Foodie Group

Jin Kuan Moo	Itgel Ganbold	Yu Feng	Minh Ly Nguyen	Jiakai Chen
17200187	18407654	22200055	22201371	21212254

Synopsis

The platform functions as a food ordering and delivery system, allowing users to log in, peruse restaurants and their menus, place orders, complete payments, and have their food delivered to them.

The project is a distributed system composed of five key microservices: User, Restaurant, Order, Payment, and Delivery. It focuses primarily on the aspects of distribution rather than on the complexities of business operations. Key functionalities include:

- User service: Facilitates user interaction including user authentication, restaurant and dishes selection, payment confirmation and order placement.
- Restaurant service: Interfaces with Order service to retrieve all available restaurants and specific dishes by ID.
- Order service: communicates with all other services to facilitate the ordering process.
- Payment service: Interfaces with the User service and Order service to handle payment methods of the order, collaborates with the User service for card verification and card details storing.
- Delivery service: Operates in sync with the Order service to receive details of the orders and coordinates with drivers.

Technology Stack

Actor (Akka)

According to [1], the Actor Model, as implemented in Akka, is particularly well-suited for modern, distributed systems like a food order and delivery system. Akka actors enforce encapsulation without resorting to locks, operating as independent entities that react to signals (messages). In a food ordering system, different components (like order processing, payment handling, and delivery management) can be encapsulated in separate actors, each managing its own state and behavior independently. This approach significantly reduces the complexity associated with concurrent modifications, which is a common challenge in such systems.

Actors in Akka interact through non-blocking message passing [1]. When an actor sends a message, it doesn't transfer its thread of execution to the receiver, allowing the actor to continue its work without waiting for a response. This model is ideal for a food ordering system where different operations (like checking restaurant availability, processing payments, updating order status) can occur simultaneously without blocking each other. This leads to a clean separation of concerns and makes the system more maintainable and flexible to changes, such as adding new features or modifying existing processes in the food ordering and delivery flow.

Akka's supervision strategy provides a robust mechanism for handling failures. In a food order and delivery system, different parts of the system can fail (like a payment gateway timeout or a problem in order processing). Akka's model allows parent actors (supervisors) to define strategies for dealing with such failures, ensuring the system's resilience and continuous operation [1].

Actors are lightweight and can be distributed across multiple nodes, making the system easily scalable [1]. This is crucial for a food ordering system, which may experience varying loads and needs to scale dynamically to handle peak times efficiently. With Akka, the scheduling of millions of actors are made efficient across a handful of threads aligns well with modern CPU architectures. This efficient use of resources is vital in a distributed system like a food ordering platform, which requires high performance and responsiveness.

Lastly, In the Actor Model, errors are handled through messages, aligning with domain-driven design [1]. This approach fits well in a food ordering system where different kinds of errors need to be handled gracefully, whether they are user input errors, service errors, or system failures.

React

React is employed as the primary framework for building the user interface of our real-time dashboard. This JavaScript library excels in creating dynamic and responsive UI's, particularly suited for single-page applications like our dashboard. In our implementation, React's useState hook is used to manage the state of order messages, which is crucial for reflecting real-time data updates. The useEffect hook handles the lifecycle of the SSE connection, establishing the connection when the component mounts and ensuring it's properly closed to prevent memory leaks when the component unmounts. Certain degree of fault tolerance is implemented, as any connection interruptions will be followed by reconnection attempts. The dashboard leverages functional components, a modern React approach that enhances simplicity and performance. Components such as 'EventCard' are designed to receive props, in this case, individual messages, enabling modular and reusable code structure.

Server-Sent Events (SSE)

Incorporating Server-Sent Events (SSE) within a Spring Boot backend and React frontend for a real-time food delivery service dashboard exemplifies a tailored approach to efficient, unidirectional data streaming [2]. SSE's capability to provide a one-way communication channel from server to client is pivotal in scenarios that demand frequent and real-time updates, such as the dynamic setting of a food delivery service. The constantly evolving information about order statuses, restaurant updates, and delivery notifications necessitates a system where updates are pushed to the user interface as they occur. This is essential for ensuring the dashboard reflects the most current view of operations, a critical aspect for both service providers and customers in this scenario.

