НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ

імені адмірала Макарова

Навчально-науковий інститут автоматики і електротехніки
(повне найменування інституту, назва факультету)

Кафедра програмованої електроніки, електротехніки і телекомунікацій
(повна назва кафедри)

«Допущений до захисту»
Завідувач кафедри _____ В. М. Рябенький
д.т.н., професор
« 16 » грудня 2024 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти «магістр»

на тему: Розробка додатку для побудови гнучкої системи управління мережею IoT-пристроїв з використанням мікросервісної архітектури

Виконав: студент 6 курсу, групи 6321М
спеціальності 171
«Електроніка»
шифр і назва спеціальності)

ОПП «Електронні системи»

Железнов В.А.
(прізвище та ініціали студента)

Керівник Солобуто Л. В.
(прізвище та ініціали)

Рецензент Рябенький В. М
(прізвище та ініціали)

м. Миколаїв – 2024 року

Національний університет кораблебудування імені адмірала Макарова

ННІ	автоматики та електротехніки
Кафедра	програмованої електроніки, електротехніки і телекомунікацій
Рівень вищої освіти	другий (магістерський)
Спеціальність	171 «Електроніка»
ОПП	Електронні системи

ЗАТВЕРДЖУЮ
Завідувач кафедри _____ В. М. Рябенький
д.т.н., професор
« 16 » грудня 2024 р.

ЗАВДАННЯ
НА МАГІСТЕРСЬКУ АТЕСТАЦІЙНУ РОБОТУ СТУДЕНТУ

Железнову Віталію Анатолійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка додатку для побудови гнучкої системи управління мережею IoT-пристроїв з використанням мікросервісної архітектури

керівник роботи Солобуто Лариса Вадимівна, к.т.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу
від « 14 » жовтня 2024 року № 1075-уч.

2. Строк подання студентом роботи 15 грудня 2024 р.

3. Вихідні дані до роботи веб-додаток та бібліотеки для IoT-пристроїв

4. Мета дослідження розробити гнучку систему для керування IoT-пристроями

5. Зміст розрахунково-пояснювальної записки
(перелік питань, які потрібно розробити) 1. Теоретичні та технічні рішення
2. Огляд схожих систем. 3. Розробка структури веб-додатку
4. Створення веб-додатку. 5. Налаштування. 6. Охорона праці

6. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
1. Порівняння та переваги.
2. Архітектура системи. Та взаємодія сервісів.
3. Інтерфейс додатку.

7. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Охорона праці	Тубальцев А.М.		

8. Дата видачі завдання « 16 » жовтня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1.	Огляд існуючих рішень	**вересень 2024р.**	**Виконано**
2.	Теоретичні та технічні рішення	**жовтень 2024р.**	**Виконано**
3.	Розробка структури веб-додатку	**жовтень 2024р.**	**Виконано**
4.	Створення веб-додатку	**листопад 2024р.**	**Виконано**
5.	Налаштування	**листопад 2024р.**	**Виконано**
6.	Охорона праці	**листопад 2024р.**	**Виконано**
7.	Оформлення пояснювальної записки	**листопад 2024р.**	**Виконано**

Студент _____ В. А. Железнов
(підпис) (прізвище та ініціали)

Керівник роботи _____ Л. В. Солобуто
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Метою даної дипломної роботи є розробка додатку для управління IoT-пристроями на основі мікросервісної архітектури, який забезпечить створення гнучкої та масштабованої системи, що підтримує інтеграцію з різними протоколами та технологіями, зокрема MQTT і Kafka. У світі сучасних IoT-рішень, де кількість підключених пристроїв швидко зростає, необхідно створювати системи, які дозволяють ефективно обробляти великі обсяги даних та забезпечують надійність і відмовостійкість. Підхід на основі мікросервісів дозволяє спростити управління, адаптацію та інтеграцію, надаючи розробникам потужні інструменти для налаштування автоматизації та моніторингу.

Основними завданнями є розробка серверної частини для реєстрації IoT-пристроїв та обробки запитів, створення веб- або мобільного додатку для зручної взаємодії з користувачами, а також впровадження інструментів для конфігурації автоматизації, що дозволяє налаштовувати сценарії взаємодії між пристроями та забезпечувати ефективну обробку даних у реальному часі. Окрім того, розробляється механізм авторизації на основі GitHub для безпечного управління доступом.

Результати даної роботи можуть бути корисними для розробки та впровадження інноваційних IoT-рішень, які відповідають сучасним вимогам щодо ефективності, масштабованості та безпеки. Створена система відкриває нові можливості для автоматизації процесів в промислових і домашніх системах, сприяє інтеграції з іншими технологіями та полегшує управління великими мережами IoT-пристроїв.

Ключові слова: IoT-пристрої, мікросервісна архітектура, автоматизація, веб-додаток, інтеграція, масштабованість, MQTT, Kafka.

ABSTRACT

The goal of this thesis is to develop an application for managing IoT devices based on microservice architecture, aimed at creating a flexible and scalable system that supports integration with various protocols and technologies, including MQTT and Kafka. In the modern world of IoT solutions, where the number of connected devices is rapidly increasing, there is a need to create systems that efficiently handle large volumes of data while ensuring reliability and fault tolerance. The microservice approach helps simplify device management, adaptation, and integration, providing developers with powerful tools for configuring automation and monitoring.

The main tasks include developing the server-side component for device registration and request processing, designing a web or mobile application for seamless user interaction, and implementing automation configuration tools that enable the setup of interaction scenarios between devices and support real-time data processing. Additionally, an authentication mechanism based on GitHub will be developed to manage system access securely.

The results of this work can be valuable for developing and implementing innovative IoT solutions that meet contemporary requirements for efficiency, scalability, and security. The system created opens up new possibilities for process automation in both industrial and home systems, facilitates integration with other technologies, and simplifies the management of large IoT device networks.

Keywords: IoT devices, microservice architecture, automation, web application, integration, scalability, MQTT, Kafka.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ ТА ТЕРМІНІВ

IoT – Internet of Things (Інтернет речей)

UI – user interface (користувацький інтерфейс)

API – Application Programming Interface (інтерфейс прикладного програмування)

REST – Representational State Transfer (архітектурний стиль для створення веб-сервісів)

JSON – JavaScript Object Notation (формат обміну даними)

DB – Database (база даних)

RDBMS – Relational Database Management System (система керування реляційними базами даних)

SQL – Structured Query Language (мова структурованих запитів)

CRUD – Create, Read, Update, Delete (операції баз даних)

Kafka – Apache Kafka (система обробки повідомлень і потоків даних)

ZK – Zookeeper (система координації, використовується з Kafka)

Docker – Платформа для контейнеризації програмного забезпечення

Container – Ізольоване середовище виконання програм у Docker

Image – Базове середовище Docker, з якого створюються контейнери

Kafka Topic – Канал передачі повідомлень у Kafka

Kafka Producer – Компонент, який надсилає повідомлення в Kafka Topic

Kafka Consumer – Компонент, який отримує повідомлення з Kafka Topic

CI/CD – Continuous Integration/Continuous Deployment (безперервна інтеграція та деплой)

TLS – Transport Layer Security (протокол безпеки для шифрування даних)

MQ – Message Queue (черга повідомлень для обміну даними між сервісами)

ORM – Object-Relational Mapping (об’єктно-реляційне відображення

ВСТУП

Актуальність. Розробка додатків для управління IoT-пристроями з використанням сучасних архітектурних підходів є важливим завданням у світі інтернету речей. В умовах постійного зростання кількості підключених пристроїв та необхідності обробки великих обсягів даних, створення гнучких і масштабованих рішень, що підтримують інтеграцію з різними протоколами та технологіями, стає особливо актуальним. У цьому контексті застосування мікросервісної архітектури дозволяє спростити управління пристроями, забезпечити високу відмовостійкість систем і створити зручний інтерфейс для розробників. Цей підхід також забезпечує гнучкість у налаштуванні та інтеграції з іншими системами, такими як MQTT і Kafka, що підвищує ефективність обробки даних та збереження інформації.

Мета дослідження. Метою даного дослідження є створення додатку для побудови гнучкої системи управління мережею IoT-пристроїв, яка використовує мікросервісну архітектуру для забезпечення ефективної інтеграції пристроїв, обробки даних у реальному часі та підтримки різних сценаріїв використання, включаючи автоматизацію взаємодії між пристроями та інтеграцію з системами збереження даних.

Завдання дослідження. Завданням дослідження є:

- Розробка архітектури мікросервісів для управління IoT-пристроями.
- Створення API, що забезпечує інтеграцію з протоколами MQTT та Kafka для отримання та обробки даних.
- Реалізація сервісів для управління даними та виконання команд на пристроях.
- Розробка механізму авторизації на основі GitHub для управління доступом до системи.

- Створення зручного інтерфейсу на основі React для моніторингу та налаштування системи управління IoT-пристроями.

Об'єкт дослідження. Об'єктом дослідження є система управління IoT-пристроями на основі мікросервісної архітектури, яка забезпечує інтеграцію з різними протоколами та підтримує функції автоматизації.

Предмет дослідження. Предметом дослідження є розробка додатку, який дозволяє управляти мережею IoT-пристроїв, інтегруючи мікросервіси, що обробляють дані та забезпечують взаємодію між пристроями. Особлива увага приділяється розробці інтерфейсу для взаємодії користувачів з системою та інтеграції системи управління з сучасними технологіями обробки даних.

Джерела дослідження. Джерелами дослідження є наукові статті, технічні документації, ресурси, що висвітлюють питання створення мікросервісних додатків, а також досвід розробників у сфері IoT. Важливими джерелами є технічні форуми, блоги розробників, документація MQTT і Kafka, а також приклади реалізацій IoT-рішень.

Наукова новизна одержаних результатів. Наукова новизна полягає у розробці інноваційного додатку, що використовує мікросервісну архітектуру для управління IoT-пристроями. Це рішення передбачає інтеграцію з різними протоколами і технологіями, які сприяють обробці даних і автоматизації взаємодії між пристроями. Створений додаток відповідає сучасним вимогам щодо ефективності, масштабованості та безпеки IoT-рішень.

Практичне значення одержаних результатів. Практичне значення полягає у створенні інструменту для розробників, який дозволяє спрощувати створення та управління мережами IoT-пристроїв, автоматизувати обробку даних і забезпечити зручність інтеграції з іншими системами. Це допоможе розробникам створювати масштабовані та відмовостійкі рішення для різноманітних застосувань у галузі IoT, зокрема для промислових і домашніх систем автоматизації.

