Team Group 77

Yuxin Zhao	Ruiqi Zhao	Miaomiao Shi	Judith Smolenski
22200204	22205500	22205800	22204071

Synopsis:

1. ***What is the application domain?***

   The application domain for the system is library management. It focuses on creating a digital library platform where various services related to library operations will interact and function together, such as user management, book services, review, and message processing.

2. ***What will the application do?***

   The application is designed to manage a library system using a microservices architecture, incorporating elements of distributed systems. It handles user management book catalogs, borrower records, inventory, and notifications. Employing Spring Boot and Spring Cloud, the system features service discovery, centralized configuration, and distributed tracing. This setup, integrated with a distributed system approach, ensures robust and scalable library operations, facilitating both synchronous and asynchronous communications among microservices.

Technology Stack:

1. ***List of the main distribution technologies you will use.***

   Spring Boot: *Used for creating stand-alone, production-grade Spring-based applications easily. It streamlines the development process of the application by providing a simplified way to set up and run Spring-based applications, making it ideal for creating individual microservices in a scalable and efficient manner.*

   Spring Cloud: *Provides tools to quickly build some of the common patterns in distributed systems. This includes service discovery, configuration management. Using Eureka, Spring Cloud enables microservices to dynamically discover and communicate with each other, which is fundamental in a distributed system where services frequently scale up and down. Spring Cloud Config is employed to manage application configurations across multiple microservices. This feature ensures that all services have a centralized, consistent, and manageable configuration setup. These implementations highlight Spring Cloud's role in enhancing the communication and operational smoothness of microservices by addressing key distributed system challenges.*

   PostgreSQL: *This database approach simplifies data management, ensuring consistent handling of structured data and complex queries across different parts of the Library application. PostgreSQL's flexibility in managing diverse data types effectively supports the varied data requirements of the system.*

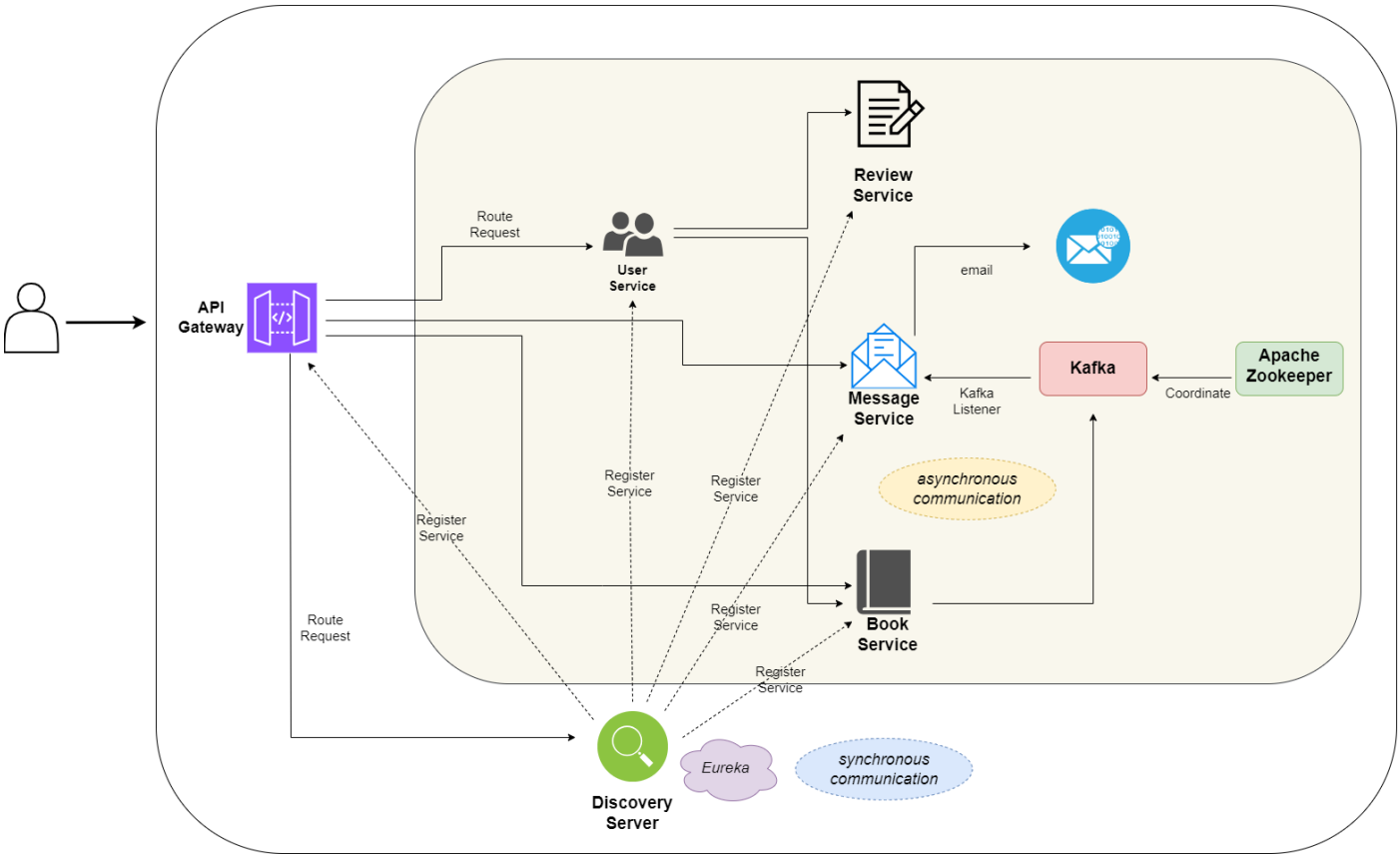
   Kafka: *Kafka is utilized for asynchronous data processing and streaming. It is particularly useful in scenarios where high-throughput and scalable message processing are required, such as event-driven architectures where multiple microservices react to state changes or updates. Kafka's ability to handle high-throughput scenarios ensures that large-scale data changes are efficiently propagated across services.*