The implementation of the EventSource API in the React frontend plays a key role in this setup. As a standard feature in modern web browsers, it is specifically designed to handle SSE connections seamlessly. In the provided code, the EventSource API is used to establish a continuous link with the server. The onmessage event listener is responsible for processing the stream of data received from the server, and the onerror handler enhances the system's robustness by managing potential connection losses, including automatic reconnection attempts. This resilience is vital for a consistent and reliable user experience.

By opting for SSE, the application purposefully sidesteps the complexities and resource demands associated with bidirectional communication protocols like WebSockets, which are often unnecessary for applications where the primary requirement is to display data from the server to the client. This choice not only streamlines the data flow but also optimizes server and network resources. Furthermore, Spring Boot's backend architecture supports this choice by efficiently managing SSE connections and ensuring the server can handle the continuous flow of data, making it a well-rounded solution for real-time data delivery in a high-demand service environment.

MongoDB

The initial choice of SQLite for the food delivery distributed system, which includes services like order, restaurant, user, delivery, and payment, was likely influenced by SQLite's simplicity, ease of setup, and adequacy for smaller-scale applications. SQLite, being a lightweight, file-based database, is excellent for rapid development and scenarios with limited concurrency and data management requirements. However, as the system expanded and scalability became a paramount concern, the shift to MongoDB marked a strategic and necessary evolution.

MongoDB, as a NoSQL database, offers substantial benefits in terms of scalability and flexibility, crucial for the growing demands of a distributed food delivery system. Unlike SQLite, MongoDB excels in handling high concurrency, large-scale data management, and complex queries, which are integral to the dynamic and data-intensive nature of food delivery services [3]. Its ability to scale horizontally allows the system to accommodate an increasing number of transactions and a growing user base without compromising performance. The schemaless nature of MongoDB is also a significant advantage, offering the flexibility to evolve the database schema without extensive rework, a critical feature for the fast-paced and evolving food delivery industry. This flexibility in managing complex data structures, along with MongoDB's robust support for geospatial queries, aligns perfectly with the needs of a service that handles diverse data types, from user profiles to geospatial data for delivery tracking. Integrating MongoDB with modern development frameworks ensures smoother development and operational processes, aligning the database technology with the future-oriented goals of the food delivery system. This transition from SQLite to MongoDB thus addresses the scalability challenges while positioning the system for continued growth and innovation.

Docker

Docker allows each service (User, Restaurant, Order, Payment, and Delivery) to be containerized independently. This means each service can be scaled, updated, or even replaced without impacting the other parts of the system. As mentioned in [4], this is crucial in a dynamic environment where different services might have varying load patterns and requirements. Isolation of the environment for each service also improves security by limiting the scope of access and potential breaches. Moreover, Docker containers are lightweight, which means they use resources more efficiently compared to virtual machines. This efficiency is key in a microservices architecture where you might be running many containers simultaneously.

Docker ensures consistency across environments, meaning the application will behave the same way in development, staging, and production. This consistency significantly reduces "it works on my machine" problems and streamlines the deployment process [4].

Kubernetes

Kubernetes excels in managing containerized applications. It can automate the deployment, scaling, and operation of these containers. For our distributed system, Kubernetes can handle load balancing, service discovery, and self-healing (automatically restarting failed containers), which are essential for maintaining high availability and reliability [4].

With Kubernetes, we achieve fault tolerance by gaining access to a robust ecosystem for monitoring and logging, which is crucial for maintaining the health and performance of your microservices. Kubernetes' ability to aggregate logs and metrics from various services simplifies the process of identifying and resolving issues [4].

The combination of Docker and Kubernetes facilitates quicker development and deployment cycles. Being able to quickly deploy and roll back services in response to issues or user feedback can be a significant advantage in the fast-paced world of online food ordering and delivery [4].

