НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут атомної та теплової енергетики

Кафедра цифрових технологій в енергетиці

«До захисту допущено»
Завідувач кафедри
_____ Наталія АУШЕВА
“___”_____2025 р.

Дипломна робота
на здобуття ступеня бакалавр

За освітньою програмою “Цифрові технології в енергетиці”

Спеціальності 122 “Комп’ютерні науки”

на тему: “IoT-платформа для зберігання та динамічної обробки телеметричних даних”

Виконав: студент 4 курсу, групи ТР-13
Мазуренко Нікіта Володимирович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник асистент Микита ГОЛОВАКІН _____
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Рецензент Доктор філософії, асист. Ольга ВЛАСЕНКО _____
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Н.контроль асистент Олександр ВОЛКОВ _____
(посада, ім’я, ПРІЗВИЩЕ) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ - 2025

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

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Рівень вищої освіти – перший (бакалаврський)
спеціальність 122 “Комп’ютерні науки”
Освітньо-професійна програма “Цифрові технології в енергетиці”

ЗАТВЕРДЖУЮ
Завідувач кафедри ЦТЕ
_____ Наталія АУШЕВА
(підпис)
“___” _____2025 р.

ЗАВДАННЯ
на дипломну роботу студенту

_____ Мазуренку Нікіті Володимировичу_____
(прізвище, ім’я, по батькові)

1. Тема роботи “IoT-платформа для зберігання та динамічної обробки телеметричних даних”
Науковий керівник Головакін Микита Андрійович, асистент
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від “ 02 ” червня 2025 року № 1875-с

2. Термін подання студентом роботи 09.06.2025

3. Вихідні дані до роботи мова програмування Java, фреймворк Spring, СУБД PostgreSQL, модель акторів Akka

4. Перелік питань, які потрібно розробити _____

1) провести аналіз серед наявних аналогів
2) визначити головні функції IoT-платформи, що обробляє телеметричні дані
3) аргументувати вибрані інструменти, алгоритми та технології для реалізації платформи
4) побудувати архітектуру програмного забезпечення та бази даних
5) створити прикладний програмний веб-інтерфейс

6) представити процес застосування веб-інтерфейсу

5. Орієнтований перелік ілюстративного матеріалу схема, що показує роботу моделі акторів Akka, схема мікросервісної архітектури, скріншоти роботи користувачів з веб-застосунком.

6. Дата видачі завдання 13.09.2024

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Вибір теми роботи		
2.	Аналіз методів та засобів розв’язання задачі		
3.	Розробка архітектури та загальної структури системи		
4.	Розробка окремих підсистем		
5.	Програмна реалізація системи		
6.	Оформлення пояснювальної записки		
7.	Захист програмного забезпечення		
8.	Передзахист		
9.	Захист		

Студент _____ Нікіта МАЗУРЕНКО
(підпис) (прізвище та ініціали)

Керівник _____ Микита ГОЛОВАКІН
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дипломна робота виконана на 64 сторінках, містить 28 ілюстрацій, 1 додаток, 17 джерел в переліку посилань.

Мета роботи – створення IoT-платформи для зберігання та динамічної обробки телеметричних даних.

Методи та засоби: мікросервісна архітектура, мова програмування Java, веб-фреймворк Spring, система управління базами даних PostgreSQL, модель акторів Akka.

Результат – IoT-платформа, що дозволяє користувачам під'єднувати пристрої, переглядати збережені телеметричні дані, а також налаштовувати обробку вхідних та вихідних повідомлень.

Ключові слова: ІОТ ПЛАТФОРМА, ОБРОБКА ТЕЛЕМЕТРИЧНИХ ДАНИХ, ЗБЕРІГАННЯ ТЕЛЕМЕТРИЧНИХ ДАНИХ.

ABSTRACT

The thesis is made on 64 pages, contains 28 illustrations, 1 appendix, 17 sources in the list of references.

The purpose of the work is to create an IoT platform for storing and dynamic processing of telemetry data.

Methods and tools: microservice architecture, Java programming language, Spring web framework, PostgreSQL database management system, Akka actor model.

The result is an IoT platform that allows users to connect devices, view stored telemetry data, and customize the processing of incoming and outgoing messages.

Keywords: IOT PLATFORM, TELEMETRY DATA PROCESSING, TELEMETRY DATA STORAGE.

ЗМІСТ

ВСТУП 7
ПЕРЕЛІК СКОРОЧЕНИХ УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ 8
1 ПОСТАНОВКА ЗАДАЧІ IOT-ПЛАТФОРМИ ДЛЯ ЗБЕРІГАННЯ ТА ДИНАМІЧНОЇ ОБРОБКИ ТЕЛЕМЕТРИЧНИХ ДАНИХ 9
1.1 Постановка прикладної задачі 9
1.2 Огляд методів, необхідних для розв'язання 10
1.3 Огляд програмних засобів, необхідних для розв'язання 11
1.4 Порівняння існуючих систем 13
1.4.1 AWS IoT Core 13
1.4.2 ThingsBoard 14
2 ЗАСОБИ РОЗРОБКИ ПЛАТФОРМИ ІНТЕРНЕТ РЕЧЕЙ 16
2.1 Мова програмування Java 16
2.2 Фреймворк Spring Boot 17
2.3 Модель акторів Akka 18
2.4 Apache Kafka для обміну повідомленнями 20
2.5 PostgreSQL як система управління базами даних 21
2.6 GraalVM для виконання JavaScript скриптів 23
2.7 Docker та Kubernetes для контейнеризації 23
2.8 Keycloak для управління ідентифікацією та доступом 25
3 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ 27
3.1 Архітектура IoT платформи 27
3.2 Сервіс ланцюжків правил 28
3.2.1 Основні функції сервісу ланцюжків правил 29
3.2.2 Використання моделі акторів Akka для побудови сервісу 29
3.2.3 Переваги та недоліки архітектури на базі моделі акторів 30
3.2.4 Роль використання Apache Kafka 31
3.3 Сервіс для підключення пристроїв 32
3.3.1 Архітектурна концепція та функціональне призначення 33

3.3.2 Протоколи інтеграції пристроїв 34
3.3.3 Інтеграція з Kafka та потокова обробка даних 34
3.3.4 Керування життєвим циклом підключень 35
3.3.5 Розподіл навантаження та масштабованість 35
3.3.6 Механізм обробки відправки повідомлень до пристроїв 35
3.4 Центральний сервіс 36
3.4.1 Архітектура та компоненти центрального сервісу 36
3.4.2 Інтеграційні механізми центрального сервісу 37
4 РОБОТА КОРИСТУВАЧА З ПРОГРАМНОЮ СИСТЕМОЮ 38
4.1 Робота з пристроями 38
4.2 Під'єднання пристроїв 41
4.3 Розділ ланцюжків правил 44
4.3.1 Вершина трансформування 46
4.3.2 Вершина фільтрування 47
4.3.3 Вершина для зберігання телеметричних даних 48
4.3.4 Вершина для надсилання повідомлень пристроям 49
4.4 Тестування навантаження 50
ВИСНОВКИ 52
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 53
ДОДАТОК А 55

ВСТУП

Сьогодення неможливо уявити без використання інтернет речей. Різні пристрої оточують нас всюди: починаючи з систем опалення, вентиляції та кондиціювання повітря, закінчуючи розумними годівницями для тварин. Зі зростанням кількості IoT-обладнання зростає навантаження на хмарні сервери, які приймають дані з цих апаратів, а також обсяг інформації, яку потрібно відфільтрувати, трансформувати та зберегти для подальшої візуалізації користувачеві. Іншою проблемою є негнучкість багатьох IoT-хмарних систем: щоб додати, видалити або змінити логіку обробки повідомлень з пристроїв, часто доводиться проходити всі етапи неперервної інтеграції, розгортання та постачання програмного продукту. Окрім того, складною задачею є утримування балансу між гнучкістю системи та обробною здатністю.

Існує багато способів вирішення проблеми недостатньої динамічності обробки телеметричних даних, зберігаючи високий рівень пропускної здатності. Одним з найпопулярніших варіантів є використання ланцюгів правил. Концепція дозволяє визначати довільну послідовність операцій обробки даних за допомогою набору вершин, кожна з яких може мати необмежену кількість вхідних і вихідних зв’язків з іншими вершинами. Ланцюги правил забезпечують миттєву реакцію на події, що виникають у реальному часі, через що здобули визнання у таких IoT проектах, як: Node-RED, ThingsBoard, AWS IoT Rules Engine.

Метою даної роботи є розробка гнучкої хмарної IoT-системи, яка здатна отримувати, зберігати, фільтрувати та трансформувати повідомлення, а також відправляти їх пристроям. Платформа здатна динамічно обробляти повідомлення та горизонтально масштабуватись за рахунок використання мікросервісної архітектури.

ПЕРЕЛІК СКОРОЧЕНИХ УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

API – Application Programming Interface – Інтерфейс програмування додатків

IT – Information technology – Інформаційні технології

IoT – Internet Of Things – Інтернет Речей

MQTT – Message Queuing Telemetry Transport – Транспорт телеметрії для обміну повідомленнями

СУБД – Система управління базами даних

1 ПОСТАНОВКА ЗАДАЧІ IOT-ПЛАТФОРМИ ДЛЯ ЗБЕРІГАННЯ ТА ДИНАМІЧНОЇ ОБРОБКИ ТЕЛЕМЕТРИЧНИХ ДАНИХ

Даний розділ присвячений аналізу предметної області та формулюванню вимог до системи управління правилами обробки IoT даних. Метою розділу є обґрунтування актуальності розробки, визначення функціональних та нефункціональних вимог до системи, а також дослідження існуючих підходів та технологій для їх реалізації.

1.1 Постановка прикладної задачі

У сучасному світі Інтернету речей (IoT) зростає потреба в ефективних системах обробки та управління даними від різноманітних пристроїв. Підприємства та організації стикаються з викликами інтеграції великої кількості IoT пристроїв, які генерують потоки телеметричних даних у режимі реального часу. Традиційні підходи до обробки таких даних часто виявляються недостатньо гнучкими та масштабованими для задоволення бізнес вимог.

Основною проблемою є необхідність створення універсальної платформи, яка б дозволила динамічно налаштовувати логіку обробки даних від IoT пристроїв без необхідності перекомпіляції та перезапуску системи. Існуючі рішення часто обмежені фіксованою логікою обробки або потребують значних технічних знань для налаштування правил обробки даних.

Актуальність розробки полягає в тому, що сучасні IoT системи потребують можливості швидкого реагування на зміни в бізнес-логіці, динамічного налаштування правил обробки даних та забезпечення надійної доставки команд до пристроїв. Особливо важливим є забезпечення можливості обробки даних у режимі реального часу з мінімальними затримками.

Метою дипломної роботи є розробка системи управління правилами обробки IoT даних, яка забезпечить гнучке налаштування логіки обробки телеметричних даних, їх трансформацію та маршрутизацію. Система повинна підтримувати інтеграцію з різними протоколами зв'язку, зокрема MQTT, та забезпечувати можливість виконання користувацьких скриптів для обробки даних.

Для досягнення поставленої мети необхідно розробити архітектуру системи правил, що дозволить створювати гнучкі ланцюги обробки телеметричних даних. Також реалізувати механізм виконання JavaScript скриптів для трансформації та фільтрації даних. Важливо забезпечити інтеграцію з MQTT брокером для отримання даних від IoT пристроїв та створити систему зберігання телеметричних даних з можливістю масштабування. Реалізація механізму відправки команд до пристроїв через downlink канали є важливою складовою будь-якої IoT системи.

Розроблювана система повинна забезпечувати високу продуктивність обробки великих обсягів даних, підтримувати розподілену архітектуру та надавати зручний інтерфейс для налаштування правил обробки. Особлива увага приділяється забезпеченню надійності доставки даних та можливості відновлення після збоїв.

1.2 Огляд методів, необхідних для розв'язання

Для розв'язання поставленої задачі створення системи управління правилами обробки IoT даних необхідно застосування комплексу сучасних методів та підходів до проектування розподілених систем.

Одним з ключових методів, що використовується в роботі, є акторна модель обчислень. Цей підхід дозволяє створювати високопродуктивні та відмовостійкі системи шляхом інкапсуляції стану та поведінки в незалежних акторах, які взаємодіють виключно через обмін повідомленнями. Акторна модель особливо ефективна для обробки великих потоків даних у режимі реального часу, оскільки забезпечує природне розпаралелювання обчислень та ізоляцію помилок.

Метод асинхронної обробки подій є фундаментальним для забезпечення відгуку системи та її масштабованості. Асинхронний підхід дозволяє системі продовжувати обробку нових запитів, не очікуючи завершення попередніх операцій, що критично важливо для IoT систем з високою інтенсивністю надходження даних.

Мікросервісна архітектура застосовується для забезпечення модульності та незалежного розгортання компонентів системи. Цей метод дозволяє розділити систему на окремі сервіси, кожен з яких відповідає за конкретну функціональність, що спрощує розробку, тестування та підтримку системи.

Потокова обробка даних є ключовим методом для роботи з безперервними потоками телеметричних даних від IoT пристроїв. Цей підхід дозволяє обробляти дані по мірі їх надходження, а не накопичувати їх для пакетної обробки, що забезпечує мінімальні затримки та актуальність інформації.

Метод publish-subscribe використовується для забезпечення слабкого зв'язку між компонентами системи та ефективної доставки повідомлень. Цей підхід дозволяє створювати гнучкі архітектури, де продуценти та споживачі даних не залежать безпосередньо один від одного.

1.3 Огляд програмних засобів, необхідних для розв'язання

Для реалізації системи управління правилами обробки IoT даних обрано комплекс сучасних програмних засобів, які забезпечують необхідну функціональність та продуктивність.

Akka є основою для реалізації акторної моделі в системі. Вона надає потужні засоби для створення відмовостійких розподілених систем, включаючи механізми супервізії акторів, кластеризації та персистентності стану. Використання Akka дозволяє ефективно обробляти великі обсяги повідомлень з мінімальними затримками та забезпечує горизонтальне масштабування системи.

Apache Kafka використовується як центральна платформа для обміну повідомленнями між мікросервісами. Kafka забезпечує високу пропускну здатність, надійність доставки повідомлень та можливість відтворення подій. Розподілена архітектура Kafka дозволяє системі витримувати високі навантаження та забезпечує відмовостійкість через реплікацію даних.

Spring Boot обрано як основний фреймворк для розробки мікросервісів завдяки його простоті налаштування, багатому екосистемі та вбудованим засобам моніторингу. Spring Boot забезпечує швидкий старт розробки та інтеграцію з іншими компонентами Spring екосистеми, включаючи Spring Data для роботи з базами даних та Spring Kafka для інтеграції з Apache Kafka.

PostgreSQL використовується як основна реляційна база даних для зберігання конфігурацій правил, метаданих пристроїв та телеметричних даних. Вибір PostgreSQL обумовлений його надійністю, підтримкою JSONB типу даних для гнучкого зберігання структурованих даних та потужними можливостями індексування для забезпечення високої продуктивності запитів.

GraalVM JavaScript Engine застосовується для виконання користувацьких скриптів обробки даних. GraalVM забезпечує високу продуктивність виконання JavaScript коду та безпечну ізоляцію скриптів, що критично важливо для системи, яка виконує користувацький код у виробничому середовищі.

Docker та Kubernetes використовуються для контейнеризації та оркестрації мікросервісів. Docker забезпечує консистентність середовища виконання на різних етапах розробки та експлуатації, а Kubernetes надає засоби для автоматичного масштабування, балансування навантаження та управління життєвим циклом контейнерів.

Hibernate ORM використовується для об'єктно-реляційного відображення та спрощення роботи з базою даних. Hibernate забезпечує ефективне управління з'єднаннями з базою даних, кешування та оптимізацію SQL запитів.

Maven застосовується як система збірки проекту та управління залежностями, що забезпечує консистентність збірки на різних середовищах та спрощує управління версіями бібліотек.

1.4 Порівняння існуючих систем

IoT хмарні платформи виникли як ключовий інструмент для централізованого керування пристроями, збору, зберігання, обробки й аналізу даних у режимі реального часу. Їх основна мета — спростити розробку та впровадження IoT-рішень, забезпечивши розробникам і компаніям готову інфраструктуру. Розглянемо найбільш популярні проекти та їх можливості, недоліки та переваги.

1.4.1 AWS IoT Core

AWS IoT Core — це керований хмарний сервіс, який дозволяє підключеним пристроям безпечно взаємодіяти з хмарними програмами та іншими пристроями (рисунок 1.1). Він підтримує різні протоколи зв'язку, такі як MQTT, HTTPS і LoRaWAN. Користувач має змогу конфігурувати правила обробки даних, логування, виклик в інші системи. [1].

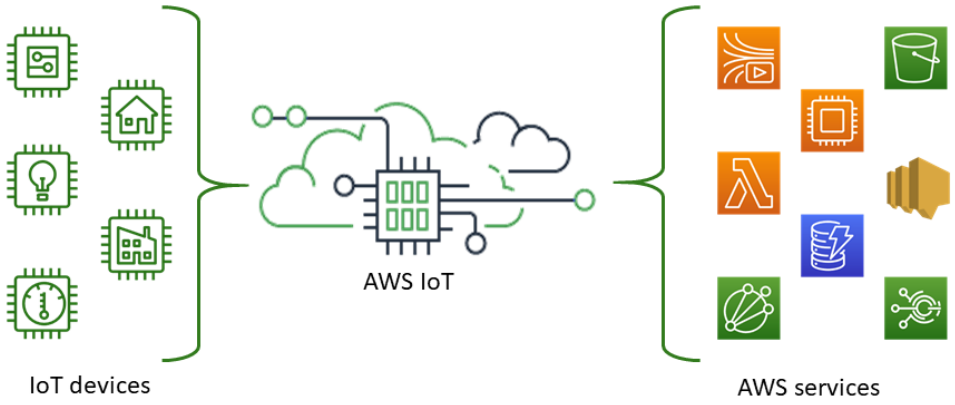


Рисунок 1.1 – Загальна архітектура AWS IoT Core платформи

Основним механізмом, який дозволяє фільтрувати, трансформувати та направляти повідомлення з MQTT брокера до інших сервісів є AWS IoT Rules Engine [1].