   Eureka: *Utilized for service discovery, Eureka acts as a registry where microservices register themselves and discover other services. This enables easy inter-service communication and load balancing.*

System Overview

1. **Diagram**

Library Management Application Architecture Diagram



2. *Explain how your system works based on the diagram.*

API Gateway: *Acts as the system's entry point, routing client requests to the correct service. It enhances security and manages traffic, ensuring efficient service utilization.*

- ***Central Entry Point:*** *The API Gateway serves as the primary interface for all client requests, efficiently routing them to designated services, like the user service. It simplifies client interactions by being the sole point of communication.*
- ***Load Balancing:*** *Leveraging Eureka for service discovery, the gateway distributes incoming requests across multiple service instances. This ensures balanced load handling, improving performance and reliability.*
- ***Routing Configuration:*** *It has predefined routes targeting specific services (e.g., user service, discovery server), optimizing request routing and ensuring requests reach their intended destinations.*
- ***Logging and Monitoring:*** *Configured logging levels (info and trace) in your gateway facilitate detailed monitoring of operations and assist in troubleshooting, allowing you to track request flows and gateway performance effectively.*

By functioning as the single point of entry, the API Gateway significantly simplifies the interaction between the client side and microservices, enhancing the system's overall manageability and security.

Discovery Server: *The part is configured as a Eureka Server, a service discovery mechanism in Spring Cloud.*

- ***Self-Registration Disabled:*** *It's set not to register itself with Eureka, as it's the service registry.*
- ***Client Fetch Registry Disabled:*** *Prevents the server from trying to fetch the registry from other Eureka nodes.*
- ***Service Port Configuration:*** *It runs on port 8761.*
- ***Service Discovery Role:*** *Other microservices register with this server, allowing them to discover each other dynamically.*

Kafka cluster with Zookeeper:

- ***Zookeeper Service:*** *Acts as the coordinator for Kafka, managing brokers and ensuring cluster stability. It runs on the default client port 2181.*
- ***Kafka Broker Service:*** *This is the Kafka server (broker) that handles message storage and transmission. It depends on Zookeeper and is configured with various environment variables for Kafka settings, like broker ID, Zookeeper connect, listener configurations, and replication factors.*

This setup creates a Kafka environment suitable for development and testing, providing a message broker system for handling high-throughput, distributed messaging or event streaming.