System Overview

Microservices

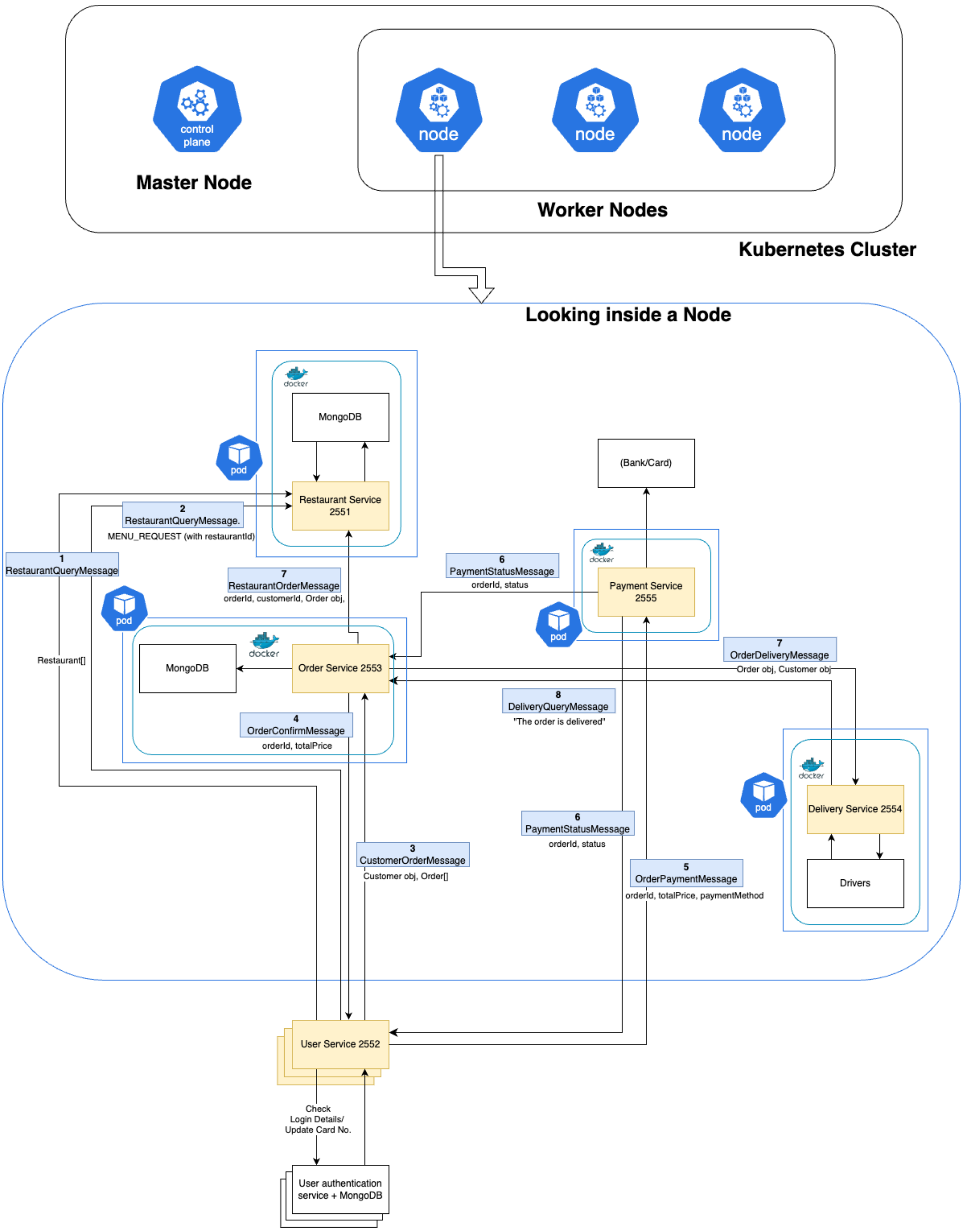
All microservices are implemented using Akka for building concurrent, distributed, and resilient message-driven applications.

