SWENG 837

BonVoyage - Real-time Ride-sharing Service

Software System Design Document

Chris Pisani

Summer Semester 2024

Problem Statement and Requirements

Business Requirements:

I. Problem Statement: The system's goal is to efficiently provide transportation to people in need by matching riders with a driver in real-time. The system needs to be able to dynamically price trips depending on the density of requests as well as optimally provide routes depending on weather and traffic conditions.
II. Main Functionalities:
    A. Real-Time Matching: A system to match passengers with available drivers in the area.
    B. Dynamic Pricing: Adjusted pricing based on the factors supply, demand, and time of day.
    C. Route Optimization: Effective routing that minimizes traffic and weather conditional hazards.
    D. Rating System: Allow passengers and drivers to to review each other for a rating out of five stars.
    E. Secure Payment: Payment processing for users supporting major credit cards, with driver direct bank deposits.
III. Users:
    A. Passengers:
        1. Quick and reliable matching with drivers.
        2. A clear and transparent view of what they would be charged for a ride.
        3. Easy booking and payment offered.
    B. Drivers:
        1. Quick and reliable matching with passengers.
        2. Ability to work whenever they need.
        3. Optimized and efficient routes.
        4. Fair and speedy payment for their work.
IV. Business Goals:

A. Satisfaction and retainability: drivers and passengers that use the system are likely to continue using it after interacting with it
B. Revenue: create a dynamic pricing model that charges passengers and pays drivers fairly, while still capitalizing on profit.
C. Market Share: Acquire a large portion of the market by providing excellent service.

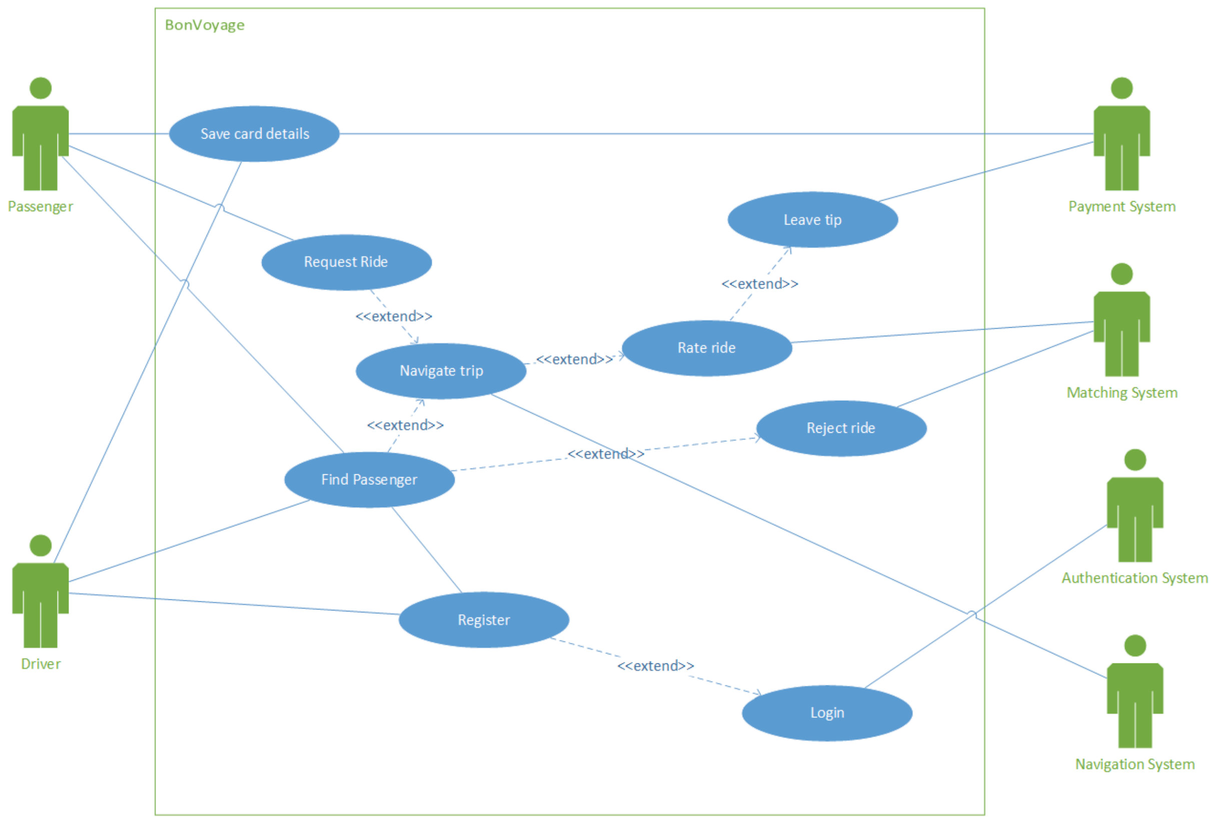
Non-functional requirements:

