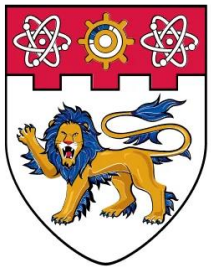


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Wee Kim Wee School of Communication and Information

IN6225: Enterprise Applications Development

2024-2025 (Semester 2)

Individual Assignment Report

Corporate Ride Management System (CRMS)

Submitted By:

[redacted]

19 April 2025

Objectives

Corporate Ride Management System is a web-based application designed for companies to manage employee transportation and shuttle services. It ensures efficient commuting, catering to businesses with regular employee transportation needs.

Main Features

1. **Ride management**
    - Employees can request, cancel rides and view ride history.
    - Drivers can start, complete assigned rides and view ride history.
    - Admins can monitor, cancel rides and view ride history.

2. **Driver Management**
    - Admins can register and view drivers.

3. **User Management**
    - Admins can register and view users.
    - Users can log in with role-based access control (RBAC).

4. **Automated driver assignments to rides**

Target Users

- Office employees with transportation needs → Passenger
- Drivers that receive ride assignments → Driver
- Businesses that provide and manage employee transportation → Admin

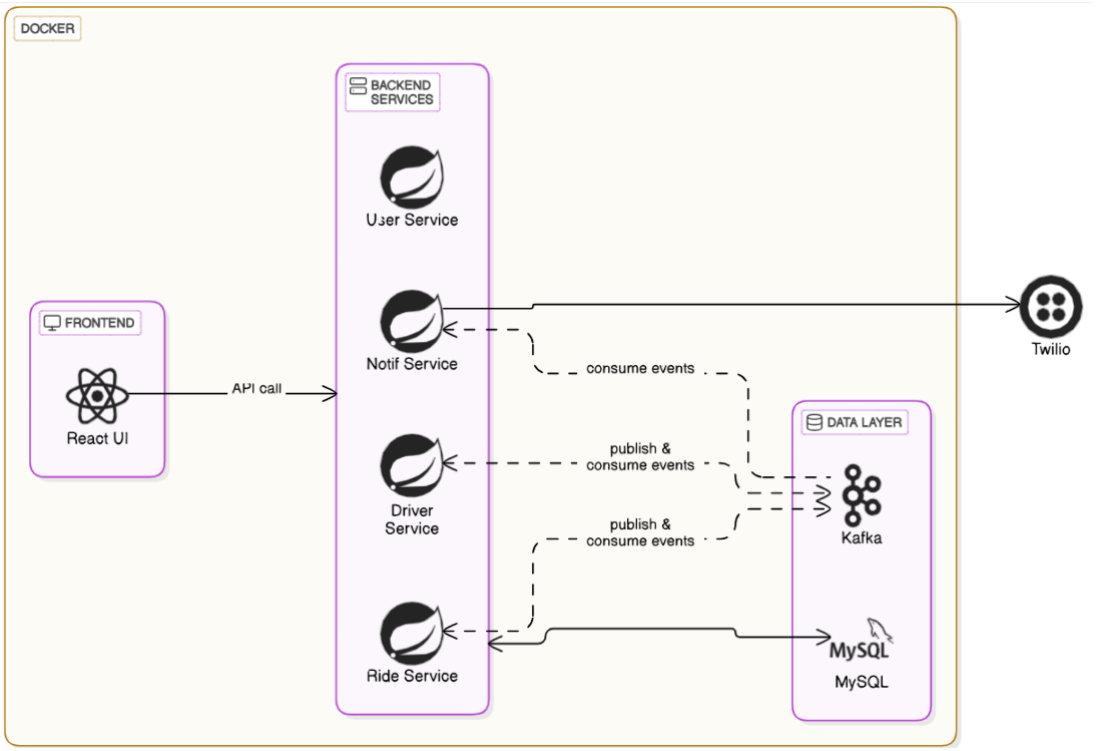
Usage Scenario Examples

- **Inter-Plant Travel Request**: A maintenance supervisor at a large manufacturing corporation, needs to travel from Plant A to Plant D for an urgent inspection.
- **Late Shift Transport**: An employee finishing a late-night shift requests a ride from the facility to the nearby staff dormitory.
- **Managerial Meeting**: A department head requests a ride to attend a scheduled meeting at the administration block.
- **Urgent Equipment Delivery**: A technician uses the system to arrange a driver to transport a part urgently from the warehouse to a repair site.