Message Service: *Manages communication between user and bookservices. It uses Kafka for handling asynchronous communication and event-driven processes, such as sending notifications via email upon book checkout and return events.*

- ***Event-Driven Communication:*** *It listens to Kafka topics for checkout and return events. When an event occurs, it triggers the corresponding message handling.*
- ***Email Notification:*** *The service sends emails to users about their book checkout or return, enhancing user engagement.*
- ***Dynamic Email Configuration:*** *It dynamically configures email settings to send notifications, ensuring reliability in communication.*
- ***Logging and Monitoring:*** *Logs actions for checkout and return events, aiding in monitoring and auditing system activity.*
- ***Service Discovery:*** *Registered as a Eureka client, it enables easy discovery and communication within the microservices ecosystem.*

Book service: *Manages the library's book catalog, handling operations like listing all books, checking stock availability, book checkouts, and returns.*

- ***Database Interaction:*** *Uses a repository to interact with the database for retrieving and updating book information.*
- ***Transactional Operations:*** *The check out and return functions are transactional, ensuring database integrity during operations like updating book stock.*
- ***Kafka Messaging:*** *For asynchronous communication, it sends messages via Kafka when books are checked out or returned, enhancing user experience.*
- ***Stock Validation:*** *It checks and updates the stock availability of books, a crucial aspect for library management.*
- ***Service Discovery:*** *Registered as an Eureka client, it enables easy discovery and communication within the microservices ecosystem.*

This service centralizes book-related operations, ensuring efficient and reliable management of the library's catalog and transactions.

Review Services: *Handles the creation and retrieval of book reviews. It allows users to post reviews and fetch reviews for a specific book.*

- ***Database Interaction:*** *Integrates with PostgreSQL for storing and retrieving review data.*
- ***Data Transfer Objects (DTOs):*** *Utilizes DTOs like ReviewRequest and ReviewResponse to manage data exchange in review operations.*
- ***Service Layer:*** *The ReviewService class encapsulates the business logic for handling reviews, interfacing with the ReviewRepository for database operations.*
- ***Eureka Client Integration:*** *As a Eureka client, it registers with the Eureka server for service discovery, enabling it to interact with other microservices in the system.*

User Service: *Manages all user-related functionalities like book checkouts, returns, and handling reviews.*

- ***Centralized Logic for User Operations:*** *Reduces complexity in handling user requests by centralizing logic related to books and reviews in one service.*
- ***Microservice Communication:*** *Utilizes WebClient for RESTful communication with other services, ensuring efficient interaction for user actions.*
- ***Load Balancing and Service Discovery:*** *As an Eureka client, it balances loads across instances and discovers other services within the microservice architecture.*

3. Explain how your system is designed to support scalability and fault tolerance.

To enhance its ability to scale and remain robust in the face of challenges, the system incorporates a range of architectural elements and methodologies:

- *Microservices Architecture: This approach divides core services (such as User, Message, Book, and Review Services) into distinct microservices. Each of these can be scaled independently, allowing for precise resource allocation according to the specific needs of each service. This modular design not only optimizes resource use but also enables targeted scaling without impacting other services.*
- *Service Discovery with Eureka: The system uses a Discovery Server configured with Eureka, a critical component for scalable and fault-tolerant design. Eureka facilitates the dynamic registration and discovery of microservices, streamlining the addition or removal of service instances. This flexibility is key in managing increased traffic loads and in the quick replacement of any failed instances, thereby bolstering fault tolerance.*
- *Load Balancing via API Gateway: The API Gateway, integrated with Eureka, ensures effective distribution of incoming requests across various service instances. This load balancing is essential in preventing overload on any single service instance and contributes to the system's scalability and fault tolerance, ensuring smooth handling of traffic.*
- *Asynchronous Communication Through Kafka: The system employs Kafka for its Message Service, enabling asynchronous communication and event-driven processes. Kafka is renowned for its scalability and fault tolerance, especially in handling high-volume, distributed messaging. This choice allows the system to efficiently process a substantial number of events and notifications, which is vital for maintaining performance under diverse load conditions.*

By combining these components and strategies, the system is structured to adaptively respond to varying demands and maintain operational integrity, even in scenarios of service disruptions or component failures.

Contributions

Provide a sub section for each team member that describes their contribution to the project. Descriptions should be short and to the point.