I. Performance Requirements:
   A. Scalability: the system shall handle one hundred thousand passengers and drivers simultaneously with the ability to dynamically scale resources to support one million concurrent users if necessary.
   B. Matching: guaranteed matching response time during peak hours at thirty seconds, at a maximum of five minutes of wait time during off peak hours.
   C. Latency: UI interactions will be no slower in populating on the screen than five hundred milliseconds, exceptions exceeding this time limit will present a working dialog.
   D. Optimization: route optimization takes no longer than three seconds to complete
   E. Throughput: able to support five thousand transactions processing throughout the system at any given time.
II. Security Requirements:
   A. Authentication: user's are able to authenticate through a username, password, and two-factor authentication.
   B. Authorization: passengers, drivers, and administrators all have roles placed on them that enable different permissions.
   C. Encryption: data transmission that is secure and safe for all information transfer using asymmetric encryption.
   D. Privacy: User data is handled responsibly in accordance with the Federal Trade Commission (FTC).

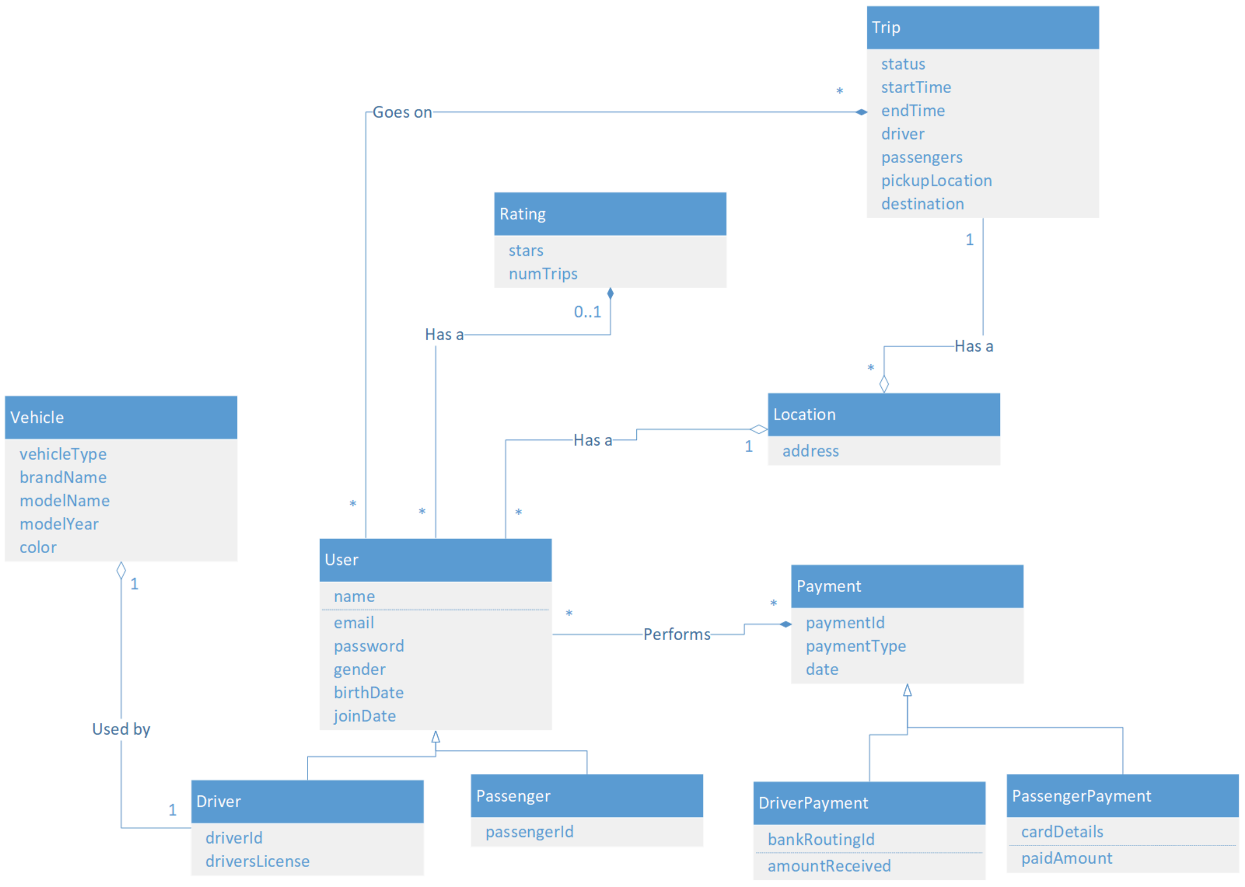
III. Maintainability Requirements:
   A. Modularity: the codebase should be partitioned into independent subsections that build up to an organized and maintainable structure that is easy to update using Git versioning control.
   B. Documentation: all classes and APIs shall be properly documented with code linting and READMEs.
   C. Testing: Proper unit and integration testing across all developed code.