Центральним компонентом системи є правила (Rules), які підтримують наступну функціональність:

- обробляти повідомлення з великої кількості пристроїв за допомогою Amazon Kinesis;
- надсилати дані повідомлення до веб-програми або сервісу;
- викликати лямбда-функцію для вилучення даних;
- доповнювати або фільтрувати дані, отримані з пристрою;
- записувати дані, до бази даних Amazon DynamoDB.

1.4.2 ThingsBoard

ThingsBoard - це IoT-платформа з відкритим вихідним кодом, яка дозволяє швидко розробляти, керувати та масштабувати IoT-проекти. Головною метою платформи є надання готового хмарного або локального IoT рішення для широкого списку потреб клієнтів [2].

Сервіс має велику кількість інтеграцій для підключення девайсів та сторонніх систем, а саме: HTTP, CoAp, MQTT, Kafka, інші (рисунок 1.2).

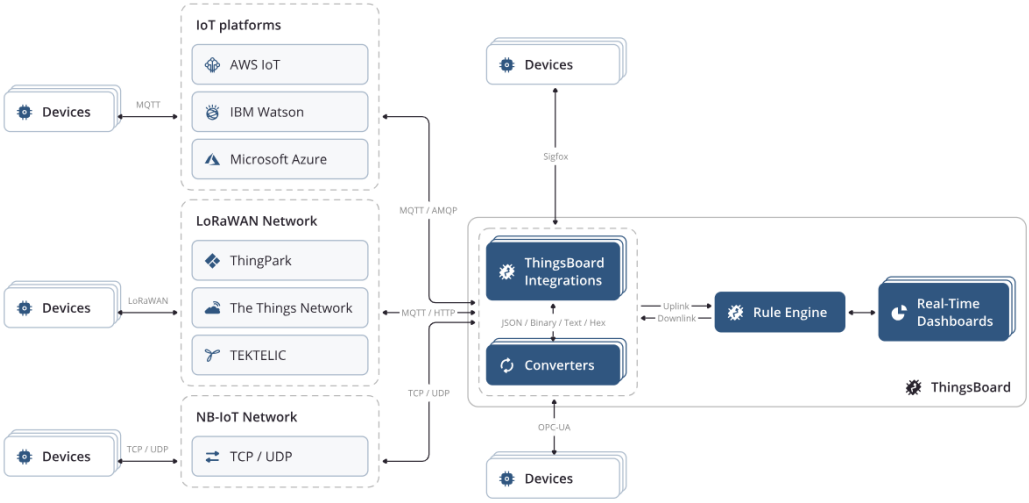


Рисунок 1.2 – Архітектура інтеграції ThingsBoard із приладами різних типів

Як тільки повідомлення надходить із зовнішньої платформи до ThingsBoard Integration, викликається конвертер даних, створений користувачем, щоб витягти підмножину значущої інформації з вхідного повідомлення. Після, трансформоване повідомлення перенаправляється в Rule Engine для подальшої обробки, який у свою чергу також може надсилати вихідне повідомлення для пристроїв та зовнішніх систем.

Як і AWS IoT платформа, ThingsBoard має свою функціональність для обробки правил. Rule Engine – це простий у використанні фреймворк для створення робочих процесів на основі подій. Існує 3 основні компоненти [3]:

- повідомлення. Це можуть бути вхідні дані від пристроїв REST API, RPC;

- вузол правила є функцією, яка виконується над вхідним повідомленням. Існує багато різних типів вузлів, які можуть фільтрувати, трансформувати або виконувати певні дії над вхідним повідомленням;

- ланцюжок правил – вузли пов'язані один з одним відносинами (рисунок 1.3).

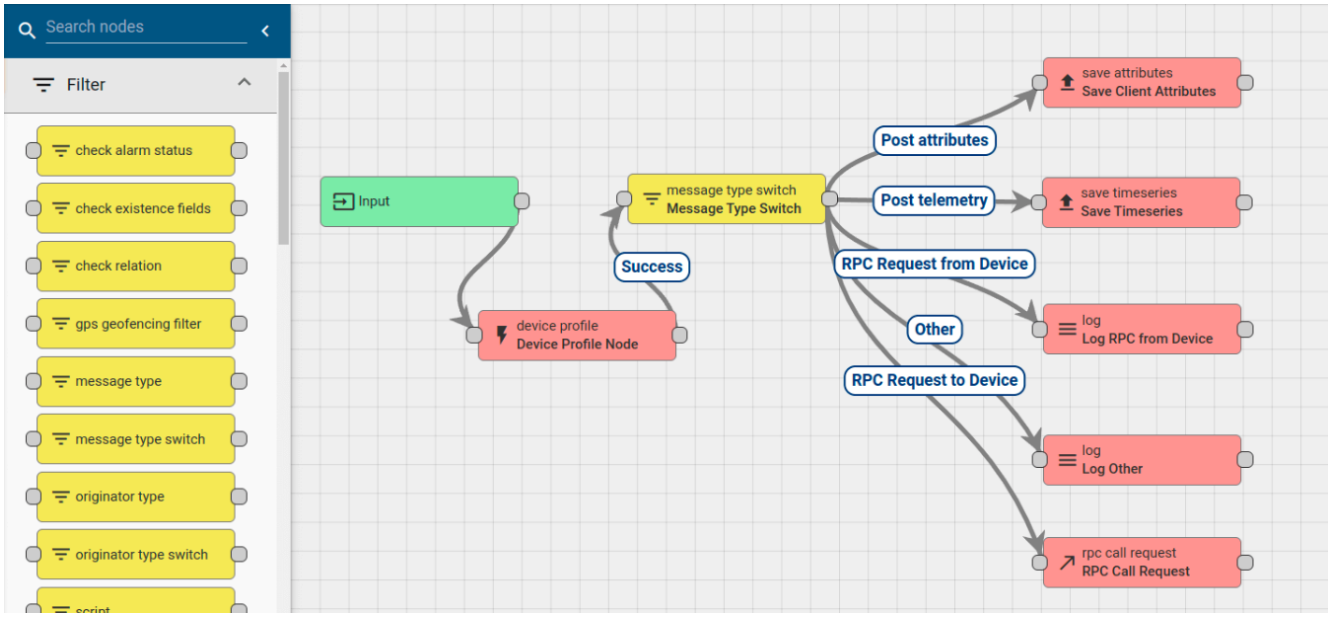


Рисунок 1.3 – Приклад ланцюжка правил

Платформа має велику кількість вузлів-обробників, серед них можна виділити основні типи: фільтраційні, модифікаційні, агрегаційні та вершини, які інтегруються з зовнішніми системами.

2 ЗАСОБИ РОЗРОБКИ ПЛАТФОРМИ ІНТЕРНЕТ РЕЧЕЙ

Важливою частиною будь-якої системи є технології, які використовуються для її побудови. Підбір правильних інструментів потребує експертизу та досвід, оскільки неправильний вибір може викликати великі негативні наслідки та збитки, особливо для великих проектів та компаній. Важливо утримувати баланс між популярністю, новизною та надійністю бібліотек та фреймворків.

2.1 Мова програмування Java

Під час вибору мови програмування для IoT проекту важливими складовими є надійність, актуальність, безпека, велика спільнота розробників та бібліотек інтернет речей. Java об’єднує в собі всі ці особливості.

Java – це широко використовувана мова програмування для створення веб-додатків. Вона є популярним вибором серед розробників вже більше двох десятиліть, і сьогодні використовуються мільйони Java-додатків. Java є багатоплатформною, об'єктно-орієнтованою та мережево-орієнтованою мовою. Це швидка, безпечна, надійна мова програмування для програмування всього, від мобільних додатків та корпоративного програмного забезпечення до додатків для роботи з великими даними та серверних технологій [4].

Окрім того, мова не стоїть на місці та стабільно підтримується. Оновлення виходять згідно з чітко визначеним шестимісячним графіком релізів (від 2017 року, з Java 9). Це означає, що нова версія Java з’являється двічі на рік — у березні та у вересні.

Останні версії мови включають новий механізм вирішення блокуючих операцій – віртуальні потоки. У порівнянні з традиційними потоками, які працюють напряму з потоками операційної системи, віртуальні потоки забезпечують значне зменшення часу виконання, особливо в ситуаціях з високим навантаженням [5]. Це демонструє ефективність і швидкість віртуальних потоків в

обробці великої кількості паралельних завдань, що є критичним для застосунків інтернет речей, які можуть одночасно оброблювати десятки тисяч повідомлень (рисунок 2.1).

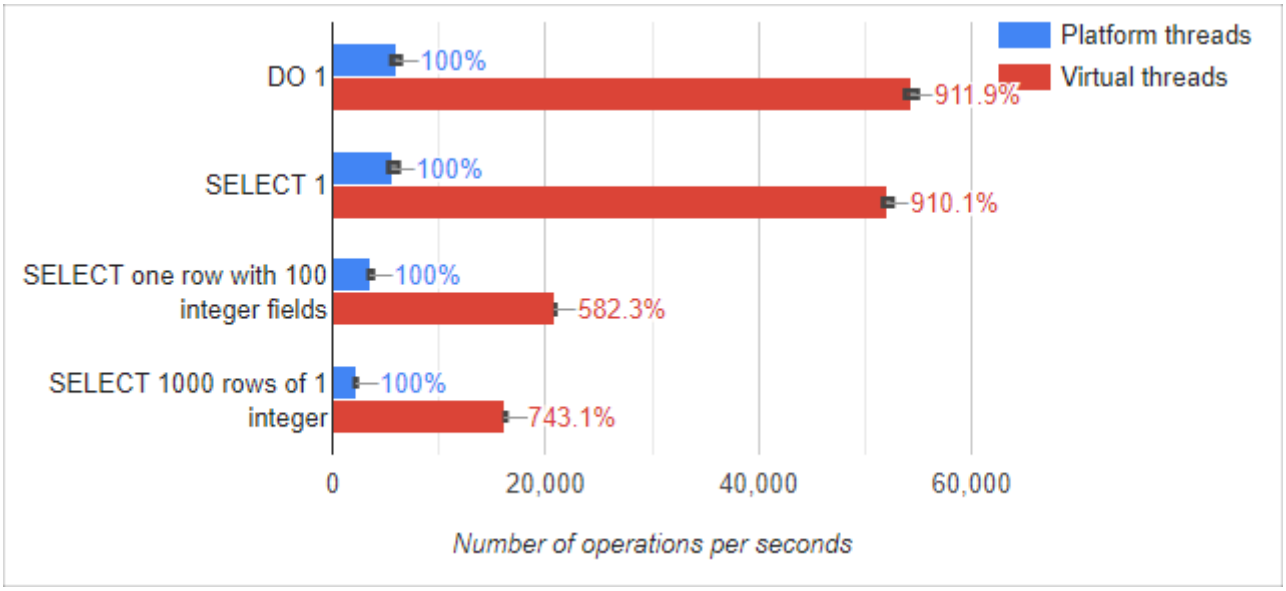


Рисунок 2.1 – Пропускна здатність виконання JDBC запитів у MariaDB з використанням традиційних та віртуальних потоків [6]

Враховуючи усі особливості мови програмування Java та інструментів, які побудовані навколо неї, можна зробити висновок, що вона добре підходить для побудови IoT проектів.

2.2 Фреймворк Spring Boot

Spring Boot є провідним фреймворком для розробки Java веб програм корпоративного рівня, який значно спрощує створення додатків на базі Spring Framework [7]. Фреймворк забезпечує автоматичну конфігурацію компонентів, вбудований веб-сервер та засоби моніторингу, що дозволяє розробникам зосередитися на бізнес-логіці замість інфраструктурних налаштувань.

Його модульна архітектура дозволяє легко інтегрувати різні компоненти системи, включаючи Spring Data JPA для роботи з базами даних, Spring Kafka для інтеграції з системою обміну повідомленнями та Spring Web для створення REST API [7].

Особливо важливою є підтримка автоматичної конфігурації, яка дозволяє мінімізувати кількість коду та забезпечує швидкий старт розробки (рисунок 2.2). Spring Boot Actuator надає готовий функціонал для моніторингу стану додатку, метрик [7].

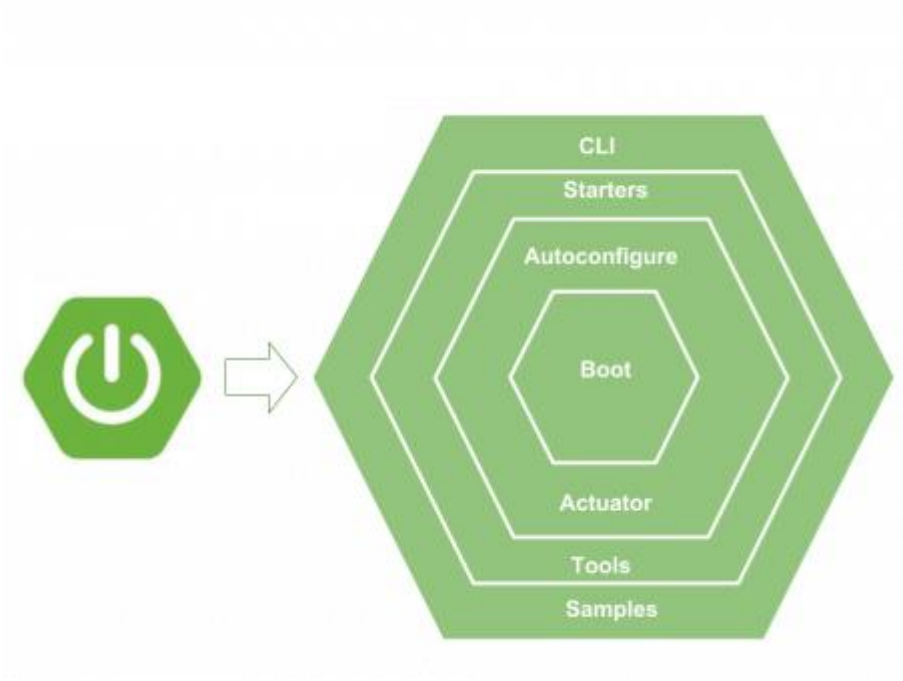


Рисунок 2.2 – Компоненти фреймворку Spring Boot

2.3 Модель акторів Akka

Akka є потужним інструментом для побудови високопродуктивних розподілених додатків на базі акторної моделі [17].

Модель акторів представляє собою парадигму конкурентних обчислень, у якій базовим елементом є актор – незалежна обчислювальна сутність з власним

станом, поведінкою та чергою повідомлень Отримавши повідомлення актор може опрацювати його, створити нових акторів, надіслати власні повідомлення, змінити власний стан, в тому числі стратегію опрацювання наступних отриманих повідомлень чи завершити свою роботу (рисунок 2.3).

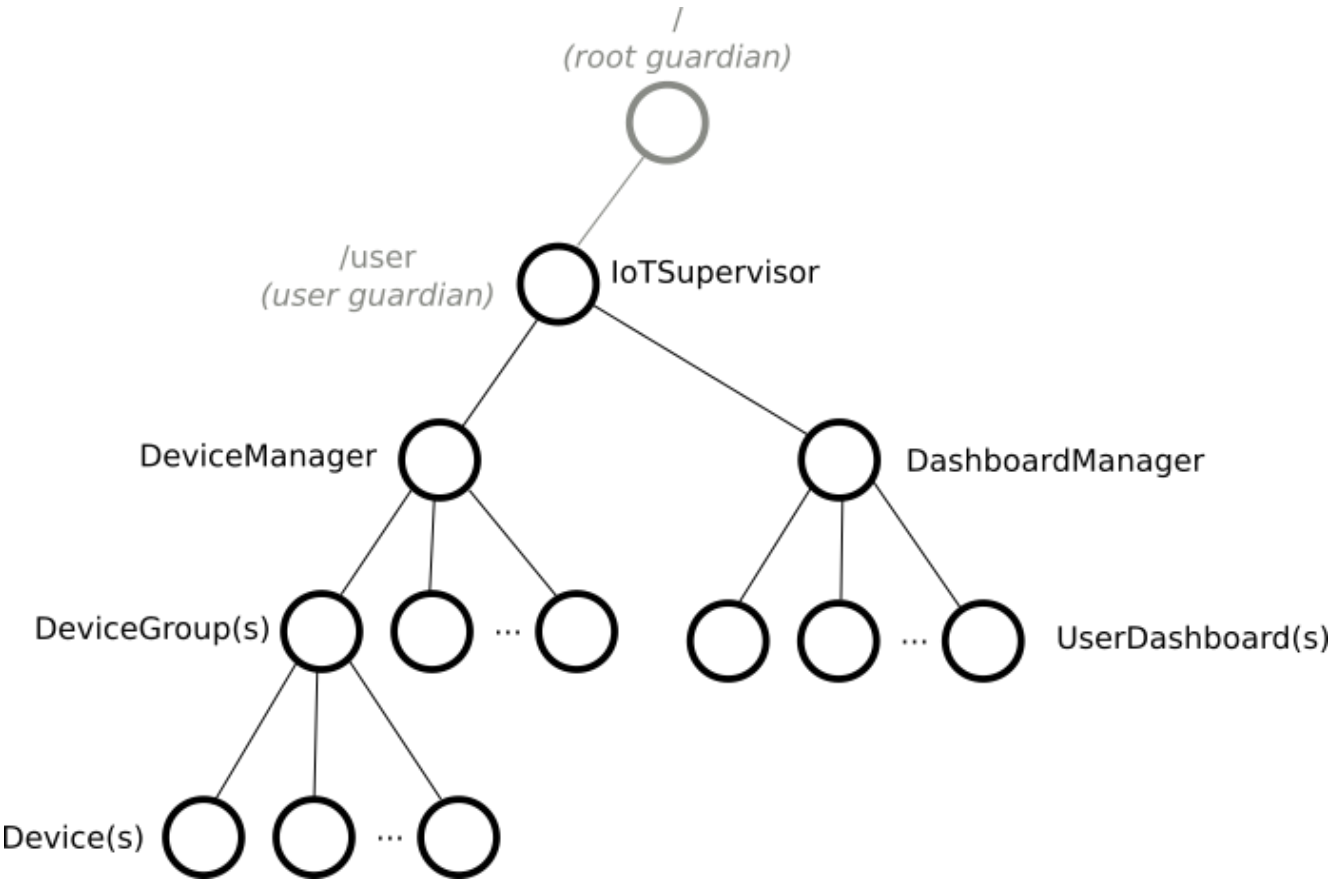


Рисунок 2.3 – Приклад системи акторів на основі Akka [14]

Актори мають наступні характеристики:

- обмін повідомленнями: актори взаємодіють лише шляхом надсилання повідомлень;

- ізоляція: актори не мають спільних ресурсів, що запобігає проблемам блокування та гонки даних;

- локальність прийняття рішень: актор самостійно вирішує, як обробляти отримані повідомлення;

- інкапсуляція стану: кожен актор має власний стан, який недоступний іншим акторам напряму.

