Avengers

Nithin Katta Kiranprakash	Bharath Bharadwaj	Praveen Kamate
22200096	22200109	22200326

Synopsis:

"To Design and Develop a distributed Food delivery application which can be used by Customers to order food, Restaurants to display menus, and Delivery Agents to deliver food."

Domain - "Online Food Delivery"

- *It allows users to select a restaurant, add food items to a cart and place an order, also view their past orders.*
- *It allows Restaurant management to create/modify the menu, accept the order, and provides updates.*
- *It allows delivery agents to get requests from restaurants for order delivery.*

Technology Stack

- *REST: All services (Restaurant, Customer, and Delivery) are built as Spring boot REST Applications.*
- *Service Discovery: This service is built as a spring boot application. All the other services register to this service on start-up. This service has a list of all the available services and helps in scalability.*
- *API Gateway: This service is built as a spring boot application, that sits between the client and all the backend services, all the requests from the client will be routed to the respective services through this service. It shall handle the load balancing (round robin) for all requests to the services.*
- *RabbitMQ: This is a broker service used for event-based communication between the services and UI.*

Reasons for choosing the above technologies

REST: REST has been chosen as it provides lightweight, scalable, and fast communication between microservices and it is format agnostic meaning requests can use JSON, XML, etc. This was the requirement for communication between the services.

Service Discovery: Communication between the microservices can become tightly coupled if all the details about the other services are hard coded. This is primarily the reason Eureka service discovery was chosen so that all the services themselves register themselves and all the information about the services will be present in Eureka.

API Gateway: API Gateway has been chosen as it allows for the dynamic routing of requests to microservices. API Gateway also takes care of load-balancing requests when multiple instances of the same service are present. This is the reason it was chosen so that dockerization or creating Kubernetes services will be accommodated.

RabbitMQ: RabbitMQ is an open-source distributed broker that allows for event-based *communication. Since food delivery application needs to be event-based i.e., customer placing orders, restaurants preparing an order to delivery agents delivering the order are all events that need to be constantly updated in the UI.*