1. **userService**

userService manages user-related functionalities, such as user authentication, processing restaurant queries, and handling order confirmations. This service also interfaces with the paymentService for handling transactions. The service uses MongoDB's Bson and Document classes for database operations, ensuring secure storage and retrieval of user data. Additionally, encryption algorithms (AES with Base64 encoding) are utilized for securing sensitive data like card numbers, enhancing the system's security.

2. **restaurantService**

Primarily handles restaurant-related data, such as fetching restaurant lists and menu items from the MongoDB database, and processing restaurant orders. The service dynamically retrieves restaurant information like names, addresses, and menus, and presents these details to users. The service uses MongoDB's Bson and Document classes for database operations, ensuring secure storage and retrieval of restaurant and menu data.



System Architecture Diagram

3. orderService

The orderService processes incoming userService requests and paymentService messages, and coordinates with restaurantService and deliveryService to ensure seamless order handling and delivery. The service uses MongoDB's Bson and Document classes for database operations, ensuring secure storage and retrieval of order data.

4. paymentService

The service processes payment messages from userService, distinguishing between card and cash payments. It simulates interaction with a payment gateway if the user chooses to pay in card, and updates the status of the order to cash-unpaid, card-unpaid or card-paid and sends the messages to orderService and userService.

5. deliveryService

Upon receiving an OrderDeliveryMessage, the DeliveryService computes and dispatches delivery tasks to available drivers, tracking the progress and updating the status of each order. It also handles DeliveryQueryMessages, forwarding updates to the relevant parties.

User flow

1. User Authentication: The userService asks for authentication information (userId & password) from the user.
2. Display Restaurants and Menus: Upon successful login, the userService fetches and displays restaurant information and menus, allowing users to make food orders.
3. Order Placement: Once the user selects their desired food items, the userService collects the order details and sends a CustomerOrderMessage to the orderService.
4. Order Processing: The orderService processes the received order. It calculates the total price and stores the order in the database. The orderService then sends an OrderConfirmMessage back to the userService, confirming the order details and total price.
5. Initiate Payment Process: The userService prompts the user to choose a payment method. If the user selects a card payment, userService interacts with the paymentService to handle the transaction. If the user selects Cash payment, the payment status is noted as Cash unpaid for later use of orderService and deliveryService.
6. Payment Processing: The paymentService processes the payment and sends a PaymentStatusMessage back to the orderService, indicating the payment status (successful or failed).
7. Order Dispatch: After payment is processed and orderService receives PaymentStatusMessage, the orderService forwards the order details to the restaurantService and deliveryService. The restaurantService prepares the order, while the deliveryService allocates a driver for delivery.
8. Delivery Management: The deliveryService, using the driverService, manages the delivery process. It updates the order status and provides real-time delivery tracking to the userService.
9. Order Completion: Once the order is delivered, the deliveryService updates the order status and informs the userService, which then notifies the user of the successful delivery.

Throughout this process, Server-Sent Events (SSE) is used to update the front-end dashboard status of all the services.

Docker

The Dockerization of our food order and delivery system is achieved using Dockerfiles for each service and a docker-compose YAML file for orchestration. The docker-compose file defines each service, specifying the build context and Dockerfile. It also configures network settings, assigning all services to the foodie-network which uses a bridge driver for inter-container communication. The ports for each service are exposed, aligning internal ports with external ones for accessibility. Docker-compose further simplifies the process of running all these services together, ensuring they are networked and function cohesively as a unified application.