Akka дозволяє досягати високого рівня паралелізму та ізоляції помилок, оскільки збій одного актора не впливає на роботу інших, при відповідних налаштуваннях [8]. Система супервізії Akka забезпечує автоматичне відновлення після помилок та підтримання стабільної роботи системи.

2.4 Apache Kafka для обміну повідомленнями

Apache Kafka є розподіленою платформою для потокової обробки даних, яка забезпечує високу пропускну здатність та надійність доставки повідомлень [8]. Kafka забезпечує горизонтальне масштабування через розбиття топіків.

В IoT платформі Kafka виконує роль центральної шини даних, через яку відбувається комунікація між мікросервісами (рисунок 2.4).

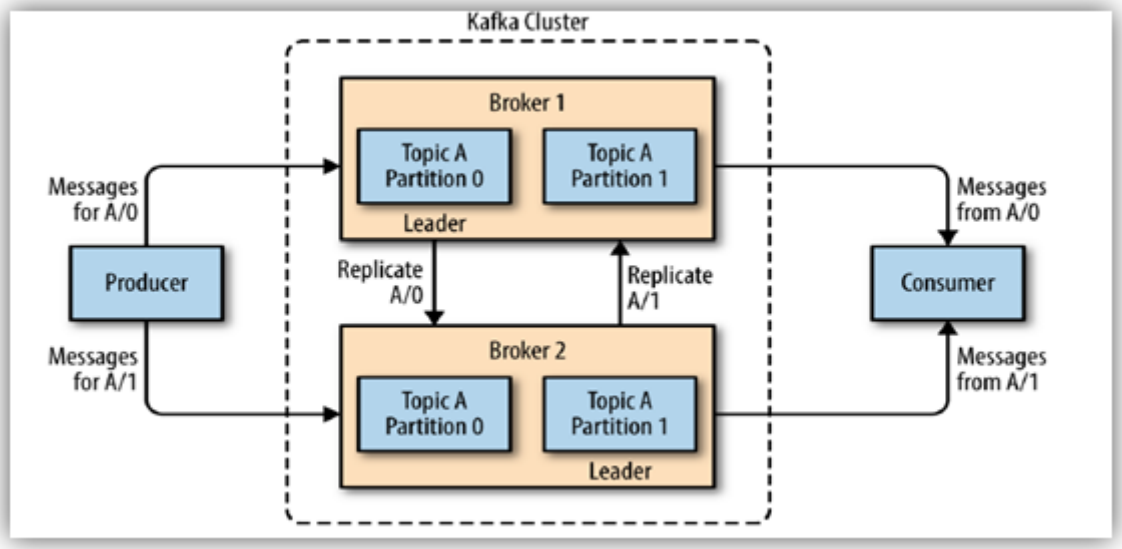


Рисунок 2.4 – Приклад архітектури Kafka кластеру

Ключовою особливістю Kafka є її підхід до зберігання повідомлень. На відміну від традиційних брокерів повідомлень, які видаляють повідомлення після їх споживання, Kafka зберігає всі повідомлення протягом заданого періоду часу. Це

дозволяє декільком клієнтам читати одні й ті ж дані, забезпечуючи можливість відновлення після збоїв без втрати даних. Producer API дозволяє додаткам публікувати потоки записів до топіків. Продюсери можуть контролювати стратегію розбиття, визначаючи, до якого розділу топіку буде направлено повідомлення, що критично важливо для забезпечення порядку обробки та ефективного розподілу навантаження. Consumer API надає можливість додаткам підписуватися на топікі та обробляти потоки записів. Клієнти організовуються в групи споживачів, що забезпечує обробку кожного окремого вхідного повідомлення лише одним із споживачів цієї групи. Це корисно, коли система містить декілька екземплярів одного мікросервісу.

2.5 PostgreSQL як система управління базами даних

PostgreSQL є однією з найпотужніших та найпопулярніших систем управління реляційними базами даних з відкритим вихідним кодом [9]. Розроблена в Університеті Каліфорнії в Берклі у 1986 році як наступник системи Ingres, PostgreSQL протягом десятиліть еволюціонувала до повнофункціональної об'єктно-реляційної системи, яка поєднує надійність традиційних реляційних баз даних з гнучкістю сучасних рішень для роботи з неструктурованими даними.

Архітектура PostgreSQL базується на багатопроцесній моделі, де кожне з'єднання з клієнтом обслуговується окремим процесом. Основними компонентами системи є головний процес-координатор, який приймає з'єднання та створює дочірні процеси для обслуговування клієнтів, фоновий процес записування, який періодично зберігає дані на диск, процес контрольних точок для забезпечення узгодженості даних, та допоміжні процеси для автоматичного очищення, збору статистики та архівування журналів транзакцій.

Модель зберігання даних у PostgreSQL використовує концепцію багатоверсійного контролю одночасності, що дозволяє різним транзакціям бачити узгоджені знімки бази даних без блокування одна одної. Кожен рядок таблиці має

метадані про версію, що дозволяє системі визначати, які дані є видимими для конкретної транзакції. Цей підхід забезпечує високий рівень одночасності та продуктивності навіть при інтенсивному навантаженні з множиною одночасних читачів та продюсерів.

Система типів даних PostgreSQL є надзвичайно розгалуженою та включає всі стандартні типи SQL, а також унікальні розширення. Окрім базових типів як цілі числа, рядки та дати, PostgreSQL підтримує геометричні типи для роботи з просторовими даними, мережеві типи для зберігання адрес та масок підмереж, типи для роботи з діапазонами значень, та JSON/JSONB для ефективного зберігання та запитів до документів у форматі JSON [9].

Індексна підсистема PostgreSQL є однією з найрозвинутіших серед усіх систем управління базами даних. Система підтримує кілька типів індексів, кожен з яких оптимізований для певних типів запитів та даних. B-tree індекси (рисунок 2.5) є стандартними для більшості випадків та ефективні для запитів на рівність, діапазони та сортування [9].

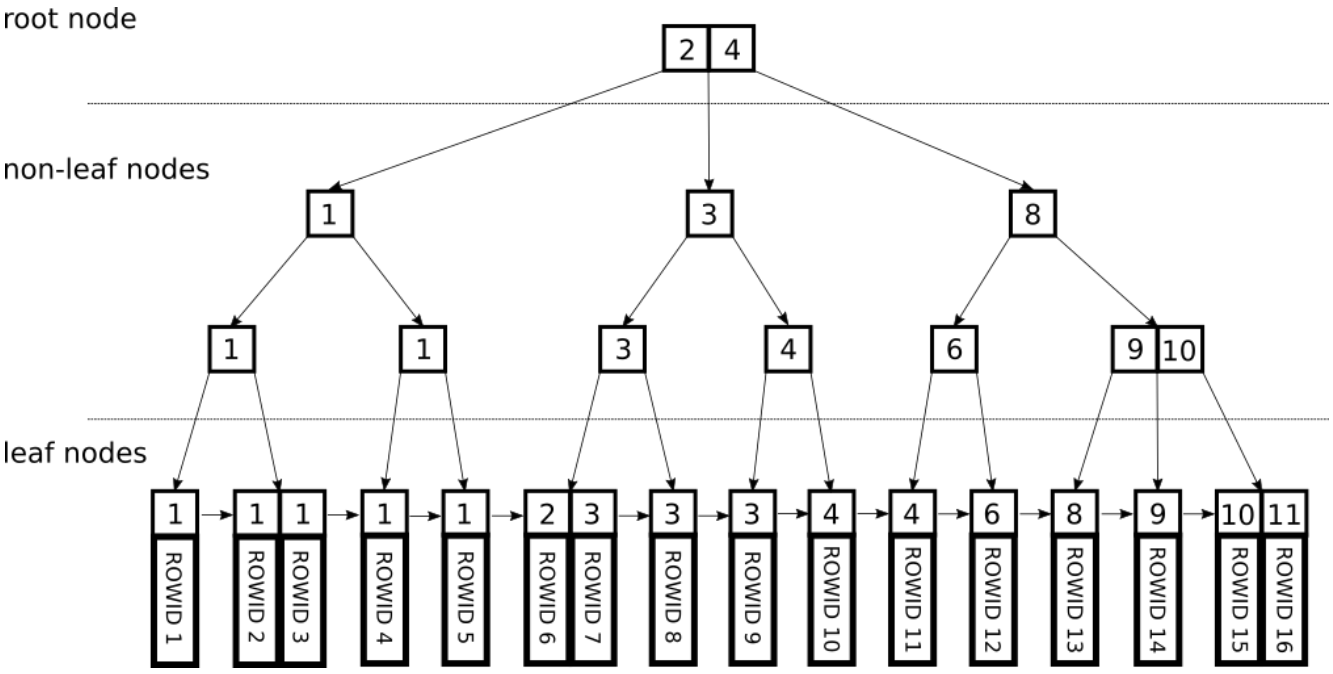


Рисунок 2.5 – Архітектура побудови B-tree індексів у PostgreSQL

Hash індекси оптимізовані для швидких запитів на рівність, але не підтримують діапазонні запити. GiST (Generalized Search Tree) індекси надають фреймворк для створення спеціалізованих структур індексування для нестандартних типів даних.

2.6 GraalVM для виконання JavaScript скриптів

GraalVM – віртуальна машина, яка підтримує виконання програм написаних різними мовами програмування, включаючи JavaScript [10]. GraalVM JavaScript engine забезпечує високу продуктивність виконання JavaScript коду та повну сумісність з ECMAScript стандартами.

В IoT платформах GraalVM використовується для виконання користувацьких скриптів обробки телеметричних даних. Це дозволяє надати користувачам гнучкий інструмент для створення унікальної логіки трансформації та фільтрації даних без необхідності перекомпіляції основного додатку [10].

Ізоляція контексту виконання в GraalVM забезпечує безпеку виконання користувацького коду та запобігає впливу на основну систему. Polyglot API дозволяє інтеграцію між Java кодом та JavaScript скриптами, забезпечуючи ефективний обмін даними.

2.7 Docker та Kubernetes для контейнеризації

Docker є революційною платформою віртуалізації, яка кардинально змінила підхід до розгортання та управління програмним забезпеченням [11]. Розроблена компанією Docker Inc. у 2013 році, ця технологія базується на концепції легких віртуальних середовищ, які дозволяють упакувати додаток разом з усіма необхідними залежностями в ізольовану одиницю виконання.

Основою Docker є технологія операційної системи Linux, зокрема механізми cgroups та namespaces, які забезпечують ізоляцію процесів та ресурсів. На відміну від традиційних віртуальних машин, які віртуалізують повну операційну систему, Docker використовує ядро хост-системи, що значно зменшує споживання ресурсів та прискорює запуск додатків. Це робить Docker ідеальним рішенням для мікросервісних архітектур, де необхідно запускати велику кількість невеликих сервісів.

Архітектура Docker складається з декількох ключових компонентів (рисунок 2.6). Docker Engine є серцем платформи – це daemon процес, який управляє життєвим циклом віртуальних середовищ виконання. Docker Client надає інтерфейс командного рядка для взаємодії з Engine через REST API. Docker Registry служить централізованим сховищем для образів додатків, найпопулярнішим з яких є Docker Hub, що містить мільйони готових образів для різноманітних додатків та операційних систем.

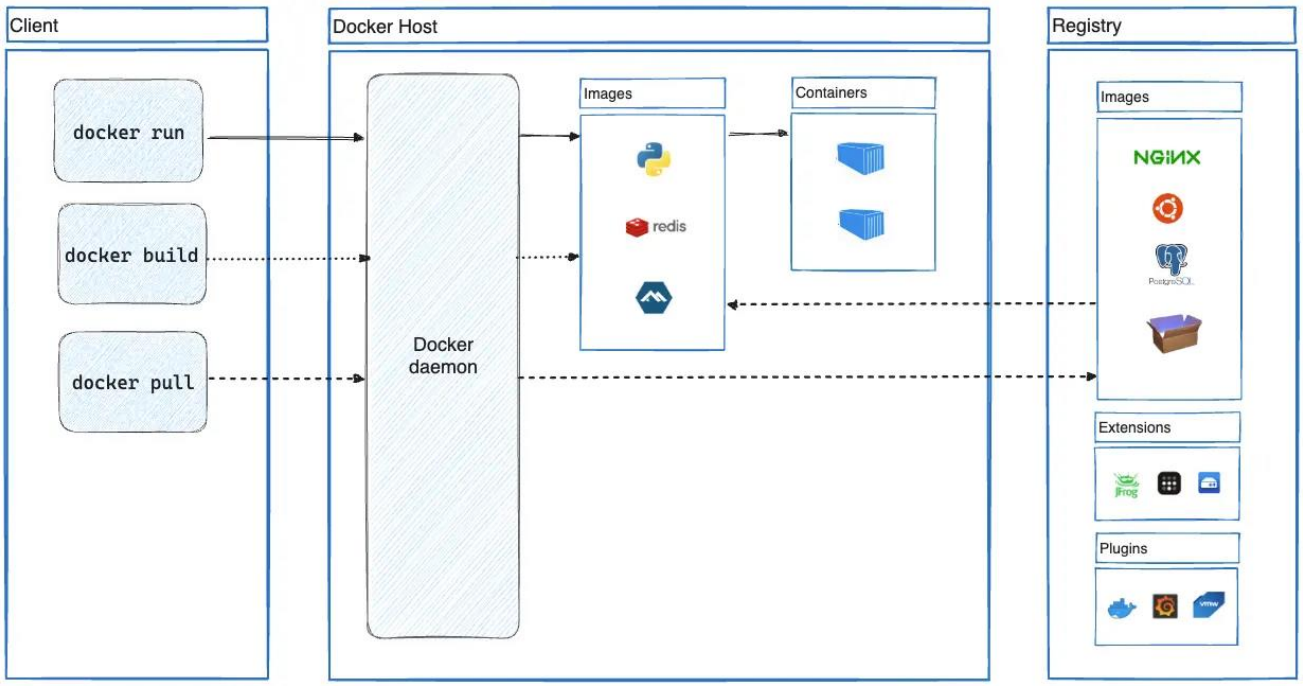


Рисунок 2.6 – Архітектура Docker

Процес створення образу Docker базується на декларативному підході через використання Dockerfile – текстового файлу, який містить послідовність інструкцій для збірки образу. Кожна інструкція в Dockerfile створює новий шар в образі, що дозволяє ефективно переконстувати образи та економити дисковий простір через повторне використання спільних шарів. Цей шаровий підхід також прискорює процес завантаження образів, оскільки незмінені шари можуть бути кешовані локально.

Однією з найважливіших переваг Docker є забезпечення повної відтворюваності середовища виконання [11]. Додаток, який працює в Docker на машині розробника, буде працювати ідентично в тестовому середовищі та в промисловій експлуатації. Це вирішує класичну проблему «на моїй машині працює», яка часто виникає через відмінності в конфігураціях середовищ, версіях бібліотек та системних залежностей.

Docker значно спрощує процес розгортання додатків. Замість складних інструкцій з налаштування середовища, встановлення залежностей та конфігурування служб, достатньо виконати одну команду для запуску повністю налаштованого додатку. Це особливо важливо для команд DevOps, які можуть автоматизувати процеси розгортання через інтеграцію Docker з системами безперервної інтеграції та доставки.

Docker Compose розширює можливості платформи для управління багатосервісними додатками. Через декларативний YAML файл розробники можуть описати всю архітектуру додатку, включаючи сервіси, мережі, томи та залежності. Це дозволяє одною командою запустити повний стек додатку локально для розробки або тестування.

