Ride-hailing and Transportation Application

Objective:

You will be building Ride-hailing and Transportation Application using React (Frontend), Spring Boot Microservices (Backend), and MySQL (database)

Project Overview:

A ride-hailing and transportation platform web application is a digital platform that connects riders with drivers for on-demand transportation services. It allows users to request rides, track their drivers' location in real-time, and make cashless payments through the application.

Key components and features typically found in Ride-hailing and Transportation Application include:

User Registration: Users need to create an account on the web application by providing their personal information, such as name, contact details, and payment information. Some applications may also require verification steps, such as email or phone number verification, to ensure the authenticity of the user.

Ride Request: Once registered, users can request a ride by entering their pickup location and destination within the application. The application then matches the user with an available driver in the vicinity.

Driver Matching: The ride-hailing platform uses algorithms to match riders with the nearest available drivers. Factors like distance, driver ratings, and vehicle type may be considered in the matching process.

Real-Time Driver Tracking: After a ride request is accepted, users can track the assigned driver's location in real-time on a map within the application. This feature provides transparency and helps users estimate the driver's arrival time.

Driver Communication: The application often provides communication channels, such as in-app messaging or calling, to facilitate communication between riders and drivers. This allows users to communicate details like pickup instructions or any specific requirements.

Fare Calculation: The application uses a fare calculation algorithm based on factors like distance traveled, travel time, and surge pricing (if applicable) to determine the fare for the ride. Users can usually see the estimated fare before confirming the ride request.

Cashless Payments: The ride-hailing platform typically offers cashless payment options, allowing users to link their credit or debit cards, or use digital

payment methods like mobile wallets. The fare is automatically deducted from the user's payment method at the end of the ride.

Ratings and Reviews: After completing a ride, both the rider and driver can provide ratings and reviews for each other. This feedback system helps maintain service quality and builds trust within the community.

Safety Features: Ride-hailing platforms often include safety features such as driver background checks, vehicle information verification, and an emergency assistance button. These features aim to enhance user safety and provide peace of mind.

Promotions and Rewards: The application may offer promotional codes, discounts, or loyalty programs to incentivize riders and drivers. These features help attract and retain users on the platform.

Accessibility and Localization: Ride-hailing applications are typically designed to be accessible on various devices, including smartphones, tablets, and desktop computers. Localization features ensure that the application supports multiple languages, currencies, and regional preferences to cater to a diverse user base.

Administrative Dashboard: The platform may include an administrative dashboard for the service provider or company. This dashboard allows administrators to manage drivers, monitor performance, track financials, and generate reports.

Integration with Mapping Services: The ride-hailing platform often integrates with mapping services like Google Maps or other navigation systems to provide accurate pickup and drop-off locations, optimize routes, and calculate distances and travel times.

DAY 1: Creating the login and registration forms using React

Phase 1: Create a UML use case diagram to represent the different actors and their interactions with the system. Use arrows to show the relationships and dependencies between the actors and use cases for the complete case study mentioned above.

- Set up the environment according to the application needs.
- Set up a new React project using Create React App or your preferred method.
- Create basic (UI) forms for registering and logging in a user using react. Give specific endpoints for the two forms.

- After registration, make the login page functional, i.e.it should redirect the user to the home page (blank as of now). In case of error in any of the credentials, the appropriate error must be displayed.
- In the process, make sure that the user is added to the database and the form reloads on submission and clear errors upon successful registration/login.

DAY 2: Getting user data into Global state and creating app components

- Use Redux to make user data available on global state. This will pass the props from parent to grandchild components without having to pass the props through all child components.
- Design page header component and side panel
- Use React Router for navigation.
- Create navigation section of the header page which includes icons and log-out functionality

DAY 3: Create and design homepage component

- The homepage of any website serves as the default page of that website. That is reason enough for the homepage to be really expressive as well as creative.
- Include a footer section with links to important pages, such as terms and conditions, privacy policy, FAQ, and contact information.
- Include social media icons or links to encourage users to follow your website on various social platforms.

DAY 4: Design other components

- Create and design all other necessary components.
- To attain reusability, pass parameters (referred to as props in React.js) to your functional component.

DAY 5: Adding more functionality to the application

- You can also use React Context API to share data values between components without having to pass a prop through every level of the app tree.
- Add more functionality to the application.

DAY 6: Design the database schema

- Determine the entities and relationships required for the application.