IV. Additional Requirements:
   A. Availability: system should be usable and operational with 99.9% uptime.
   B. Recovery: data backups are frequently taken and stored with the ability to quickly restore lost data in the event of system failure or data loss.

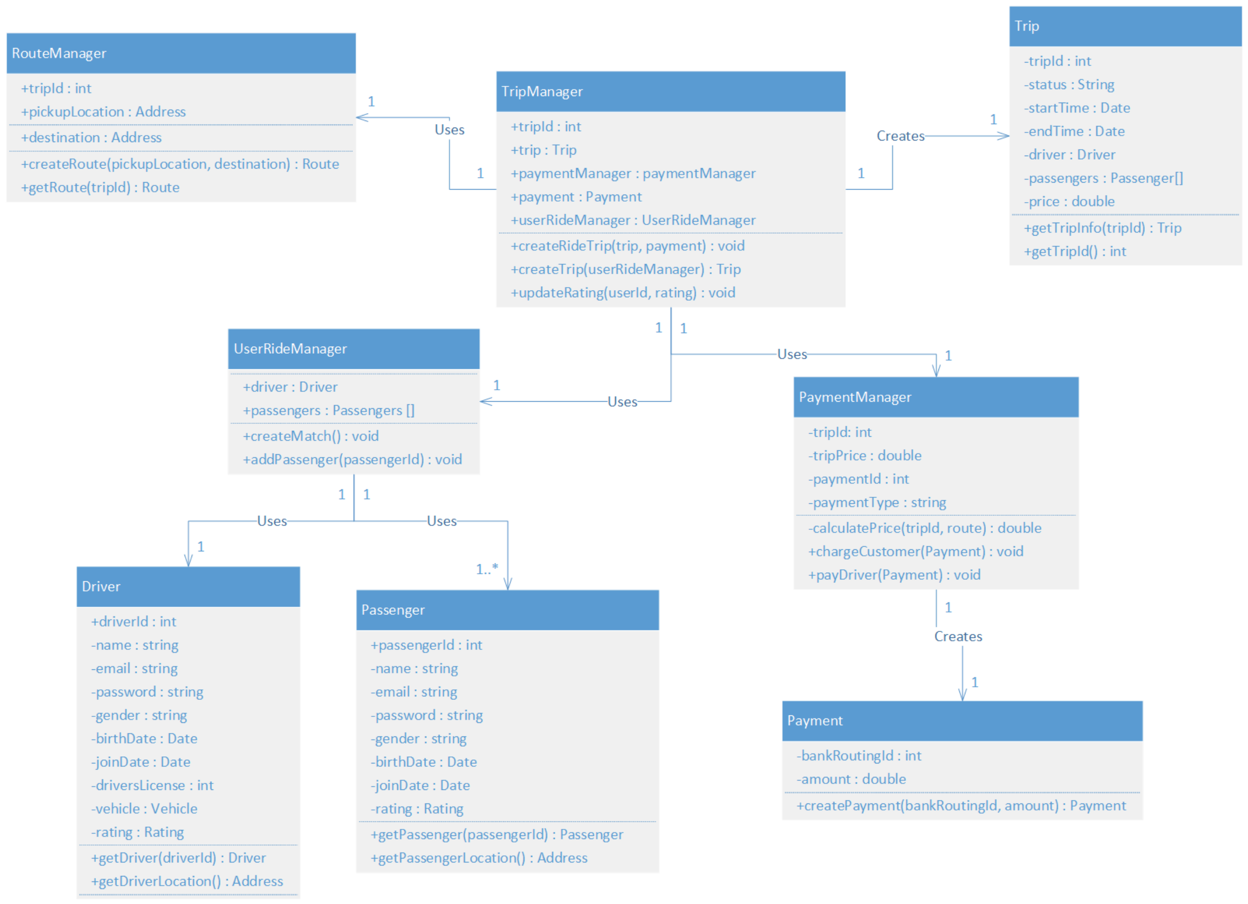
Use Case Diagram