2.8 Keycloak для управління ідентифікацією та доступом

Keycloak є комплексним рішенням з відкритим вихідним кодом для управління ідентифікацією та доступом, розробленим компанією Red Hat [12]. Ця

платформа надає централізовані служби автентифікації та авторизації для сучасних додатків, підтримуючи широкий спектр стандартів безпеки та протоколів. Keycloak дозволяє організаціям реалізувати принципи єдиного входу та централізованого управління користувачами без необхідності розробки власних механізмів безпеки.

Архітектура Keycloak базується на концепції областей, які представляють ізольовані простори для управління користувачами, ролями, групами та клієнтськими додатками. Кожна область функціонує як окрема організаційна одиниця з власними налаштуваннями безпеки, політиками паролів та конфігураціями інтеграції. Це дозволяє одному екземпляру Keycloak обслуговувати множину організацій або підрозділів з різними вимогами до безпеки.

Система підтримує широкий спектр протоколів автентифікації та авторизації, включаючи OpenID Connect, OAuth 2.0, SAML 2.0 та власний протокол Keycloak. OpenID Connect. OAuth 2.0 використовується для авторизації доступу до захищених ресурсів, дозволяючи додаткам отримувати обмежений доступ до даних користувача без розкриття облікових даних.

Адаптери клієнтських додатків забезпечують інтеграцію з різними технологічними платформами та фреймворками. Keycloak надає готові адаптери для Java EE, Spring Boot, Node.js, Python та інших популярних технологій, що значно спрощує процес інтеграції існуючих додатків з системою управління ідентифікацією. Адаптери автоматично обробляють процеси автентифікації, отримання та перевірки токенів, звільняючи розробників від необхідності імплементації складної логіки безпеки [12].

Багатофакторна автентифікація є невід'ємною частиною сучасних систем безпеки, і Keycloak надає комплексну підтримку різних методів додаткової перевірки. Система підтримує одноразові паролі через мобільні додатки як Google Authenticator та FreeOTP, SMS повідомлення, електронну пошту та біометричну автентифікацію через WebAuthn стандарт. Адміністратори можуть налаштовувати гнучкі потоки автентифікації, які враховують рівень ризику та контекст доступу.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ

Імплементація IoT системи потребує чіткого плану та архітектури, щоб стабільно витримувати високе навантаження обробки телеметричних даних. Окрім цього, опрацювання даних має виконуватися настільки швидко, наскільки можливо. Це досягається за рахунок підвищення пропускної здатності системи.

3.1 Архітектура IoT платформи

Архітектура системи – це структурне уявлення про її компоненти, зв'язки між ними, принципи взаємодії, розгортання та розвитку. Архітектура є критично важливою на всіх етапах створення складних програмних рішень, зокрема для IoT-платформ, хмарних сервісів або корпоративних систем.

Популярним рішенням для горизонтального масштабування платформи є використанням мікросервісної архітектури (рисунок 1.5).

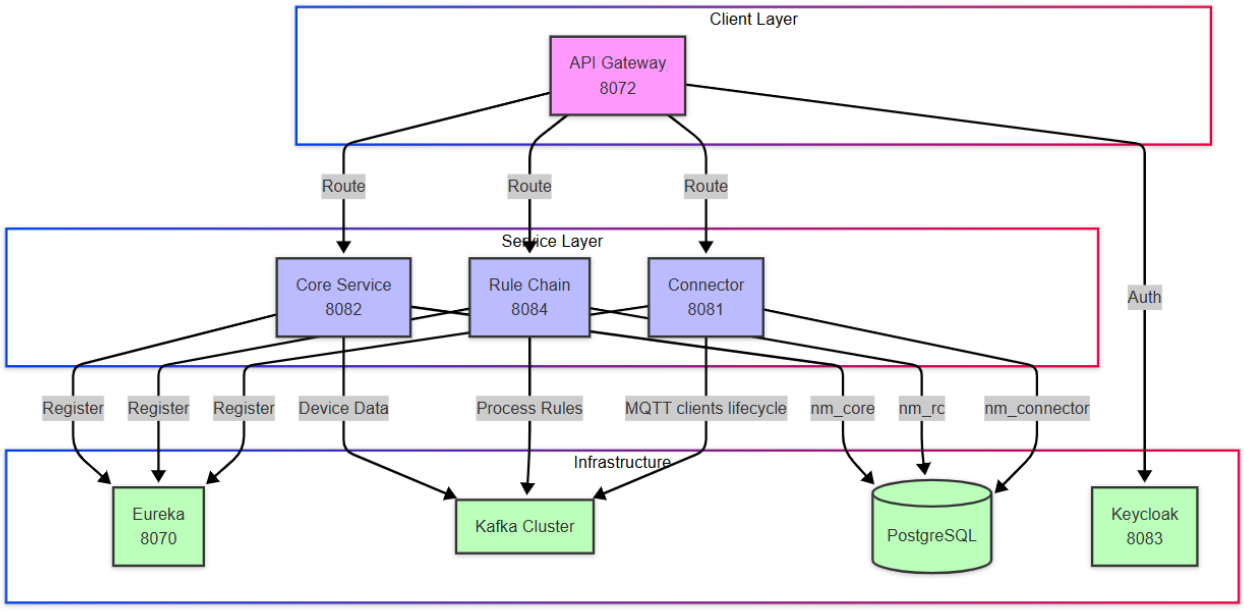


Рисунок 3.1 – Архітектура IoT платформи.

Підняття декількох екземплярів одного сервіса дає нам змогу підвищити пропускну здатність системи збільшуючи лише обчислювальні ресурси. Це дозволяє опрацьовувати більше користувачів та вхідних повідомлень в реальному часі. Варто зазначити, що горизонтальне масштабування може суттєво зменшити затримки у черзі (queue latency) шляхом розподілу навантаження між вузлами [7]. Крім того, у багатьох випадках це також дозволяє зменшити загальну затримку виконання (execution latency), оскільки обчислення можуть виконуватись паралельно. Проте, у випадках, коли час виконання зумовлений складністю самого алгоритму, масштабування може мати обмежений вплив.

Компоненти мікросервісної архітектури системи складаються з трьох основних мікросервісів, Kafka кластеру, сервіс виявлення служб, баз даних PostgreSQL, систему авторизації Keycloak та API-шлюз.

3.2 Сервіс ланцюжків правил

Rule Chain Service є центральним компонентом IoT платформи, що відповідає за обробку та маршрутизацію повідомлень від пристроїв згідно з визначеними бізнес-правилами. Цей сервіс реалізує основний механізм обробки даних та подій у системі. Також він має власну базу даних, де зберігається інформація про самі ланцюжки правил та зв’язки між вершинами, які і створюють логіку обробки повідомлень.

У межах архітектури мікросервісів Rule Chain Service взаємодіє з іншими компонентами через механізми Kafka та RESTful API. Сервіс отримує події від Core Service та Connector Service через повідомлення Kafka, що забезпечує асинхронну та масштабовану обробку. Результати обробки можуть передаватися іншим сервісам для збереження, візуалізації або подальшої інтеграції з зовнішніми системами.

3.2.1 Основні функції сервісу ланцюжків правил

Основними функціями сервісу є:

- обробка потоків даних: сервіс приймає телеметрію та події від пристроїв та обробляє їх згідно з налаштованими правилами;

- маршрутизація повідомлень: спрямовує повідомлення між різними вузлами обробки в залежності від їх типу та змісту;

- фільтрація даних: дозволяє відфільтровувати непотрібні дані або події;

- трансформація даних: перетворює формат даних для подальшої обробки або зберігання;

- прийняття рішень: на основі заданих умов приймає рішення про подальші дії.

3.2.2 Використання моделі акторів Akka для побудови сервісу

Для імплементації сервісу було обрано фреймворк Akka, що базується на математичній моделі акторів. У побудованій системі додано декілька видів акторів, кожен з яких виконує чітку функцію, що дозволяє легко розширювати код функціонально.

ActorSystem є центральним компонентом, який ініціалізується при запуску сервісу та координує роботу всіх акторів. Система акторів організована ієрархічно, де головним актором є MsgDelegatorActor, який делегує повідомлення відповідним ланцюжкам правил. Також він має команди, які здатні зупинити ланцюжок правил, а також видалити старий і створити новий.

Кожен вузол правил представлений як окремий актор RuleNodeActor, що інкапсулює логіку обробки конкретного типу вузла. Ці актори взаємодіють виключно через обмін типізованими повідомленнями, реалізованими через інтерфейс RuleNodeCommand.

Під час ініціалізації ланцюжка правил створюється граф акторів, що відповідає структурі ланцюжка. Коренева вершина ініціює потік обробки, а кожен

актор, отримавши повідомлення, приймає рішення про його обробку та подальшу маршрутизацію відповідно до налаштованих правил.

3.2.3 Переваги та недоліки архітектури на базі моделі акторів

Використання моделі акторів Akka в побудованому сервісі забезпечує низку суттєвих архітектурних переваг [14]:

- архітектура сервісу демонструє високу конкурентність завдяки можливості паралельної обробки великої кількості повідомлень без блокування. Актори, як легковагові сутності, дозволяють ефективно використовувати обчислювальні ресурси, де кожен актор обробляє своє повідомлення незалежно від інших;

- масштабованість системи досягається через відсутність спільного мутабельного стану між акторами. Система може горизонтально масштабуватися шляхом розподілу акторів між різними фізичними серверами, а завдяки іммутабельності повідомлень забезпечується консистентність даних;

- відмовостійкість є вбудованим концептом Akka через механізми нагляду та відновлення. Система акторів реалізує патерн «let it crash», де супервізори відстежують стан підпорядкованих акторів і можуть перезапускати їх у разі помилки, не впливаючи на роботу всієї системи.

Проте, архітектура на базі моделі акторів має певні недоліки, які важливо враховувати [14, 17]:

- модель програмування на основі акторів має високу криву навчання та вимагає відмінного розуміння конкурентних патернів, що усуває можливість простого імперативного програмування;

- налагодження систем на основі акторів є комплексним завданням, оскільки стан розподілений між багатьма незалежними сутностями, а традиційні інструменти налагодження часто не пристосовані для роботи з асинхронними моделями. Також асинхронна природа комунікації акторів створює виклики для трасування та налагодження потоків обробки. Без спеціалізованих інструментів

складно відстежити послідовність обробки повідомлень та ідентифікувати джерело проблеми.

3.2.4 Роль використання Apache Kafka

Apache Kafka виступає центральним елементом комунікаційної інфраструктури сервісу, забезпечуючи надійний, масштабований механізм обміну повідомленнями між компонентами системи. В архітектурі IoT платформи Kafka функціонує як магістраль даних, що єднає генератори подій (пристрої, сенсори, сервіси) з процесорами подій (ланцюжки правил, аналітичні модулі).

Rule Chain Service інтегрований з Kafka через два основні архітектурні патерни: виробник-споживач (producer-consumer) для асинхронного обміну повідомленнями та потокової обробки даних (stream processing [8]) для неперервного аналізу та трансформації телеметрії.

Кластер Kafka функціонує як розподілений реєстр подій, пов'язаних з управлінням ланцюжками правил. Спеціалізовані топіки, такі як RULE_CHAIN_INIT_TOPIC та RULE_CHAIN_STOP_TOPIC, використовуються для ініціалізації, оновлення та зупинки ланцюжків правил, оскільки кожен екземпляр сервісу містить в собі стан цих ланцюжків. При публікації таких подій Kafka гарантує збереження порядку повідомлень в межах розділу (partition), що критично для правильної послідовності операцій з ланцюжками правил.

Ключовою функцією Kafka в сервісі ланцюжків правил є забезпечення транспортного шару для потоку телеметрії та подій від пристроїв. Повідомлення від пристроїв публікуються в спеціалізовані топіки, структуровані за типами даних або категоріями пристроїв. Сервіс ланцюжків правил споживає ці повідомлення, забезпечуючи їх маршрутизацію через відповідні ланцюжки правил. Оскільки кожен екземпляр сервісу відноситься до єдиної групи споживачів, лише один з них обробить повідомлення надіслане пристроєм, що унеможливлює обробку одного і того самого повідомлення декілька разів.

Архітектурна перевага цього підходу полягає в зменшенні зв’язності генераторів, тобто екземплярів мікросервісу, який відповідає за підключення пристроїв, та споживачів даних, тобто екземплярів мікросервісу ланцюжків правил, що дозволяє незалежно масштабувати обидві сторони системи та забезпечувати буферизацію при пікових навантаженнях [15].

Можна перерахувати основні переваги використання Apache Kafka для IoT сервісу ланцюжків правил:

- висока доступність та надійність досягається завдяки розподіленій природі Kafka, де дані копіюються між брокерами, що забезпечує стійкість до відмов окремих вузлів. Реалізація кластеру з декількох брокерів підвищує відмовостійкість системи та забезпечує безперервність обробки даних;

- горизонтальна масштабованість є інтегральною частиною архітектури Kafka. Сервіс ланцюжків правил може масштабуватися шляхом збільшення кількості партицій в топіках та запуску додаткових екземплярів сервісу, які утворюють групи споживачів. Це дозволяє системі адаптуватися до зростаючого навантаження без зміни архітектури;

- гарантована доставка повідомлень забезпечується механізмами підтвердження та зберігання повідомлень у Kafka. Це критично для IoT систем, де втрата даних телеметрії або подій може призвести до некоректної роботи бізнес-логіки або втрати важливої інформації;

- буферизація та згладжування пікових навантажень є природним наслідком архітектури Kafka. При раптовому збільшенні кількості повідомлень від пристроїв, Kafka зберігає їх до моменту, коли сервіс зможе їх обробити, запобігаючи втраті даних та перевантаженню системи.

3.3 Сервіс для підключення пристроїв

Сервіс для підключення пристроїв є ключовим компонентом IoT архітектури, що забезпечує комунікаційний інтерфейс між фізичними IoT пристроями та

платформою. Цей сервіс вирішує фундаментальну проблему інтеграції різнорідних пристроїв та протоколів у єдину екосистему, виступаючи в ролі універсального перекладача між пристроями та бізнес-логікою платформи. Реалізовано два протоколи підключення: MQTT та HTTP.

3.3.1 Архітектурна концепція та функціональне призначення

Архітектурно сервіс для підключення пристроїв реалізований згідно з принципами гнучкого, компонентного проектування, що дозволяє підтримувати різноманітні протоколи IoT та легко інтегрувати нові. Основна концепція будується навколо абстракцій підключення та обміну повідомленнями, які інкапсулюють специфіку конкретних протоколів.

В рамках мікросервісної архітектури цей сервіс відповідає за:

- встановлення та підтримку з'єднань з фізичними пристроями;
- отримання телеметрії та подій від пристроїв (uplink);
- відправку команд та конфігураційних повідомлень до пристроїв (downlink);
- конвертацію специфічних форматів повідомлень у канонічну форму для платформи.

Сервіс імплементує шаблон проектування «Стратегія» для обробки різних типів з'єднань. Кожен тип протоколу, тобто MQTT та HTTP, реалізується окремим менеджером, який імплементує відповідний інтерфейс. Це дозволяє єдиному мікросервісу підтримувати множинні протоколи без зміни основної архітектури, а збільшення кодової бази системи не буде впливати на час розробки нових протоколів комунікацій.

Іншим ключовим шаблоном є «Фабрика», яка відповідає за створення та ініціалізацію з'єднань на основі конфігурації абстрактного типу. Кожен вищевказаний менеджер має імплементувати метод ініціалізації з конкретним типом конфігурації.

3.3.2 Протоколи інтеграції пристроїв

На даний момент Сервіс для підключення пристроїв підтримує наступні протоколи:

- MQTT є легким протоколом обміну повідомленнями за моделлю «видавець-підписник» [16], оптимізований для IoT. Реалізовано через Mqtt3ClientManager, що забезпечує підтримку аутентифікації, підписки на теми та отримання/відправки повідомлень;

- HTTP є REST-подібним інтерфейсом для пристроїв, які не підтримують постійне з'єднання.

Архітектура передбачає розширення на інші протоколи (CoAP, AMQP, Modbus та інші) через імплементацію відповідних менеджерів з'єднань.

3.3.3 Інтеграція з Kafka та потокова обробка даних

Центральним елементом інтеграції сервісу для підключення пристроїв з іншими компонентами платформи є Apache Kafka. Архітектурна концепція базується на повному розділенні відповідальності між прийомом повідомлень від пристроїв та їх обробкою.

При отриманні uplink-повідомлення від пристрою, Сервіс для підключення пристроїв виконує наступні кроки:

- конвертує формат повідомлення з протокол-специфічного у канонічний для платформи (використовуючи JavaScript-перетворювачі). Логіка конвертації задається користувачами платформи та зберігається в базі даних;

- формує метадані повідомлення (назва пристрою, тип, часова мітка);

- публікує повідомлення у відповідний Kafka-топік для подальшої обробки сервісом ланцюжків правил.

Цей патерн забезпечує ефективний розподіл навантаження та масштабованість системи. Сервіс для підключення пристроїв може фокусуватися на забезпеченні стабільних з'єднань з пристроями, в той час як обробка даних делегується сервісу ланцюжків правил через асинхронний механізм Kafka.

3.3.4 Керування життєвим циклом підключень

Важливим архітектурним аспектом є керування життєвим циклом підключень в розподіленому середовищі. Для цього реалізовано механізм координації через Kafka-топіки. Такий підхід дозволяє динамічно створювати та зупиняти з'єднання з пристроями в кластерному середовищі, забезпечуючи консистентність стану між різними екземплярами сервісу, виключаючи можливість дублювання обробки одного і того самого повідомлення.