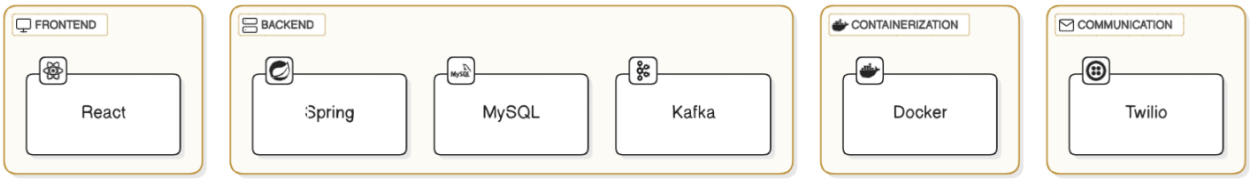
Technical implementation

Architecture

The application uses a microservices-based approach with event-driven communication.



Tech Stack



Libraries Used

React: axios, bootstrap, jwt-decode, react, react-dom, react-router-dom

Spring Boot: web, jpa, validation, security, mysql, kafka, swagger, jjwt, lombok, servlet, twilio

Microservices Overview

1. Ride Service

Handles ride creation & lifecycle updates and publishes events for other services to react to.

Endpoints:

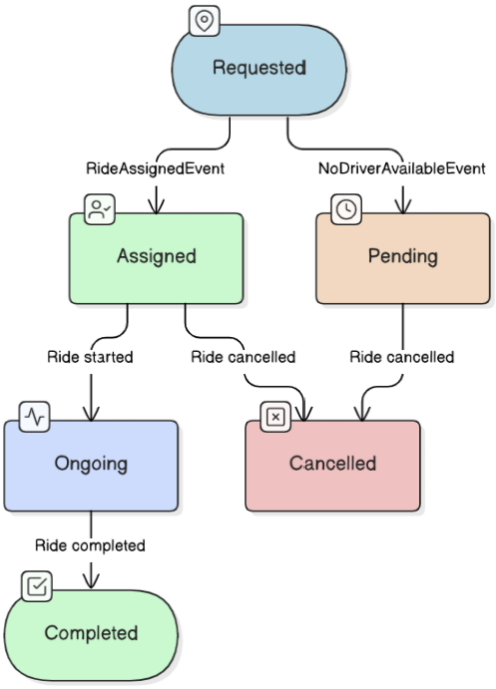
- POST /rides/request → creates a ride (Requested); publishes RideRequestedEvent
- PATCH /rides/{id}/start → marks ride as Ongoing; publishes RideStartedEvent
- PATCH /rides/{id}/complete → marks ride as Completed; publishes RideCompletedEvent
- PATCH /rides/{id}/cancel → marks ride as Cancelled; publishes RideCancelledEvent
- GET /rides → returns all rides (or filters by userId, driverId, status)
- GET /rides/{id} → returns ride by id

Event handlers:

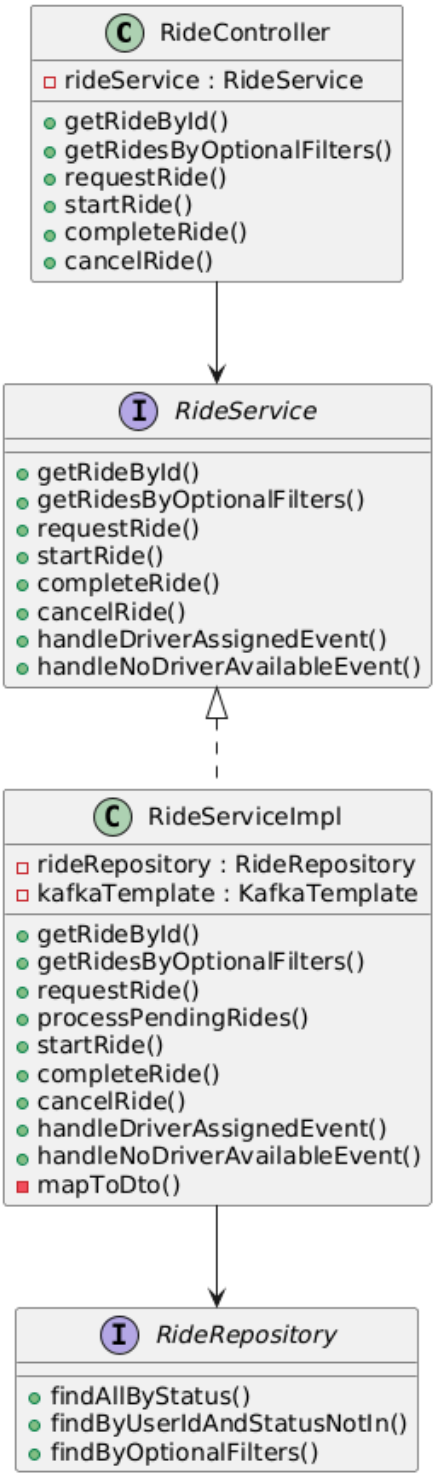
- DriverAssignedEvent → assigns driver and marks ride as assigned
- NoDriverAvailableEvent → marks ride as pending

Ride Data Model

Ride State Flow



Class Diagram



2. Driver Service

Manages drivers and their availability.

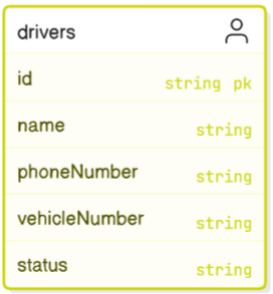
Endpoints:

- POST /drivers/register → creates a driver
- GET /drivers → returns all drivers
- GET /drivers/{id} → returns driver by id

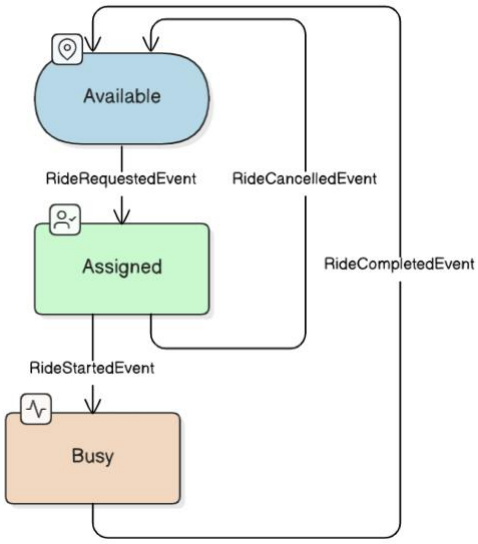
Event handlers:

- RideRequestedEvent → marks first available driver as busy and publishes DriverAssignedEvent or NoDriverAvailableEvent if there are no available drivers
- RideStartedEvent → marks driver as busy
- RideCompletedEvent → marks driver as available
- RideCancelledEvent → marks driver as available or ignored if no driver was already assigned (for pending rides)

