ANOUSHKA DASH (B080)
PRANAV DHAWALE (B084)
ARYAN CHAUHAN (B087)
B. Tech Computer Engineering Div. B

Vehicle Rental System

Phase 1: Synopsis (Microservices and Architecture)

Introduction to the Project

The **Vehicle Rental System** is a robust and scalable platform designed using a microservices architecture and built with Spring Boot. This system aims to showcase the core concepts of microservices, such as effective service communication, fault tolerance, and streamlined deployment. By utilizing cloud-native technologies and containerization, the platform ensures a smooth and flexible experience for users renting vehicles. Each microservice is dedicated to specific operations, including rental management, inventory tracking, payment processing, and invoicing. This architecture guarantees high scalability, resilience, and ease of maintenance, enabling the system to efficiently handle increasing demands and support large-scale operations.

Problem Statement

As businesses expand, traditional monolithic applications often struggle with scalability and maintenance. The interdependence of components in such systems makes it difficult to manage and update services independently, leading to performance bottlenecks and complex deployment processes. To overcome these challenges, the vehicle rental system adopts a scalable and maintainable microservices architecture. This approach allows the system to perform various functions more efficiently while offering enhanced flexibility, resilience, and scalability.

The challenge is to build a scalable and resilient vehicle rental system that can effectively handle multiple microservices for various functions, such as car rental, payment processing, inventory management, and invoice generation. The system must ensure smooth communication between services while maintaining fault tolerance, allowing it to remain operational even if certain services go down or encounter issues. Additionally, it requires a centralized configuration management system to manage different settings for both development and production environments. The system must also leverage modern technologies to manage service discovery, authentication, logging, and monitoring, ensuring seamless operations, high performance, and scalability.

Objectives

The objectives of the vehicle rental system are as follows:

- **Microservices Architecture**: Implement a scalable and modular microservices-based system to handle key functionalities such as car rental, payment processing, inventory management, and invoicing.
- **Centralized Configuration**: Utilize a configuration management system to maintain consistent settings across all services and environments, ensuring ease of management and updates.
- **API Gateway**: Establish a unified gateway to manage and route requests to the appropriate microservices, simplifying client-side interaction and improving security.
- **Database Management**: Implement different database systems based on each microservice's requirements, ensuring optimized performance and scalability.
- **Fault Tolerance and Resilience**: Ensure system reliability and uptime by incorporating mechanisms to handle service failures and maintain operation despite disruptions.
- **Security**: Implement authentication and authorization solutions to protect sensitive user data and ensure secure access to various services within the system.
- **Monitoring and Performance Tracking**: Integrate monitoring tools to track the health and performance of microservices, enabling proactive issue resolution and performance optimization.

Scope of the Project

Functional Scope

- Administrators should be able to update vehicle information, including pricing and other relevant details.
- Administrators should have the ability to view and manage all active and past leases.
- Vehicles should be able to be sent for maintenance when required.
- The system should verify the current condition of vehicles before initiating a rental transaction.
- Users should be able to rent the vehicle of their choice.
- Users should have the ability to filter and search for vehicles based on their desired features.
- Users should be able to view both current and previous rental transactions.
- Users should be able to view both current and past invoices.
- Users should have the flexibility to pay using any available method.

Non-Functional Scope

- Services should be optimized for low latency.
- The system should be highly available and consistent, allowing users to rent vehicles instantly and receive immediate transaction confirmations.
- The system must be scalable to accommodate multiple car rentals across major cities.