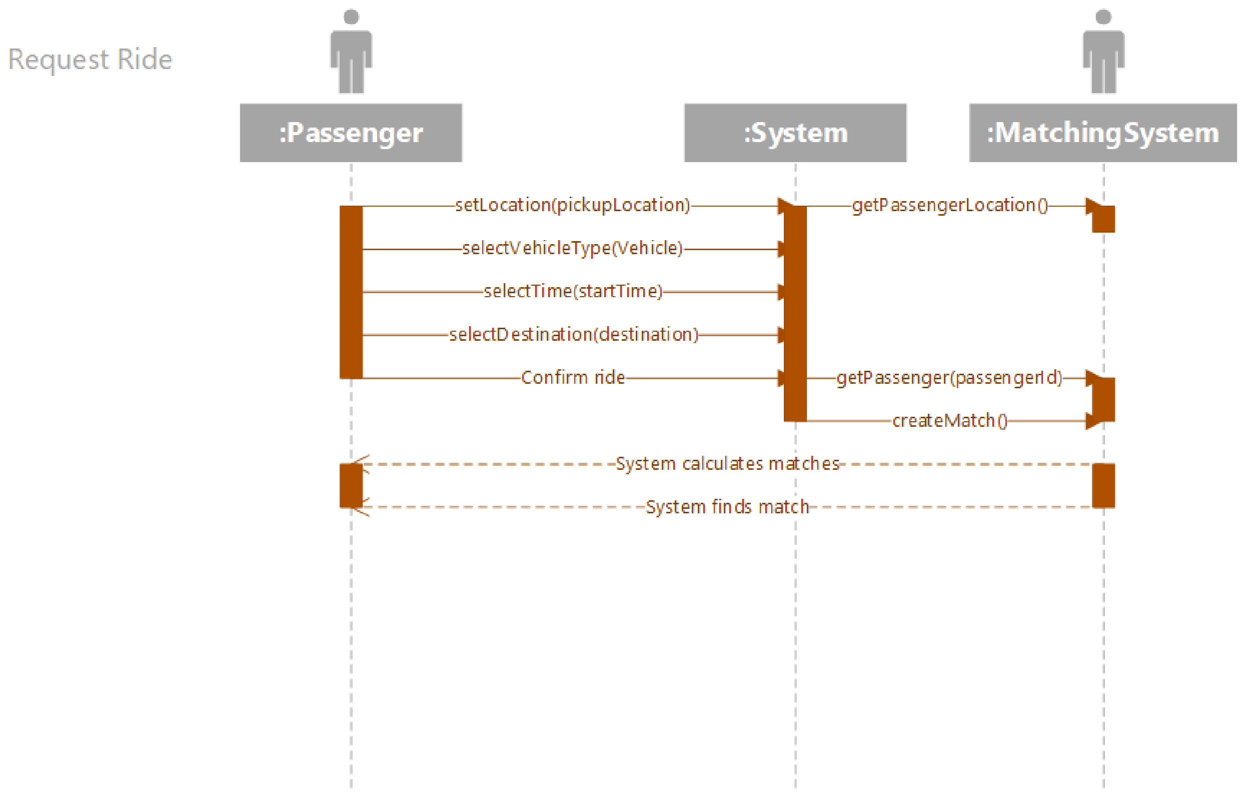
Domain Model



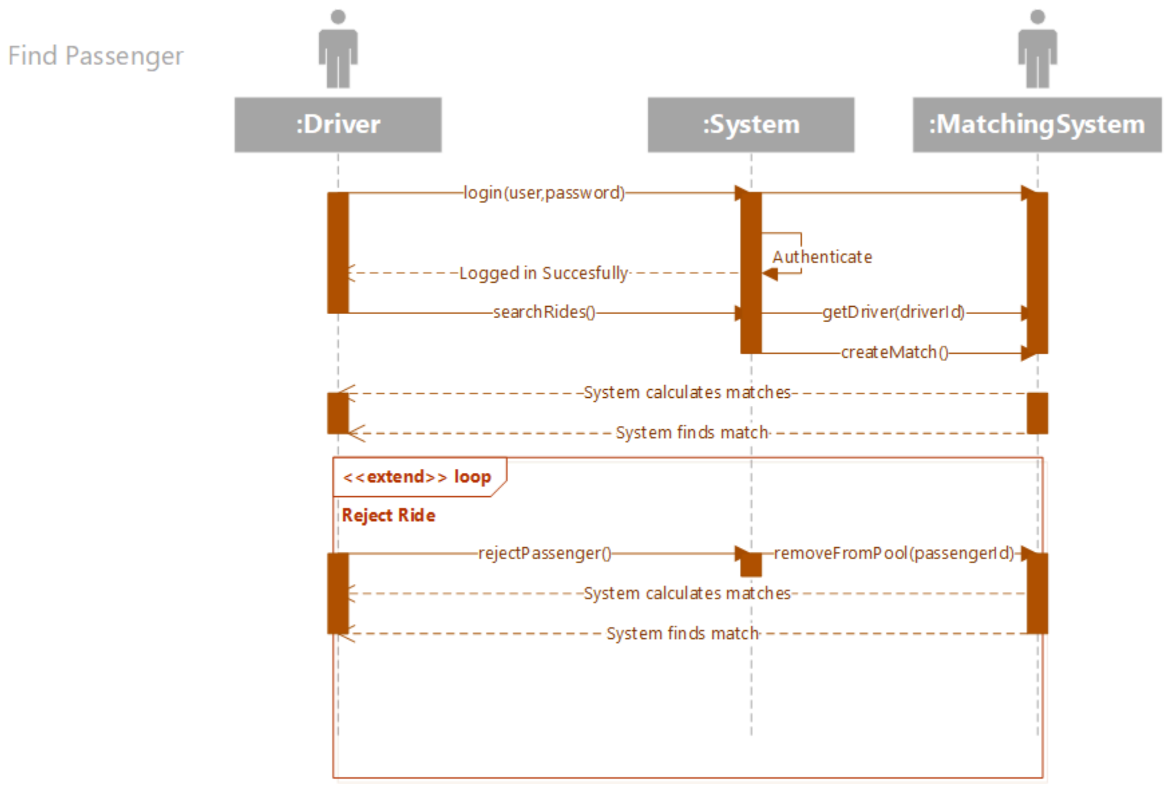
Class Diagram

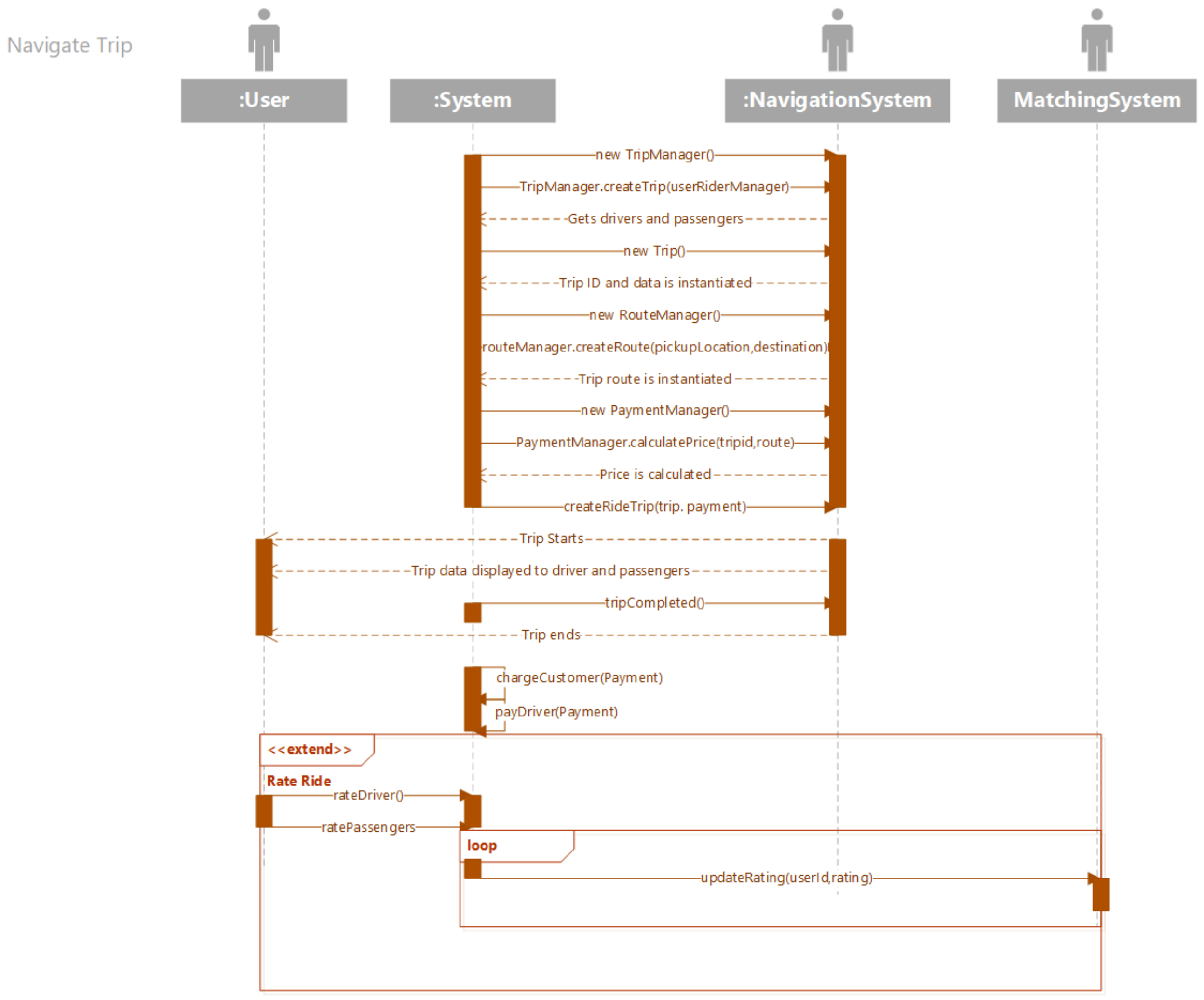


Sequence Diagrams

