INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Project report
Service architecture

MAGANA Amalia, 5ISS

DA COSTA BENTO Marie-Line, 5ISS

INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Project report

Service architecture

MAGANA Amalia, 5ISS

DA COSTA BENTO Marie-Line, 5ISS

Summary

Introduction

This report presents the Spring Boot project developed as part of the *Service Architecture* course, which is a component of the *Middleware and Services* module. The main objective of this project was to develop a web application (Proof-of-Concept) for managing INSA's coffee machines. Through software services (microservices), the application retrieves data from sensors (such as temperature and presence), and based on the values of the retrieved data, actions on actuators can be triggered.

The organisation of the code was facilitated by the use of Git, while continuous integration (DevOps) was implemented to automate the build and validate the functionalities throughout the development process. This report will detail these different aspects of the work carried out as well as the tools and methods used throughout the project.

The project's source code is available on the following GitHub repository: https://github.com/millillitre/coffee_management/tree/main.

I. Project Requirements

The project required the following:

- Design an application based on microservices.
- Implement the various services and service calls.
- (Optional) Implement a web interface for viewing action history.
- Technologies Used:
    - Language: Java
    - Framework: Spring Boot

II. Architecture

a) Microservices overview

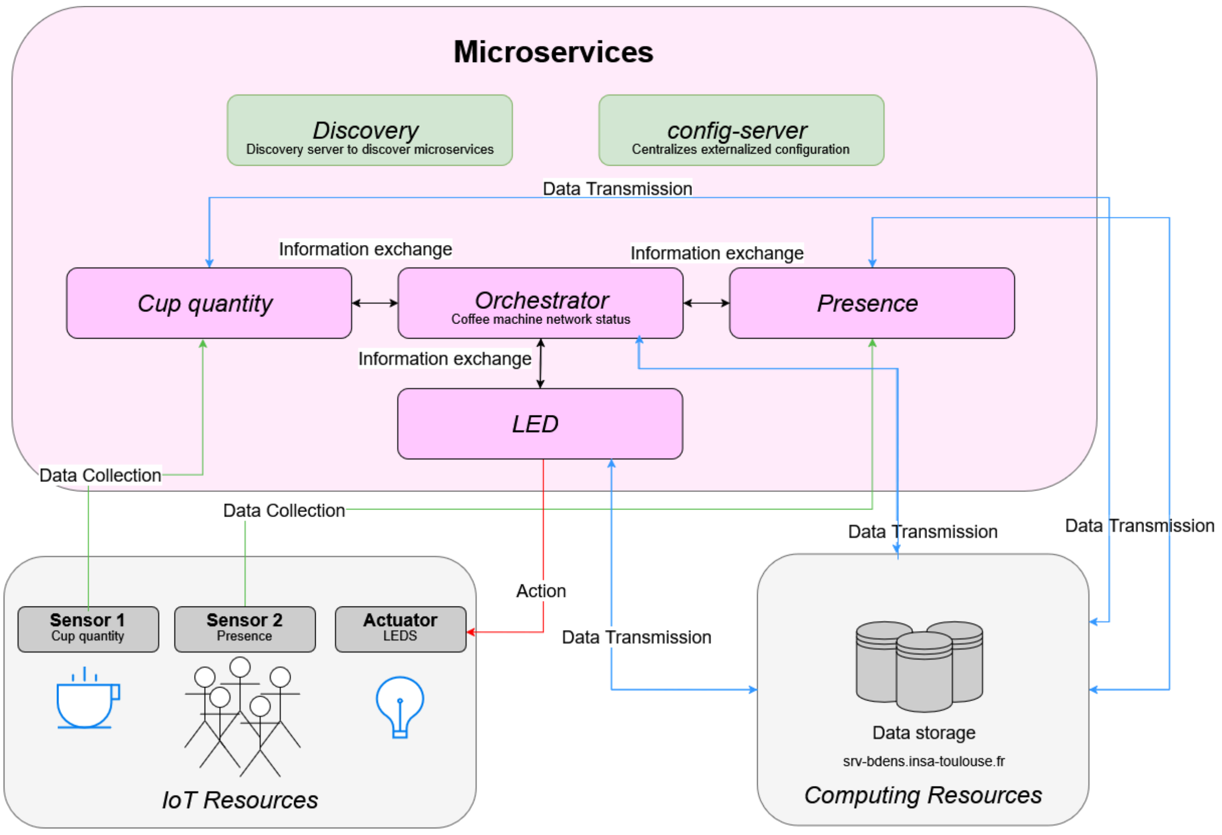

Figure 1: Microservices architecture overview

The system does not manage the coffee machines directly; instead, it focuses on data collection and action triggering. A dedicated microservice for machine management could be added in a more comprehensive implementation, rather than relying on the orchestrator to fetch machine data from the database. The system is designed as a collection of microservices, each responsible for a specific function within the coffee machine management application. Below is a detailed explanation of each microservice:

1. *Cup Quantity Recording*

It tracks the number of cups dispensed by each coffee machine. The microservice receives and stores cup quantity data from sensors. It also provides endpoints to add new cup data and retrieve historical records:

```
$ curl -X POST "http://localhost:8081/api/cup-Ms?machineId=1&value=10"

$ curl http://localhost:8081/api/cup-Ms/1
```

2. *Presence Analysis*

The presence microservice records the presence of people near the coffee machines. It captures and stores presence data (number of people detected). It provides endpoints to log presence data and fetch historical records:

```
$ curl -X POST "http://localhost:8082/api/presence-Ms?machineId=1&value=3"

$ curl http://localhost:8082/api/presence-Ms/history/1
```

3. *LED Management*

The LED management microservice controls the LED indicators on coffee machines based on sensor data. It receives input from the orchestrator (cup quantity and presence data). and determines the LED status (green, red, blue) and updates the machine's LED. LED status green is equivalent to everything is fine ok, blue to degraded service (no more cup) and orange to a crowded machine area (over 15 people). Here are some endpoints:

```
$ curl -X POST "http://localhost:8083/api/led-Ms/update?machineId=1&cup

Quantity=5&presenceValue=15"

$ curl http://localhost:8083/api/led-Ms/1
```

4. *Machine Network Status (Orchestrator)*

It acts as the central coordinator for the system. The orchestrator aggregates data from the Cup Quantity and Presence Analysis microservices. It analyses the combined data to determine the appropriate LED status and then sends commands to the LED Management microservice to update the machine's LED. Additionally, this microservice manages static machine information (building location, condition, and last maintenance visit).

5. *Improvements*

Our application is designed to manage multiple coffee machines, such as those found on a university campus. To fully embrace a microservices architecture, several improvements can be made to the current system.

Currently, the Orchestrator directly queries the database to fetch the list of machine IDs. This approach is not ideal for a microservices setup, as it creates a direct dependency on the database. A better solution would be to introduce a dedicated Machine Microservice. This service would handle all machine-related operations, including retrieving machine IDs, adding new machines, and removing existing ones. By doing so, we achieve better separation of concerns, reduce coupling, and make it easier to extend functionality in the future.

The LED management system also needs refinement. In the current version, a single LED is used to display all status information, which can be confusing. For example, if there is a low cup quantity and a crowded area, the LED will only indicate the low cup quantity by turning blue. To improve clarity, we could add a second LED or introduce more colour-coded cases to provide more precise feedback to users.

Finally, to gain a comprehensive, real-time view of each machine's status, we could integrate additional sensors. These might include sensors for coffee, sugar, and water levels, as well as status sensors to detect malfunctions or maintenance needs. Presence sensors could also be added to analyse foot traffic and optimise maintenance schedules. These enhancements would improve responsiveness, enable predictive maintenance, and provide valuable data for better decision-making.

b) Database

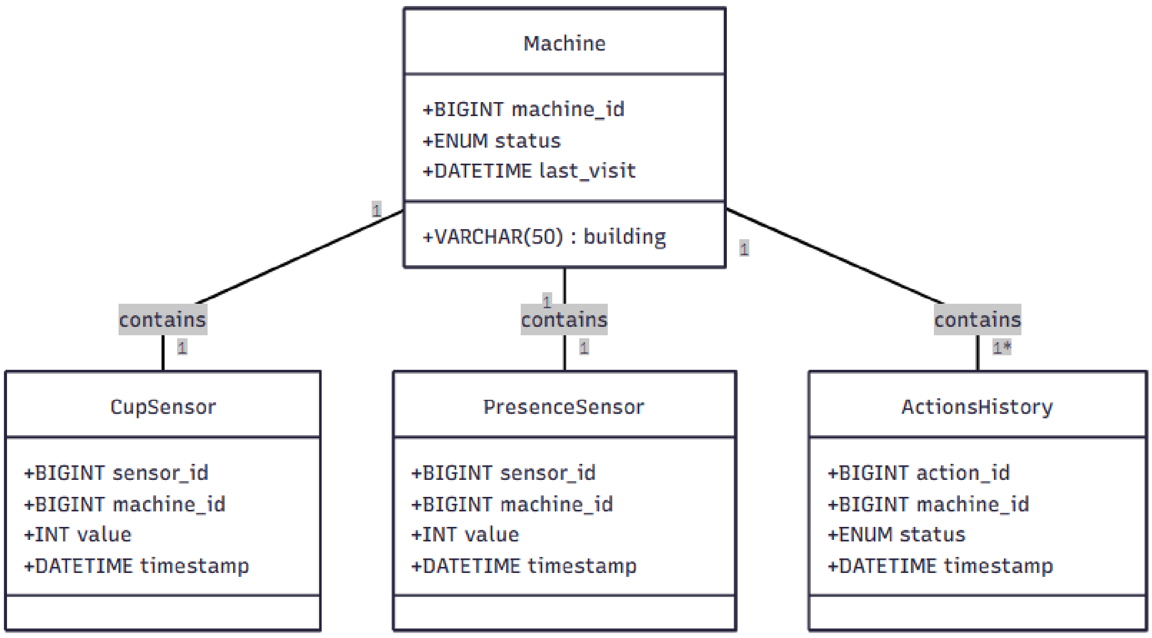

Figure 2: Database Diagram

The database tables are the following:

- machine: Stores static information about coffee machines (ID, building, condition, last visit date).
- cup_sensor: Stores cup sensor data (sensor ID, machine ID, value, timestamp).
- presence_sensor: Stores presence sensor data (sensor ID, machine ID, value, timestamp).
- actions_history: Logs actions taken (action ID, machine ID, status, timestamp).

c) Discovery and configuration

1. ***Discovery Service (Eureka)***

It acts as a service registry where all microservices register themselves upon startup. Microservices register their network locations (IP, port) with Eureka, and other services query Eureka to discover and communicate with each other dynamically. It eliminates hardcoded URLs, enabling scalability and resilience. Without Eureka, microservices would need to manually track each other's locations, which is impractical in dynamic environments. It must start first because other services depend on it to register and discover each other.

2. ***Configuration Service***

It centralizes externalized configuration (application.properties) for all microservices. It stores configuration files in a Git repository (https://github.com/millillitre/coffee_management/tree/main/config). Microservices fetch their configurations from this server at startup and it supports environment-specific configurations. It avoids duplicating configuration files across services and enables dynamic updates without redeploying services. It starts after the Discovery Service (Eureka) because it may need to register itself with Eureka for high availability.

III. Organisation

a) Directory Structure

- **.github/workflows/**: Scripts for continuous integration.
- **CupMS/**: Source code for the cup management microservice.
- **DiscoveryMS/:** Eureka discovery microservice.
- **LEDMS/**: Source code for the LED management microservice.
- **OrchestratorMS/**: Source code for the supervision microservice.
- **PresenceMS/**: Source code for the presence microservice.
- **config-server/**: Configuration microservice.
- **config/**: Configuration files.
- **docs/**: Images for the README.
- **.gitignore**: Files to ignore in Git.
- **README.md**: Project documentation.

b) Source code management with git

For source code management, we used Git with a strategy adapted to working in pairs:

- **Branching strategy:** Used branches for main, config, led, orchestrator, cup, and presence.
- **Commit:** Each commit was documented to facilitate tracking and review.
- **Pull Requests (P-R):** All feature additions, bug fixes, and significant changes were submitted via pull requests. This ensured peer review, discussion, and validation before merging into the main branch, improving code quality and collaboration.

- **Development approach:** Focused on clear documentation and structured commits for maintainability.

IV. Implementation of Spring Boot

a) Managing dependencies with Maven

We used **Maven** to manage dependencies, compile the project. Here are the main dependencies defined in each pom.xml:

- **Spring Boot Starter Web**: provides everything needed to build web applications, including embedded Tomcat, Spring MVC, and RESTful support. It enables the development of RESTful APIs and web services for our microservices.
- **Spring Cloud Config Client:** it allows microservices to fetch configuration from a centralized Spring Cloud Config Server. It centralizes configuration management, making it easier to update settings across all services without redeploying.
- **MySQL Connector:** JDBC driver for connecting to MySQL databases. It enables our microservices to interact with the MySQL database for data storage and retrieval.
- **Spring Boot Starter JDBC**: simplifies JDBC-based database access with Spring Boot. It provides a lightweight way to interact with SQL databases without using ORMs like JPA/Hibernate (which we preferred to avoid because of its complexity).
- **Eureka Client**: it enables service discovery and registration with Netflix Eureka. It allows microservices to dynamically discover and communicate with each other.
- **Spring Cloud**: it provides tools for building cloud-native applications, including service discovery, configuration management, and circuit breakers.

We have also added the following plugins to automate certain tasks:

- **Spring Boot Maven Plugin:** it automates the build process and ensures the application can be run as a standalone JAR.

b) Automation with GitHub Actions

We have configured GitHub Actions workflows to automate critical steps:

1. **Compilation:** Ensures code compiles correctly on each push.
2. **Packaging:** Generates a JAR file for stable versions.
3. **Continuous Delivery:** Prepares artifacts for deployment.

Conclusion

This project successfully demonstrates the implementation of a microservice-based architecture for managing INSA’s coffee machines. By leveraging Spring Boot and microservices, the system achieves modularity, scalability, and maintainability. Each microservice is designed to handle a specific function while communicating seamlessly through RESTful APIs.

The use of Git for version control and GitHub Actions for continuous integration ensures a robust development workflow, enabling automated builds, testing, and deployment. The database schema supports efficient data storage and retrieval, while the Eureka discovery service and Spring Cloud Config provide dynamic service registration and centralized configuration.