Technical Scope

- Managing centralized configuration files for all microservices with Spring Cloud Config, handling different environments like development and production.
- Routing requests with Spring Cloud Gateway, while Eureka Service Registry dynamically registers microservices for seamless communication.
- Handling event-driven communication with Kafka, ensuring smooth data flow between services like lease transactions and invoice generation.
- Ensuring consistent and portable deployment of microservices, databases, and Kafka using Docker containers.
- Storing relational and non-relational data with MySQL and MongoDB, respectively, based on service requirements.
- Reducing boilerplate code with Lombok, which automatically generates constructors, getters, and setters at compile time.
- Simplifying REST communication with OpenFeign, while ensuring reliability with the Retry pattern for fault tolerance using Resilience4j.
- Facilitating efficient database interaction and object-relational mapping (ORM) using Spring Data and JPA.
- Monitoring resource usage and system health using Prometheus and Grafana, providing real-time performance insights.

Technologies Used (Technology Stack)

- **Config Server** is implemented using Spring Cloud Config to manage YAML configuration files for microservices over a port. It allows fetching different YAML files for Production and Development environments.
- **API Gateway** is built using Spring Cloud Gateway, while **Eureka Service Registry** handles service discovery. These tools bring all microservices together under one roof via a single port, registering their dynamic ports as they are created or changed.
- **Kafka** is employed for asynchronous communication, facilitating scenarios like delayed data insertions. For instance, after a lease transaction, Kafka is used to send an event to the invoice service, which processes the event and adds a record.
- **Docker** is used to containerize the databases and Kafka, ensuring consistent deployment and environment portability.
- **MySQL** and **MongoDB** are utilized for data storage, with MySQL handling relational data and MongoDB catering to non-relational data.

- **Lombok** is leveraged to generate constructors, getters, and setters at compile-time, reducing boilerplate code.
- **OpenFeign** simplifies REST calls between microservices, while **Resilience4j** is implemented to manage the Retry pattern for handling intermittent failures.
- **Spring Data** is employed to implement JPA (Java Persistence API) for Object-Relational Mapping, making data access and interaction more efficient.
- **Prometheus** and **Grafana** are used for monitoring and visualizing the resource usage and performance metrics of each microservice, providing real-time insights into system health.