- Create the necessary tables and define the relationships between them.

DAY 7: Set up the backend with Spring Boot

- Create a new Spring Boot projects using your preferred IDE.
- Set up the project dependencies, including Spring Web, Spring Data JPA, and any additional dependencies required for your database (e.g., MySQL).
- Configure the database connection in the application.properties file.
- Create entity classes representing your database tables, with appropriate annotations for mapping to the database.

DAY 8: Spring Boot CRUD operation implementation

- Implement repository interfaces using Spring Data JPA for database operations for all the applications.
- Create RESTful APIs using Spring Web to handle CRUD operations for different entities.
- Create APIs for all other functionality which is required for the application.

DAY 9: Implement Spring security

- Add authentication and authorization mechanisms to secure your APIs,
- Make use of JSON Web Tokens (JWT) or OAuth 2.0.

DAY 10: Implement security measures:

- Implement user registration, login, and session management functionalities.
- Make sure you build a secure application.

DAY 11: Build microservices architecture

- Identify different microservices within your application.
- Create separate modules or projects for each microservice.
- Implement business logic and functionality within each microservice.
- Set up database connections for each microservice using Spring Data JPA or your preferred method.
- Implement RESTful APIs for each microservice, focusing on specific functionality

DAY 12: Setup Service Discovery and Communication

- Choose a service discovery mechanism like Netflix Eureka
- Configure service registration and discovery within each microservice.
- Implement communication between microservices using RESTful APIs or messaging queues.

DAY 13: Implement API gateway with Spring Cloud Gateway

- Create a new Spring Boot project for your API Gateway.
- Include the Spring Cloud Gateway dependency.
- Configure the API routes, load balancing, and routing rules in the API Gateway project.

DAY 14: Set up messaging with RabbitMQ

- Configure RabbitMQ connection details in each microservice project.
- Define message queues, exchanges, and bindings for the communication between microservices.
- Implement event producers and consumers within microservices to publish and consume messages.

Day 15: Integrate frontend with microservices via API Gateway

- Connect the frontend components with the API Gateway.
- Implement API calls to interact with the API Gateway using libraries like Axios.
- Make HTTP requests to the API Gateway routes, which will route the requests to the appropriate microservices.
- Handle responses and update the UI accordingly.

DAY 16: Implement additional features

- Design and implement features like search functionality, filters, sorting, and pagination for product listings.
- Implement features like user reviews, ratings, and order tracking.
- Integrate Stripe payment in your React application.
- Make application more creative and attractive by adding more features.
- Give a new name to your application and make a new logo for it

DAY 17: Write test cases

- Identify the critical paths and functionalities in your application.

- Write unit tests for each microservice to verify the behaviour of individual components.
- Use testing frameworks like JUnit to write the test cases.
- Mock external dependencies using tools like Mockito to isolate the components under test.

DAY 18: Test and debug

- Consider writing end-to-end tests to simulate user interactions and verify the overall system behaviour.
- Execute the written test cases and verify that they pass successfully.
- Debug any failures or unexpected behaviours and fix the issues in the code.
- Conduct integration testing to verify the communication between microservices and the API Gateway.
- Debug and fix any issues that arise during testing.

Day 19: Dockerizing Spring Boot (Microservices) App and deployment

- Build Docker images for each of your microservices and the API Gateway.
- Create a Dockerfile in each microservice and API Gateway project to define the image build process.
- Include all necessary dependencies and configurations in the Docker images.
- Push the Docker images for your microservices and API Gateway to the container registry (Docker hub).
- Log in to the AWS Management Console and navigate to Amazon ECS.
- Create a new ECS cluster that will host your Docker containers.

Day 20: Deploy the Microservices, API Gateway, and RabbitMQ

- Create an ECS task definition and ECS service for each microservice and the API Gateway.
- Setup load balancing by creating an Application Load Balancer (ALB) in AWS
- Configure the ALB to distribute incoming traffic to your microservices and API Gateway.
- Set up listeners and target groups to route requests to the appropriate ECS services.
- Create security groups to control inbound and outbound traffic for your ECS instances.

- Use the AWS Management Console or the AWS Command Line Interface (CLI) to deploy your ECS services.
- Create ECS service tasks based on the task definitions you created earlier.
- Monitor the deployment process and ensure that the tasks are running successfully.

Congrats!!! You have successfully built and deployed Ride-hailing and Transportation Application.