3.3.5 Розподіл навантаження та масштабованість

Архітектура Сервісу для підключення пристроїв передбачає горизонтальне масштабування для обробки великої кількості одночасних з'єднань. Замість того, щоб один сервер обробляв усі пристрої, система розподіляє їх між кількома серверами. Розподіл відбувається за принципом партиціонування через Kafka – кожен MQTT клієнт потрапляє до певної партиції на основі свого унікального ідентифікатора. Це забезпечує рівномірний розподіл з'єднань між серверами та стійкість до відмов окремих екземплярів. Також забезпечено оновлення MQTT клієнтів кожного екземпляру сервісу при зміні партицій в системі.

3.3.6 Механізм обробки відправки повідомлень до пристроїв

Для забезпечення двостороннього зв'язку з пристроями, Сервіс для підключення пристроїв імплементує механізм відправки команд до пристроїв. Архітектурно це реалізовано через компонент DownlinkMessageDelegator, який маршрутизує команди до відповідних менеджерів пристроїв на основі типу протоколу, який міститься в команді. Ці повідомлення потрапляють в делегатор через Kafka-топік з сервісу ланцюжків правил, забезпечуючи асинхронне опрацювання між двома мікросервісами.

3.4 Центральний сервіс

Центральний сервіс є фундаментальним компонентом архітектури IoT платформи, що виконує роль центрального сховища даних та метаданих системи. Цей сервіс встановлює єдину точку доступу до інформації про пристрої, їхні типи та історичні дані телеметрії, формуючи основу для всіх інших компонентів платформи.

Архітектура побудована згідно з принципами доменно-орієнтованого програмування, які набули неабиякої популярності. Система також має властивість горизонтального масштабування за рахунок збільшення екземплярів сервісу.

3.4.1 Архітектура та компоненти центрального сервісу

Архітектура Центрального сервісу побудована згідно з принципами доменно-орієнтованого проектування (DDD), де чітко розділені рівні доступу до даних, бізнес-логіки та представлення.

Основу архітектури формує модель даних, що включає наступні ключові сутності:

- пристрій, що представляє фізичний або віртуальний IoT пристрій, що взаємодіє з платформою. Кожен пристрій має унікальний ідентифікатор, назву та належить до певного типу пристроїв;

- тип пристрою, що категоризує групи подібних пристроїв та визначає їхні характеристики, включаючи зв'язок з ланцюжком правил обробки. Тип пристрою є центральним елементом, що пов'язує фізичні пристрої з логікою їх обробки;

- телеметрія пристрою зберігає часові ряди даних, отриманих від пристроїв. Кожен запис телеметрії включає ідентифікатор пристрою, ключ даних, тип значення, фактичне значення та часову мітку;

- остання телеметрія пристрою виступає спеціалізованою сутністю для швидкого доступу до останніх значень телеметрії для кожного пристрою.

Центральний сервіс структурований у класичну трирівневу архітектуру. Рівень даних реалізований через репозиторії, що забезпечують доступ до бази даних PostgreSQL. Репозиторії використовують Spring Data JPA для абстрагування роботи з БД та включають оптимізовані запити для різних сценаріїв використання. Сервісний рівень містить бізнес-логіку та координує взаємодію між репозиторіями, зовнішніми клієнтами та контролерами. Сервіси забезпечують транзакційну цілісність та валідацію даних. Рівень API – представлений RESTful контролерами, що надають стандартизований інтерфейс для інших компонентів системи. API структуроване за ресурсами та підтримує повний спектр CRUD операцій.

Архітектура доповнена механізмами Spring Security для авторизації та аутентифікації запитів, а також глобальними обробниками винятків для централізованого управління помилками.

3.4.2 Інтеграційні механізми центрального сервісу

Сервіс інтегрується з двома іншими: ланцюжків правил та сервісом для підключення пристроїв. Синхронна комунікація для інтеграції з першим використовується при створенні та модифікації типів пристроїв. Для обробки телеметрії пристроїв використовується асинхронний механізм обміну через Kafka, що дозволяє масштабувати обробку даних незалежно від їх зберігання. центральний сервіс надає інформацію про типи пристроїв та їхні конфігурації для сервіс, який відповідає за підключення пристроїв. Коли пристрій під'єднується до платформи, сервіс пристроїв перевіряє його існування та конфігурацію через API центрального сервісу.

Усі інтеграційні взаємодії реалізовані з урахуванням принципів надійності та відмовостійкості системи. У разі недоступності одного із сервісів центральний сервіс використовує механізми повторних спроб та кешування критично важливої інформації.

4 РОБОТА КОРИСТУВАЧА З ПРОГРАМНОЮ СИСТЕМОЮ

Для комфортного користування платформою, додано веб інтерфейс, побудований на фреймворку React. Платформа також містить функціональність для створення та редагування ланцюжків правил на основі React Flow.

4.1 Робота з пристроями

Для роботи з пристроями додано розділи: пристрої і їх типи (рисунок 4.1).

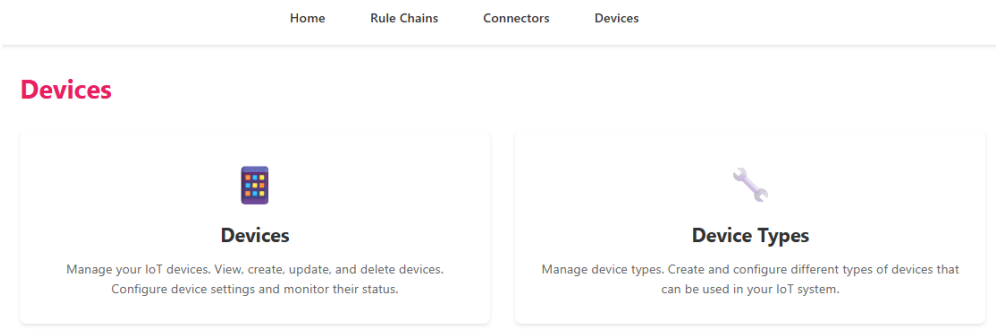


Рисунок 4.1 – Сторінка з розділами пристроїв

При переході на сторінку пристроїв, можна побачити список із існуючими пристроями, а також кнопку для додавання нових. Кожна картка апарату містить функціональність для редагування та видалення (рисунок 4.2).

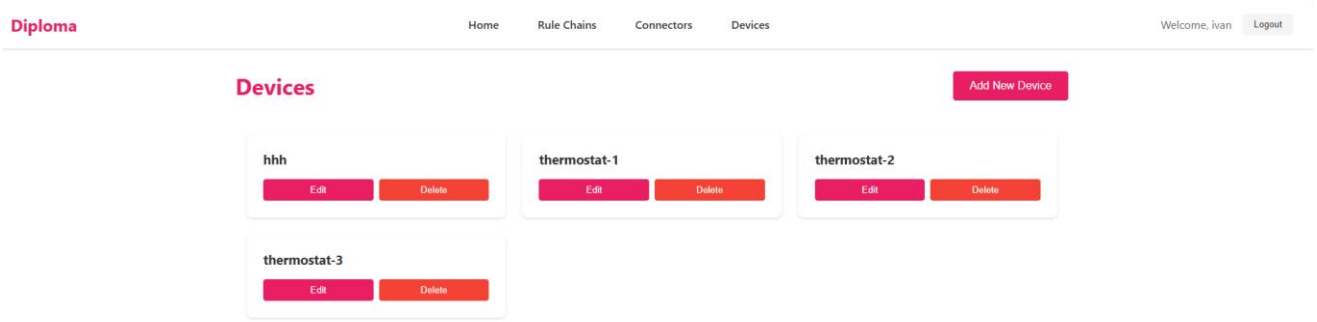


Рисунок 4.2 – Список з пристроями

При натисканні на карточку пристрою, можна побачити вікно із останніми телеметричними даними (рисунок 4.3).

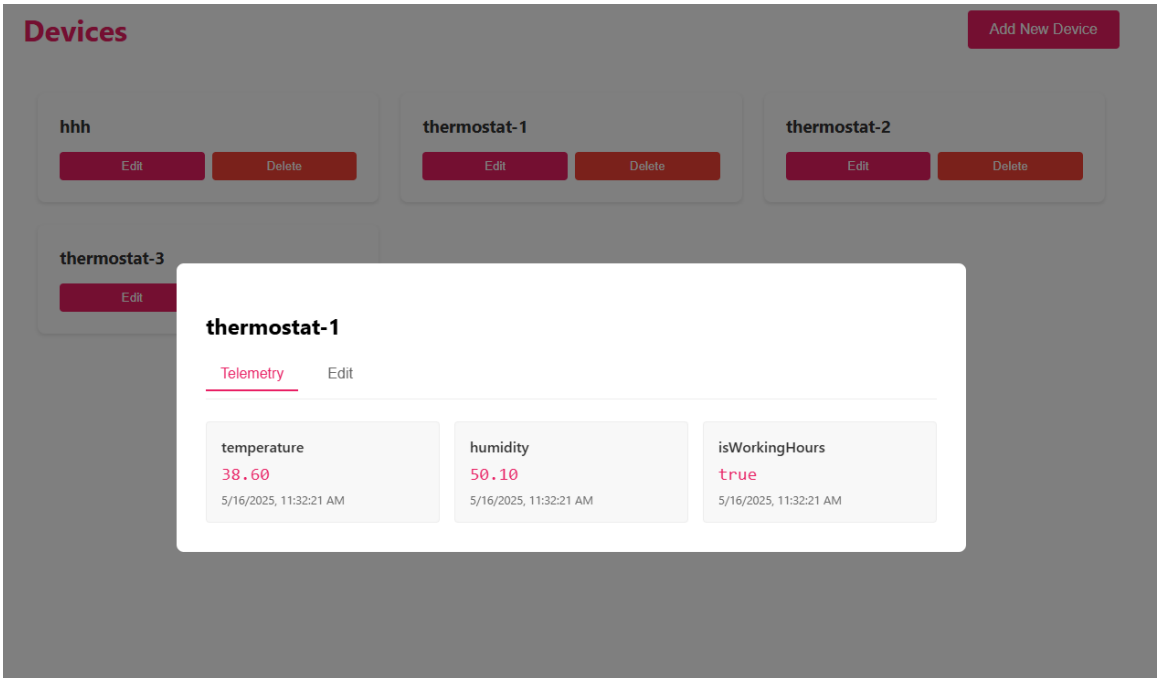


Рисунок 4.3 – Приклад останніх телеметричних даних пристрою

При натисканні на вкладку редагування, можемо побачити поля введення назви та опису пристрою, а також випадаючий список із доступними для вибору типами (рисунок 4.4).

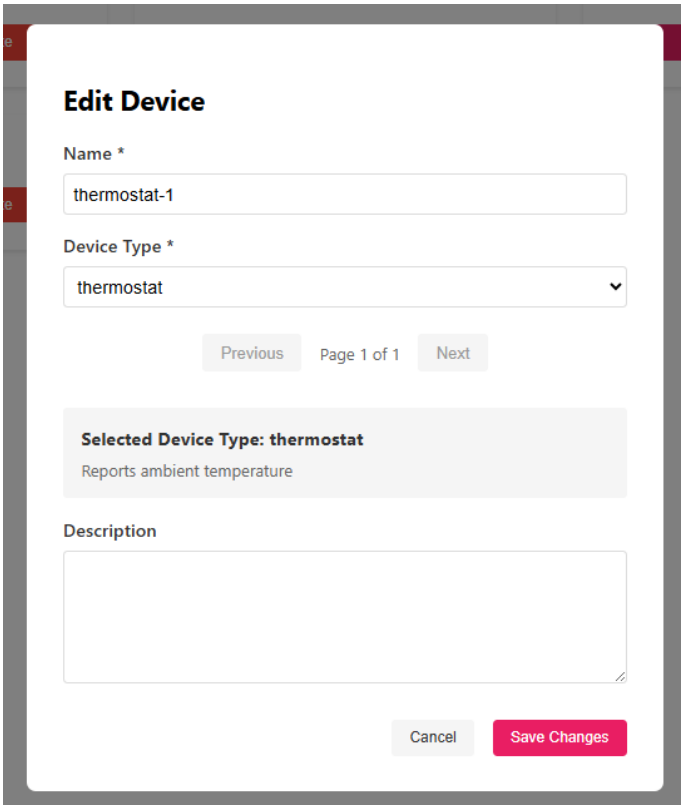


Рисунок 4.4 – Вікно редагування пристрою

При переході на сторінку з типами апаратів, бачимо їх список (рисунок 4.5).

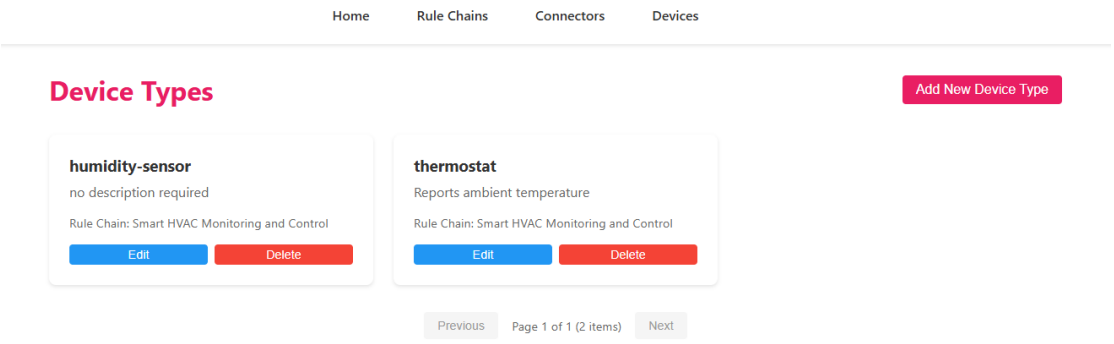


Рисунок 4.5 – Сторінка з типами пристроїв

При натисканні на карточку, можна побачити вікно редагування, яке містить поля для введення назви та опису, а також випадаючий список із доступними ланцюжками правил (рисунок 4.6).

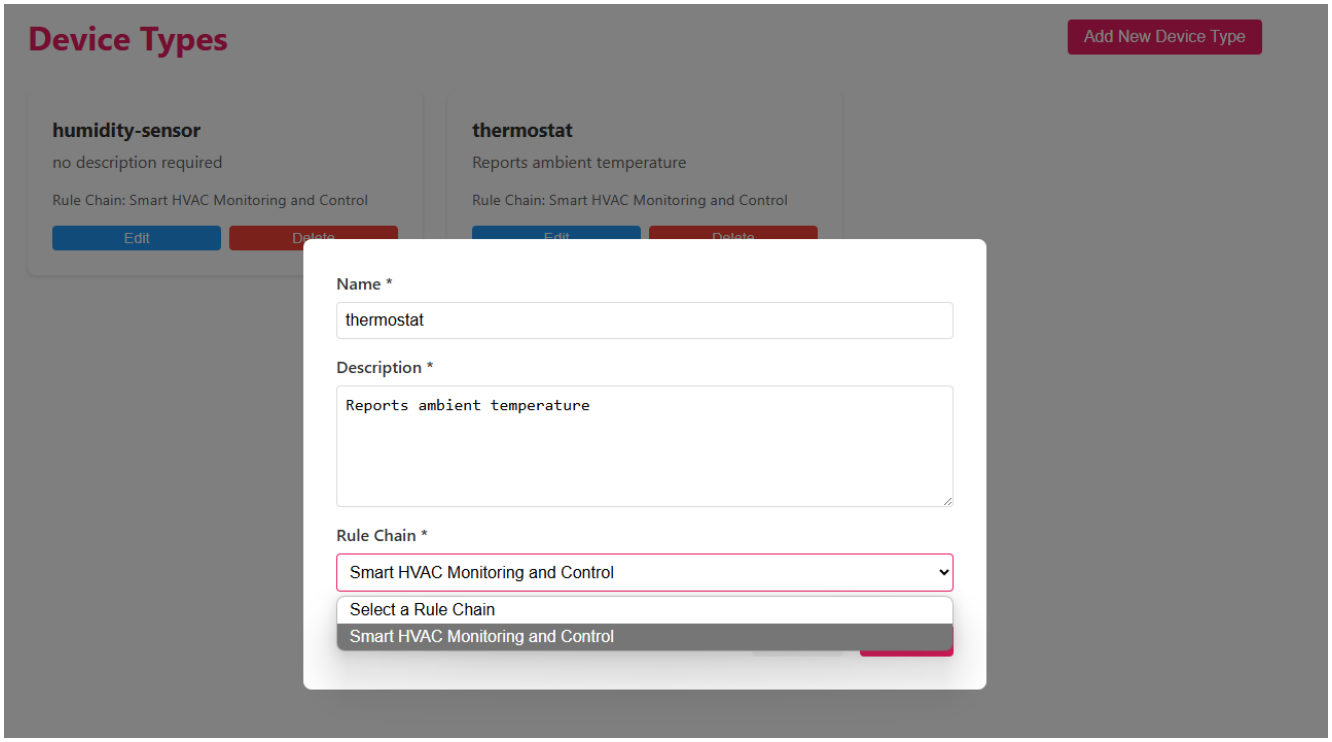


Рисунок 4.6 – Вікно редагування типу пристроїв

Кожна картка типу містить назву, опис, кнопки для редагування та видалення, а також посилання на ланцюжок правил, який оброблятиме повідомлення пристроїв з цим типом.

4.2 Під'єднання пристроїв

Платформа має розділ з’єднувачів. При натисканні на кнопку розділу, користувач може побачити дві картки, які символізують протокол комунікації. Наразі доступні два типи: MQTT та HTTP. Кожна картка містить опис та маленький логотип (рисунок 4.7).