РОЗДІЛ 1. ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1. Огляд та порівняльна характеристика існуючих рішень

У сучасних системах управління IoT-пристроями використовуються різні підходи та технології. Ось кілька прикладів відомих рішень:

OpenHAB — платформа для автоматизації будинку, яка дозволяє інтегрувати різноманітні IoT-пристрої та автоматизувати їхню взаємодію. OpenHAB має модульну архітектуру, що забезпечує підключення компонентів через плагіни та сервіси. Центральний сервер обробляє дані та реалізує сценарії автоматизації.

Home Assistant — платформа для домашньої автоматизації, яка підтримує інтеграцію з більш ніж 1800 різними пристроями та протоколами. Мікросервісна архітектура Home Assistant дозволяє налаштовувати систему, підключати нові пристрої та створювати автоматизовані сценарії за допомогою інтеграційних модулів і API.

AWS IoT Core — рішення для управління IoT-пристроями, яке забезпечує підключення, управління та обробку даних від тисяч пристроїв у реальному часі. AWS IoT Core інтегрується з іншими сервісами AWS, такими як Lambda для обробки даних, DynamoDB для збереження інформації і S3 для архівації даних. Підтримка протоколів MQTT і HTTP забезпечує ефективну комунікацію між пристроями та сервісами.

Mosquitto — легкий брокер повідомлень, який часто використовується для передачі даних між пристроями за допомогою протоколу MQTT. Mosquitto є основою для реалізації систем, що потребують підключення численних датчиків і актуаторів, а також надійності та масштабованості при взаємодії між сервісами.

Apache Kafka — розподілена система обміну повідомленнями, що використовується для обробки потоків даних у реальному часі. Kafka забезпечує високу продуктивність, масштабованість і надійність при передачі

великих обсягів даних. Цей підхід часто використовується для обробки даних від різних пристроїв та розподілу їх між мікросервісами.

Порівняння з даним рішенням

Кожне з перерахованих рішень має свої переваги та обмеження. Платформи, як OpenHAB і Home Assistant, підходять для швидкого впровадження та управління IoT-пристроями, але можуть мати обмеження при розробці складних промислових рішень, які потребують масштабованості та гнучкості. Використання AWS IoT Core забезпечує потужну інфраструктуру для управління IoT-пристроями, але може бути дорогим і залежить від хмарних сервісів.

Дане рішення спрямоване на створення гнучкої системи управління IoT-мережею, що поєднує мікросервісну архітектуру, інтеграцію з MQTT і Kafka для обробки даних у реальному часі та інтерфейс на основі React для взаємодії з користувачем. Це забезпечує масштабованість, відмовостійкість та швидкість обробки даних, а також можливість інтеграції з іншими платформами та протоколами.

1.2. Технічні аспекти реалізації систем IoT

У цьому підрозділі розглянемо ключові технічні рішення, що використовуються в системах Інтернету речей (IoT). Аналізуватимемо архітектурні особливості, протоколи зв'язку, інструменти для обробки та збереження даних, а також модулі для інтеграції та розширення функціоналу.

Архітектурні особливості.

- Монолітні архітектури: У монолітних системах усі компоненти розробляються та розгортаються як єдине ціле. Це може бути ефективним для малих проектів, де простота та швидкість розробки є пріоритетом. Однак, такі системи можуть стати важкими для масштабування та підтримки в міру зростання

функціоналу, оскільки будь-яка зміна вимагає повторного розгортання всього додатку.

- Мікросервісні архітектури: Мікросервіси дозволяють розділити систему на незалежні компоненти, кожен з яких відповідає за конкретну функцію. Це забезпечує кращу масштабованість та гнучкість, оскільки окремі сервіси можуть бути оновлені або замінені без впливу на всю систему. Мікросервісна архітектура також полегшує використання різних технологій для різних частин системи.

Використання протоколів.Протоколи комунікації є критично важливими для систем IoT, оскільки вони визначають, як пристрої обмінюються даними. Основні протоколи включають:

- MQTT (Message Queuing Telemetry Transport): Легкий протокол, спеціально розроблений для обмежених мереж з низькою пропускною здатністю. Використовується для передачі повідомлень між пристроями в режимі "публікація-підписка", що дозволяє зменшити навантаження на мережу.
- HTTP (HyperText Transfer Protocol): Загальновідомий протокол, що використовується для передачі даних в Інтернеті. Хоча HTTP не є оптимальним для IoT через його велику накладну витрату, він все ще широко використовується завдяки своїй простоті та інтеграції з веб-технологіями.
- WebSockets: Протокол, що забезпечує двосторонню комунікацію між клієнтом і сервером. Він дозволяє зменшити затримки в передачі даних, що є важливим для реального часу в IoT-додатках.

Інструменти для обробки даних і їхнього збереження.

Обробка та зберігання даних є важливими аспектами IoT-систем, оскільки дані, зібрані з пристроїв, повинні бути доступними для аналізу та прийняття рішень. Основні інструменти включають:

- Бази даних: Використовуються для зберігання структурованих та неструктурованих даних. Реляційні бази даних (наприклад, PostgreSQL, MySQL) підходять для структурованих даних, тоді як NoSQL бази даних (наприклад, MongoDB, Cassandra) є більш гнучкими для роботи з великими обсягами неструктурованих даних.
- Системи обробки потокових даних: Такі системи, як Apache Kafka або Apache Flink, дозволяють обробляти дані в реальному часі, що є критично важливим для IoT-додатків, які потребують миттєвих реакцій на події.

Модулі для інтеграції та розширення функціоналу

Для розширення функціоналу IoT-систем часто використовуються різні модулі та платформи:

- API (Application Programming Interface): API дозволяють інтегрувати різні сервіси та компоненти, що забезпечує можливість взаємодії між ними. Це може включати інтеграцію з хмарними платформами, сторонніми сервісами або іншими IoT-пристроями.
- Платформи IoT: Такі платформи, як AWS IoT, Microsoft Azure IoT або Google Cloud IoT, надають готові рішення для управління пристроями, збору даних та їхнього аналізу. Вони спрощують розробку та впровадження IoT-додатків, надаючи інструменти для масштабування та безпеки.

Узагальнюючи, технічні аспекти реалізації систем IoT є багатогранними і вимагають ретельного підбору архітектур, протоколів, інструментів для обробки даних та модулів для інтеграції, щоб забезпечити ефективність, масштабованість та надійність системи.

1.3. Визначення ключових напрямків для розвитку IoT

Вивчення існуючих рішень у сфері IoT показує, що сучасні системи часто стикаються з низкою обмежень, які впливають на їхню ефективність, масштабованість та інтеграцію. Для покращення управління IoT-мережами необхідно враховувати кілька ключових аспектів, що забезпечать прогрес у розвитку систем.

- Інтеграція між різними системами: Сучасні IoT-рішення часто мають обмежену здатність до інтеграції з іншими системами та платформами. Вдосконалення протоколів та створення адаптивних API, що підтримують різноманітні типи пристроїв і мереж, може значно полегшити процес взаємодії.
- Масштабування та управління великими даними: Для ефективного управління великими обсягами даних, що генеруються IoT-пристроями, необхідні технології, які підтримують масштабованість і реальну обробку даних в режимі реального часу. Це включає використання сучасних систем обробки потокових даних, таких як Apache Kafka, та оптимізацію баз даних для збереження структурованих і неструктурованих даних.
- Безпека та захист даних: Підвищення безпеки залишається важливим напрямком розвитку. Впровадження комплексних механізмів шифрування, автентифікації та авторизації для захисту даних і забезпечення конфіденційності користувачів є необхідністю для будь-якої IoT-системи.
- Гнучкість архітектурних рішень: Перехід від монолітних систем до мікросервісних архітектур забезпечує гнучкість і можливість швидкої модифікації окремих компонентів без впливу на всю систему. Це дозволяє реалізовувати нові функції швидше та з меншими витратами.

- Підтримка різноманітних протоколів комунікації: Розширення підтримки протоколів, таких як MQTT, CoAP та WebSockets, дозволяє знизити навантаження на мережу, забезпечити оперативність передачі даних і спростити інтеграцію між пристроями різних виробників.
- Оптимізація витрат на інфраструктуру: Вибір між використанням хмарних платформ і локальних серверів впливає на загальні витрати системи. Гібридні рішення, що комбінують обидва підходи, можуть дати більшу гнучкість і економічність.
- Підвищення доступності та надійності системи: Створення високодоступних рішень з резервуванням та автоматичним відновленням у разі збою може забезпечити безперебійну роботу IoT-систем.

Врахування цих напрямків дозволяє створити систему управління IoT, яка буде ефективною, масштабованою та адаптивною до змінюваних умов ринку. Інноваційні підходи до розробки таких систем мають значний потенціал для підвищення їхньої ефективності та розширення можливостей у сфері IoT.

Висновок

В результаті огляду було прийнято рішення створити систему управління IoT-пристроями, яка відрізняється високою гнучкістю, масштабованістю та можливістю індивідуального налаштування. Основні переваги цієї системи:

GitHub-авторизація:

- Спрощений процес входу для користувачів і розробників.
- Генерація унікальних ідентифікаторів для пристроїв, прив'язаних до користувача.

Інтерактивне управління пристроями:

- Додавання та налаштування IoT-пристроїв через веб-інтерфейс.
- Автоматизація сценаріїв взаємодії між пристроями без програмування.

Масштабована обробка даних:

- Використання Kafka для обробки великих обсягів даних у реальному часі.
- Надійне зберігання та передача даних між сервісами.
- MQTT-брокер:
- Ефективна взаємодія між пристроями через протокол MQTT.
- Гнучкі налаштування підписок і публікацій для пристроїв.

Мікросервісна архітектура:

- Легкість додавання нових функцій без впливу на інші частини системи.
- Висока стійкість до відмов і простота масштабування.
- Основні відмінності від існуючих систем

На відміну від популярних платформ, таких як OpenHAB, Home Assistant чи AWS IoT Core, запропонована система:

- Пропонує GitHub-авторизацію, що робить її більш зручною для розробників.
- Підтримує налаштування взаємодії між пристроями без програмування, що дозволяє швидко створювати індивідуальні сценарії.
- Використовує Kafka, забезпечуючи ефективну обробку даних для масштабованих IoT-систем.

Ці характеристики роблять систему ідеальним рішенням як для розробників, так і для кінцевих користувачів, які потребують простих та потужних інструментів для управління IoT-пристроями.

					171.6321М.КР.ПЗ.Р2		
Змн.	*Арк.*	*№ документу*	*Підпис*	*Дата*			
Розробник		Железнов В.А.			Архітектура та технології розробки системи	*Літ.*	*Арк.* 16 / *Аркушів*
Консультант							
Н. контр.		Добрінова Л. П.				НУК імені адмірала Макарова	
Керівник		Солобуто Л. В.					
Зав. каф.		Рябенький В. М.					