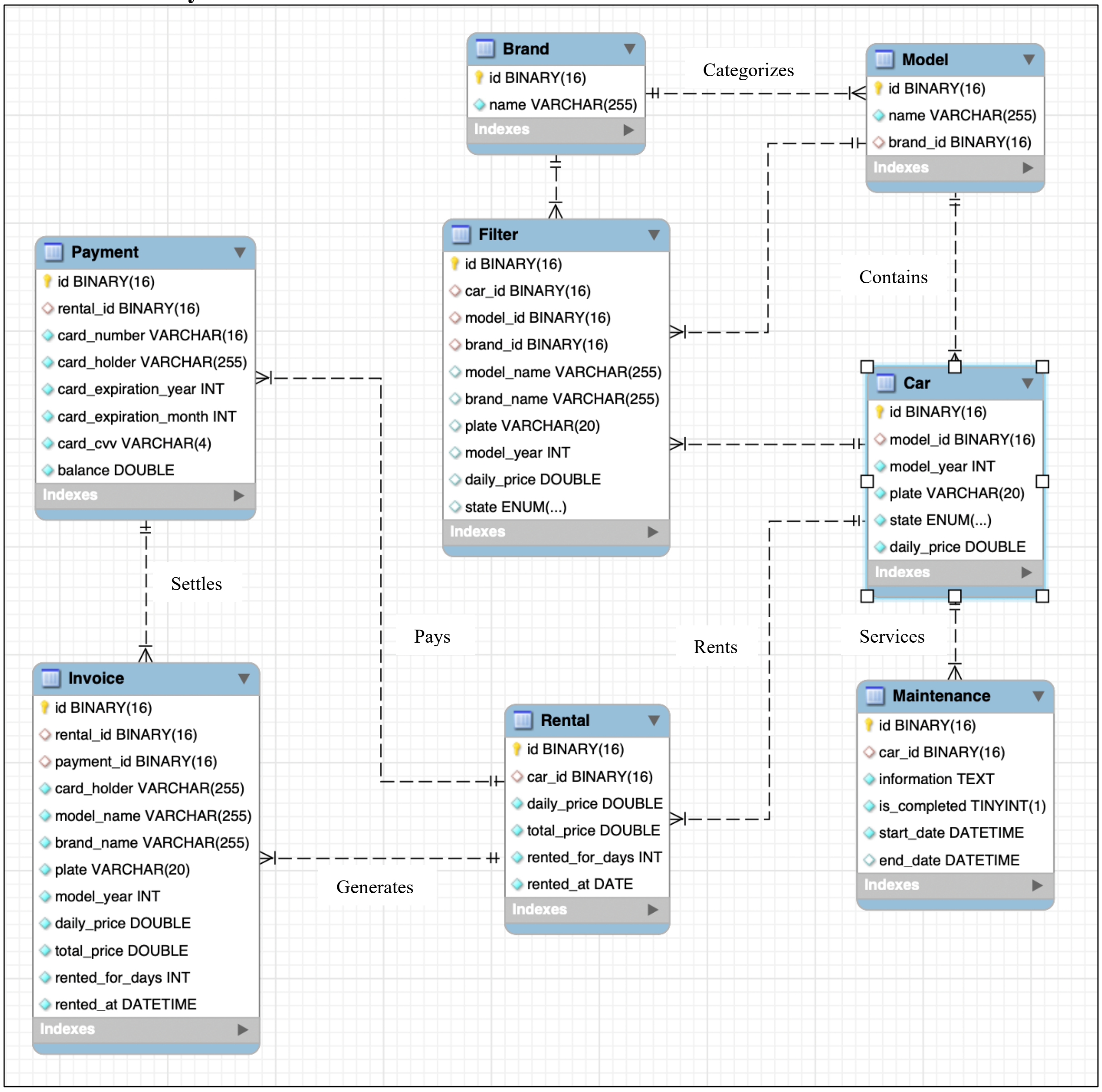
Expected Outcome

The expected outcomes are as follows:

- **Seamless Rental Experience:** Users will have a smooth and efficient process for renting vehicles, with easy access to available vehicles, reservation, and payment options.
- **Scalable Infrastructure:** The system will scale to accommodate growing numbers of users, vehicles, and transactions, ensuring efficient performance as the business expands.
- **Reliable Operations:** The system will remain operational even during failures, ensuring continuous availability of services like car rental, payment processing, and inventory management.
- **Centralized Management:** Administrators will be able to manage vehicles, leases, and prices in a centralized system, streamlining daily operations.
- **Real-time Data Processing:** The system will handle real-time rental transactions, updating inventory and generating invoices promptly.
- **Enhanced Security:** The system will protect user data and transaction details, ensuring secure access and confidentiality.
- **Proactive Monitoring and Issue Resolution:** Continuous monitoring and performance tracking will allow for quick identification of issues, ensuring smooth system operation at all times.

ERD (Entity-Relationship Diagram)

Vehicle Rental System



Brand

- Represents a car brand, with an ***id*** and ***name***.
- A one-to-many relationship exists between Brand and Model; one brand can have many models.

Model

- Represents a specific model of a car with an ***id*** and ***name***.
- Linked to Brand through a foreign key (***brand_id***).
- Has a one-to-many relationship with Car; one model can have many cars.

Car

- Represents an individual car with attributes such as ***id***, ***model_year***, ***plate***, ***state***, and ***daily_price***.
- Connected to Model via the foreign key (***model_id***).
- One-to-many relationships with both Rental and Maintenance; one car can have many rentals and maintenance records.

Rental

- Represents a rental transaction, including ***id***, ***daily_price***, ***total_price***, ***rented_for_days***, and ***rented_at***.
- Linked to Car through a foreign key (***car_id***).
- Has a one-to-one relationship with Invoice, where each rental generates one invoice.

Maintenance

- Represents maintenance records for cars, with attributes like ***id***, ***information***, ***start_date***, ***end_date***, and ***is_completed***.
- Linked to Car via the foreign key (***car_id***).
- A one-to-many relationship exists; one car can have multiple maintenance records.