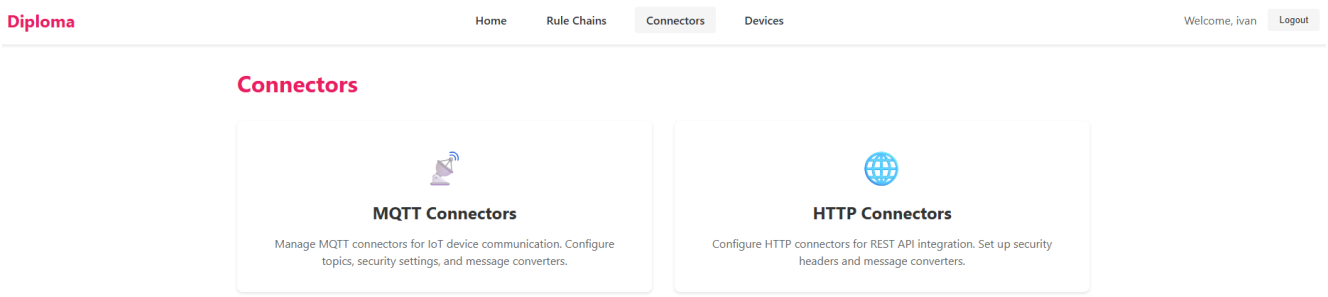


Рисунок 4.7 – Сторінка розділу під'єднання пристроїв із доступними типами комунікації

При натисканні на картку з MQTT типом, користувач може побачити список із створеними з'єднувачами. Кожна картка списку містить в собі назву, статус з'єднання з MQTT брокером, а також його хост та порт і топіки, за якими здійснюється підписка для зчитування даних (рисунок 4.8).

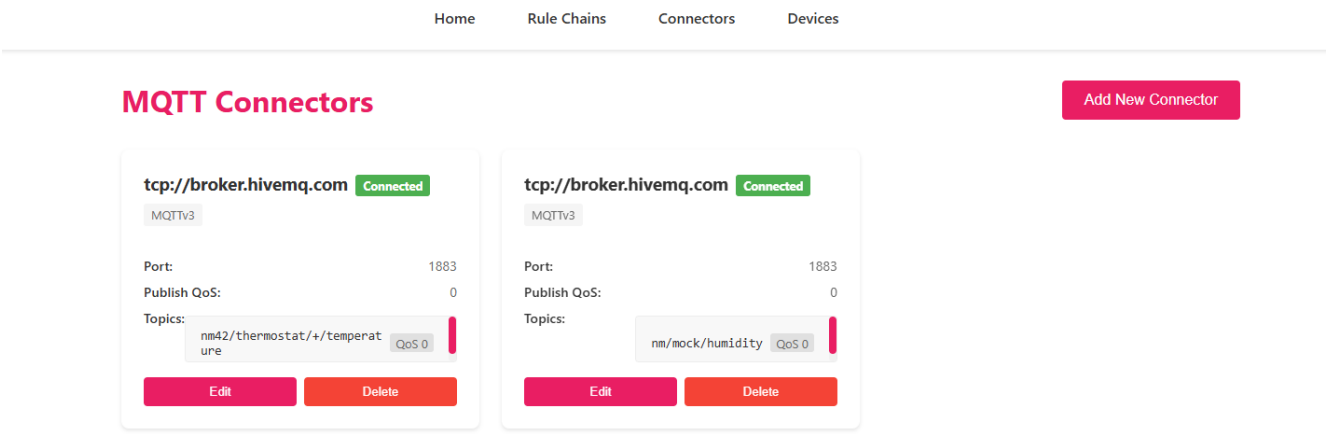


Рисунок 4.8 – Сторінка зі списком існуючих MQTT клієнтів

При натисканні на картку з'єднувача, користувач бачить вікно редагування. Окрім полів для налаштування інтеграції з MQTT брокером (рисунок 4.9), користувач має змогу створити конвертер вхідних повідомлень у вигляді байтів.

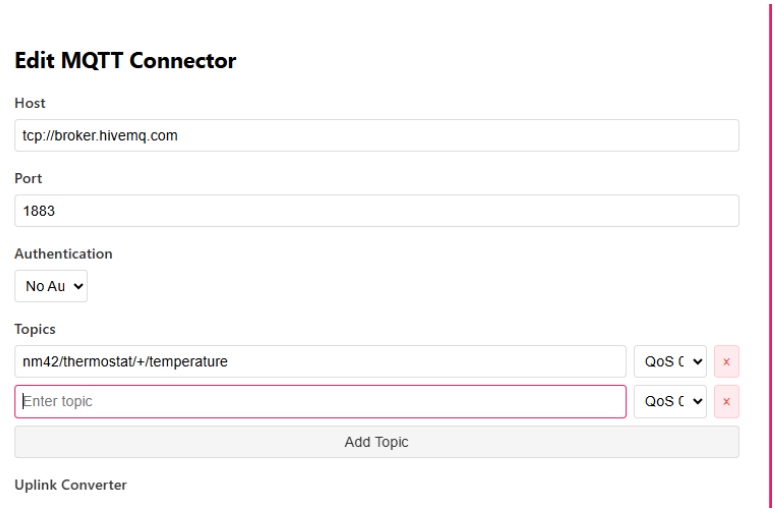


Рисунок 4.9 – Налаштування інтеграції з MQTT брокером

Окрім вхідного конвертера, налаштувати можна й вихідний. (рисунок 4.10).

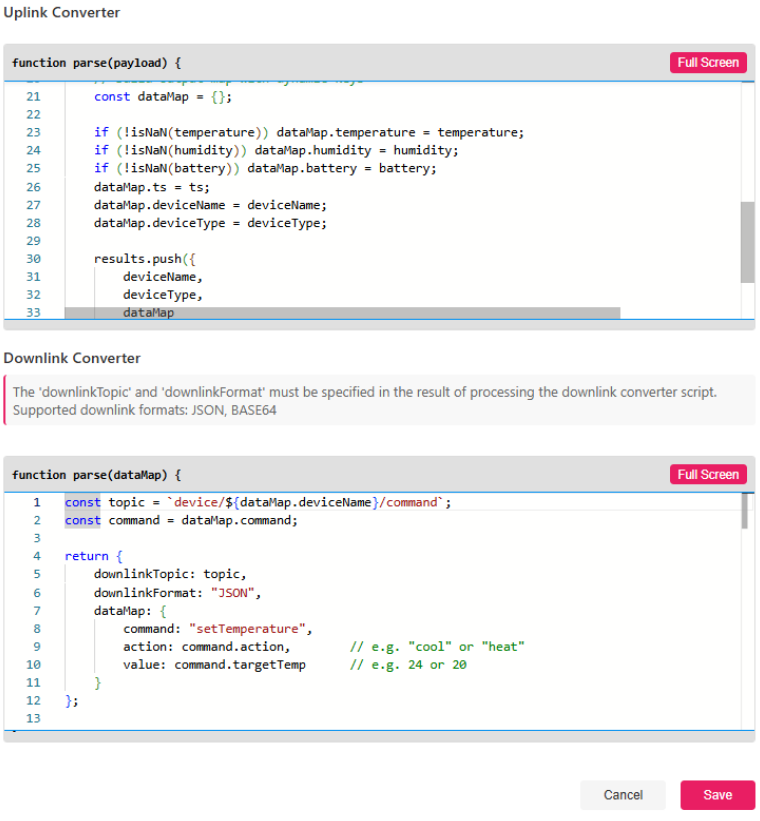


Рисунок 4.10 – Налаштування конвертерів вхідних та вихідних повідомлень

Ці перетворювачі слугують буфером між ланцюжками правил та реальними пристроями. Вони також є доволі гнучкими і дозволяють не прив’язуватись до корисного навантаження окремого виробника апаратів.

4.3 Розділ ланцюжків правил

Ланцюжки правил є останнім розділом у платформі. Потрапляючи на сторінку, можна побачити список із створеними ланцюжками правил, разом з їх назвою та описом (рисунок 4.11). Також є можливість їх видалення та редагування назви та опису.

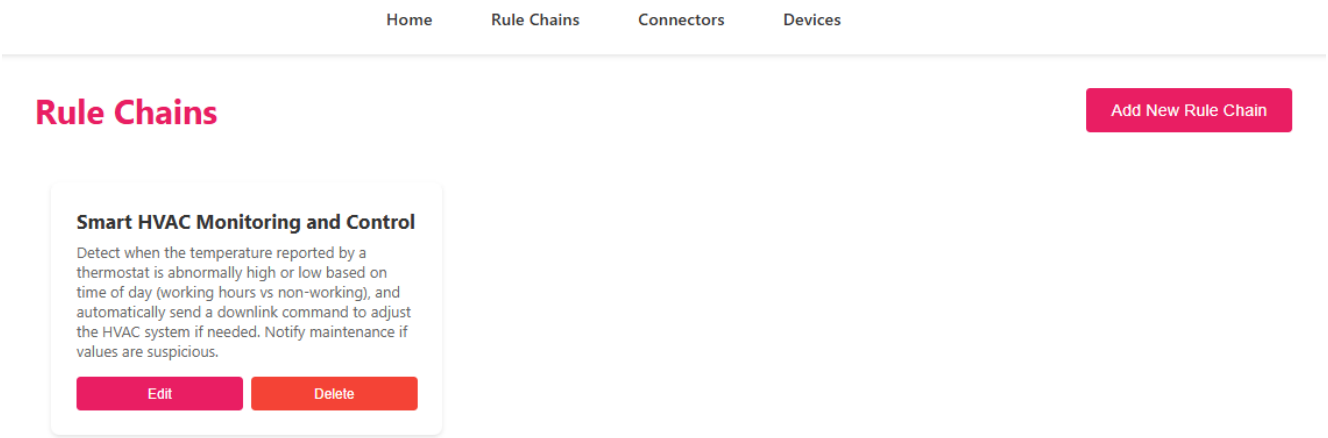


Рисунок 4.11 – Сторінка із списком створених ланцюжків правил

При натисканні на картку окремого ланцюжка, ми переходимо на сторінку з його вершинами (рисунок 4.12). Зліва можна побачити список із доступних типів вершин. Їх можна перетягувати на дошку з усіма вершинами.

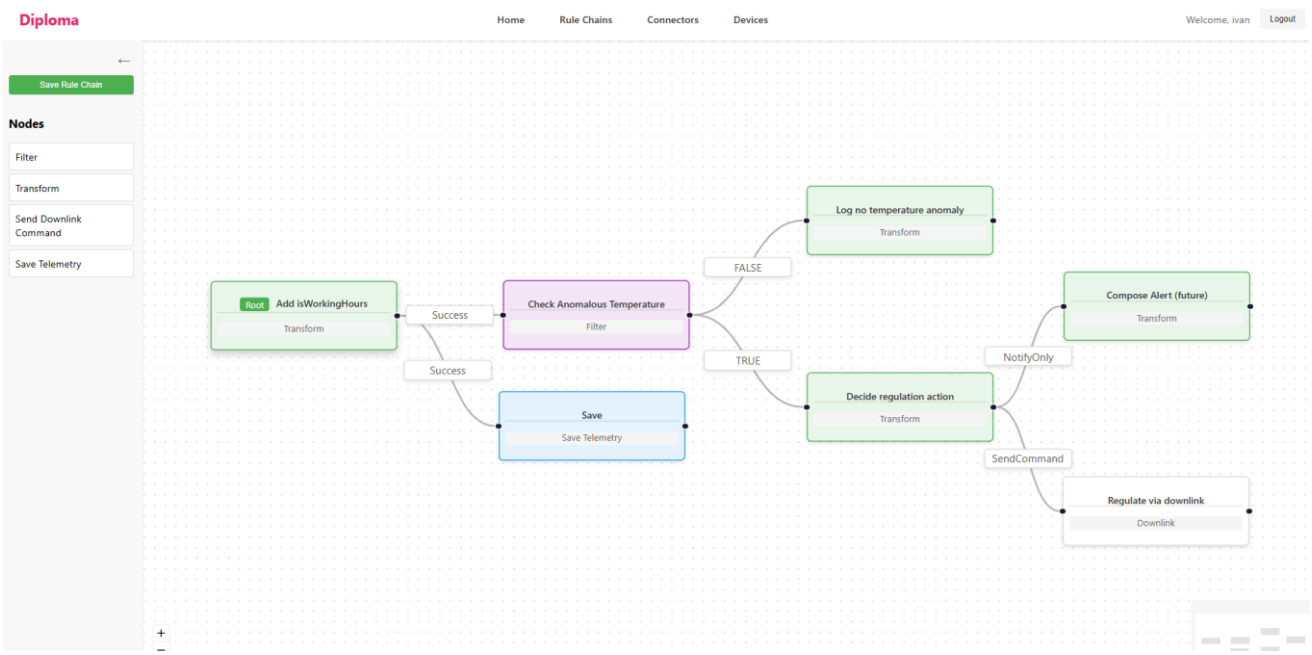


Рисунок 4.12 – Приклад налаштованого ланцюжка правил із вершинами

Наразі платформа підтримує 4 види вершин: фільтраційна, трансформаційна, зберігання телеметричних даних та надсилання повідомлень до пристроїв (рисунок 4.13). Розглянемо кожну окремо.

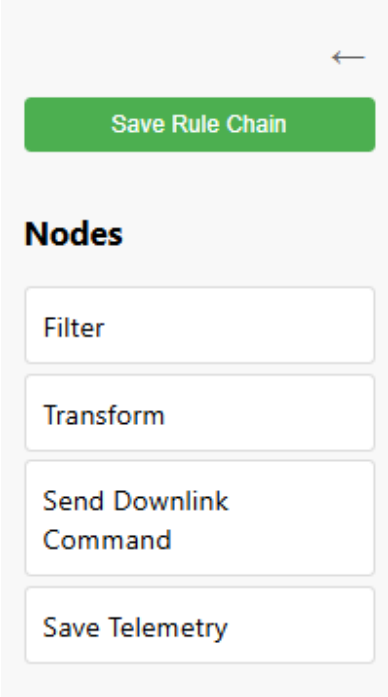
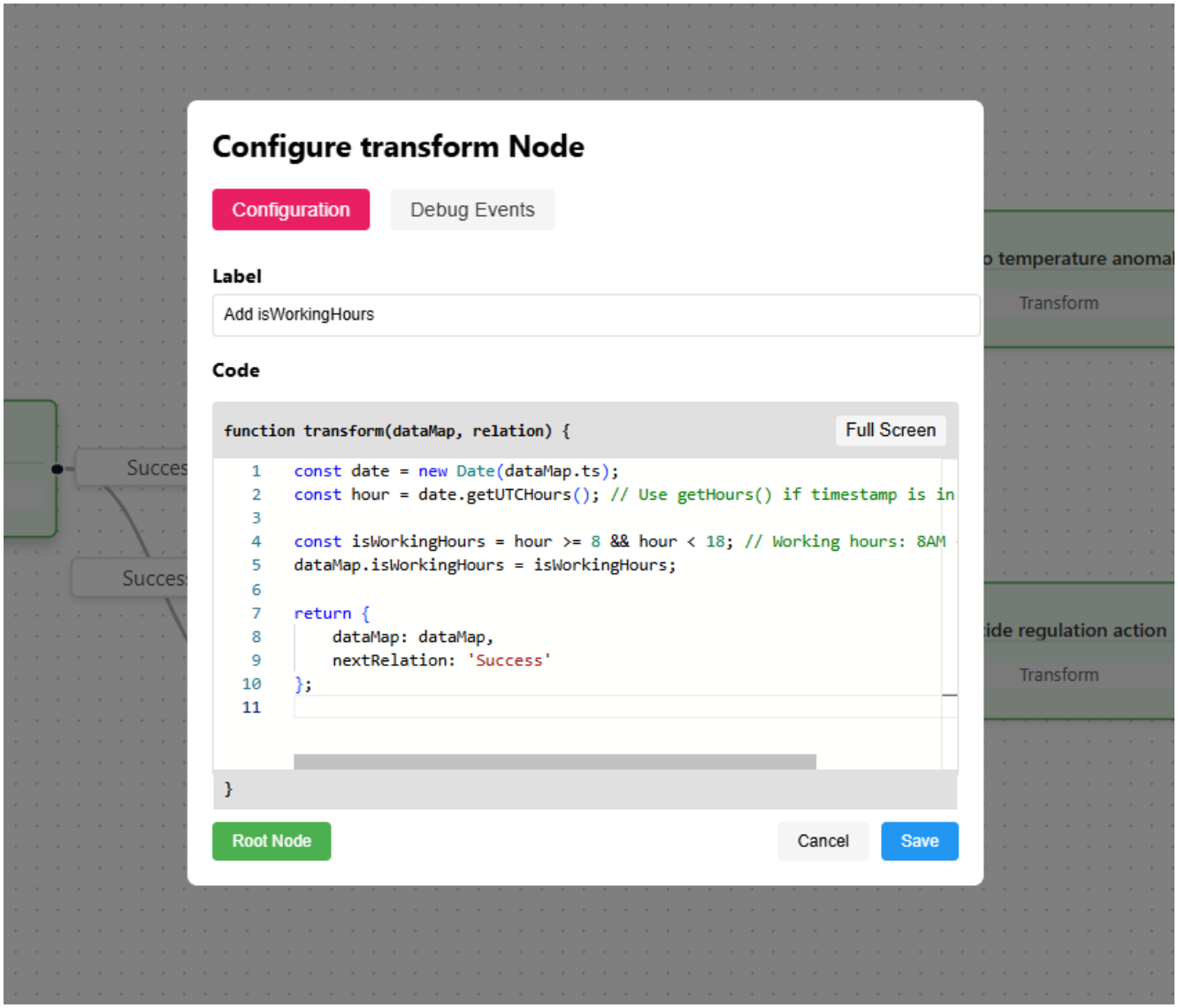


Рисунок 4.13 – Список з доступних типів вершин ланцюжків правил

4.3.1 Вершина трансформування

Трансформаційна вершина представляє собою довільну функцію, параметрами якої є об’єкт та зв’язок. Обидва параметрами є сутностями створеними або батьківською (попередньою) вершиною, або вхідним конвертером у випадку, якщо вершина є кореневою (рисунок 4.14).



