31.05.2024

Göktuğ Özdoğan

Plan S Case Study Report

Project Links

GitHub: https://github.com/goktugozdogan/PlanS-CaseStudy

Git HTTPS: https://github.com/goktugozdogan/PlanS-CaseStudy.git

Mongo Express: http://localhost:8081/

Swagger API: http://localhost:8086/swagger-ui/index.html (ports: 8086, 8087, 8088)

Frontend: http://localhost:3000/ (does not work)

Technology Stack

Java Main Stack: Java 17, Spring Boot 3.3.0, Maven

Database: MongoDB

Synchronous: RESTful

Asynchronous: Kafka

Containerization: Docker, Docker compose

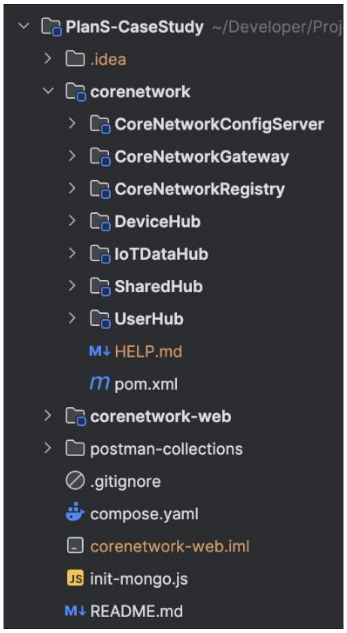
Frontend: React

Introduction

This report outlines project structure and implementation details of a microservices-based project developed using Spring Boot. Frontend is not finished in time, hence I wont be talking about frontend in this report. What is in the case study project as follows, more details are in the report.

- There are 3 core microservice and 3 infrastructure services. Core services can do CRUD operations. All of them can create, read, update or delete their related documents in MongoDB.
- Each core microservice exposes its related endpoints. Services can send RESTful requests to each other.
- Each core microservice can send and listen Kafka messages from specified topics.
- Project run with ‘docker-compose up -d’

Project Structure



```
PlanS-CaseStudy ~/Developer/Proj
  .idea
  corenetwork
    CoreNetworkConfigServer
    CoreNetworkGateway
    CoreNetworkRegistry
    DeviceHub
    IoTDataHub
    SharedHub
    UserHub
    HELP.md
    pom.xml
  corenetwork-web
  postman-collections
  .gitignore
  compose.yaml
  corenetwork-web.iml
  init-mongo.js
  README.md
```

I used single-repo-project structure for *Core Network* case study project. **corenetwork** module contains backend related services and **corenetwork-web** contains frontend related services. In this part, I will be talking about backend module.

When I read the requirements for *Core Network* case study project, the design I thought would work for the case study is, develop 3 core business services which will handle Device related operations, IoT Data related operations and User related operations. I call these services **DeviceHub**, **IoTDataHub** and **UserHub**. These 3 microservice should be able to handle all the requirements given in the case study and possible future feature implementations.

Services communicate synchronously over REST calls and communicate asynchronously with Kafka. Currently, **DeviceHub** and **IoTDataHub** only have CRUD operations, however **UserHub** has both CRUD operations and endpoints for generic device related operations that a user may want to perform on the device. These generic device operations are */link_a_device* (to User) and */toggle_sensor* (on/off temperature sensor on device).

I also implement a shared library, **SharedHub**, which contains utilities, constants, object models, and other common classes used by core 3 microservices. It promotes code reusability and reduces redundancy.

When 3 core microservices want to send requests to each other, they need to know where others' IP addresses are. In order to discover where other services are, I implement the **CoreNetworkRegistry** using Eureka which acts as a service registry where all services register themselves upon startup. This enables dynamic discovery of services and facilitates load balancing. Since, in production environment, core business services will be deployed as needed, this registry service plays an important role.

Also, this project requires a gateway, which is a bridge between other clients (other systems) and *Core Network*. I called it **CoreNetworkGateway**, and it represents the single entry point for all requests. It routes client requests to the appropriate microservices.

Since there are a couple microservices and they might have different kind of configurations, a config server is also needed. I call this service **CoreNetworkConfigServer**. The config server manages configurations of all microservices in a single place.

With the help of these 5 services, all of them registered in eureka, Core Network backend should be able to work properly.