Scalability

1. **Horizontal Pod Autoscaler (HPA) in Kubernetes**

The Horizontal Pod Autoscaler (HPA) in Kubernetes stands as a fundamental feature for dynamic application scalability [5]. It is designed to automatically adjust the number of Pods in a Deployment or ReplicaSet, based on observed metrics like CPU utilization or custom-defined metrics. The HPA operates by continuously monitoring these metrics for each Pod via a metrics server. When it detects that the average CPU utilization across all Pods surpasses a predefined threshold, the HPA responds by increasing the number of Pods, thereby distributing the workload more effectively. Conversely, should the utilization fall below this threshold, the HPA reduces the Pod count, conserving resources.

Users have the ability to finely tune HPA's behavior by setting target metrics, such as desired CPU utilization percentages, and by defining the minimum and maximum number of Pods that can be deployed. This configuration flexibility is key in managing the application's scalability, enabling it to adeptly handle varying loads autonomously. The HPA's adaptive scaling is especially beneficial for applications with fluctuating workloads. During times of high traffic, HPA scales up the services to maintain performance levels, while in periods of lower activity, it scales down to optimize resource usage and reduce operational costs [5]. This intelligent scaling mechanism ensures that resources are efficiently used and that the application consistently performs well, adapting to the changing demands of the environment.

2. **Load Balancing with Kubernetes Services**

Kubernetes Services play a vital role in distributing network traffic and ensuring the high availability of applications within the Kubernetes ecosystem [6]. These services function as an internal load balancer, intelligently routing incoming requests to multiple Pods within a Deployment. Kubernetes offers various types of Services to cater to different needs: ClusterIP for internal cluster communication, NodePort for external traffic exposure, and LoadBalancer, which integrates with external cloud load balancers for broader accessibility.

An important aspect of Kubernetes Services is their ability to provide stable access points to the application. They do this through stable endpoints, like DNS names or virtual IPs, that do not change despite scaling operations or Pod restarts. This stability is crucial for maintaining consistent access to the application, regardless of the underlying infrastructure changes.

Fault Tolerance

1. **Supervision Strategy in Akka Actors**

In our food delivery service application, the 'FoodieActor' class, extending Akka's AbstractActor, demonstrates a strategic approach to fault tolerance using Akka's supervision strategy. The class employs Akka's 'OneForOneStrategy', which addresses failures of individual child actors independently, rather than applying the same recovery action across all children, as in 'AllForOneStrategy' [7].

This supervision strategy within 'FoodieActor' is tailored to respond differently to various types of exceptions. For transient errors like 'ArithmeticException', it chooses to resume the actor, keeping its state intact. In contrast, a 'NullPointerException' triggers a restart of the affected actor, effectively resetting its state to recover from potential inconsistencies. For more critical issues, such as an 'IllegalArgumentException', the strategy stops the actor, used when an error is deemed irrecoverable or indicative of a serious flaw. For all other exceptions, the strategy escalates the error, passing the responsibility up the actor hierarchy, allowing higher-level actors to decide on the appropriate course of action.

This nuanced implementation of fault tolerance in 'FoodieActor' underlines the importance of tailored error handling in distributed systems. By carefully differentiating between types of exceptions and assigning appropriate recovery mechanisms, the system ensures resilience and stability, maintaining uninterrupted service in the face of individual actor failures.

2. Fault Tolerance with Kubernetes

In our food delivery distributed system, Kubernetes plays a crucial role in enhancing fault tolerance. It does this primarily through the management of Pod replication and self-healing capabilities. By using Deployments and ReplicaSets, Kubernetes ensures that multiple replicas of each Pod are maintained. This replication is vital for service continuity, as it allows other Pods to take over if one fails. Additionally, Kubernetes automatically restarts any crashed or unresponsive Pods, minimizing downtime and maintaining consistent service availability.

Kubernetes Services aid in fault tolerance by acting as load balancers, evenly distributing network traffic among all operational Pods. In the event of Pod failures, these Services reroute traffic to healthy Pods, ensuring uninterrupted user access [8]. For stateful components, Kubernetes offers StatefulSets, which maintain data consistency and Pod identity even after failures. Moreover, Kubernetes' node health checks and automatic failover mechanisms ensure that the system swiftly recovers from Node failures, maintaining the desired state of the application. Collectively, these features make Kubernetes an effective platform for ensuring high availability and resilience in distributed systems.