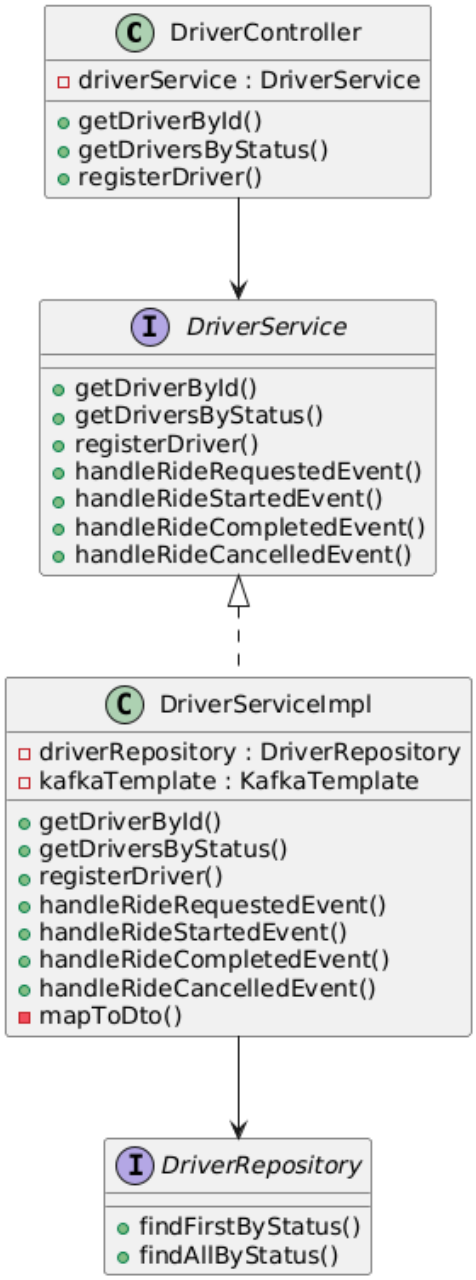
Driver data model



Driver State Flow



Class Diagram



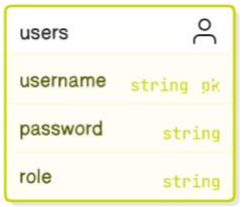
3. **User Service**

   Manages users' registration, authentication.

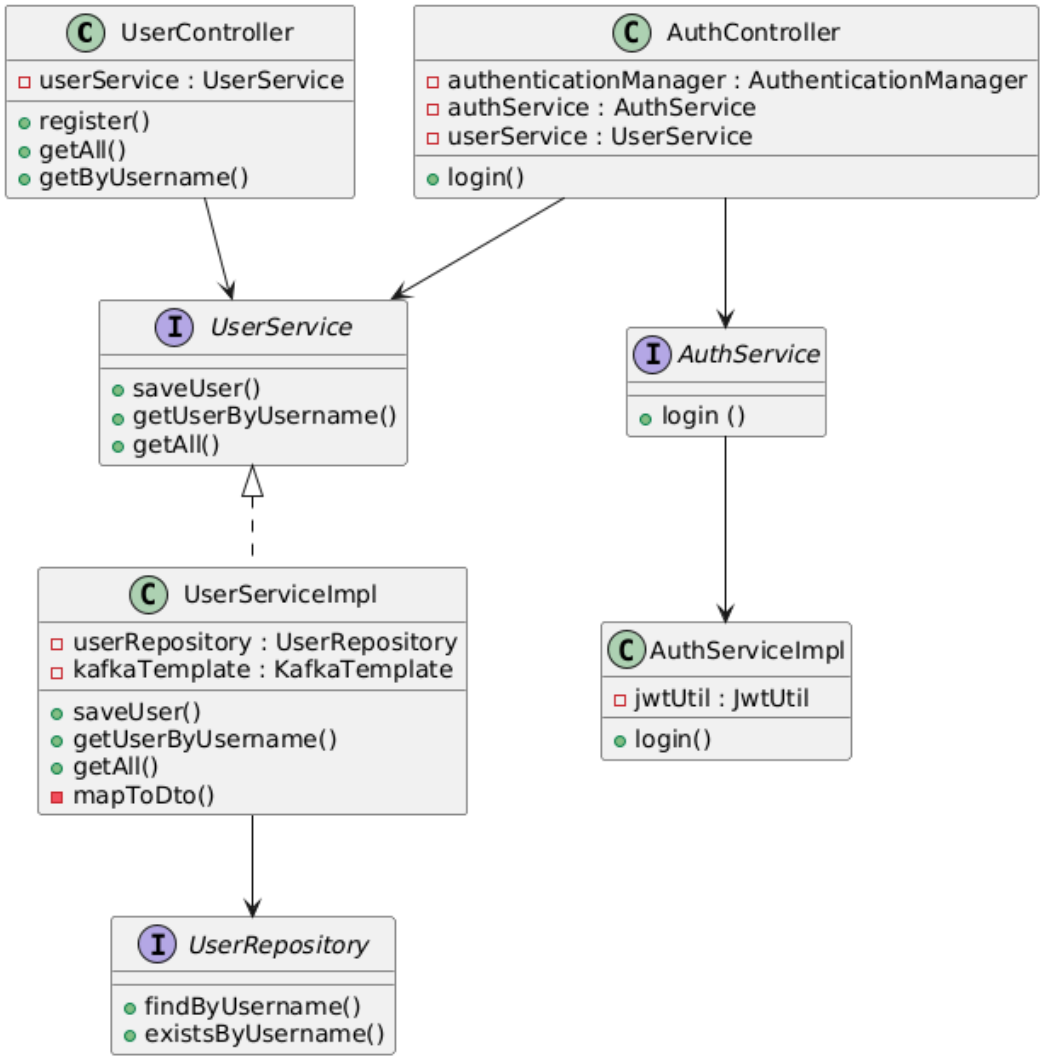
   Endpoints:

   - POST /users/register → creates a user
   - GET /users → returns all users
   - GET /users/{username} → returns user by username
   - POST /auth/login → authenticates and returns JWT token

User Data Model



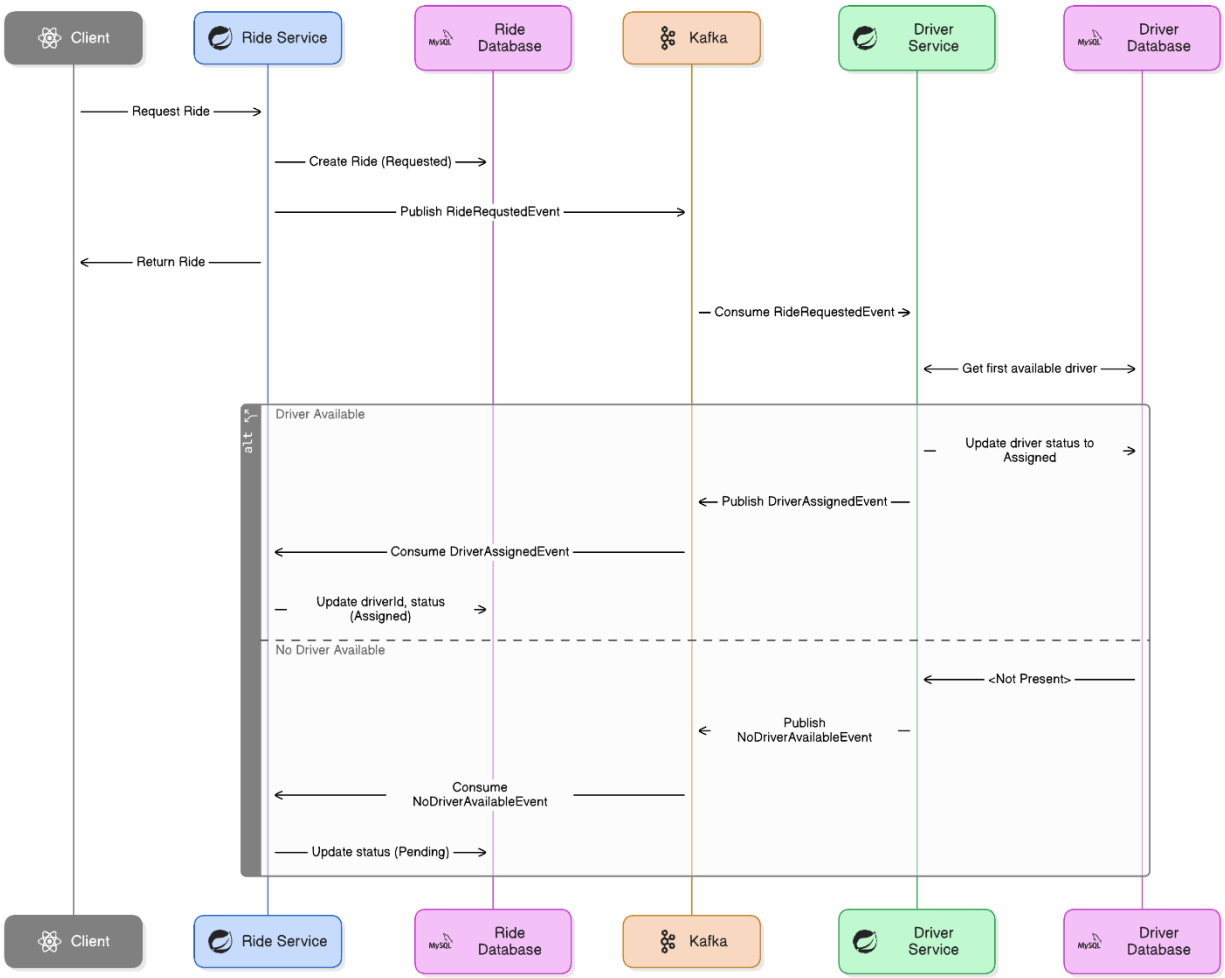
Class Diagram



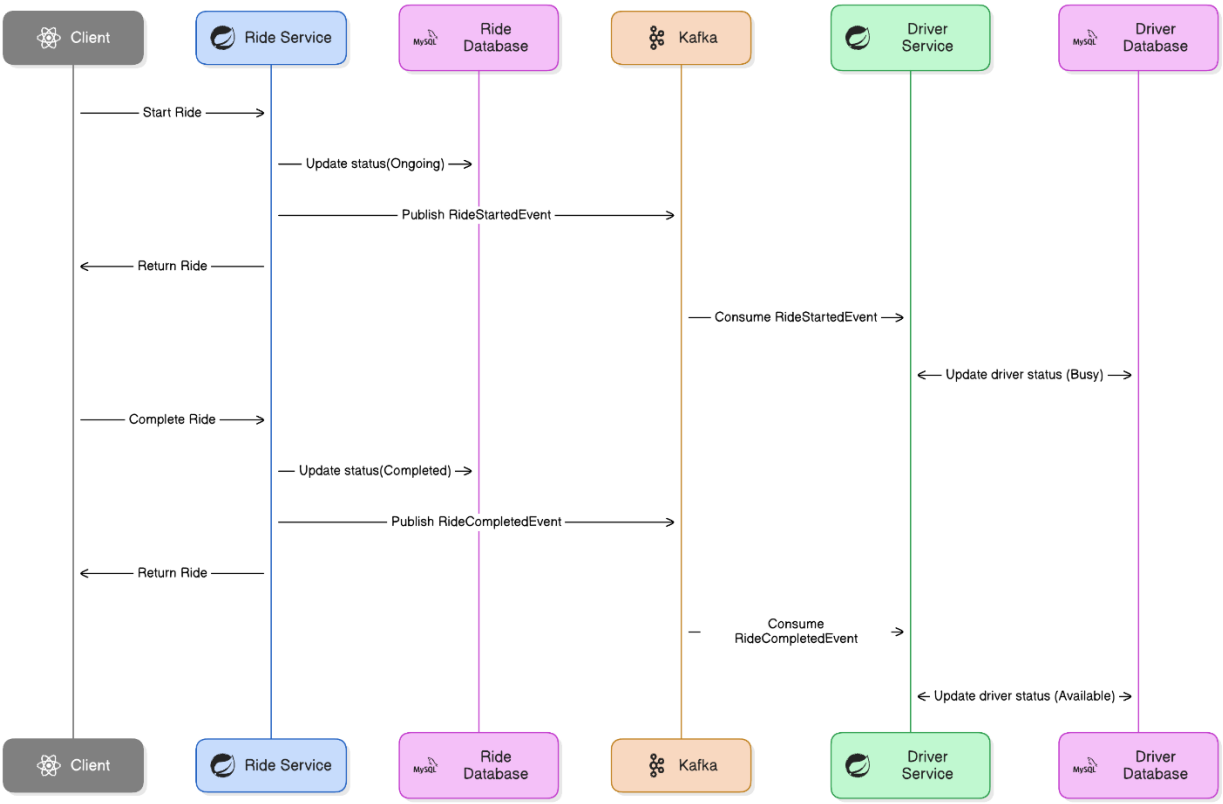
4. **Notification Service**
   Manages notifications for ride status changes; uses Twilio API.

