestJS WsAdapter format
    let eventData = data;

    // If the message has a data property and it's an object, it might be in NestJS WsAdapter format
    if (data.data && typeof data.data === 'object') {
      // Extract the event data from the data property and preserve device_id if it exists
      eventData = {
        ...data.data,
        event: data.event,
        // Preserve device_id from the original data if it exists and is not in data.data
        device_id: data.data.device_id || data.device_id,
        // Ensure data is properly set
        data: data.data.data || data.data
      };
    }

    // Emit the event to all registered handlers
    if (this.eventHandlers.has('message')) {
      this.eventHandlers.get('message').forEach(handler => handler(eventData));
    }

    // Emit specific event type if available
    if (eventData.event && this.eventHandlers.has(eventData.event)) {
      this.eventHandlers.get(eventData.event).forEach(handler => handler(eventData));
    }
  }

  /**
   * Attempt to reconnect to the WebSocket server
   * @private
   */
  _attemptReconnect() {
    if (this.reconnectAttempts >= this.maxReconnectAttempts) {
      console.error('Maximum reconnection attempts reached');
      return;
    }

    this.reconnectAttempts++;
    const delay = Math.min(1000 * Math.pow(2, this.reconnectAttempts), 30000);

    console.log(`Attempting to reconnect in ${delay}ms (attempt ${this.reconnectAttempts}/${this.maxReconnectAttempts})`);

    this.reconnectTimeout = setTimeout(() => {
      this.connect().catch(error => {
        console.error('Reconnection failed:', error);
      });
    }, delay);
  }
```

```
  /**
   * Connect to the triggers WebSocket server
   * @returns {Promise} Resolves when connected, rejects on error
   */
  connectToTriggers() {
    return new Promise((resolve, reject) => {
      if (this.triggersSocket && this.isTriggersConnected) {
        resolve();
        return;
      }

      // Add authentication token to WebSocket URL if user is authenticated
      let url = this.triggersUrl;
      if (authService.isAuthenticated()) {
        const token = authService.getToken();
        // Add token as query parameter
        url = `${url}?token=${token}`;
      }

      this.triggersSocket = new WebSocket(url);

      this.triggersSocket.onopen = () => {
        console.log('Triggers WebSocket connected');
        this.isTriggersConnected = true;
        this.reconnectAttempts = 0;
        resolve();
      };

      this.triggersSocket.onclose = (event) => {
        console.log(`Triggers WebSocket disconnected: ${event.code} ${event.reason}`);
        this.isTriggersConnected = false;
        this._attemptTriggersReconnect();
      };

      this.triggersSocket.onerror = (error) => {
        console.error('Triggers WebSocket error:', error);
        reject(error);
      };

      this.triggersSocket.onmessage = (event) => {
        try {
          const data = JSON.parse(event.data);
          this._handleTriggersMessage(data);
        } catch (error) {
          console.error('Error parsing Triggers WebSocket message:', error);
        }
      };
    });
  }

  /**
   * Disconnect from the triggers WebSocket server
   */
  disconnectFromTriggers() {
    if (this.triggersSocket) {
      this.triggersSocket.close();
      this.triggersSocket = null;
      this.isTriggersConnected = false;
    }

    if (this.triggersReconnectTimeout) {
      clearTimeout(this.triggersReconnectTimeout);
      this.triggersReconnectTimeout = null;
```

```
  }
 }

 /**
  * Subscribe to trigger events
  * @param {Object} options Subscription options
  * @param {string} options.deviceId Optional device ID to filter events
  * @returns {string} Subscription ID
  */
 subscribeToTriggers(options = {}) {
  if (!this.isTriggersConnected) {
   throw new Error('Triggers WebSocket not connected');
  }

  const subscriptionId = this._generateId();
  const message = {
   event: 'subscribe',
   data: {
    subscriptionId,
    deviceId: options.deviceId || null
   }
  };

  this.triggersSocket.send(JSON.stringify(message));
  return subscriptionId;
 }

 /**
  * Register a trigger event handler
  * @param {string} eventType Event type to handle
  * @param {Function} handler Handler function
  */
 onTrigger(eventType, handler) {
  if (!this.triggersEventHandlers.has(eventType)) {
   this.triggersEventHandlers.set(eventType, []);
  }

this.triggersEventHandlers.get(eventType).push(handler);
 }

 /**
  * Remove a trigger event handler
  * @param {string} eventType Event type
  * @param {Function} handler Handler function to remove
  */
 offTrigger(eventType, handler) {
  if (!this.triggersEventHandlers.has(eventType)) return;

  const handlers = this.triggersEventHandlers.get(eventType);
  const index = handlers.indexOf(handler);

  if (index !== -1) {
   handlers.splice(index, 1);
  }
 }

 /**
  * Handle incoming trigger WebSocket messages
  * @private
  * @param {Object} data Message data
  */
 _handleTriggersMessage(data) {
  // Emit the event to all registered handlers
  if (this.triggersEventHandlers.has('message')) {
```