Reflections

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

Reflecting on the project, we encountered several significant challenges, particularly in managing the messaging service and asynchronous processing. Here's how we navigated these issues:

Handling Messaging (Message Service):

- *Kafka Setup and Configuration: Initially, configuring Kafka for the Message Service was complex. We tackled this by diving deep into Kafka's documentation and online tutorials, which provided the insights needed to properly set up topics, partitions, and brokers.*
- *Dynamic Email Configuration: For dynamic email settings in notifications, we developed a configuration management system. This system was designed to be flexible, allowing changes in email settings without interrupting message delivery.*

- ***Handling Asynchronous Processing :***
    - *Concurrency and Parallelism: Implementing asynchronous processing raised concurrency and parallelism issues. We addressed these by carefully designing our asynchronous components to allow simultaneous processing of multiple tasks without conflicts.*
    - *Error Handling: Asynchronous operations posed a challenge in error management. We implemented comprehensive error logging and monitoring to track and address errors effectively.*
    - *Ensuring Data Consistency: Maintaining data consistency in an asynchronous environment was challenging. We resolved this by employing transactional operations and message sequencing to preserve data integrity.*
    - *Testing and Debugging: Debugging asynchronous code was more intricate than synchronous code. We relied heavily on extensive unit and integration testing, alongside detailed logging, to identify and resolve issues.*

In both messaging and asynchronous processing, thorough planning, research, and the use of appropriate tools and frameworks were key to overcoming challenges. Additionally, a focus on error handling, testing helped ensure the reliability and robustness of the asynchronous components within the system. Collaboration among team members and leveraging community resources also played a vital role in addressing challenges and achieving project success.

2. ***What would you have done differently if you could start again?***

Looking back at the project, there are a few key areas where a different approach might have enhanced our system's design and efficiency. Initially, integrating UML diagrams from the project's start would have been a game-changer. These diagrams would have offered a clear, visual blueprint of the system's architecture, greatly aiding in both planning stages and team communication.

Regarding database technology, a mixed approach incorporating both NoSQL and SQL databases would have been ideal. Leveraging MongoDB for its flexibility with unstructured data, coupled with PostgreSQL's robust handling of structured data and complex queries, would have provided a more dynamic and efficient database solution. This blend of database technologies would have catered to diverse data storage and processing needs more effectively.

Lastly, employing both RabbitMQ and Kafka could have optimized our messaging infrastructure. RabbitMQ's strength in managing lightweight messaging tasks would have complemented Kafka's capability in handling high-volume, event-driven data streams. This dual approach to message brokering would have offered a more comprehensive and adaptable messaging system.

3. ***What have you learnt about the technologies you have used? Limitations? Benefits?***

Throughout the development of our system, we've gained valuable insights into various technologies, each bringing its unique set of benefits and limitations. The use of Spring Boot and Spring Cloud has been central to this learning experience. Spring Boot simplifies the bootstrapping and development of new Spring applications, significantly reducing development time and increasing productivity. However, it sometimes leads to challenges in fine-tuning configurations for specific needs.

The integration of Eureka for service discovery in Spring Cloud simplifies the management of microservices, but it also introduces challenges in network reliability and resilience. We encountered situations where network fluctuations caused Eureka clients to de-register temporarily, leading to service unavailability. Ensuring consistent service discovery in the face of network inconsistencies required additional safeguards, such as implementing circuit breakers and fallback mechanisms.

Our API Gateway effectively manages traffic and routes client requests, but under high load, we observed challenges in throughput and latency. The gateway became a bottleneck during peak traffic, slowing down

request processing. Optimizing its configuration for high-load scenarios, such as tweaking timeout settings and concurrency limits, was necessary to maintain performance.

Kafka's role in handling high-throughput, distributed messaging is undeniable, but its complexity in configuration and cluster management posed significant challenges. Ensuring consistent message delivery and order, particularly in failure scenarios, required careful planning of topic partitions and replication strategies. Additionally, managing Zookeeper's coordination with Kafka for broker management added another layer of operational complexity.

While the microservices architecture offers scalability and fault tolerance, managing data consistency across services was a major challenge. Implementing distributed transactions to ensure data integrity across the User and Book services, without compromising on performance, required intricate coordination. Moreover, network latency and inter-service communication overhead sometimes affected the overall response time of the system.

In conclusion, these technologies have taught us the delicate balance between leveraging advanced features for efficiency and scalability, and the complexities they introduce in terms of setup, configuration, and management. The key lesson is that a well-thought-out architectural plan, alongside a deep understanding of each component, is essential for the system.