Disclaimer: In midway of the development, I start to get issues with registry and config server. It was taking too much of my time to fix it, thus I disabled it before submitting.

Implementation Details

What features have this project are as follows;

- 3 core microservice, **DeviceHub**, **IoTDataHub** and **UserHub,** can do CRUD operations. All of them can create, read, update or delete their related documents in MongoDB.
- Each core microservice exposes its related endpoints via Controller class. Services can send RESTful requests to each other.
- Each core microservice can send and listen Kafka messages from specified topics given in application.yml file.
- **UserHub** has **/signup** and **/signin** endpoints. In order to **/signin** given password must be matched with pass in database. Encryption and decryption are handled by Spring Security. Created User's password encrypted and stored in database.
- Other spring related things;
    - There is an aspect called TimingAspect in **SharedHub.** It wraps responses from Controller classes with **"timestamp"** and **"cost".**

Database

I used MongoDB as a database. Each core microservice has its own database. In each database there might be multiple collections. For example, for the **IoTDataHub** there are 2 collections. ***iot_data*** where stores IoT device data packages coming from Ground Station and ***device_operations*** where stores IoT device operations data, such as result of *toggle_sensor* operation. MongoDB makes sense to me for **IoTDataHub** related data, because there might be a lot of data packages coming from Ground Station and if database need to expend due to availability, NoSQL base databases can be expanded horizontally with ease due to sharding and no need for complex join operations. Another reason why I choose MongoDB is that data packages might be too complex and vary in detail a lot. I think a document based data storage suits for IoT data packages.

Although I used MongoDB to store all data, I kind a regret not to use JDPA for **Device**, **User** and **Role** purpose data. Since **Device** (device info devEUI, location, sensors etc.), **User** (email, pass etc.) and **Roles** (admin, operator, client etc.) are relational data, they suit more for SQL based databases. PostgreSQL or MySQL would be my choice. According to the relation between **User** and **Device** *ManyToMany* or *OneToMany* can be used. In the current version with MongoDB, I choose *ManyToMany* where a User can have multiple devices and a device can be owned by multiple users. Each **User** document has an array field of **“devices”: []** and each **Device** document has **“owners”: [].**

Fail-Safe Operations

- **Backup Strategies**:
  To ensure data integrity and availability, backup strategies can be implemented. This should include regular backups of MongoDB databases (or possibly PostgreSQL database), Kafka topics, and configuration files.

  MongoDB has a built-in backup utility, however, there are powerful technics for backup data, **Replication** and **cloud-based *Automation, Retention Point-in-Time Recovery***. With AWS, or any cloud, a schedule can be set up to backup data **automatically** without human intervention. It can also be specified how long backup data should be **retained**, after a while they can be deleted. If anything goes wrong, **point-int-time recovery** allows MongoDB data to be recovered in the given time. This is particularly useful for recovering from data corruption or accidental deletion.

  Kafka's mirroring features (using **MirrorMaker**) can be employed. Additionally, periodic snapshots and off-site storage should be considered for disaster recovery.

- **Validation Techniques**:
  Input validation should be enforced for all microservices to ensure data integrity and prevent malicious data entry. This includes validating request parameters, payloads (request body), and inter-service communication. Techniques such as using Java Bean Validation with annotations like **@NotNull, @NotEmpty, @Email, @Size**. It is also possible to create custom validators according to needs.

- **Service Failure**: I will discuss this in the Maintenance and Reliability part.

