COMP41720-Distributed Systems-Group- 16

Fan Jiang	Wenjie Cai	Guiyang Fan	Meihui Wu
< 22212850>	< 23206906>	< 23205403>	<22209051>

Synopsis:

We are looking to develop a foundational e-commerce application system based on the Spring Boot and Spring Cloud frameworks. The system will encompass core services such as user services, product services, and order services. It will offer functionalities including user registration, product listings, adding order, and viewing order lists and details. This comprehensive system aims to streamline online shopping operations by integrating essential e-commerce components into a cohesive service architecture.

Technology Stack:

List of the main distribution technologies We use:

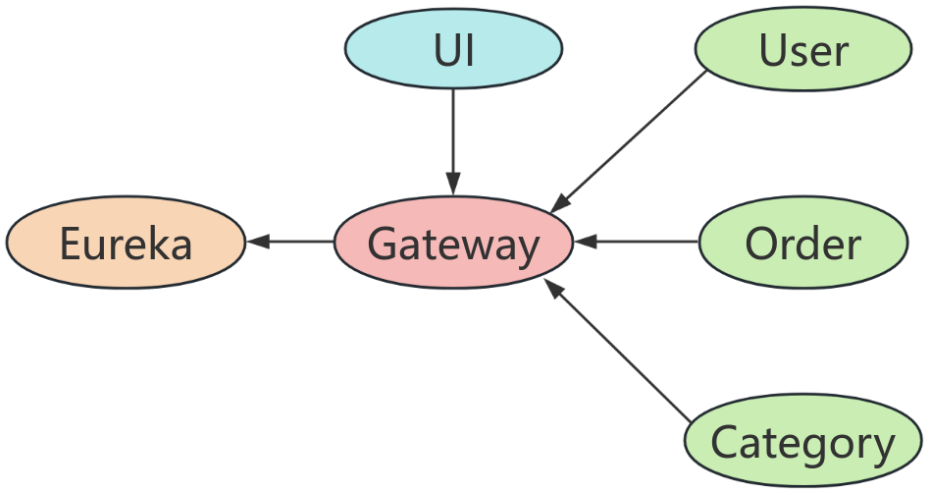
1. **Spring Boot**: Spring Boot is an extension of the Spring framework that simplifies the initial setup and development of new Spring applications. Its primary goal is to ease the development of stand-alone, production-grade Spring-based applications with minimal effort. In our project, it is used to create microservices with necessary dependencies such as **spring-boot-starter-web**, **spring-boot-starter-thymeleaf**, and **spring-boot-starter-test**.
2. **Spring Cloud**: This is a set of tools for quickly building some of the common patterns in distributed systems. In our project, we're using **spring-cloud-starter-config**, **spring-cloud-starter-netflix-zuul**, and **spring-cloud-starter-netflix-eureka-client**. Chosen for its ability to simplify the bootstrapping and development of new Spring applications. Features like auto-configuration and standalone deployment reduce development time and increase productivity. Essential for developing distributed systems and microservices architecture. It provides tools for configuration management, service discovery, and other patterns useful in a cloud environment
3. **MySQL Connector**: This is the official JDBC driver for MySQL. In our **order-service**, we're using this driver to connect to the MySQL database. Necessary for connecting our Java application with MySQL database, ensuring a reliable and efficient database communication.
4. **Rest**: The REST framework is used for distributing the services into microservices that can be accessed by a centralised application. REST framework allows each service to be accessible by the centralised application through a given URI. POST and GET methods allow data to be written and read in order to allow these services to communicate with each other.
5. **Vue:** Vue.js is a progressive JavaScript framework used for building user interfaces. Known for its simplicity and flexibility, Vue empowers developers to create dynamic and responsive web applications. With a reactive data-binding system and a component-based architecture, Vue facilitates the seamless integration of interactive features. It's lightweight and easy to pick up, making it an ideal choice for both beginners and experienced developers.
6. **Axios:** Axios is a popular JavaScript library used for making HTTP requests in both browser and Node.js environments. With a simple and intuitive API, Axios streamlines the process of handling asynchronous operations, such as fetching data from APIs. It supports various HTTP methods and automatically transforms responses into JavaScript objects, simplifying data manipulation.

System Overview

Describe the main components of your system.

1. **Registration Center (Discovery)**: Acts as the core of service discovery. It is responsible for registering and managing all instances of microservices, ensuring seamless communication between them. This central registry facilitates service discovery and load balancing, which is crucial for scalability and fault tolerance.
2. **Gateway**: Provides a unified entry point to the system. It is responsible for routing requests to the appropriate microservices and handling global functionalities like authentication and authorization. The gateway also acts as a reverse proxy, aggregating the responses from various services.
3. **Category Service**: Handles business logic related to products. This includes acquiring product lists and querying product details. It's a standalone service that can be scaled independently based on demand related to product information.
4. **Order Service**: Manages the creation of orders, viewing order lists, and order details. It is designed to handle high volumes of transaction data and can be scaled independently to manage load during peak times.
5. **User Service**: Deals with user-related operations, including user registration and authentication. It's crucial for personalizing user experience and securing user data.
6. **UI**: Front end where users can interact with the system easily.