System Overview

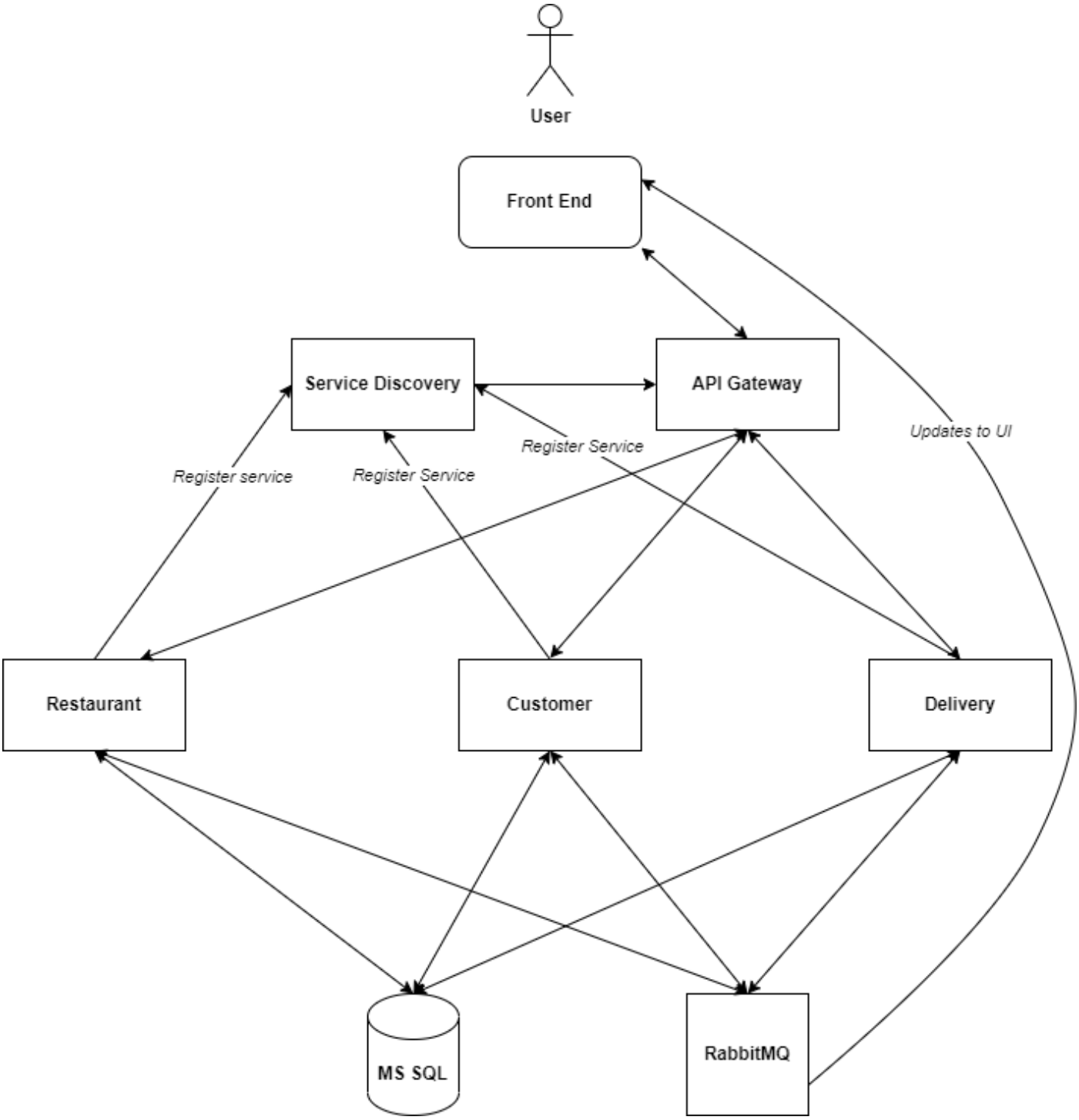

Figure 1: Avengers Food App - System Architecture

Restaurant: Restaurant service is a RESTful Spring Boot application. It consists of all the APIs which will be required by the restaurant user to view, and update items (name, price, and status) in the menu. It also allows the user to update the status of a customer order once the order has been prepared.

Customer: It is a REST Service that interacts with the database using JPA. All the APIs needed for the customer view, add to cart, and place an order are available. It allows the customer to view various restaurants, their menus, and create an order. It also allows the updating of the customer.

Delivery: Delivery service is a RESTful Spring Boot application. It consists of all the APIs which will be used by the delivery agents to view the current orders assigned to them. It can also be used to view past orders assigned and it assigns an available agent with the order once prepared by the restaurant.

Service Discovery: Eureka is a Spring Boot application that contains information about all the microservices. All the microservices (API Gateway, restaurant, customer, and delivery) will register their service to the Eureka server. It is a central service that contains the global view of services along with their addresses.

API Gateway: API gateway is a Spring Boot application that sits between the frontend and collection of backend services acting as a reverse proxy. On receiving the request from frontend (client) it communicates with the Eureka service to fetch the address of the required service and routes the request to the service based on the pattern specified and returns the result back to the client.

MS SQL Server: Microsoft SQL Server is a relational database management system used for storing and updating data and it has been used in the context of this application for storing and updating data related to customers, restaurants, delivery agents, customer orders, etc. which has been depicted using ER diagram shown below