3. Fault Tolerance with MongoDB

In a MongoDB deployment, we are utilizing three nodes which significantly enhances fault tolerance, a key aspect of database resilience and reliability. This setup typically constitutes a replica set, where one node functions as the primary node and the other two serve as secondary nodes [9]. The primary node handles all write operations, while the secondaries maintain copies of the primary's data, ensuring real-time data redundancy and consistency across the nodes.

The strength of this three-node configuration lies in its automatic failover capability. If the primary node encounters a failure, one of the secondary nodes is automatically and promptly elected to become the new primary. This transition ensures minimal interruption in database operations, maintaining continuous data availability and write capability. The election process is managed internally by MongoDB, requiring no manual intervention, which is crucial for maintaining high availability, especially in critical applications where downtime can have significant repercussions.

Moreover, the data redundancy inherent in this setup safeguards against data loss. Even if one node goes down, the remaining nodes continue to hold the complete data set. This replication across multiple nodes means there's no single point of failure, enhancing the overall resilience of the system. Additionally, having multiple nodes allows for read scaling; read operations can be distributed across the secondary nodes, optimizing performance and balancing the load.

Contributions

Jin Kuan Moo(Student ID:17200187)

- Responsible for the user service development and its interactions with all other services
- Implemented a login function by utilising mongoDB as the database to store user's information
- Carried out various integration tests between services, primarily between user and other services, to ensure the overall operability of the system
- Introduced several logics in the user interaction to facilitate user experience
- Refactored the code to accommodate multi users scenario

Itgel Ganbold (Student ID: 18407654)

- Developed Restaurant Service and its related MongoDB collection on Atlas. The service connects to User Service and provides list of restaurants. Upon a restaurant selection, the service sends a list of menu items.
- The service also receives messages from Order Service to inform the restaurants of the order.
- Investigated WebSocket and Server-sent Events as a suitable front-end technology.
- Created the front-end system dashboard.
- Wrote the back-end code that allow live update of the system to front-end.

Yu Feng (Student ID: 22200055)

- Developed Order Service and integrated with User, Restaurant, Payment and Delivery services to ensure seamless communication.
- MongoDB Connection Setup and create methods for creating, reading, updating, and deleting order data.
- Established standardized messaging protocols in core.
- Created Supervision Strategy in Akka Actors for fault tolerance.
- Wrote Part of the report.

Minh Ly Nguyen (Student ID: 22201371)

- Developed Payment Service and refine related codes in Order Service to ensure smooth communication.
- Developed part of User Service which requests the payment by handling user input of payment method, encrypting and decrypting card number, and updating card number in Users MongoDB.
- Carried out multiple integration tests between Payment, Order & User service.
- Wrote Part of this report and part of the script for the video.

Jiakai Chen (Student ID: 21212254)

- Developed Delivery Service and designed the MongoDB collections of drivers and order tracking history.
- Developed part of Messages and User Service to ensure the system goes with appropriate multi-user implementation.
- Deployed the system to Docker Engine and Kubernetes.
- Recorded the video for briefly demonstrating how the system will work.

Reflections

In our food delivery project, we faced several challenges with using Maven, an essential tool for project management and build automation. These challenges primarily revolved around managing dependencies and ensuring consistent build processes across various independently developed services like user, restaurant, and order management. To tackle these issues, we standardized the Maven project structure for all services and utilized a parent POM (Project Object Model) to manage common dependencies. This approach helped maintain uniformity in builds and reduced conflicts caused by different library versions.

In our project, integrating different services like user, restaurant, and order management was a major challenge, particularly during testing. We faced issues in ensuring smooth communication between these services, caused by differences in how they exchanged messages. To solve this, we standardized our messaging protocols, as exemplified by the ‘CustomerOrderMessage’, ‘OrderDeliveryMessage’, ‘OrderConfirmMessage’, and ‘RestaurantOrderMessage’ classes. This standardization ensured consistent and clear communication across all services. Through thorough integration testing, we confirmed that each service could correctly send and receive messages. This approach allowed us to successfully integrate all services into a cohesive and functioning system.