Payment

- Contains payment details with attributes such as ***id***, ***card_number***, ***card_holder***, ***card_expiration_year***, and ***balance***.
- Connected to Invoice through a one-to-one relationship, where each payment corresponds to one invoice.

Invoice

- Represents an invoice generated for a rental transaction, with attributes like ***id***, ***card_holder***, ***model_name***, ***plate***, ***total_price***, and ***rented_at***.
- Linked to Rental and Payment through one-to-one relationships; each invoice is tied to one rental and one payment.

Filter

- Serves as a search filter, holding attributes like **carId**, ***modelId***, ***brandId***, ***modelName***, ***brandName***, ***plate***, and ***modelYear***. Linked to brand, model and car.

Normalisation

1st Normal Form (1NF)

- Each table has a **primary key**.
- Each column contains only **atomic values** (no multi-valued attributes).
- All records are **unique** (no duplicate rows).

Entities After 1NF: All of the tables already contain atomic attributes (such as id, name, model_year, etc.), and each table has a **primary key**. Thus, they are already in **1NF**.

2nd Normal Form (2NF)

- The entity should be in **1NF**.
- Every non-key attribute must be **fully dependent** on the **primary key** (no partial dependencies).

Changes

- **Car** and **Model**: The **daily price** and **state** are attributes of a car but are being stored at the model level as well. We will store daily_price and state in the **Car** table, as they apply to individual cars rather than the entire model.
- **Rental**: The **total_price** and **rented_for_days** are fully dependent on the rental transaction (not on the individual car), so no changes are required for 2NF.

Entities in 2NF: The tables are already normalized to **2NF**, as all non-key attributes are fully dependent on their respective primary keys.

3rd Normal Form (3NF)

- The entity should be in **2NF**.
- No transitive dependencies (non-key attributes must not depend on other non-key attributes).

Changes

- **Invoice**: The **model_name**, **brand_name**, and **plate** attributes are derived from the **Car** and **Model** entities. These should not be repeated in the **Invoice** table; instead, we should reference the **Car** and **Model** entities. This removes redundancy and ensures the **Invoice** table stores only the transaction-related data.
- **Rental**: The **daily_price** should be stored in the **Car** table (as it's an attribute of the car and not the rental transaction). We'll remove the **daily_price** from the **Rental** table.

Final Entities in 3NF

1. **Brand**
   id (Primary Key)
   name

2. **Model**
   id (Primary Key)
   name
   brand_id (Foreign Key to Brand)

3. **Car**
   id (Primary Key)
   model_id (Foreign Key to Model)
   model_year
   plate
   state (Enum: 'AVAILABLE', 'RENTED', 'UNDER_MAINTENANCE')
   daily_price

4. **Rental**
   id (Primary Key)
   car_id (Foreign Key to Car)
   rented_for_days
   rented_at (Date)
   total_price

5. **Maintenance**
   id (Primary Key)
   car_id (Foreign Key to Car)
   information
   is_completed
   start_date (DateTime)
   end_date (DateTime)

6. **Payment**
   id (Primary Key)
   card_number
   card_holder

card_expiration_year
card_expiration_month
card_cvv
balance

7. **Invoice**
id (Primary Key)
rental_id (Foreign Key to Rental)
payment_id (Foreign Key to Payment)
card_holder
total_price
rented_for_days
rented_at (DateTime)

8. **Filter**
id (Primary Key)
car_id (Foreign Key to Car)
model_id (Foreign Key to Model)
brand_id (Foreign Key to Brand)
model_name
brand_name
plate
model_year

Resulting Changes after 3NF

- **Invoice** now references **Car** and **Model** through foreign keys, and no redundant attributes like model_name, brand_name, or plate are stored.
- **Rental** no longer stores daily_price (since it's stored in **Car**).
- **Filter** now stores references to **Car**, **Model**, and **Brand** as foreign keys, keeping it normalized.