Application Security and Authentication

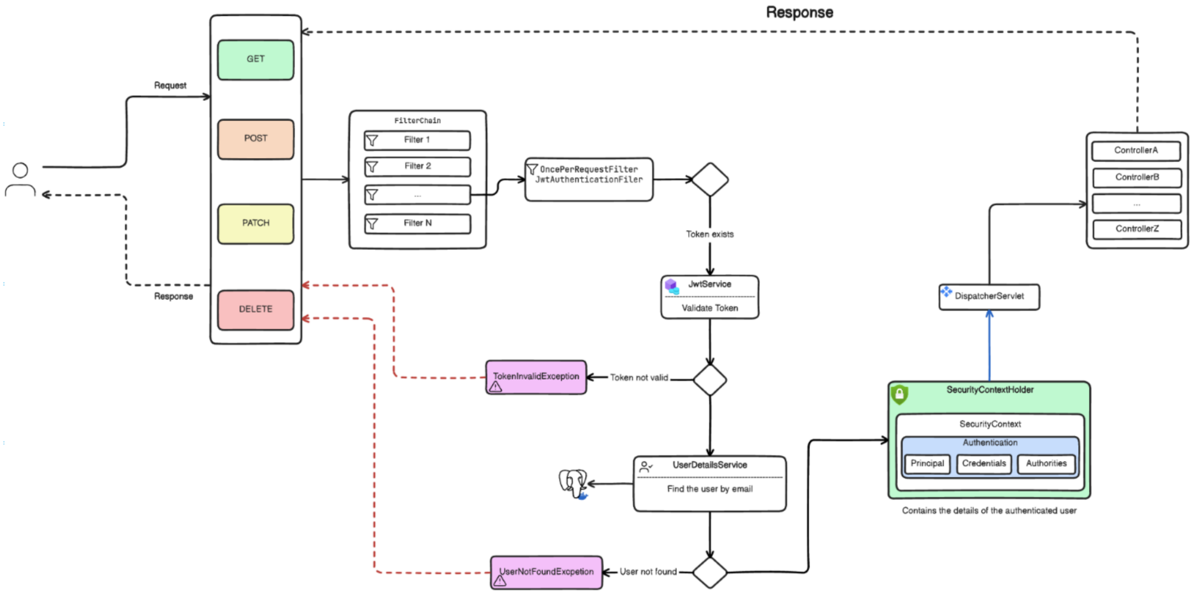
To secure the microservices, **Spring Security** and **JWT tokens** can be integrated to handle authentication and authorization.

- **Service-to-service Authentication:** When one microservice sends a request to another, it includes authentication credentials, such as a **JWT token**, in the request ***headers*** **(e.g., Authorization: Bearer <token>)**. The receiving microservice verifies these credentials to ensure that the request is coming from an authenticated and authorized microservice. In order to microservice2 to verify token, microservice1 and microservice2 need to know a **secret key,** which is generally stored in config server, such as **CoreNetworkConfigServer.**

  CoreNetworkGateway is a gateway to core network single entry point for other clients (other systems). It can be extended to enforce security measures such as **rate limiting, request validation**. It also should authenticate and authorize requests before forwarding them to the appropriate microservice. This adds an extra layer of security to the communication between microservices.

- **User Authentication & Roles:** JWT can be used to provide stateless authentication, the server only needs to verify the token's integrity and validity. Tokens will be created upon successful login and verified with each request. This setup will protect the services from unauthorized access and ensure secure communication between services.

  With the help of Spring Security can handle this, diagram below shows how. When a request is received, first interception will be in FilterChain. If the request is related to Authentication, a JWTService will validate token and given credentials (email, password). User details, credentials and authorities will be stored in SecurityContext and request directed to related endpoint in Controller. If service will have role based actions, authorities will be used from context.



Scalability Maintenance Reliability

Scalability:

- Since docker is used, containerized services can be deployed with ease using Kubernetes. Using a cloud system like AWS, auto-scaling can be achieved as well. If the server is in heavy load, new containers can be deployed. With the help of Eureka, newly deployed containers will be registered automatically as well.
- Same principals can be applied to MongoDB or other databases and Kafka brokers.
- Another improvement can be adding a caching system, like **Redis.** Frequently accessed queries will be stored in **Redis** and decrease system load.

Maintenance:

- Every service should be monitored. System health and availability can be checked by Spring Actuator or cloud systems.
- System restarts should be added. In docker-compose file **restart: unless-stopped** is added for all containers.
- Code related:
    - Unit test should be written.
    - In the CI/CD pipelines, every build should be automatically tested, scanned for vulnerabilities and verified.
    - API documentations

Reliability:

- Multi-zone deployments in cloud environments to protect against zone failures.
- If watched metrics (request failure limit, timeouts etc.) are above threshold, necessary notifications should be sent to operators.
- To ensure data integrity and availability, backup strategies can be implemented. This should include regular backups of MongoDB databases (or possibly PostgreSQL database), Kafka topics, and configuration files.