Sequence Flow

Ride Request



Ride Commencement & Completion



Scope for further improvements

- **Notifications for Ride Status**
  Notify users on key ride status changes (e.g., assigned, ongoing, completed).

- **Admin Reporting Dashboard**
  Provide metrics such as total rides requested, completed, cancelled, and driver performance.

- **Driver Mobile App**
  Develop a dedicated mobile application for drivers to manage rides and receive real-time ride requests and notifications.

- **User Profile Management**
  Allow users to update their personal information and reset their passwords.

- **Intelligent Ride Matching**
  Use location-aware algorithms to assign the nearest available driver efficiently.

- **Ride Scheduling Functionality**
  Allow passengers to schedule rides in advance for future dates and times.

- **Reliable Transaction Management for Driver Assignments**
  Ensure consistency and reliability when assigning drivers to rides.

- **Live Vehicle Tracking**
  Enable real-time tracking of vehicles on a map during active rides.

Challenges faced and how they were resolved

- **Security Token Validation Strategy for Microservices**

  Option 1: Centralized authentication, decentralized token validation
  Login happens in the User Service, which issues a JWT. Other microservices validate the JWT locally using the shared secret (HS256).

  Option 2: Centralized authentication and token validation
  Microservices forward the JWT (or just the Authorization header) to the User Service for validation on each request.

  Chosen Approach: Option 1
  Option 1 is chosen to avoid bottlenecks and latency caused by calling the User Service on every request, while also promoting microservice independence by allowing each service to handle token validation locally.

- **CORS Handling**
  To enable communication between the frontend (React with Vite) and the backend (Spring Boot), Cross-Origin Resource Sharing (CORS) must be addressed.

  Option 1: Configure CORS in the Backend

  Create a global CORS configuration class and update SecurityConfig to allow CORS requests.

  Option 2: Frontend Proxy with Vite

  Set up a proxy in vite.config.js to forward API calls to the backend, avoiding CORS issues during development.

  Chosen Approach: Option 1

  Option 1 is preferred for production-ready setups as it provides consistent, centralized CORS control across environments, enhancing security and reducing reliance on frontend-specific workarounds.

Lessons learnt - from a development perspective

- **Minimal Boilerplate**
  Spring Boot's auto-configuration and starters reduced setup time and boilerplate code significantly.

- **Smooth Kafka Integration**
  Setting up Kafka was simple using spring-kafka—just a few properties and annotations like @KafkaListener and KafkaTemplate.

- **Simplified Database Access with JPA**
  Spring Data JPA allowed quick development with built-in CRUD methods and easy custom queries via method names or @Query.

- **Automatic Schema Management**
  Entities automatically mapped to database tables, helping in rapid prototyping without writing DDL scripts.

- **Easy Environment Configuration**
  Centralized config via application.properties made managing Kafka topics, DB settings, and environment-specific values straightforward.

- **Decoupled Services via Kafka**
  Kafka enabled asynchronous, event-driven communication between services, improving resilience and scalability.

References

IN6225: Enterprise Applications Development Course Chapters

VMWare Tanzu, “Spring,” Spring. Available: https://spring.io

“How to run Kafka locally with Docker,” *Confluent*. Available: https://developer.confluent.io/confluent-tutorials/kafka-on-docker/

“About | UI for Apache Kafka,” *Provectus.io*. Available: https://docs.kafka-ui.provectus.io/

“Twilio,” Twilio, 2024. Available: https://twilio.com