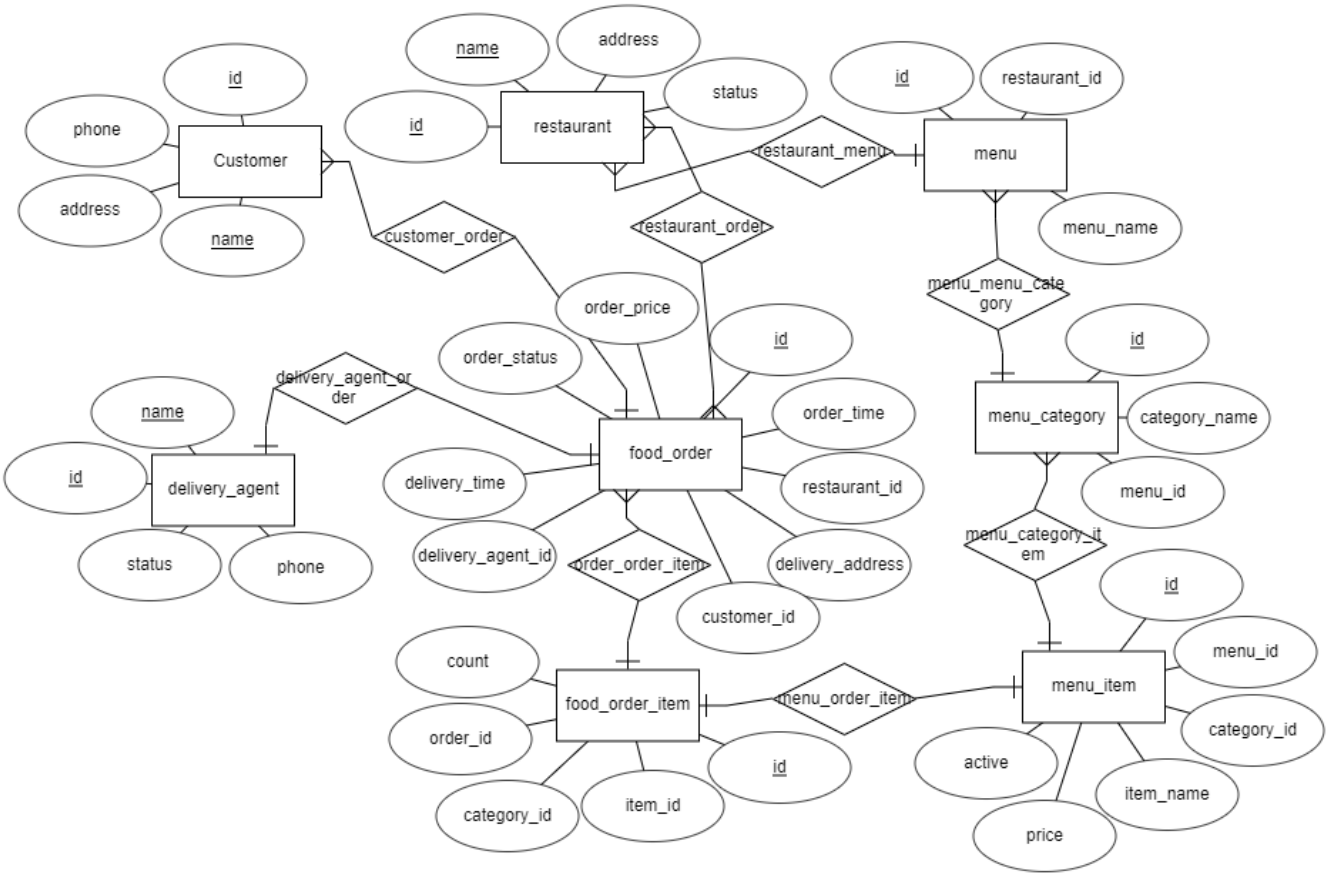

Figure 2: ER Diagram

RabbitMQ: RabbitMQ is the message broker that facilitates the exchange of messages between the services. The messages are exchanged between the Customer and Restaurant service to place the order, the Restaurant, and Delivery service once the order is prepared and the Delivery service and Customer service once the order is delivered.

React Frontend: Customer facing UI interface where users can select a restaurant and view the menu of the restaurants, add items to a basket and place an order, restaurant users can view the menu of their restaurant and update the prices and availability of items, the delivery agent can update the status of orders and view their past delivered orders.