РОЗДІЛ 2. АРХІТЕКТУРА ТА ТЕХНОЛОГІЇ РОЗРОБКИ СИСТЕМИ

2.1. Основи мікросервісної архітектури

Мікросервісна архітектура (МСА) є сучасним підходом до проєктування програмних систем, який базується на розділенні функціональності додатка на окремі, незалежно розроблювані й розгортані сервіси. Кожен мікросервіс виконує чітко визначену задачу, що відповідає принципу Single Responsibility (єдина відповідальність), і взаємодіє з іншими сервісами через стандартизовані інтерфейси, наприклад, REST API або системи обміну повідомленнями, такі як Kafka.

Основні принципи МСА включають:

- Ізольованість сервісів: кожен мікросервіс є автономним і може оновлюватися чи масштабуватися незалежно від інших.
- Децентралізоване управління даними: кожен сервіс має свою власну базу даних, що забезпечує мінімальну залежність між сервісами.
- Стійкість до збоїв: завдяки незалежності сервісів помилка одного з них не впливає на роботу всієї системи.
- Масштабованість: сервіси можуть масштабуватися горизонтально залежно від навантаження.

Мікросервісна архітектура має численні переваги, серед яких гнучкість, масштабованість і здатність адаптуватися до змінних бізнес-вимог. Проте впровадження МСА не є тривіальним завданням. Воно потребує ретельного планування, забезпечення безпеки та організації ефективної взаємодії між сервісами. Для подолання цих викликів і зменшення ризиків, пов’язаних із розробкою складних розподілених систем, використовуються спеціальні патерни мікросервісів.

Патерни — це перевірені рішення для типових проблем, які виникають під час розробки, розгортання та експлуатації додатків на основі мікросервісної архітектури (МСА). Вони охоплюють різні аспекти роботи з

мікросервісами, зокрема комунікацію, організацію даних, забезпечення стійкості та масштабованості. Серед найбільш поширених патернів можна виділити:

- API Gateway: виступає єдиним пунктом входу в систему, який відповідає за маршрутизацію запитів, автентифікацію та балансування навантаження.
- Database per Service: кожен мікросервіс має свою власну базу даних, що забезпечує ізоляцію даних і зменшує залежності між сервісами.
- Event Sourcing: використання подій як основного джерела інформації для відстеження змін у стані системи.
- Circuit Breaker: запобігає каскадним відмовам системи, обмежуючи запити до сервісу, який вийшов з ладу.
- Saga: управління транзакціями між сервісами шляхом координації подій або управління процесами.

Патерни є ключовим інструментом для впровадження МСА, оскільки вони не тільки спрощують реалізацію, а й сприяють стандартизації систем. Водночас їх застосування має бути ретельно продуманим, з урахуванням конкретних бізнес-вимог і технічних обмежень.

Висновок

Принципи мікросервісної архітектури забезпечують надійність, масштабованість і модульність програмних систем, роблячи їх адаптивними до сучасних потреб. Успішне впровадження МСА можливе лише за умови правильного планування структури додатка та використання відповідних технологій.

2.2. Основні технології для розробки мікросервісів.

У попередньому розділі було розглянуто мікросервісну архітектуру. А в даному аналізуються інструменти, що використовуються для створення програмного забезпечення в цілому.

Першим етапом є розробка користувацького інтерфейсу, який може бути реалізований як у вигляді додатку для конкретної операційної системи, так і у вигляді веб-додатку, доступного через браузер. У даному випадку обирається другий варіант.

Для розробки веб-інтерфейсу використовується бібліотека React, що базується на мові програмування JavaScript. React є ефективним інструментом для створення односторінкових застосунків (Single Page Applications, SPA), які дозволяють оновлювати контент без перезавантаження сторінки, що забезпечує швидкий і плавний досвід взаємодії для користувача.

Далі розглядається приклад підключення React до застосунку та базові методи роботи з ним. Для початку необхідно створити базовий проект за допомогою інструменту Create React App. Команда для ініціалізації проекту має наступний вигляд:

```
npx create-react-app my-app
cd my-app
npm start
```

Після виконання зазначених команд проект запускається в браузері за замовчуванням за адресою: http://localhost:3000/. На екрані відображається початкова сторінка React. Наступним етапом є аналіз структури проекту та додавання базової функціональності. Після ініціалізації проекту структура має вигляд, представлений на (рисунку-2.1):

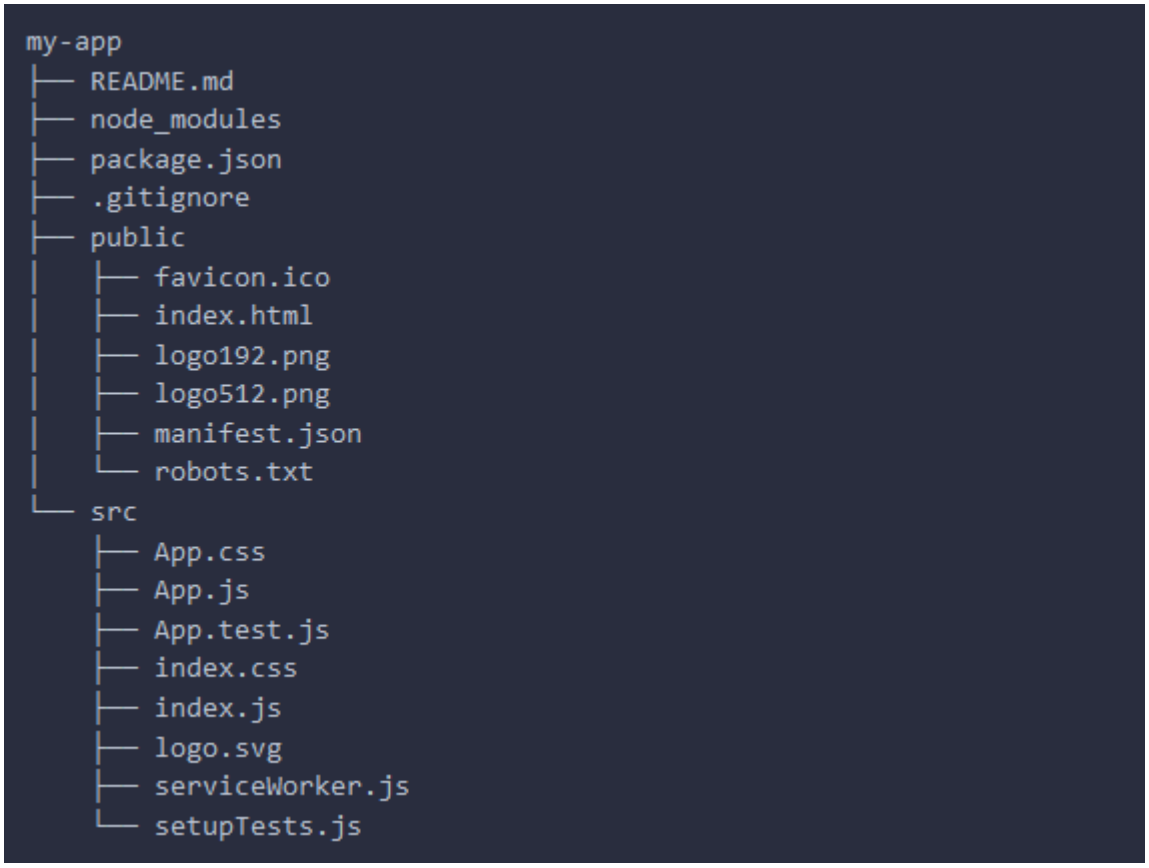

Рисунок 2.1 – Приклад файлової структури проекту.

Компонент App.js є початковою точкою додатку. Він зазвичай містить основну логіку та структуру інтерфейсу. Стандартний вигляд компонента App.js представлено у наступному коді:

```
import React from "react";
import "./App.css";

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <h1>Мій перший React-додаток</h1>
        <p>Це стартова сторінка для вашого додатка!</p>
      </header>
    </div>
  );
}

export default App;
```

React базується на використанні компонентів, що дозволяє створювати нові компоненти та зберігати їх у файлах для подальшого використання в додатку. Наприклад, компонент DeviceList.js може бути розроблений для відображення списку IoT-пристроїв. Стандартний вигляд цього компонента може виглядати наступним чином:

```
import React from "react";

const DeviceList = ({ devices }) => {
  return (
    <div>
      <h2>Список пристроїв</h2>
      <ul>
        {devices.map((device, index) => (
          <li key={index}>
            {device.name} - Статус: {device.status}
          </li>
        ))}
      </ul>
    </div>
  );
};

export default DeviceList;
```

Подальше підключення компонента DeviceList до App.js забезпечує інтеграцію та використання цього компонента в основному додатку. Стандартний вигляд підключення може виглядати наступним чином:

```
import React from "react";
import DeviceList from "./DeviceList";

function App() {
  const devices = [
    { name: "Датчик температури", status: "Активний" },
    { name: "Лампа", status: "Вимкнена" },
    { name: "Розумна розетка", status: "Активна" },
  ];

  return (
    <div className="App">
      <header className="App-header">
        <h1>IoT-Додаток</h1>
```

```
        <DeviceList devices={devices} />
      </header>
    </div>
  );
}

export default App;
```

Для стилізації React-додатку можна використовувати CSS або бібліотеки, такі як Bootstrap чи Material-UI. Для встановлення Bootstrap необхідно виконати наступні дії:

```
npm install bootstrap
```

Додайте його до index.js:

```
import "bootstrap/dist/css/bootstrap.min.css";
```

Тепер існує можливість використовувати класи Bootstrap у компонентах React-додатку.

Далі розглядається інструмент для створення мікросервісів, зокрема Flask. Flask є легким і гнучким веб-фреймворком для Python, що дозволяє швидко розробляти веб-додатки та API. Його простота у використанні та можливість розширення роблять його оптимальним вибором для розробки стартапів і мікросервісів. Наступні кроки описують процес створення простого веб-додатку на базі Flask.

Перед початком необхідно переконатися, що Python встановлений. Завантажити його можна з офіційного сайту. Після цього слід відкрити термінал і виконати команду для встановлення Flask:

```
pip install Flask
```

Наступним кроком є створення нової папки для даного прикладу, після чого слід перейти до неї. Це дозволить організувати проект у відокремленому каталозі. Для цього можна виконати наступні команди в терміналі:

```
mkdir my_flask_app
cd my_flask_app
```

Для написання додатку в першу чергу треба створити файл app.py, а потім в редакторі коду додати своє рішення, наприклад:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'

if __name__ == '__main__':
    app.run(debug=True)
```

Для запуску у локальній мережі на порту 5000 необхідно ввести наступну команду в консолі:

```
py або python адресса_директорії_проекту/app.py
```

Наступним етапом є збірка створених мікросервісів у контейнер. Для цього використовуватиметься Docker, що забезпечує упаковку Flask-додатку разом з усіма його залежностями в ізольоване середовище. Це сприяє спрощенню процесів розгортання та масштабування додатку в різних середовищах.

Для створення образу Docker необхідно створити файл Dockerfile у директорії сервісу та налаштувати його:

```
# Вибір базового образу
FROM python:3.9-slim

# Встановлення робочої директорії
WORKDIR /app

# Копіювання файлів
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
COPY . .

# Визначення команди запуску
CMD ["python", "app.py"]
```

Необхідно створити файл requirements.txt, що міститиме перелік усіх необхідних бібліотек. У головній директорії проекту слід також створити файл docker-compose.yml, який дозволяє визначити та запустити багатоконтейнерні Docker-додатки. Приклад структури файлу docker-compose.yml для двох мікросервісів: web та db (для бази даних) наведено нижче.