Риснок 4.14 – Вікно налаштування трансформаційної вершини

Виходячи з назви типу вершини, її головною метою є трансформація вхідного повідомлення. Користувач може використовувати як логічні операції, так і цикли для реалізації своїх бізнес потреб.

4.3.2 Вершина фільтрування

За своєю конфігурацією фільтруюча вершина має схожість із попередньо розглянутою вершиною. Вона також містить можливість введення довільної функції, параметрами якої є об’єкт та зв’язок. Відмінність полягає у вихідному типу даних: для вершини фільтрування це має обов’язково бути булеве значення, тобто значення true або false (рисунок 4.15).

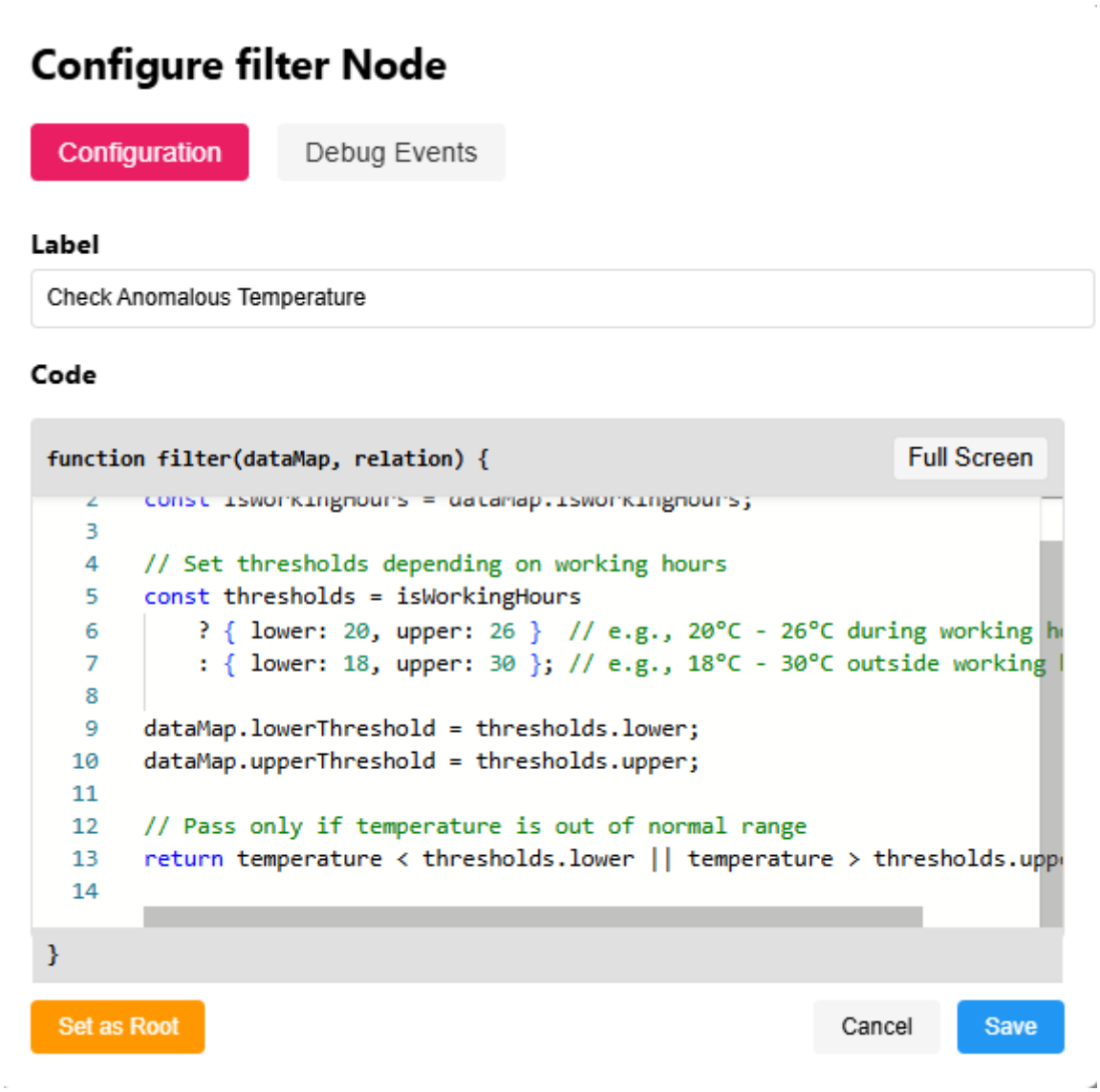


Рисунок 4.15 – Вікно налаштування фільтраційної вершини

Вершина також дозволяє будь-які операції в функції, що дозволяє користувачам реалізовувати комплексні перевірки на основі власних потреб. Популярним прикладом є визначення відхилення даних від норми.

4.3.3 Вершина для зберігання телеметричних даних

Вершина для зберігання телеметричних даних не містить в собі конфігурації, окрім введення назви (рисунок 4.16).

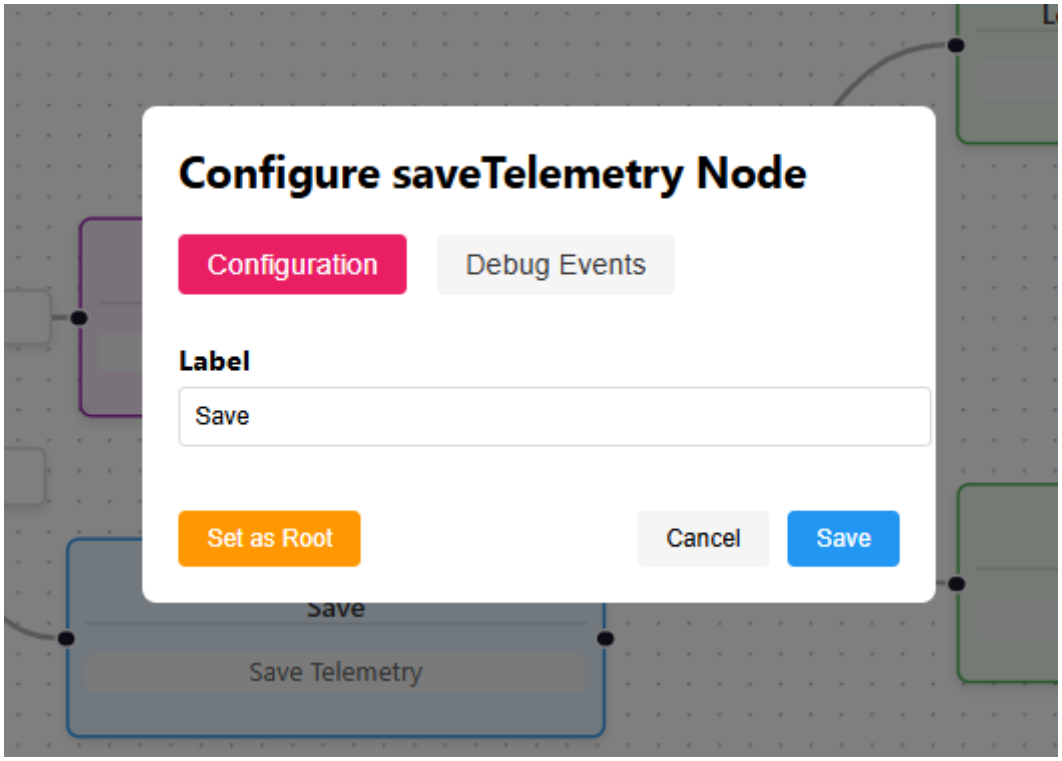


Рисунок 4.16 – Вікно конфігурації вершини, яка зберігає телеметричні дані пристрою

Процес збереження телеметрії для пристрою відбувається таким чином, що кожна пара ключ-значення вхідного повідомлення у вершину буде вважатись окремою точкою в базі даних. Останні збережені дані можна побачити при натисканні на карточку пристрою на відповідній сторінці.

Протягом усього шляху повідомлення з вхідного MQTT/HTTP конвертера та в повідомленнях між вершинами, програма зберігає унікальний ідентифікатор пристрою, за допомогою якого і відбувається зберігання даних.

Розглянута причина є критичною для будь-якого користувача, оскільки пристрої неможливо уявити без даних, а також їх зберігання в базі даних платформи.

4.3.4 Вершина для надсилання повідомлень пристроям

Останньою вершиною, яку наразі підтримує платформа, є вершина для надсилання вихідних повідомлень пристроям. Конфігурація містить назву, протокол обробника (наразі підтримується тільки MQTT) та випадаючий список самих обробників (рисунок 4.17).

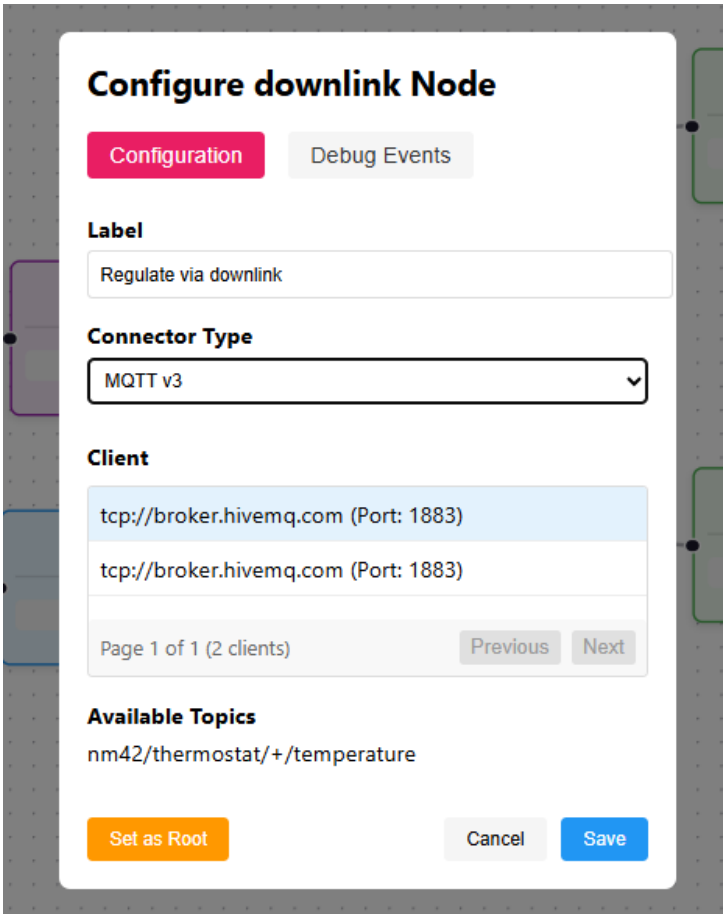


Рисунок 4.17 – Конфігурація вершини надсилання повідомлень пристроям

Ця вершина є корисною для багатьох бізнес випадків, починаючи від систем охолодження та вентиляції, коли потрібно збільшити, або зменшити температуру окремого апарату, або приміщення, закінчуючи системами поливу, які можуть відслідковувати стан вологості газону і у випадку засухи надсилати команду для зволоження.

4.4 Тестування навантаження

Для проведення тесту навантаження було розроблено сценарій, що імітував роботу реальних IoT-пристроїв. Протягом 5 хвилин з інтенсивністю 100 повідомлень на секунду здійснювалась відправка даних на визначений топік MQTT брокера. Кожне повідомлення містило телеметричні дані від двох емульованих IoT-пристроїв, що дозволило оцінити можливості системи з обробки паралельних потоків даних.

Розроблена платформа була налаштована на підписку вхідного топіку MQTT брокера та виконувала наступні операції для кожного отриманого повідомлення відбувається його парсинг, збереження телеметричних даних з повідомлення та генерація й публікація вихідних повідомлень у відповідні топіки для кожного з пристроїв;

Для моніторингу результатів обробки було створено спеціалізований MQTT-клієнт на мові Python, який здійснював підписку на вихідні топіки платформи та фіксував кількість отриманих повідомлень.

За результатами проведеного навантажувального тестування було зафіксовано наступні показники (рисунок 4.18):

- загальна кількість відправлених повідомлень: 30 000;

- загальна кількість отриманих вихідних повідомлень: 60 000;

- коефіцієнт трансформації: 2:1 (відповідно до логіки обробки двох пристроїв);

- втрати повідомлень: 0%.

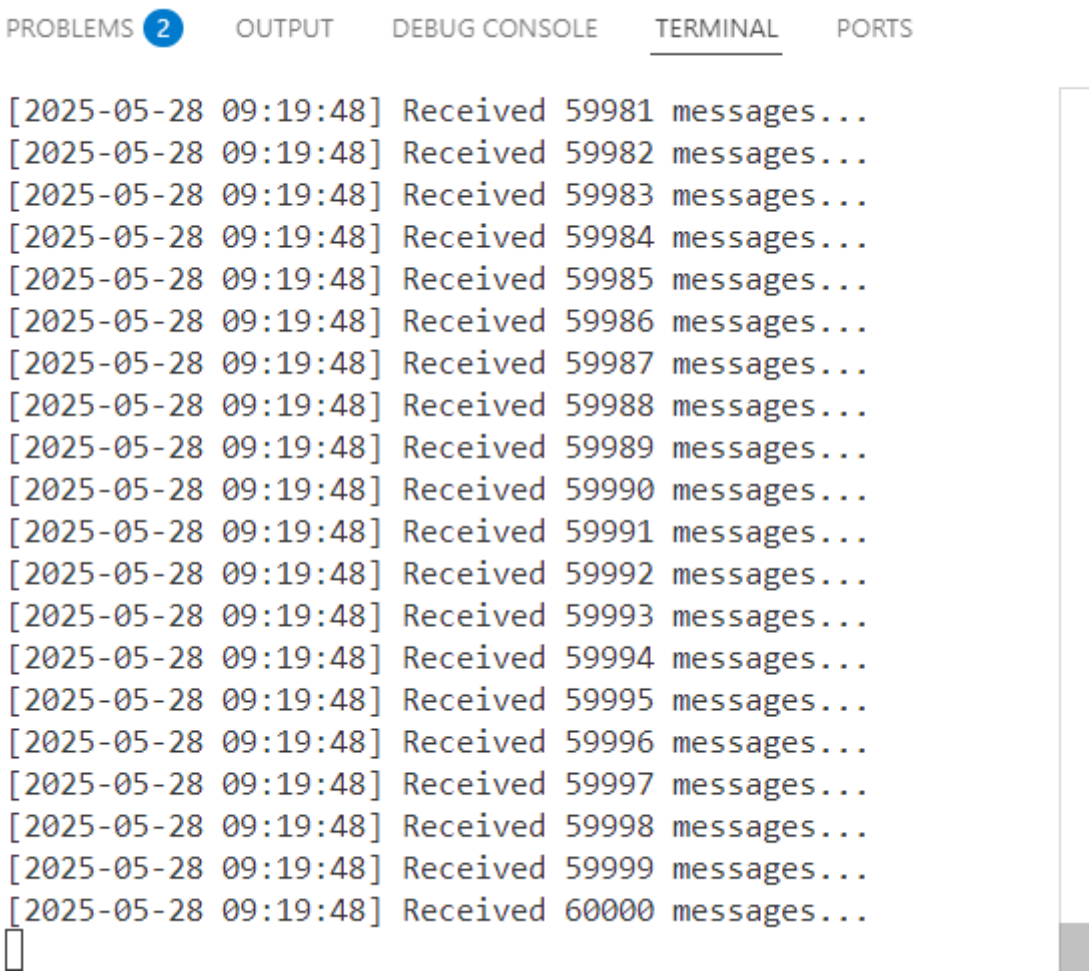


Рисунок 4.18 – Отримані повідомлення MQTT-клієнтом на мові Python

Отримані результати підтверджують стабільність роботи платформи під значним навантаженням та коректність обробки всіх вхідних повідомлень без втрат даних.

ВИСНОВКИ

В ході виконання дипломної роботи було успішно розроблено та впроваджено сучасну мікросервісну платформу для обробки та зберігання даних IoT-пристроїв. Архітектура системи побудована на основі сучасних технологій та практик розробки, що дозволяє забезпечити високу надійність, масштабованість та гнучкість системи.

Платформа включає в себе набір взаємопов'язаних мікросервісів, кожен з яких відповідає за певний функціональний аспект системи. Першим компонентом є discovery-service, який забезпечує реєстрацію та пошук мікросервісів у системі. Центральний сервіс відповідає за базову функціональність пристроїв, зберігання телеметричних даних та управління даними. Сервіс ланцюжків правил забезпечує обробку правил та логіку бізнес-процесів, заданими користувачами платформи, а сервіс підключення пристроїв відповідає за взаємодію із зовнішніми апаратами та системами.