The front-end dashboard of our application is designed with a minimalistic approach, effectively representing the system’s inner workings. While the use of SSE enables efficient unidirectional communication from the server to the client, it's important to note that SSE doesn't support bidirectional communication. This means that while the server can easily send data to the client, the client cannot send data back to the server via SSE. To enhance the application's interactivity and realism, considering the implementation of a technology that supports bidirectional communication, such as WebSockets, might be beneficial. This would enable a more dynamic interaction between the client and server, particularly for the user service's front-end.

We integrated Server-Sent Events (SSE) in a unique way within our Maven-based architecture, which uses the Spring Boot framework. Spring Boot allows us to create stand-alone applications, typically with a single entry point annotated with @SpringBootApplication. However, to align with our goal of creating a distributed system, we have adopted a different approach. Each service in our system has its own dedicated SSE handler, deviating from the more centralized model commonly used in Spring Boot applications. This design choice promotes decoupling between services, thereby reducing single points of failure and enhancing the system's fault tolerance. This architectural decision reflects our commitment to a microservices-oriented structure, where each component operates independently while contributing to the system's resilience.

The shift from SQLite to MongoDB in our food delivery project was driven mainly by the need for greater scalability and the ability to handle more complex data. SQLite is well-suited for smaller applications but struggles with the high transaction volumes and growth demands of a large-scale food delivery service. MongoDB, a NoSQL database, addresses these issues effectively. It can manage large amounts of data and supports scaling out through sharding, which is essential for handling our growing user base and the complex operations involved in our service. MongoDB's document-oriented model is also more adaptable for managing the varied data structures we encounter, like nested orders and user profiles, compared to SQLite's more rigid, table-based structure [3]. This flexibility makes it easier to work with complex data. However, it's important to note that MongoDB's distributed nature and advanced features add some complexity to its setup and management, and it may require more computing resources as the system grows. Despite these challenges, moving to MongoDB is a strategic fit for our project's needs, offering the scalability and flexibility necessary for our food delivery system's continued growth and development.

REFERENCES

[1] Akka documentation (no date) How the Actor Model Meets the Needs of Modern, Distributed Systems • Akka Documentation. Available at: https://doc.akka.io/docs/akka/current/typed/guide/actors-intro.html.

[2] Limbu, U. (2022) Server-sent event (SSE) chat application using spring boot and REACT JS, Medium. Available at: https://umes4ever.medium.com/server-sent-event-sse-chat-application-using-spring-boot-and-react-js-945b7fdf88f.

[3] Comparing the differences - mongodb vs mysql (no date) MongoDB. Available at: https://www.mongodb.com/compare/mongodb-mysql.

[4] A complete guide to build Docker microservices architecture (no date) Third Rock Techkno. Available at: https://www.thirdrocktechkno.com/blog/microservices-with-docker/.

[5] Horizontal pod autoscaling (2023) Kubernetes. Available at: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[6] Overview (no date) Kubernetes. Available at: https://kubernetes.io/docs/concepts/overview/#:~:text=Kubernetes%20provides%20you%20with%3A,that%20the%20deployment%20is%20stable.

[7] Akka documentation (no date) Supervision and Monitoring • Akka Documentation. Available at: https://doc.akka.io/docs/akka/2.5/general/supervision.html.

[8] Limbu, U. (2022) Server-sent event (SSE) chat application using spring boot and REACT JS, Medium. Available at: https://umes4ever.medium.com/server-sent-event-sse-chat-application-using-spring-boot-and-react-js-945b7fdf88f.

[9] Overview (no date) Kubernetes. Available at: https://kubernetes.io/docs/concepts/overview/#:~:text=Kubernetes%20provides%20you%20with%3A,that%20the%20deployment%20is%20stable.