```
version: '3.8'

services:
  web:
    build: ./web
    ports:
      - "5000:5000"
    depends_on:
      - db

  db:
    image: postgres:13
    environment:
      POSTGRES_DB: mydatabase
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
    ports:
      - "5432:5432"
```

- version: Вказує версію Docker Compose.
- services: Визначає мікросервіси, які будуть запущені.
- web: Це наш веб-сервіс, який будується з Dockerfile, розташованого в папці ./web. Порт 5000 на контейнері буде доступний на порту 5000 хоста.
- depends_on: Вказує, що сервіс web залежить від сервісу db.
- db: Це сервіс бази даних, який використовує образ PostgreSQL.

PostgreSQL — це потужна об'єктно-реляційна система управління базами даних (СУБД), яка забезпечує високу надійність, гнучкість та масштабованість. Вона підтримує широкий спектр функціональних можливостей, таких як транзакції, зберігання даних у JSON-форматі, повнотекстовий пошук, а також розширюваність через користувацькі типи даних і функції.

PostgreSQL забезпечує можливість зберігання та обробки великих обсягів даних, а також підтримує різноманітні типи запитів. Завдяки своїй архітектурі, PostgreSQL може використовуватися як для малих, так і для

великих проектів, що потребують надійного зберігання даних. Система підтримує стандарт SQL, а також надає можливість написання запитів на різних мовах програмування, таких як PL/pgSQL, PL/Perl, PL/Python та інші.

Для роботи з PostgreSQL зазвичай використовуються клієнтські бібліотеки, які дозволяють виконувати запити до бази даних, обробляти результати та управляти з'єднаннями. Зазвичай використовуються такі бібліотеки, як :

- psycopg2 - Python,
- pg - Node.js
- JDBC - Java.

Приклад простого додатку на Python, який використовує бібліотеку psycopg2 для роботи з PostgreSQL:

```
import psycopg2

# Підключення до бази даних
connection = psycopg2.connect(
    dbname="mydatabase",
    user="user",
    password="password",
    host="localhost",
    port="5432"
)

# Створення курсора для виконання запитів
cursor = connection.cursor()

# Створення таблиці
cursor.execute("""
CREATE TABLE IF NOT EXISTS users (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100)
)
""")

# Додавання даних
cursor.execute("""
INSERT INTO users (name, email) VALUES (%s, %s)
""", ("John Doe", "john.doe@example.com"))
```

```
# Збереження змін
connection.commit()

# Отримання даних
cursor.execute("SELECT * FROM users")
rows = cursor.fetchall()

for row in rows:
    print(row)

# Закриття курсора та з'єднання
cursor.close()
connection.close()
```

PostgreSQL є потужним інструментом для зберігання та обробки даних, що робить його підходящим вибором для різноманітних проектів. Завдяки своїй гнучкості та можливостям розширення, система може адаптуватися до потреб користувачів, забезпечуючи ефективну роботу з даними.

У цьому розділі розглянуто основні інструменти для розробки програмного забезпечення, зокрема веб-додатків на базі React та мікросервісів з використанням Flask. React забезпечує створення динамічних односторінкових застосунків, а Flask є легким фреймворком для швидкої розробки API. Використання Docker спрощує розгортання мікросервісів, а PostgreSQL надає надійний механізм для зберігання даних. Ці інструменти формують потужну екосистему для розробки сучасних рішень, що відповідають вимогам. Наступні розділи присвячені інтеграції цих компонентів у єдину систему.

2.3. Забезпечення взаємодії між мікросервісами

У цьому підрозділі розглядаються різні підходи та технології, що забезпечують ефективну взаємодію між мікросервісами, зокрема в контексті архітектури, описаної в попередньому розділі.

У сучасній архітектурі мікросервісів забезпечення ефективної комунікації між сервісами є критично важливим. Одними з найбільш популярних рішень для реалізації асинхронної взаємодії є Apache Kafka та MQTT, кожен з яких має свої особливості, переваги та недоліки.

Apache Kafka — це розподілена платформа для потокової обробки даних, яка забезпечує високу пропускну здатність, надійність та масштабованість. Kafka ідеально підходить для мікросервісів, дозволяючи їм обмінюватися повідомленнями через концепцію тем (topics) за допомогою моделі "publish-subscribe".

Топіки — це категорії, в які публікуються повідомлення. Кожен топік може мати кілька партицій, що дозволяє розподіляти навантаження та забезпечувати паралельну обробку. Споживачі (consumers) підписуються на топіки і отримують повідомлення в реальному часі.

Щоб розгорнути Apache Kafka в Docker, можна використовувати Docker Compose. Ось приклад **docker-compose.yml** файлу для розгортання Kafka разом із Zookeeper:

```
version: '2'
services:
  zookeeper:
    image: wurstmeister/zookeeper:3.4.6
    ports:
      - "2181:2181"

  kafka:
    image: wurstmeister/kafka:latest
    ports:
      - "9092:9092"
    environment:
      KAFKA_ADVERTISED_LISTENERS: INSIDE://kafka:9092,OUTSIDE://localhost:9092
      KAFKA_LISTENERS: INSIDE://0.0.0.0:9092,OUTSIDE://0.0.0.0:9092
```

```
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
    depends_on:
      - zookeeper
```

Для запуску Kafka і Zookeeper достатньо виконати команду:

```
docker-compose up -d
```

Мікросервіси можуть інтегруватися з Kafka через бібліотеки, такі як:

- kafka-python - Python
- kafkajs - Node.js.

Ось приклад коду на Python для публікації повідомлень у топік:

```
from kafka import KafkaProducer

producer = KafkaProducer(bootstrap_servers='localhost:9092')
producer.send('my_topic', b'Hello, Kafka!')
producer.close()
```

MQTT (Message Queuing Telemetry Transport) — легкий протокол обміну повідомленнями, оптимізований для роботи в умовах обмежених ресурсів, що робить його ідеальним для IoT-пристроїв.

Для підключення IoT-пристроїв, таких як ESP32, до брокера MQTT, можна використовувати Arduino IDE, яка дозволяє програмувати ESP32 за допомогою мови C/C++.

Для початку потрібно встановити бібліотеку PubSubClient, яка забезпечує підтримку MQTT для ESP32. В Arduino IDE це можна зробити через "Менеджер бібліотек".

Ось приклад коду, який демонструє, як підключити ESP32 до MQTT-брокера та публікувати повідомлення:

```
#include <WiFi.h>
#include <PubSubClient.h>

// Задайте ваші дані Wi-Fi
const char* ssid = "YOUR_SSID"; // Ваша назва Wi-Fi мережі
```

const char* password = "YOUR_PASSWORD"; // Ваш пароль Wi-Fi

// Задайте дані для MQTT

const char* mqttServer = "mqtt_broker_address"; // Адреса вашого MQTT брокера

const int mqttPort = 1883; // Порт MQTT брокера

const char* mqttUser = "YOUR_MQTT_USERNAME"; // Ваше ім'я користувача MQTT (якщо потрібно)

const char* mqttPassword = "YOUR_MQTT_PASSWORD"; // Ваш пароль MQTT (якщо потрібно)

```
WiFiClient espClient;
PubSubClient client(espClient);

void setup() {
  Serial.begin(115200);

  // Підключення до Wi-Fi
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Підключення до Wi-Fi...");
  }
  Serial.println("Підключено до Wi-Fi");

  // Налаштування MQTT сервера
  client.setServer(mqttServer, mqttPort);
}

void loop() {
  // Перевірка з'єднання з MQTT
  if (!client.connected()) {
    reconnect();
  }
  client.loop();

  // Публікація повідомлення
  client.publish("my/topic", "Hello from ESP32!");
  delay(5000); // Затримка між публікаціями
```

```
}

void reconnect() {
  // Цикл до підключення
  while (!client.connected()) {
    Serial.print("Підключення до MQTT...");
    if (client.connect("ESP32Client", mqttUser , mqttPassword)) {
      Serial.println("підключено");
    } else {
      Serial.print("Помилка підключення, код: ");
      Serial.print(client.state());
      delay(2000);
    }
  }
}
```

В випадку зустрічі з придбаними IoT-пристроями, які підтримують MQTT, часто є можливість налаштувати з'єднання з брокером через веб-інтерфейс або мобільний додаток. Зазвичай це включає:

- Вхід до налаштувань пристрою: Зазвичай через IP-адресу пристрою в браузері або через спеціальний додаток.
- Знайти налаштування MQTT: Вони можуть бути в розділі "Налаштування мережі" або "IoT".
- Введення параметрів MQTT:
  - Адреса брокера: Введіть IP-адресу або доменне ім'я брокера.
  - Порт: Зазвичай 1883 для незахищених з'єднань.
  - Ім'я користувача та пароль: Якщо ваш брокер вимагає аутентифікацію, введіть ці дані.
  - Топіки: Вкажіть, на які топіки пристрій буде підписуватися або публікувати повідомлення.
- Збереження налаштувань: Після введення всіх даних збережіть налаштування та перезавантажте пристрій, якщо це необхідно.

Мікросервіси можуть підписуватися на топіки брокера MQTT для отримання повідомлень. Це можна реалізувати за допомогою тих же бібліотек, наприклад, paho-mqtt.

Ось приклад коду на Python для підписки на топік:

```
import paho.mqtt.client as mqtt

def on_message(client, userdata, message):
    print(f"Received message: {message.payload.decode()} on topic: {message.topic}")

client = mqtt.Client()
client.on_message = on_message

client.connect("mqtt_broker_address", 1883, 60)
client.subscribe("my/topic")
client.loop_forever()
```

Використання ESP32 для підключення до MQTT-брокера є простим і ефективним, дозволяючи реалізувати різноманітні IoT-додатки. Налаштування заводських IoT-пристроїв також може бути легким, якщо вони підтримують MQTT, що дозволяє легко інтегрувати їх у вашу систему, а Apache Kafka є оптимальним для мікросервісів, які потребують високої продуктивності та надійності.

Висновок

В цьому розділі були охоплені основи мікросервісної архітектури, технології для розробки мікросервісів, а також забезпеченна їх взаємодія. Мікросервісна архітектура забезпечує надійність, масштабованість і модульність програмних систем, роблячи їх адаптивними до сучасних потреб. Використання React, Flask, Docker та PostgreSQL формує потужну екосистему для розробки сучасних рішень. Інтеграція таких технологій, як Apache Kafka і MQTT, дозволяє реалізувати ефективну комунікацію між мікросервісами, що є критично важливим для побудови складних розподілених систем. Успішне впровадження цих технологій можливе лише за умови правильного планування та врахування конкретних бізнес-вимог.

					171.6321М.КР.ПЗ.РЗ		
Змн.	*Арк.*	*№ документу*	*Підпис*	*Дата*			
Розробник		Железнов В.А.			Розробка додатку. Налаштування	*Літ.*	*Арк.* 32 / *Аркушів*
Консультант							
Н. контр.		Добрінова Л. П.				НУК імені адмірала Макарова	
Керівник		Солобуто Л. В.					
Зав. каф.		Рябенький В. М.					

РОЗДІЛ 3. РОЗРОБКА ДОДАТКУ. НАЛАШТУВАННЯ

3.1. Розробка структури додатку

API Gateway:

- Роль: Керує маршрутизацією запитів, аутентифікацією та авторизацією.
- Технології: Flask з Flask-RESTful для створення REST API.
- Функції:
  - Перенаправлення запитів до мікросервісів.
  - Обробка авторизації через GitHub.
  - Управління токенами JWT для захисту API.

Мікросервіс авторизації через GitHub:

- Роль: Забезпечує аутентифікацію користувачів через GitHub і генерує дані для підключення IoT-пристроїв.
- Технології: Flask.
- Функції:
  - Авторизація за допомогою OAuth2.
  - Генерація унікальних ідентифікаторів для пристроїв та прив'язка їх до користувача.
  - Управління профілями користувачів та їхніми даними.

Сервіс управління пристроями:

- Роль: Керує пристроями, їхніми налаштуваннями та станом, а також топіками та авто-відповідями.
- Технології: Flask.
- Функції:
  - Додавання та видалення пристроїв.
  - Управління топіками та налаштуваннями підписок.
  - Управління налаштуваннями авто-відповідей.

Сервіс моніторингу та управління топіками MQTT:

- Роль: Керує підписками на MQTT-топіки та налаштуваннями для отримання та відправки повідомлень.
- Технології: Flask з використанням бібліотеки paho-mqtt.
- Функції:
  - Підключення до MQTT-брокера.
  - Налаштування підписок і публікацій на топіки.
  - Обробка вхідних повідомлень та управління авто-відповідями.

Сервіс обробки даних:

- Роль: Отримує дані від пристроїв через MQTT, обробляє їх і передає в Kafka.
- Технології: Flask.
- Функції:
  - Отримання та форматування даних з MQTT.
  - Передача даних у Kafka для подальшої обробки.
  - Взаємодія з базами даних для зберігання проміжних результатів.

Сервіс обробки команд:

- Роль: Обробляє команди від користувача та відправляє їх пристроям.
- Технології: Flask.
- Функції:
  - Отримання команд через API.

- Форматування та відправка команд на пристрої через MQTT.
- Управління авто-відповідями.

UI (Frontend):

- Роль: Інтерфейс користувача для управління пристроями, налаштуваннями та даними.
- Технології: React (можна використовувати Next.js для серверного рендерингу).
- Функції:
  - Відображення даних про пристрої та їхні стани.
  - Управління підписками та налаштуваннями топіків.
  - Управління авто-відповідями та отриманими даними.

Kafka:

- Роль: Брокер повідомлень для передачі даних між сервісами та їх зберігання.
- Технології: Apache Kafka.
- Функції:
  - Зберігання даних у топіках, пов'язаних з користувачами та їхніми пристроями.
  - Обробка великих обсягів даних у реальному часі.
  - Підтримка патрицій для ізоляції даних пристроїв.

MQTT брокер:

- Роль: Забезпечує зв'язок між IoT-пристроями та мікросервісами.
- Технології: Mosquitto, EMQ X, HiveMQ.
- Функції:

- Отримання та відправка повідомлень між пристроями та сервісами.
- Підтримка підписок і публікацій на топіки.
- Забезпечення надійного зв'язку між пристроями та системою.

Взаємодія між компонентами:

UI підключається до API Gateway для отримання даних і відправки команд.

API Gateway перенаправляє запити до відповідних мікросервісів, включаючи авторизацію та управління пристроями.

Сервіс авторизації використовує GitHub для аутентифікації та управління користувачами, а також генерує ідентифікатори для пристроїв.

Сервіс управління пристроями та сервіс моніторингу топіків підписуються на MQTT-топіки для обробки даних.

Сервіс обробки даних отримує повідомлення від MQTT-брокера, обробляє їх і відправляє в Kafka.

Сервіс обробки команд взаємодіє з MQTT-брокером для відправки команд і авто-відповідей.

Kafka слугує для передачі даних між мікросервісами та зберігання їх для аналітики та звітності.

MQTT брокер забезпечує зв'язок між пристроями та мікросервісами.

Тепер, коли визначені основні сервіси, можна перейти до створення відповідної структури директорій для даного проекту. Це дозволить організувати код у зрозумілій та логічній формі, що полегшить подальшу розробку та підтримку проекту.

Нижче представлена структура директорій, що відображає організацію мікросервісної частини IoT проекту:

```
IoT_Project/
│
├── api_gateway/
│   ├── app.py
│   ├── Dockerfile
│   ├── requirements.txt
│   ├── .dockerignore
│   └── venv/
│
├── auth_service/
│   ├── app.py
│   ├── Dockerfile
│   ├── requirements.txt
│   ├── .dockerignore
│   └── venv/
│
├── device_service/
│   ├── app.py
│   ├── Dockerfile
│   ├── requirements.txt
│   ├── .dockerignore
│   └── venv/
│
├── mqtt_service/
│   ├── app.py
│   ├── Dockerfile
│   ├── requirements.txt
│   ├── .dockerignore
│   └── venv/
│
├── data_processing_service/
│   ├── app.py
│   ├── Dockerfile
│   ├── requirements.txt
│   ├── .dockerignore
│   └── venv/
│
├── command_service/
│   ├── app.py
│   ├── Dockerfile
│   ├── requirements.txt
│   ├── .dockerignore
│   └── venv/
│
├── ui/
```

```
|   ├── App.js
|   ├── package.json
|   ├── package-lock.json
|   └── node_modules/
|
└── docker-compose.yml
```

Ця структура дозволяє чітко розділити кожен мікросервіс та його компоненти, що полегшує роботу над проектом і забезпечує легкість в подальшій розробці.

3.2. Розробка мікросервісної частини

У цьому розділі детально викладено реалізацію кожного з мікросервісів, описаних у попередньому розділі. Кожен мікросервіс буде реалізовано відповідно до специфікацій, а також будуть представлені технології, що використовуються, та основні функції.

API Gateway :

Встановлення залежностей: Встановлено Flask та необхідні бібліотеки для роботи з REST API та JWT.

```
pip install Flask Flask-RESTful Flask-JWT-Extended
```

Створення основного файлу: Створено файл app.py, в якому визначено основні маршрути для API.

Налаштування JWT: Реалізовано механізм аутентифікації за допомогою JWT, що дозволяє захищати маршрути API.

Маршрутизація запитів: Налаштовано маршрути для перенаправлення запитів до відповідних мікросервісів.

Примітка: код даного рішення у ДОДАТОК А.

Опис коду:

Ініціалізація Flask: Створюється екземпляр Flask і налаштовується JWT.

Модель користувача: Простий словник для зберігання користувачів та паролів (для демонстраційних цілей).

Клас Auth: Обробляє аутентифікацію користувачів. При успішному вході генерує токен доступу.

Класи DeviceManagement та DataProcessing: Захищені маршрути, доступ до яких можливий лише за наявності дійсного токена JWT.

Маршрути: Додаються маршрути для аутентифікації та доступу до мікросервісів.

Валідація даних: Додано бібліотеку marshmallow для валідації вхідних даних. Створено клас DeviceSchema, який описує структуру даних для пристроїв.

Обробка запитів до мікросервісів: У класі DeviceManagement реалізовано метод get, який робить запит до іншого мікросервісу. Додано обробку помилок для запитів.

Обробка помилок: Додано обробку помилок для валідації даних та запитів до мікросервісів.

Для реалізації мікросервісу **авторизації через GitHub**, було зроблено наступні кроки:

- код для налаштування OAuth з GitHub
- функціонал для генерації унікальних ідентифікаторів для IoT-пристроїв.

Примітка: код у додатку А.2

Налаштування OAuth: Використовується Flask-OAuthlib для налаштування OAuth з GitHub. Необхідно вказати consumer_key та consumer_secret, які ви отримаєте після реєстрації свого додатку на GitHub.

Маршрути:

/login: Перенаправляє користувача на GitHub для авторизації.

/callback: Обробляє відповідь від GitHub, отримує токен доступу та інформацію про користувача.

/generate_id: Генерує унікальний ідентифікатор для IoT-пристроїв за допомогою uuid.

Сесії: Використовується сесія для зберігання токена доступу.

Сервіс управління пристроями. Реалізація сервісу управління пристроями з підтримкою CRUD операцій для додавання, видалення та оновлення IoT-пристроїв.

Примітка: код у додатку А.3

Опис коду:

Список пристроїв: Використовується список devices для зберігання інформації про IoT-пристрої. Кожен пристрій є словником, що містить його дані.

Маршрути:

POST /devices: Додає новий пристрій до списку. Приймає JSON-дані та генерує унікальний ідентифікатор.

DELETE /devices/<int:device_id>: Видаляє пристрій за його ідентифікатором.

PUT /devices/<int:device_id>: Оновлює дані пристрою за його ідентифікатором.

GET /devices: Повертає список усіх пристроїв.

Лічильник ідентифікаторів: Використовується змінна device_id_counter для генерації унікальних ідентифікаторів для нових пристроїв.

SQLAlchemy: Додано імпорт SQLAlchemy для роботи з базою даних.

Модель Device: Створена модель Device, яка відображає таблицю в базі даних.

Створення таблиць: Використовується db.create_all() для створення таблиць на основі моделей.

CRUD Операції: Оновлені маршрути для роботи з базою даних замість списку.

Реалізація сервісу моніторингу та управління топіками MQTT:

Примітка: код у додатку А.4

Опис коду:

Налаштування MQTT: Встановлюється з'єднання з MQTT брокером за допомогою paho-mqtt.

Обробка підключення: Функція on_connect підписується на всі топіки, які є у списку subscriptions.

Обробка повідомлень: Функція on_message обробляє вхідні повідомлення, в даному випадку просто виводячи їх у консоль.

Маршрути:

- POST /subscribe: Додає новий топік до підписок.
- POST /unsubscribe: Видаляє топік з підписок.
- GET /subscriptions: Повертає список активних підписок.

Реалізація сервісу обробки даних, який отримує дані з MQTT і передає їх у Kafka для подальшої обробки:

Примітка: код у додатку А.5

Опис коду

Налаштування Kafka: Встановлюється з'єднання з Kafka за допомогою KafkaProducer.

Налаштування MQTT: Встановлюється з'єднання з MQTT брокером.

Обробка підключення: Функція on_connect підписується на топік, з якого будуть отримуватися дані.

Обробка повідомлень: Функція on_message обробляє вхідні повідомлення, передаючи їх у функцію process_data.

Логіка обробки даних: Функція process_data обробляє отримані дані і передає їх у Kafka.

Маршрут /start: Запускає MQTT клієнт.

Реалізація сервісу обробки команд:

Примітка: код у додатку А.6

Опис коду

Налаштування MQTT: Встановлюється з'єднання з MQTT брокером.

Обробка підключення: Функція on_connect підтверджує підключення до брокера.

Маршрут /commands:

Приймає POST запити з JSON, що містить команду.

Перевіряє, чи надана команда.

Відправляє команду на вказаний топік через MQTT.

Запуск сервісу: Запускає Flask сервер.

3.3. Розробка фронтенду

У цьому розділі описується створення клієнтської частини системи, яка забезпечує інтерактивну взаємодію з користувачем. Основні завдання на цьому етапі включають:

- Реалізація основних компонентів: Розроблено структуру UI, включаючи елементи для авторизації, управління пристроями та відображення даних.
- Інтеграція з бекендом: Забезпечено інтеграцію з API Gateway для отримання даних та надсилання запитів до мікросервісів.

Аутентифікація користувачів:

Використано JWT для захисту приватних маршрутів.

Реалізовано вхід через GitHub OAuth (використовуючи клієнтську частину, що взаємодіє з бекендом).

Додано компонент **LoginPage**, який перенаправляє користувача до GitHub для авторизації.

Примітка: код у додатку Б.1

Створено компонент **DeviceManagement**, що дозволяє переглядати, додавати, оновлювати та видаляти IoT-пристрої.

Додано форми валідації введених даних із використанням бібліотеки Formik.

Налаштовано інтеграцію з бекендом через Axios.

Примітка: код у додатку Б.2

Інтерактивна візуалізація даних:

Реалізовано компонент для моніторингу MQTT-топіків у реальному часі, використовуючи WebSocket-з'єднання.

Додано графічні віджети для відображення статистики, побудовані за допомогою бібліотеки Chart.js.

Примітка: код у додатку Б.3

3.4. Налаштування додатку

У цьому розділі налаштовується мікросервісна архітектура, використовуючи Docker для контейнеризації різних компонентів, таких як API Gateway, служби аутентифікації, управління пристроями, MQTT, обробки даних, командна служба та інтерфейс користувача. Також налаштовуються Kafka та Zookeeper для обробки повідомлень.

Файл docker-compose.yml