Для забезпечення надійної комунікації між мікросервісами використовується Kafka, що дозволяє реалізувати асинхронну обробку повідомлень та забезпечити високу продуктивність системи. Зберігання даних реалізовано з використанням PostgreSQL, що забезпечує надійність, атомарність та цілісність даних.

API шлюз забезпечує єдину точку входу для клієнтських застосунків, що спрощує взаємодію з системою та забезпечує додатковий рівень безпеки. Всі сервіси інтегровані в єдину мережу, що забезпечує безпечну комунікацію між компонентами.

В результаті розробки було створено сучасну, масштабовану та надійну платформу, яка відповідає сучасним вимогам до IoT-систем та може бути успішно використана для обробки даних від різноманітних пристроїв та сенсорів. Система демонструє високу продуктивність, надійність та здатність до масштабування, що робить її придатною для використання в реальних умовах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. AWS IoT Core Documentation. URL: https://docs.aws.amazon.com/iot/

2. Rules for AWS IoT. URL: https://docs.aws.amazon.com/iot/latest/developerguide/iot-rules.html

3. What is ThingsBoard Rule Engine? URL: https://thingsboard.io/docs/user-guide/rule-engine-2-0/re-getting-started/

4. What is Java? URL: https://aws.amazon.com/what-is/java/

5. Benchmarking Java Virtual Threads: A Comprehensive Analysis. URL: https://www.kloia.com/blog/benchmarking-java-virtual-threads-a-comprehensive-analysis

6. Benchmark JDBC connectors and Java 21 virtual threads. URL: https://mariadb.com/resources/blog/benchmark-jdbc-connectors-and-java-21-virtual-threads/

7. Walls C. Spring in Action, 6th Edition. Shelter Island: Manning Publications, 2022. 520 p.

8. Shapira G., Palino T., Sivaram R., Petty K. Kafka: The Definitive Guide, 2nd Edition. Sebastopol: O'Reilly Media, 2021.

9. Younan K. PostgreSQL 15 Cookbook. Birmingham: Packt Publishing, 2023. 450 p.

10. Seijas Á. GraalVM for Java Developers. Shelter Island: Manning Publications, 2021. 300 p.

11. Hightower K., Burns B., Beda J. Kubernetes: Up & Running, 3rd Edition. Sebastopol: O’Reilly Media, 2022.

12. Ferreira P. Keycloak - Identity and Access Management for Modern Applications. Birmingham: Packt Publishing, 2022. 340 p.

13. Ejsmont A. Web Scalability for Startup Engineers. New York: O'Reilly Media, 2016. 200 p.

14. Vernon V. Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka. Boston: Addison-Wesley Professional, 2015. 480 p.

15. Maison M., Stanley K. Kafka Connect. Sebastopol: O'Reilly Media, 2023.

16. Hillar G. C. MQTT Essentials: A Lightweight IoT Protocol. Birmingham: Packt Publishing, 2017. 280 p.

17. Akka. Architecture model. URL: https://doc.akka.io/concepts/architecture-model.html

ДОДАТОК А
РЕАЛІЗАЦІЯ БІЗНЕС ЛОГІКИ УПРАВЛІННЯ ЛАНЦЮЖКАМИ ПРАВИЛ

Програмні засоби:

- Мова програмування – Java
- Веб-фреймворк – Spring
- Модель акторів: Akka

```
@Service
@RequiredArgsConstructor
public class RuleChainManager {

    private final KafkaTemplate<String, Object> kafkaTemplate;
    private final ApplicationEventPublisher applicationEventPublisher;
    private final RuleMsgProcessorDelegator ruleMsgProcessorDelegator;
    private final RuleChainRepository ruleChainRepository;
    private final RuleChainMapper ruleChainMapper;

    private ActorSystem<RuleChainCommand> actorSystem;

    @EventListener(ApplicationReadyEvent.class)
    public void startUp() {
        HashMap<Long, ActorRef<RuleNodeCommand>> ruleChainIdAndRootRefs = new
HashMap<>();
        actorSystem = ActorSystem.create(MsgDelegatorActor.create(ruleChainIdAndRootRefs),
RuleChainConstants.RULE_CHAIN_SYSTEM);

        ruleChainRepository.getAllBy()
                .forEach(this::initRuleChainInActorSystem);
    }

    @KafkaListener(topics = RuleChainConstants.RULE_CHAIN_INIT_TOPIC, groupId =
"${kafka.self.rule-chain.group-id}",
```

```
            containerFactory = "kafkaListenerContainerFactory",
            concurrency = "${kafka.rule-chain.init-topic-concurrency:2}")
    public void kafkaInitRuleChainListener(RuleChainInitEvent ruleChainInitEvent) {
        RuleChain rc = ruleChainMapper.mapFromDtoToEntity(ruleChainInitEvent.requestRc());
        initRuleChainInActorSystem(rc);
    }

    public void publishInitRuleChainEvent(RuleChainDto rc) {
        kafkaTemplate.send(RuleChainConstants.RULE_CHAIN_INIT_TOPIC, rc.getId().toString(), new
RuleChainInitEvent(rc));
    }

    @KafkaListener(topics = RuleChainConstants.RULE_CHAIN_STOP_TOPIC, groupId =
"${kafka.self.rule-chain.group-id}",
            containerFactory = "kafkaListenerContainerFactory",
            concurrency = "${kafka.rule-chain.stop-topic-concurrency:2}")
    public void kafkaStopRuleChainListener(RuleChainStopEvent ruleChainStopEvent) {
        actorSystem.tell(new StopRuleChainCommand(ruleChainStopEvent.ruleChainId()));
    }

    public void publishStopRuleChainEvent(Long ruleChainId) {
        kafkaTemplate.send(RuleChainConstants.RULE_CHAIN_STOP_TOPIC, ruleChainId.toString(),
new RuleChainStopEvent(ruleChainId));
    }

    public void processIncomingRuleMessage(RuleMessageCommand msg) {
        actorSystem.tell(msg);
    }

    private void initRuleChainInActorSystem(RuleChain ruleChain) {
        actorSystem.tell(new InitRuleChainCommand(ruleChain, applicationEventPublisher,
ruleMsgProcessorDelegator));
    }
}
```

```
public class MsgDelegatorActor extends AbstractBehavior<RuleChainCommand> {

    private final Map<Long, ActorRef<RuleNodeCommand>> ruleChainIdAndRootRefs;

    public static Behavior<RuleChainCommand> create(Map<Long, ActorRef<RuleNodeCommand>>
ruleChainIdAndRootRefs) {
        return Behaviors.setup((context) -> new MsgDelegatorActor(context, ruleChainIdAndRootRefs));
    }

    private MsgDelegatorActor(ActorContext<RuleChainCommand> context, Map<Long,
ActorRef<RuleNodeCommand>> ruleChainIdAndRootRefs) {
        super(context);
        this.ruleChainIdAndRootRefs = ruleChainIdAndRootRefs;
    }

    @Override
    public Receive<RuleChainCommand> createReceive() {
        return newReceiveBuilder()
                .onMessage(InitRuleChainCommand.class, (cmd) -> {
                    boolean terminated = removeRuleChainFromContextIfExists(cmd);
                    if (terminated) {
                        getContext().getLog().info("Rule chain {} with id {} was terminated",
cmd.getRuleChainName(), cmd.getRuleChainId());
                        return Behaviors.same();
                    }
                    return initRuleChain(cmd);
                })
                .onMessage(PostTerminatedRuleChainCommand.class, (cmd) -> {
                    InitRuleChainCommand initCmd = cmd.initCmd;
                    getContext().getLog().info("Updating Rule Chain {} with id {}",
initCmd.getRuleChainName(), initCmd.getRuleChainId());
                    initRuleChain(initCmd);
                    return Behaviors.same();
                })
                .onMessage(RuleMessageCommand.class, (cmd) -> {
```

```
                // todo: lock if rule chain is being initialized

Optional.ofNullable(ruleChainIdAndRootRefs.get(cmd.getRuleChainId())).ifPresentOrElse(
                        ref -> ref.tell(cmd),
                        () -> logRuleChainNotFound(cmd.getRuleChainId())
                );
                return Behaviors.same();
            }).onMessage(StopRuleChainCommand.class, (cmd) -> {
                getContext().getLog().info("Stopping Rule Chain with id {}", cmd.ruleChainId());

Optional.ofNullable(ruleChainIdAndRootRefs.remove(cmd.ruleChainId())).ifPresentOrElse(
                        ref -> getContext().stop(ref),
                        () -> logRuleChainNotFound(cmd.ruleChainId())
                );
                return Behaviors.same();
            })
            .build();
    }

    private Behavior<RuleChainCommand> initRuleChain(InitRuleChainCommand cmd) {
        Optional<RuleNode> rootRuleNode = cmd.ruleChain().getRootRuleNode();
        rootRuleNode.ifPresent(ruleNode -> initRuleNodes(cmd, ruleNode));
        initGeneratorNodes(cmd);
        return Behaviors.same();
    }

    private void initGeneratorNodes(InitRuleChainCommand cmd) {
        // TODO: init generator nodes
    }

    private void initRuleNodes(InitRuleChainCommand cmd, RuleNode rootNode) {
        ActorRef<RuleNodeCommand> rootActor = initRootActor(cmd, rootNode);
        ruleChainIdAndRootRefs.put(cmd.getRuleChainId(), rootActor);

        initGeneratorNodes(cmd);
```

```
    }

    private ActorRef<RuleNodeCommand> initRootActor(InitRuleChainCommand cmd, RuleNode
rootNode) {
        var ruleNodeAndItsRelationsMap = getRuleNodeAndItsRelationsMap(cmd);
        ActorRef<RuleNodeCommand> rootActor = getContext().spawn(RuleNodeActor.create(),
rootNode.getActorName());
        rootActor.tell(getInitRuleNodeCmd(cmd, rootNode, ruleNodeAndItsRelationsMap));

        return rootActor;
    }

    private InitRuleNodeCommand getInitRuleNodeCmd(
            InitRuleChainCommand cmd, RuleNode rootNode, Map<RuleNode, List<ToNodeRelation>>
ruleNodeAndItsRelationsMap) {
        return new InitRuleNodeCommand(
                rootNode, ruleNodeAndItsRelationsMap, cmd.applicationEventPublisher(),
                cmd.ruleMsgProcessorDelegator(), new ActorNodesContext());
    }

    private boolean removeRuleChainFromContextIfExists(InitRuleChainCommand cmd) {
        ActorRef<RuleNodeCommand> rootNodeRef =
ruleChainIdAndRootRefs.remove(cmd.getRuleChainId());
        boolean terminated = false;

        if (rootNodeRef != null) {
            getContext().watchWith(rootNodeRef, new PostTerminatedRuleChainCommand(cmd));
            getContext().stop(rootNodeRef);
            terminated = true;
        }
        return terminated;
    }

    private Map<RuleNode, List<ToNodeRelation>>
getRuleNodeAndItsRelationsMap(InitRuleChainCommand cmd) {
```

```
        return cmd.ruleChain().getRuleNodeRelations()
                .stream()
                .collect(Collectors.groupingBy(
                        RuleNodeRelation::getFromNode,
                        Collectors.mapping(ToNodeRelation::new, Collectors.toList())
                ));
    }

    private void logRuleChainNotFound(Long ruleChainId) {
        getContext().getLog().warn("Rule Chain '{}' not found", ruleChainId);
    }

    private record PostTerminatedRuleChainCommand(InitRuleChainCommand initCmd) implements
RuleChainCommand { }
}

@Slf4j
public class RuleNodeActor {

    public static Behavior<RuleNodeCommand> create() {
        return Behaviors.setup(InitRuleActorReceive::new);
    }
}

@Slf4j
public class InitRuleActorReceive extends AbstractBehavior<RuleNodeCommand> {

    public InitRuleActorReceive(ActorContext<RuleNodeCommand> context) {
        super(context);
    }

    @Override
    public Receive<RuleNodeCommand> createReceive() {
        return newReceiveBuilder()
                .onMessage(InitRuleNodeCommand.class, this::onInitRuleNode)
```

```
            .build();
    }

    private Behavior<RuleNodeCommand> onInitRuleNode(InitRuleNodeCommand cmd) {
        // todo: handle errors(children instantiation, ruleMsgProcessor init, etc)
        Map<String, List<ActorRef<RuleNodeCommand>>> children = spawnChildActors(cmd);
        RuleMsgProcessor ruleMsgProcessor = initRuleMsgProcessorForCurrentNode(cmd);

        Behavior<RuleNodeCommand> postInitBehavior = PostInitRuleActorReceive.create(children,
ruleMsgProcessor);
        return Behaviors.supervise(postInitBehavior)
                .onFailure(Exception.class, SupervisorStrategy.resume());
    }

    private RuleMsgProcessor initRuleMsgProcessorForCurrentNode(InitRuleNodeCommand cmd) {
        RuleNode ruleNode = cmd.selfRuleNode();
        Map<String, String> config = ruleNode.getConfig();
        return cmd.ruleMsgProcessorDelegator().buildRuleMsgProcessor(ruleNode, config);
    }

    private Map<String, List<ActorRef<RuleNodeCommand>>>
spawnChildActors(InitRuleNodeCommand cmd) {
        Map<String, List<ActorRef<RuleNodeCommand>>> relationAndChildMap = new
HashMap<>();

        for (var relationToChild : cmd.getNodeChildRelations()) {
            tryInitChildActorAndLogEvent(cmd, relationToChild, relationAndChildMap);
        }
        return relationAndChildMap;
    }

    private void tryInitChildActorAndLogEvent(InitRuleNodeCommand cmd, ToNodeRelation
relationToChild, Map<String, List<ActorRef<RuleNodeCommand>>> relationAndChildMap) {
        try {
            if (relationToChild.getToRuleNode() == null) {
```

```
                log.warn("Child node is null for relation {}", relationToChild);
                return;
            }
            var childActor = initChildActor(cmd, relationToChild); // todo: make resilient if one of child
nodes can't be initialized
            String relation = relationToChild.getRelation();
            relationAndChildMap.computeIfAbsent(relation, k -> Lists.newArrayList()).add(childActor);
        } catch (Exception e) {
            String debugMessage = String.format(
                    "Error while initializing child actor for relation %s: %s", relationToChild,
e.getMessage());
            saveDebugEvent(cmd, debugMessage);
        }
    }

    private void saveDebugEvent(InitRuleNodeCommand cmd, String debugMessage) {
        cmd.applicationEventPublisher().publishEvent(new
NodeDebugEvent(cmd.selfRuleNode().getId(), debugMessage));
    }

    private ActorRef<RuleNodeCommand> initChildActor(InitRuleNodeCommand cmd,
ToNodeRelation relationToChild) {
        var childNode = relationToChild.getToRuleNode();
        return cmd.actorNodesContext().initIfNotExists(childNode, cn -> createChildActor(cmd, cn));
    }

    private ActorRef<RuleNodeCommand> createChildActor(InitRuleNodeCommand cmd, RuleNode
childNode) {
        var childActor = getContext().spawn(RuleNodeActor.create(), childNode.getActorName());
        var initChildNodeCmd = InitRuleNodeCommand.fromParent(childNode, cmd);

        childActor.tell(initChildNodeCmd);
        return childActor;
    }
}
```

```
@Slf4j
public class PostInitRuleActorReceive extends AbstractBehavior<RuleNodeCommand> {

    private final Map<String, List<ActorRef<RuleNodeCommand>>> children;
    private final RuleMsgProcessor ruleMsgProcessor;

    public PostInitRuleActorReceive(ActorContext<RuleNodeCommand> context,
                                    Map<String, List<ActorRef<RuleNodeCommand>>> children,
                                    RuleMsgProcessor ruleMsgProcessor) {
        super(context);
        this.children = children;
        this.ruleMsgProcessor = ruleMsgProcessor;
    }

    public static Behavior<RuleNodeCommand> create(Map<String,
List<ActorRef<RuleNodeCommand>>> children,
                                                   RuleMsgProcessor ruleMsgProcessor) {
        return Behaviors.setup(ctx -> new PostInitRuleActorReceive(ctx, children, ruleMsgProcessor));
    }

    @Override
    public Receive<RuleNodeCommand> createReceive() {
        return newReceiveBuilder()
                .onMessage(RuleMessageCommand.class, this::onRuleMessage)
                .onMessage(TellNextMessageCommand.class, this::onTellNext)
                .build();
    }

    private Behavior<RuleNodeCommand> onRuleMessage(RuleMessageCommand cmd) {
        ruleMsgProcessor.process(this.getContext().getSelf(), cmd);
        return this;
    }

    private Behavior<RuleNodeCommand> onTellNext(TellNextMessageCommand tellNextCmd) {
```

```
        RuleMessageCommand nextCmd = tellNextCmd.getDelegate();
        List<ActorRef<RuleNodeCommand>> nullableNextChildren =
children.get(nextCmd.getNextType());

        Optional.ofNullable(nullableNextChildren).ifPresentOrElse(
                nextChildren -> doTellNext(nextCmd, nextChildren),
                () -> log.info("Finished processing message '{}', in message processor with type: '{}'",
                        nextCmd, ruleMsgProcessor.getType()) // todo: persist logging for node
        );
        return this;
    }

    private void doTellNext(RuleMessageCommand nextCmd, List<ActorRef<RuleNodeCommand>>
nextChildren) {
        if (CollectionUtils.isEmpty(nextChildren)) {
            throw new IllegalStateException("There's no next child for type " + nextCmd.getNextType());
// todo: persist logging for node
        } else if (nextChildren.size() == 1) {
            nextChildren.getFirst().tell(nextCmd);
        } else {
            nextChildren.forEach(c -> c.tell(nextCmd.copy()));
        }
    }
}
```