```
this.triggersEventHandlers.get('message').forEach(handler => handler(data));
    }

    // Emit specific event type if available
    if (data.event && this.triggersEventHandlers.has(data.event)) {
this.triggersEventHandlers.get(data.event).forEach(handler => handler(data));
    }
  }

  /**
   * Attempt to reconnect to the triggers WebSocket server
   * @private
   */
  _attemptTriggersReconnect() {
    if (this.reconnectAttempts >= this.maxReconnectAttempts) {
      console.error('Maximum reconnection attempts reached for triggers WebSocket');
      return;
    }

    this.reconnectAttempts++;
    const delay = Math.min(1000 * Math.pow(2, this.reconnectAttempts), 30000);

    console.log(`Attempting to reconnect to triggers WebSocket in ${delay}ms (attempt ${this.reconnectAttempts}/${this.maxReconnectAttempts})`);

    this.triggersReconnectTimeout = setTimeout(() => {
      this.connectToTriggers().catch(error => {
        console.error('Triggers WebSocket reconnection failed:', error);
      });
    }, delay);
  }

  /**
   * Generate a random ID for subscriptions
   * @private
   * @returns {string} Random ID
   */
  _generateId() {
    return Math.random().toString(36).substring(2, 15) + Math.random().toString(36).substring(2, 15);
  }
}

// Create a singleton instance
const webSocketService = new WebSocketService();

export default webSocketService;
```

src/router/index.js

```
import { createRouter, createWebHistory } from 'vue-router'
import authService from '../services/AuthService'

// Use dynamic imports for code splitting
const HomeView = () => import(/* webpackChunkName: "home" */ '../views/HomeView.vue')
const DevicesView = () => import(/* webpackChunkName: "devices" */ '../views/DevicesView.vue')
const EventsView = () => import(/* webpackChunkName: "Події» */ '../views/EventsView.vue')
```

```
const TriggersView = () => import(/* webpackChunkName: "triggers" */ '../views/TriggersView.vue')
const AdaptersView = () => import(/* webpackChunkName: "adapters" */ '../views/AdaptersView.vue')
const LoginView = () => import(/* webpackChunkName: "auth" */ '../views/LoginView.vue')
const SignupView = () => import(/* webpackChunkName: "auth" */ '../views/SignupView.vue')

const routes = [
  {
    path: '/',
    name: 'home',
    component: HomeView,
    meta: { requiresAuth: true }
  },
  {
    path: '/devices',
    name: 'devices',
    component: DevicesView,
    meta: { requiresAuth: true }
  },
  {
    path: '/events',
    name: 'events',
    component: EventsView,
    meta: { requiresAuth: true }
  },
  {
    path: '/triggers',
    name: 'triggers',
    component: TriggersView,
    meta: { requiresAuth: true }
  },
  {
    path: '/adapters',
    name: 'adapters',
    component: AdaptersView,
    meta: { requiresAuth: true }
  },
  {
    path: '/login',
    name: 'login',
    component: LoginView,
    meta: { guest: true }
  },
  {
    path: '/signup',
    name: 'signup',
    component: SignupView,
    meta: { guest: true }
  }
]

const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes
})

// Navigation guard to check authentication
router.beforeEach((to, from, next) => {
  const requiresAuth = to.matched.some(record => record.meta.requiresAuth)
  const isGuestRoute = to.matched.some(record => record.meta.guest)
  const isAuthenticated = authService.isAuthenticated()
```

```
  // If route requires auth and user is not authenticated, redirect to login
  if (requiresAuth && !isAuthenticated) {
    next({
      path: '/login',
      query: { redirect: to.fullPath }
    })
  }
  // If guest route and user is authenticated, redirect to home
  else if (isGuestRoute && isAuthenticated) {
    next({ path: '/' })
  }
  // Otherwise, proceed as normal
  else {
    next()
  }
})


export default router;
```