System Workflow:

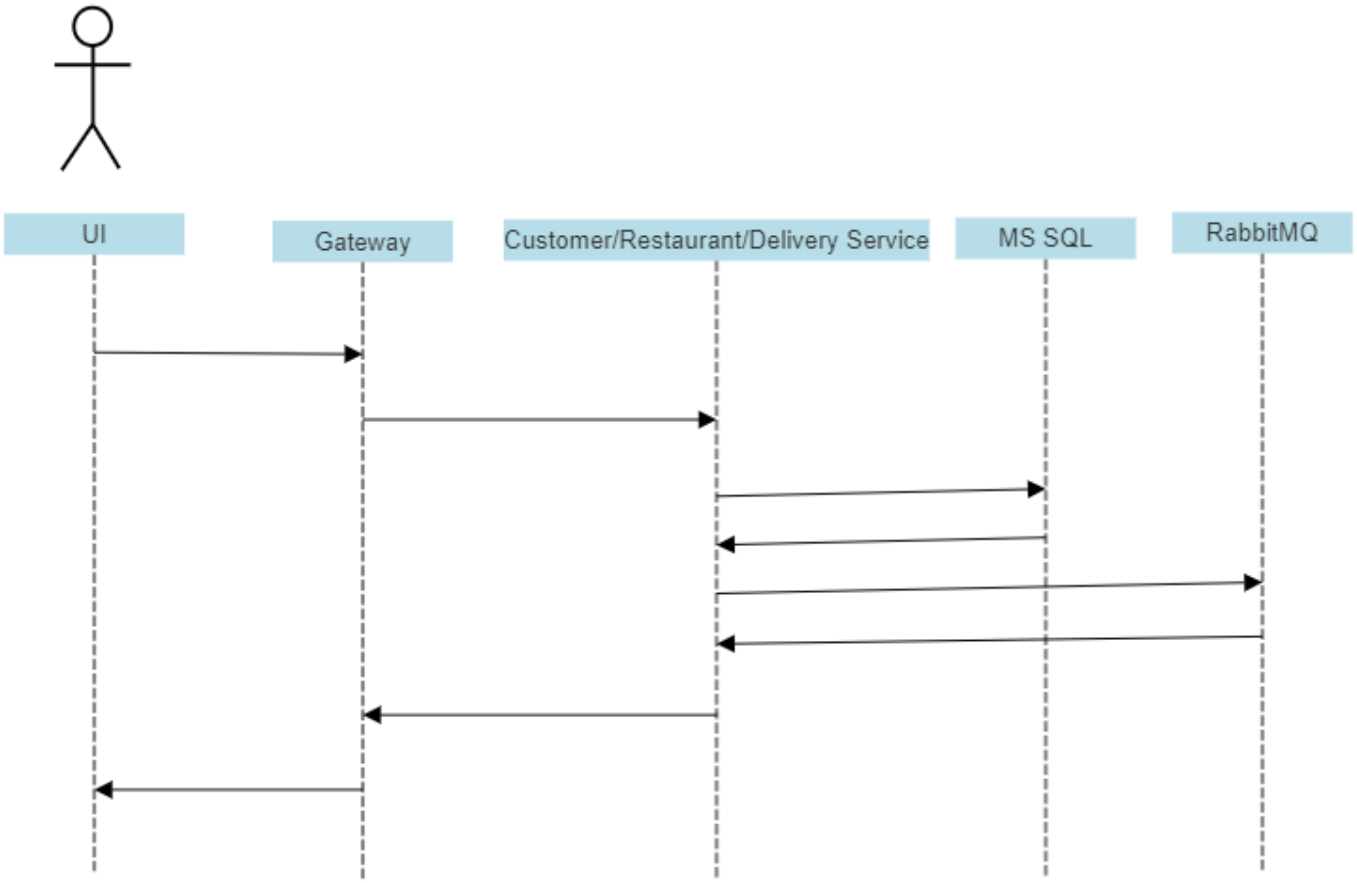

Figure 3: UML workflow sequence for the services

1. All the requests from UI shall hit the spring gateway service.
2. Spring gateway service will handle load balancing and routing of the request to the respective service.
3. After receiving the request, the service shall interact with the database and returns the update to UI via Gateway service.
4. Any updates to or from other services are updated through the RabbitMQ service as a message. Respective services shall take care of listening and handling these messages.