```
version: '3'
services:
  api_gateway:
    build: ./api_gateway
    ports:
      - "5000:5000"

  auth_service:
    build: ./auth_service
    ports:
      - "5001:5000"

  device_service:
    build: ./device_service
    ports:
      - "5002:5000"

  mqtt_service:
    build: ./mqtt_service
    ports:
      - "1883:1883"

  data_processing_service:
```

```
    build: ./data_processing_service
    ports:
      - "5004:5000"

  command_service:
    build: ./command_service
    ports:
      - "5005:5000"

  ui:
    build: ./ui
    ports:
      - "3000:3000"

  kafka:
    image: wurstmeister/kafka
    ports:
      - "9092:9092"
    environment:
      KAFKA_ADVERTISED_LISTENERS: INSIDE://kafka:9092,OUTSIDE://localhost:9092
      KAFKA_LISTENERS: INSIDE://0.0.0.0:9092,OUTSIDE://0.0.0.0:9092
      KAFKA_ZOOKEEPER: zookeeper:2181

  zookeeper:
    image: wurstmeister/zookeeper
    ports:
      - "2181:2181"
```

Приклад Dockerfile для одного з сервісів (наприклад, auth_service):

```
# Dockerfile для auth_service
FROM python:3.8-slim

# Встановлення робочого каталогу
WORKDIR /app

# Копіювання файлів
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt

COPY . .

# Запуск сервісу
CMD ["python", "app.py"]
```

Приклад requirements.txt:

```
Flask==2.0.1
```

```
Flask-Cors==3.0.10
requests==2.25.1
```

Приклад `.dockerignore`

```
__pycache__
*.pyc
*.pyo
*.pyd
.env
*.db
```

Для налаштування топіків у MQTT та Kafka використовуються відповідні клієнти. Наприклад, для MQTT:

```
import paho.mqtt.client as mqtt

client = mqtt.Client()
client.connect("mqtt_service", 1883, 60)

client.loop_start()
client.subscribe("your/topic")
client.loop_stop()
```

Для Kafka:

```
from kafka import KafkaProducer

producer = KafkaProducer(bootstrap_servers='kafka:9092')
producer.send('your_topic', b'Hello, Kafka!')
```

Цей набір конфігураційних файлів дозволяє налаштувати мікросервісну архітектуру з використанням Docker. Забезпечується легкість запуску, зупинки та масштабування сервісів, а також налаштування обробки повідомлень за допомогою MQTT та Kafka. Це забезпечує надійність та ефективність веб-додатку.

Висновок

Розробка мікросервісного додатку для управління IoT-пристроями є складним, але водночас захоплюючим процесом, що дозволяє інтегрувати різноманітні технології та підходи. У даному проекті було реалізовано архітектуру, що складається з декількох мікросервісів, кожен з яких виконує

свою специфічну роль, забезпечуючи ефективність та масштабованість системи.

Основні компоненти, такі як API Gateway, сервіси авторизації, управління пристроями, моніторингу даних та обробки команд, були реалізовані з використанням Flask, що дозволило створити легкі та швидкі RESTful API. Використання Docker для контейнеризації кожного мікросервісу спростило процес налаштування та розгортання, забезпечивши легкість у масштабуванні та управлінні сервісами.

Фронтенд, розроблений на React, надає зручний інтерфейс для користувачів, дозволяючи їм легко взаємодіяти з системою, управляти IoT-пристроями та візуалізувати дані в реальному часі. Інтеграція з OAuth2 для авторизації через GitHub забезпечує безпечний доступ до системи.

Використання Kafka та MQTT для обробки повідомлень між сервісами забезпечує надійний і швидкий обмін даними, що є критично важливим для IoT-додатків. Це дозволяє системі ефективно обробляти великі обсяги даних у реальному часі, що підвищує її продуктивність та швидкість реакції.

Узагальнюючи, реалізація даного проекту демонструє можливості сучасних технологій у сфері IoT, а також підтверджує важливість мікросервісної архітектури для створення гнучких і масштабованих рішень. Наступними кроками можуть стати вдосконалення системи, впровадження нових функцій та розширення можливостей для інтеграції з іншими сервісами та платформами.

					171.6321М.КР.ПЗ.Р4		
Змн.	Арк.	№ документу	Підпис	Дата			
Розробник		Железнов В.А.			Охорона праці	Літ.	Арк. 47 / Аркушів
Консультант		Тубальцев А. М.					
Н. контр.		Добрінова Л. П.				НУК імені адмірала Макарова	
Керівник		Солобуто Л. В.					
Зав. каф.		Рябенький В. М.					

РОЗДІЛ 4. ОХОРОНА ПРАЦІ

4.1. Загальні положення

Охорона праці в Україні регулюється Законом України "Про охорону праці", а також низкою підзаконних актів, зокрема державними стандартами і нормативами, які визначають вимоги до безпечних умов праці. Метою охорони праці є запобігання травмам, професійним захворюванням і створення безпечних умов для виконання робіт.

Дотримання принципу пріоритетності охорони життя та здоров'я працюючих є основою законодавства в галузі охорони праці. Всі заходи повинні спрямовуватися на зниження ризику та усунення потенційних загроз для працівників.

Основні положення законодавства про охорону праці в Україні включають:

Гарантії прав працівників. Працівники мають право на безпечні та здорові умови праці, на отримання необхідних засобів індивідуального та колективного захисту, а також на компенсації у випадку нещасних випадків чи професійних захворювань.

Обов'язки роботодавців. Роботодавці зобов'язані забезпечити дотримання норм і правил охорони праці, проводити навчання працівників з питань безпеки, забезпечувати регулярні перевірки стану охорони праці та вживати заходів для усунення ризиків.

Державний нагляд і контроль. Органи державного нагляду, зокрема Державна служба України з питань праці, здійснюють перевірки дотримання вимог законодавства та можуть застосовувати санкції до порушників.

Підзаконні акти та нормативні документи охоплюють:

- Державні санітарні норми і правила.
- Нормативи щодо використання засобів індивідуального захисту.
- Інструкції з охорони праці для окремих галузей та видів робіт.

Сучасні виклики та тенденції:

- Впровадження систем управління безпекою праці на підприємствах (наприклад, стандарту ISO 45001).
- Цифровізація процесів моніторингу та контролю за станом охорони праці.
- Активізація навчання працівників щодо новітніх технологій безпеки та оцінки ризиків.

Дотримання принципів охорони праці не лише зберігає здоров'я і життя працівників, але й підвищує продуктивність праці, зменшує фінансові втрати від аварійних ситуацій і сприяє стійкому розвитку підприємств.

4.2. Робоче місце та його оснащення

Робоче місце повинно відповідати вимогам ГОСТів, державних стандартів та інших нормативних документів, що регулюють організацію робочого простору. Правильне облаштування робочого місця має на меті забезпечення комфортності, безпеки та збереження здоров'я працівників, а також підвищення їх продуктивності.

Основні аспекти організації робочого місця:

- Ергономічність розташування елементів:
  - Монітор повинен бути розташований на відстані 600–700 мм від очей, на рівні або трохи нижче рівня очей.
  - Клавіатура, мишка та інші пристрої введення мають бути розміщені так, щоб забезпечити комфортний кут згину зап'ястя.
  - Висота столу та стільця має регулюватися відповідно до зросту працівника, щоб спина залишалася прямою, а ноги рівно стояли на підлозі або підставці.
- Освітлення та мікроклімат:

- Робоче місце повинно бути добре освітлене, бажано природним світлом, доповненим локальними джерелами освітлення.
- Температура, вологість і вентиляція робочого приміщення мають відповідати санітарним нормам.

- Контроль стану обладнання:
  - Усі пристрої на робочому місці повинні регулярно перевірятися згідно з плановим графіком технічного обслуговування.
  - Своєчасне обслуговування дозволяє уникнути аварій, забезпечує стабільну роботу обладнання та продовжує його експлуатаційний термін.
- Профілактика перенапруження:
  - Рекомендується робити перерви кожні 1–2 години роботи за комп'ютером для виконання фізичних вправ для очей та розслаблення м'язів.
  - Використання ергономічних меблів та аксесуарів (підставки для ніг, підлокітники) сприяє зменшенню навантаження на опорно-рухову систему.

Важливість дотримання вимог: Забезпечення відповідності робочого місця встановленим стандартам позитивно впливає на здоров'я працівників, зменшує кількість випадків професійного вигорання та травматизму, а також підвищує загальну ефективність роботи. Дотримання цих вимог є ключовим фактором створення комфортного, безпечного та продуктивного робочого середовища.

4.3. Вимоги до електробезпечності

Усі електропристрої повинні підключатися до розеток, які відповідають чинним стандартам безпеки, мають надійне заземлення і сертифіковані

відповідно до вимог якості. Використання подовжувачів дозволяється лише за умови, що вони відповідають необхідним технічним характеристикам, зокрема мають достатню потужність і захист від перенапруги.

Контроль за станом електромережі.

Регулярний огляд:

- Електропроводка та обладнання повинні періодично перевірятися на предмет зношення, механічних пошкоджень або ознак перегріву.

Дії у разі виявлення пошкоджень:

- У випадку виявлення несправностей потрібно негайно припинити використання пристрою, повідомити відповідального за електробезпеку та забезпечити ремонт або заміну пошкоджених елементів.

Технічне обслуговування:

- Планове технічне обслуговування мережі й обладнання допомагає уникнути аварійних ситуацій і забезпечує їхню стабільну роботу.

Правила безпеки при роботі з електропристроями

Стан пристроїв:

- Забороняється використовувати електропристрої з пошкодженими кабелями, вилками, корпусами чи іншими елементами.
- Електроприлади, що демонструють нестабільну роботу (іскріння, шум), слід негайно відключити.

Підключення та розташування:

- Кабелі повинні бути прокладені так, щоб уникнути їх перегину, механічного напруження чи випадкового пошкодження.
- Не допускається прокладання кабелів у місцях інтенсивного руху без додаткового захисту.

Заземлення:

- Усі пристрої повинні мати надійне заземлення відповідно до норм ДСТУ та чинних стандартів.
- Розетки для пристроїв з великим енергоспоживанням повинні мати заземлювальний контакт.

Запобіжні заходи:

- Заборонено торкатися електропристроїв вологими руками.
- Перед проведенням будь-яких маніпуляцій з обладнанням (чистка, заміна деталей) його слід вимкнути з розетки.

Дотримання цих вимог гарантує:

- Зниження ризику електротравм.
- Безперебійне функціонування електромережі.
- Підвищення довговічності обладнання.

Дотримання правил безпеки є обов’язковим для всіх працівників і повинно контролюватися відповідальними особами на підприємстві.

4.4. Вимоги до умов роботи в приміщенні

Освітлення:

- Приміщення повинно бути обладнане належним освітленням, яке забезпечує рівномірне розподілення світла на робочій зоні без створення тіней чи бликів.
- Яскравість освітлення має відповідати санітарно-гігієнічним нормам, щоб запобігти перенапруженню зору.
- Для штучного освітлення слід використовувати світильники з м’яким розсіяним світлом.
- Робочі місця біля вікон потрібно захищати від прямого сонячного світла за допомогою жалюзі або штор.

Мікроклімат:

- Оптимальна температура в приміщенні для роботи з комп’ютерною технікою повинна бути в межах 18–22 °C.
- Відносна вологість повітря — 40–60%. Надто сухе або вологе повітря може негативно впливати на здоров'я працівників та роботу обладнання.
- Вентиляція повинна забезпечувати регулярний обмін повітря, а приміщення слід провітрювати кілька разів на день.

Гігієна приміщення:

- Потрібно проводити регулярне прибирання, включаючи вологе очищення поверхонь для зменшення кількості пилу.
- Забезпечити доступ до свіжого повітря, уникаючи при цьому протягів.
- Установлення системи кондиціонування повітря або зволожувачів дозволить підтримувати комфортні умови в будь-яку пору року.

Рекомендації для працівників:

- Робота за комп’ютером не повинна перевищувати 1–2 години без перерви. Після цього рекомендується зробити паузу тривалістю 10–15 хвилин.
- Під час перерв слід виконувати вправи для очей (фокусування на віддалених предметах, обертання очима) та розминку для розвантаження опорно-рухового апарату.
- При тривалій роботі сидячи варто змінювати положення тіла, щоб уникнути застою крові.

Дотримання цих вимог:

Забезпечення відповідних умов у приміщенні сприяє підвищенню продуктивності праці, збереженню здоров’я працівників та забезпеченню тривалого функціонування технічного обладнання.

Висновок

Охорона праці в Україні є важливим аспектом, що регулюється законодавством, яке визначає вимоги до безпечних умов праці та захисту прав працівників. Закон України "Про охорону праці" та супутні підзаконні акти встановлюють чіткі гарантії прав працівників на безпечні умови праці та обов’язки роботодавців щодо їх забезпечення. Дотримання принципів охорони праці не лише захищає життя та здоров'я працівників, але й підвищує загальну продуктивність роботи, зменшує ризики та фінансові втрати, пов’язані з нещасними випадками.

Сучасні виклики, такі як цифровізація процесів моніторингу та впровадження систем управління безпекою праці, вимагають адаптації до нових умов і технологій. Це підкреслює необхідність активного навчання працівників та впровадження новітніх технологій безпеки.

Таким чином, забезпечення безпеки праці є не лише юридичним обов'язком, але й важливим елементом розвитку підприємств, що сприяє створенню комфортного та продуктивного робочого середовища. Дотримання вимог охорони праці є ключовим для збереження здоров'я працівників та забезпечення стабільної роботи підприємств в умовах сучасних викликів.

ВИСНОВКИ

Висновок до роботи

У даній роботі було проведено комплексний аналіз і розробку системи управління IoT-пристроями, яка базується на сучасних технологіях та підходах. Основна мета проекту полягала у створенні гнучкої, масштабованої та надійної архітектури, здатної ефективно інтегрувати різноманітні IoT-пристрої та забезпечити їхнє управління в реальному часі.

У першому розділі було здійснено огляд існуючих рішень для управління IoT-пристроями, таких як OpenHAB, Home Assistant, AWS IoT Core, Mosquitto та Apache Kafka. Аналіз показав, що, незважаючи на різноманітність підходів, сучасні системи стикаються з проблемами інтеграції, масштабування, безпеки та гнучкості. У відповідь на ці виклики було розроблено рішення, яке поєднує мікросервісну архітектуру з інтеграцією протоколів MQTT і Kafka, а також інтерфейс на основі React для взаємодії з користувачами. Це забезпечує відмовостійкість, швидкість обробки даних та можливість інтеграції з іншими платформами.

Другий розділ розглянув основи мікросервісної архітектури та основні технології для її реалізації. Було підкреслено важливість вибору відповідних інструментів, таких як Flask для розробки мікросервісів, React для фронтенду, а також Docker для контейнеризації. Окрему увагу було приділено забезпеченню взаємодії між мікросервісами за допомогою Apache Kafka та MQTT, що є критично важливим для побудови ефективних систем.

У третьому розділі було детально описано розробку додатку, включаючи структуру, реалізацію мікросервісів, інтеграцію з API Gateway, а також створення фронтенду. Зокрема, було реалізовано функціонал для управління IoT-пристроями, аутентифікації користувачів через GitHub, моніторингу даних у реальному часі та обробки команд.

В результаті виконаної роботи була створена система, що забезпечує ефективне управління IoT-пристроями, інтеграцію з різними сервісами та можливість масштабування. Використання сучасних технологій та підходів дозволило досягти високого рівня продуктивності, надійності та безпеки.

Подальші напрямки розвитку включають вдосконалення системи, впровадження нових функцій, таких як підтримка нових протоколів, покращення механізмів безпеки та інтеграція з додатковими платформами. Інноваційні рішення, розроблені в рамках цього проекту, мають значний потенціал для розширення можливостей IoT-систем і їхнього використання в різних галузях.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

Літературні джерела:

1. Моргунов, І. В., Єфімов, В. П., Григор'єва, Л. І. Охорона праці: навчальний посібник. Харків: Харківський національний університет радіоелектроніки, 2018. - 256 ст.
2. Коваленко, О. В., Левченко, А. В. Інтернет речей: технології, архітектури, застосування. Київ: Наукова думка, 2020. - 312 ст.
3. Шевченко, В. І., Петров, С. О. Основи IoT: теорія та практика. Львів: Львівський національний університет імені Івана Франка, 2019. - 280 ст.
4. Morabito, R., & Karp, B. (2018). Internet of Things: Technologies and Applications. Amsterdam: Elsevier. - 350 p.
5. Evans, D. (2011). The Internet of Things: How Smart TVs, Smart Cars, Smart Homes, and Smart Cities Are Changing the World. New York: Cisco Internet Business Solutions Group. - 40 p.

Інтернет джерела:

1. Docker Documentation. (n.d.). Retrieved from https://docs.docker.com
2. Flask Documentation. (n.d.). Retrieved from https://flask.palletsprojects.com
3. Python Documentation. (n.d.). Retrieved from https://docs.python.org
4. IoT Documentation. (n.d.). Retrieved from https://www.iot.gov
5. React Documentation. (n.d.). Retrieved from https://reactjs.org
6. PostgreSQL Documentation. (n.d.). Retrieved from https://www.postgresql.org/docs
7. Kafka Documentation. (n.d.). Retrieved from https://kafka.apache.org/documentation/
8. MQTT Documentation. (n.d.). Retrieved from https://mqtt.org/documentation

ДОДАТОК А

Microservices

1.Gateway:

```
from flask import Flask, request, jsonify
from flask_restful import Api, Resource
from flask_jwt_extended import JWTManager, create_access_token, jwt_required
from marshmallow import Schema, fields, ValidationError
import requests

app = Flask(__name__)
app.config['JWT_SECRET_KEY'] = 'your_secret_key'  # Задайте свій секретний ключ
jwt = JWTManager(app)
api = Api(app)

# Модель користувача для демонстрації
users = {
    "user1": "password1"
}

# Схема валідації для даних
class DeviceSchema(Schema):
    device_id = fields.Int(required=True)
    device_name = fields.Str(required=True)

class Auth(Resource):
    def post(self):
        username = request.json.get('username')
        password = request.json.get('password')

        if username in users and users[username] == password:
            access_token = create_access_token(identity=username)
            return {"access_token": access_token}, 200
        return {"message": "Invalid credentials"}, 401

class DeviceManagement(Resource):
    @jwt_required()
    def get(self):
        # Приклад запиту до іншого мікросервісу
        try:
            response = requests.get('http://example-microservice/devices')
            response.raise_for_status()  # Перевірка на помилки
```