Our system architecture diagram in this section.



➢ *This diagram shows how we decided to implement the services and the overall structure of the project.*

The front-end provides users with intuitive and convenient input methods. Through the User, Order and Category modules in the microservice architecture, the flexible management and interaction of user information, order data and commodity categories are realized.

Our system works based on the diagram.

- **Service Discovery**: Microservices, such as Category Service, Order Service, and User Service, register themselves with the Discovery module. This registration allows services to discover and communicate with each other dynamically.

- **Gateway Routing**: All external requests are routed through the Gateway. It directs these requests to the appropriate microservice. The Gateway also balances the load and secures the back-end services by providing an additional layer of abstraction.
- **Microservices Interaction**: Each microservice operates independently but can communicate with other services as needed. For example, the Order Service might need to interact with the User Service for customer details or with the Category Service for product information.

Our system is designed to support scalability and fault tolerance in the blow aspects:

- **Independent Scaling**: Each microservice can be scaled independently according to demand. This means that during high demand for a particular service (like Order Service during a sale), only the necessary service is scaled without affecting others.
- **Load Balancing**: The Discovery and Gateway modules facilitate load balancing. This ensures that no single instance of a service gets overwhelmed, thus maintaining system performance and reliability.
- **Decentralized Architecture**: The decentralized nature of microservices allows for fault isolation. If one service fails, it doesn't bring down the entire system. This contributes to the overall fault tolerance of the system.
- **Dynamic Service Discovery**: The Discovery module allows for dynamic registration and deregistration of service instances. This means the system can adapt to the addition or removal of instances in real-time, enhancing fault tolerance and scalability.

Contributions

Fan Jiang 22212850: Designing the project framework including the database, service architecture, entity design, and business logic; writing the entire back-end code; crafting and testing interfaces; leading the crafting of the project report.

Wenjie Cai* 23206906*: write front-end codes, set cross region in the rest project, record the video

***Guiyang Fan* 23205403:** front-end design, assist report writing and project checking, check and test code

Meihui Wu* 22209051*: draw the diagram of report and assist report writing, check and test code

Reflections

What were the key challenges you have faced in completing the project? How did you overcome them?

The first challenge we face is **difficulty in choosing the right theme**. Initially, we planned to develop a travel consulting system that would offer different quotes and detailed information based on user input. However, we encountered significant challenges in implementing the logic for this system. The complexity of integrating diverse travel data and generating customized responses proved to be more intricate than anticipated.

The other big problem for us is **switching to an E-commerce System**: In response to these challenges, we decided to shift our focus to a more comprehensible theme - an e-commerce system. This change allowed us to leverage our strengths in creating a structured and user-friendly shopping platform, where the logic and functionalities were more straightforward and within our technical capabilities.

What would you have done differently if you could start again?

Future Enhancements: Having successfully implemented the e-commerce application using modern technologies like Spring Boot and Spring Cloud, there are areas we could further develop if starting again like integrating additional services such as a payment gateway would enrich the application by accommodating more complex e-commerce transactions.

Focus on Performance and Security: Another area of improvement would be to place a greater emphasis on performance optimization and security reinforcement from the outset. Ensuring the stability and efficiency of the system is crucial, especially as it scales and faces increasing security threats.

What have you learnt about the technologies you have used? Limitations? Benefits?

The benefit is that we learn a lot about the techniques available. First up is the **Spring Boot and Spring Cloud**. These technologies significantly simplified the development and management of microservices. Spring Boot's auto-configuration and standalone capability made it easier to set up and develop applications, while Spring Cloud provided essential support for building resilient and scalable distributed systems. The second one is **Microservices Architecture**. Adopting a microservices approach allowed for the development of independently deployable services, making the system more scalable and easier to manage.

The limitation is that our technology needs to be strengthened. For one thing, **distributed systems are relatively complex.** While microservices offer numerous advantages, they also bring complexity, especially in terms of inter-service communication and data management. On the other hand, there are **security challenges.** Securing a microservices architecture can be more challenging compared to a monolithic architecture due to the increased number of endpoints and the distributed nature of the system.

Conclusion

Through this project, we gained valuable insights into the practical application of modern software development methodologies and technologies. We learned not only about the strengths and limitations of these technologies but also about the importance of flexibility in project planning and execution. This experience has provided a solid foundation for tackling similar complex projects in the future, with a more informed approach towards system design, scalability, and security.

Project Link: https://github.com/JoyJiang98/ShoppingOnlineSystem

Link to video: https://youtu.be/tTdxvPLeXYQ