Support for scalability and fault tolerance

- Any number of similar service instances which are started will register themselves to Service Discovery which inherently updates the Spring gateway service and gets load balanced individually.
- System is designed to work normally even if other services are not healthy. As the communication between these services happens through RabbitMQ which is asynchronous, any problem with the other services can be resolved after they come up, and other services workflows shall not be interrupted.

Contributions

Bharath Bharadwaj:

- Responsible for the creation of the restaurant Spring Boot service along with the restaurant UI screen on the react frontend application.
- Responsible for REST-based communication between the react frontend application and the restaurant service.
- Responsible for dockerization of Eureka, API Gateway, RabbitMQ, MSSQL, and restaurant service.
- Responsible for connecting Restaurant service to RabbitMQ and sending events to delivery service.
- Responsible for connecting the React frontend application to RabbitMQ using MQTT.

Nithin Katta Kiranprakash:

- Responsible for the creation of the Customer Spring Boot service along with the Customer UI screen on the react frontend application.
- Have integrated Eureka Service Discovery and Spring Gateway service with other services.
- Responsible for implementing the SQL Script for the creation of schema, tables, and views, dockerization of MSSQL
- Responsible for dockerization of RabbitMQ, and customer service.
- Responsible for connecting Customer service to RabbitMQ and sending events to restaurant service, listening to RabbitMQ updates from delivery service, and updating UI with order data.

Praveen Kamate:

- Responsible for the creation of the delivery Spring Boot service along with the delivery agent's user interface using the react framework.
- Responsible for REST-based communication between the frontend application and the delivery service.
- Responsible for creating the SQL view used by the delivery service.
- Responsible for the dockerization of the delivery service.
- Responsible for connecting the delivery service to the RabbitMQ, receiving messages from the restaurant service, and sending a message to customer service once the order is delivered.

Reflections

key challenges and solutions

- During dockerization of the services, tables, and views need to be created in MSSQL Server for the services to work, but in docker, the commands are run immediately after pulling the base image, hence the container was failing to start since the MSSQL server was not started and it was trying to connect to it. This was overcome by adding a script that checked for the status of the MSSQL server and then executing the script for creating tables and views.
- After the creation of the MSSQL container SQL script is run by docker using Shell scripts which were failing to run throwing LF error caused due to the git default formatting of files this was overcome by deleting the git local workspace, running 'git config –global core.autocrif' and pulling the code again.
- Event-based communication using RabbitMQ is primarily built for server-to-server-based communication and is not integrated well with using the same in a browser. This was overcome by enabling MQTT plugin in RabbitMQ and using the stompjs client in the UI which uses WebSocket-based communication to connect to RabbitMQ.
- During the dockerization of the services, all the backend services depend on the RabbitMQ service to be started, but it was failing since the services were trying to connect to RabbitMQ before it was started. This was handled by adding Health Check to the RabbitMQ container and only starting the dependent services once the RabbitMQ container was completely initialized.

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

- We could have implemented a git pipeline which could have helped in mitigating build failure issues.
- We could have written test automation which would have reduced manual testing effort.
- We could have used the scrum methodology to have better working practices.
- We could have done POCs using different technologies which could have yielded a better solution.

Technology Limitations and Benefits

RabbitMQ:

Limitation – Event based communication using RabbitMQ is not integrated well in the browser. The workaround of using web sockets to connect is tedious. It can become the single point of failure if deployment is done as a single standalone service, instead it can be deployed as a cluster.

Benefits – RabbitMQ works excellently during event-based communication i.e., it does not matter if the service listening to events is healthy or not, once the service comes up the message will be delivered.

Eureka and API Gateway:

Limitation – Requires highly available server to be used, if not all the incoming requests will fail and none of them will be forwarded to the respective service even if they are running.

Benefits – Basic Load balancing, filtering, validating, and routing requests.

MSSQL:

Current deployment leads to a single point of failure which can be mitigated by deploying multiple MSSQL servers and enabling SQL AO (always on) which ensures availability.