```
            return response.json(), 200
        except requests.exceptions.RequestException as e:
            return {"message": str(e)}, 500

    @jwt_required()
    def post(self):
        # Валідація даних
        schema = DeviceSchema()
        try:
            data = schema.load(request.json)
            # Логіка для обробки даних
            return {"message": "Device added", "data": data}, 201
        except ValidationError as err:
            return {"message": err.messages}, 400

class DataProcessing(Resource):
    @jwt_required()
    def get(self):
        return {"message": "Data processing information"}, 200

# Додати маршрути
api.add_resource(Auth, '/auth')
api.add_resource(DeviceManagement, '/devices')
api.add_resource(DataProcessing, '/data')

if __name__ == '__main__':
    app.run(debug=True)
```

2. git_auth

```
from flask import Flask, redirect, url_for, session, request, jsonify
from flask_oauthlib.client import OAuth
import uuid

app = Flask(__name__)
app.secret_key = 'your_secret_key'  # Задайте свій секретний ключ
oauth = OAuth(app)

github = oauth.remote_app('github',
    consumer_key='YOUR_CONSUMER_KEY',
    consumer_secret='YOUR_CONSUMER_SECRET',
    request_token_params={'scope': 'user:email'},
    base_url='https://api.github.com/',
    request_token_url=None,
    access_token_method='POST',
```

```
        access_token_url='https://github.com/login/oauth/access_token',
        authorize_url='https://github.com/login/oauth/authorize',
    )

    @app.route('/')
    def index():
        return 'Welcome to the Auth Service!'

    @app.route('/login')
    def login():
        return github.authorize(callback=url_for('authorized', _external=True))

    @app.route('/callback')
    def authorized():
        response = github.authorized_response()
        if response is None or 'access_token' not in response:
            return 'Access denied: reason={} error={}'.format(
                request.args['error_reason'],
                request.args['error_description']
            )

        session['github_token'] = (response['access_token'], '')
        user_info = github.get('user')
        return jsonify(user_info.data)

    @app.route('/generate_id')
    def generate_id():
        unique_id = str(uuid.uuid4())  # Генерація унікального ідентифікатора
        return jsonify({"unique_id": unique_id})

    @github.tokengetter
    def get_github_oauth_token():
        return session.get('github_token')

    if __name__ == '__main__':
        app.run(debug=True)
```

3 . device_control

```
    from flask import Flask, request, jsonify
    from flask_sqlalchemy import SQLAlchemy

    app = Flask(__name__)
    app.config['SQLALCHEMY_DATABASE_URI']                                    =
'postgresql://username:password@localhost/iot_devices'
    app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
```

```
db = SQLAlchemy(app)

# Модель для пристроїв
class Device(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), nullable=False)
    type = db.Column(db.String(50), nullable=False)

    def to_dict(self):
        return {'id': self.id, 'name': self.name, 'type': self.type}

# Створення таблиць
with app.app_context():
    db.create_all()

@app.route('/devices', methods=['POST'])
def add_device():
    device_data = request.json
    device = Device(name=device_data['name'], type=device_data['type'])
    db.session.add(device)
    db.session.commit()
    return jsonify(device.to_dict()), 201

@app.route('/devices/<int:device_id>', methods=['DELETE'])
def delete_device(device_id):
    device = Device.query.get(device_id)
    if device:
        db.session.delete(device)
        db.session.commit()
        return jsonify({'message': 'Device deleted successfully'}), 204
    return jsonify({'message': 'Device not found'}), 404

@app.route('/devices/<int:device_id>', methods=['PUT'])
def update_device(device_id):
    device_data = request.json
    device = Device.query.get(device_id)
    if device:
        device.name = device_data.get('name', device.name)
        device.type = device_data.get('type', device.type)
        db.session.commit()
        return jsonify(device.to_dict()), 200
    return jsonify({'message': 'Device not found'}), 404

@app.route('/devices', methods=['GET'])
def get_devices():
```

```
    devices = Device.query.all()
    return jsonify([device.to_dict() for device in devices]), 200

if __name__ == '__main__':
    app.run(debug=True)
```

4. MQTT

```
from flask import Flask, jsonify, request
import paho.mqtt.client as mqtt

app = Flask(__name__)

# Налаштування MQTT
mqtt_broker_address = "mqtt_broker_address"  # Заміна на адресу вашого MQTT брокера
mqtt_client = mqtt.Client()

# Список підписок
subscriptions = []

def on_connect(client, userdata, flags, rc):
    for topic in subscriptions:
        client.subscribe(topic)

def on_message(client, userdata, msg):
    # Логіка обробки повідомлень
    print(f"Received message '{msg.payload.decode()}' on topic '{msg.topic}'")

mqtt_client.on_connect = on_connect
mqtt_client.on_message = on_message

@app.route('/subscribe', methods=['POST'])
def subscribe():
    topic = request.json.get('topic')
    if topic and topic not in subscriptions:
        subscriptions.append(topic)
        mqtt_client.subscribe(topic)
        return jsonify({'message': f'Subscribed to {topic}'}), 201
    return jsonify({'message': 'Invalid topic or already subscribed'}), 400

@app.route('/unsubscribe', methods=['POST'])
def unsubscribe():
    topic = request.json.get('topic')
    if topic in subscriptions:
```

```
            subscriptions.remove(topic)
            mqtt_client.unsubscribe(topic)
            return jsonify({'message': f'Unsubscribed from {topic}'}), 200
        return jsonify({'message': 'Topic not found in subscriptions'}), 404

@app.route('/subscriptions', methods=['GET'])
def get_subscriptions():
    return jsonify(subscriptions), 200

if __name__ == '__main__':
    mqtt_client.connect(mqtt_broker_address)
    mqtt_client.loop_start()  # Запуск циклу MQTT
    app.run(debug=True)
```

5. MQTT to Kafka bridge

```
from flask import Flask, jsonify
import paho.mqtt.client as mqtt
from kafka import KafkaProducer

app = Flask(__name__)

# Налаштування Kafka
producer = KafkaProducer(bootstrap_servers='localhost:9092')

# Налаштування MQTT
mqtt_broker_address = "mqtt_broker_address"  # Заміна на адресу вашого MQTT брокера

def on_connect(client, userdata, flags, rc):
    client.subscribe("data_topic")  # Заміна на ваш топік

def on_message(client, userdata, msg):
    data = msg.payload.decode()
    process_data(data)

def process_data(data):
    # Логіка обробки даних
    print(f"Processing data: {data}")
    producer.send('processed_data_topic', value=data.encode('utf-8'))

mqtt_client = mqtt.Client()
mqtt_client.on_connect = on_connect
mqtt_client.on_message = on_message

@app.route('/start', methods=['GET'])
```

```
def start_mqtt():
    mqtt_client.connect(mqtt_broker_address)
    mqtt_client.loop_start()  # Запуск циклу MQTT
    return jsonify({'message': 'MQTT client started'}), 200

if __name__ == '__main__':
    app.run(debug=True)
```

6. command_processed

```
from flask import Flask, request, jsonify
import paho.mqtt.client as mqtt

app = Flask(__name__)

# Налаштування MQTT
mqtt_broker_address = "mqtt_broker_address"  # Заміна на адресу вашого MQTT брокера
client = mqtt.Client()

def on_connect(client, userdata, flags, rc):
    print("Connected to MQTT Broker")

client.on_connect = on_connect
client.connect(mqtt_broker_address)
client.loop_start()  # Запуск циклу MQTT

@app.route('/commands', methods=['POST'])
def send_command():
    command = request.json
    if 'command' not in command:
        return jsonify({"error": "Command not provided"}), 400

    client.publish("device/topic", command['command'])  # Заміна на ваш топік
    return jsonify({"status": "command sent"}), 200

if __name__ == '__main__':
    app.run(debug=True)
```

ДОДАТОК Б

UI

1. LoginPage:

```
import React from 'react';
import { useHistory } from 'react-router-dom';

const LoginPage = () => {
    const history = useHistory();

    const handleLogin = () => {
        const clientId = 'YOUR_CLIENT_ID';
        const redirectUri = 'http://localhost:3000/auth/github/callback';
        window.location.href                                                    =
`https://github.com/login/oauth/authorize?client_id=${clientId}&redirect_uri=${redirectUri}`;
    };

    return (
        <div className="container mt-5">
            <h2 className="text-center mb-4">Welcome Back!</h2>
            <div className="text-center">
                <button className="btn btn-primary btn-lg" onClick={handleLogin}>
                    Login with GitHub
                </button>
            </div>
        </div>
    );
};

export default LoginPage;
```

2. DeviceManagement:

```
// DeviceManagement.js
import React, { useEffect, useState } from 'react';
import axios from 'axios';

const DeviceManagement = () => {
    const [devices, setDevices] = useState([]);
    const token = localStorage.getItem('token');

    useEffect(() => {
        const fetchDevices = async () => {
```

```
                const response = await axios.get('/api/devices', { headers: { Authorization:
`Bearer ${token}` } });
                setDevices(response.data);
            };
            fetchDevices();
        }, [token]);

        return (
            <div className="container mt-5">
                <h2 className="text-center mb-4">Device Management</h2>
                <table className="table table-striped table-bordered">
                    <thead className="thead-dark">
                        <tr>
                            <th>ID</th>
                            <th>Name</th>
                            <th>Actions</th>
                        </tr>
                    </thead>
                    <tbody>
                        {devices.map(device => (
                            <tr key={device.id}>
                                <td>{device.id}</td>
                                <td>{device.name}</td>
                                <td>
                                    <button className="btn btn-warning btn-sm">Edit</button>
                                    <button       className="btn       btn-danger       btn-
sm">Delete</button>
                                </td>
                            </tr>
                        ))}
                    </tbody>
                </table>
            </div>
        );
    };

    export default DeviceManagement;
```

3. DataVisualization:

```
// DataVisualization.js
import React, { useEffect } from 'react';
import { Chart } from 'react-chartjs-2';
```

```
const DataVisualization = () => {
    useEffect(() => {
        const socket = new WebSocket('ws://YOUR_MQTT_SERVER');

        socket.onmessage = (event) => {
            const data = JSON.parse(event.data);
            // Обробка даних для графіків
        };

        return () => {
            socket.close();
        };
    }, []);

    return (
        <div className="container mt-5">
            <h2 className="text-center mb-4">Data Visualization</h2>
            <div className="chart-container">
                <Chart data={/* ваші дані */} />
            </div>
        </div>
    );
};

export default DataVisualization;
```