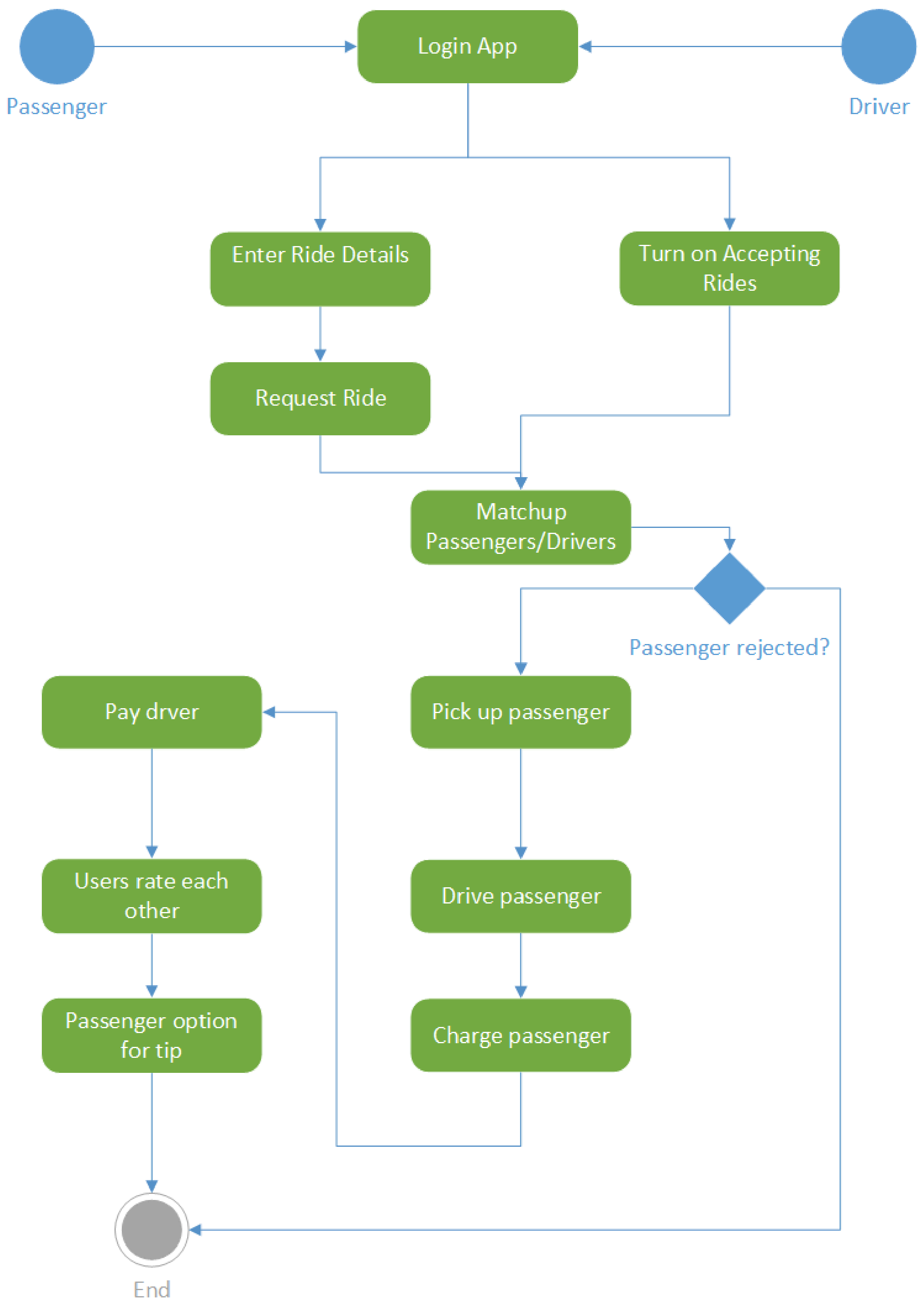


Request Ride
:Passenger
:System
:MatchingSystem
setLocation(pickupLocation)
getPassengerLocation()
selectVehicleType(Vehicle)
selectTime(startTime)
selectDestination(destination)
Confirm ride
getPassenger(passengerId)
createMatch()
System calculates matches
System finds match

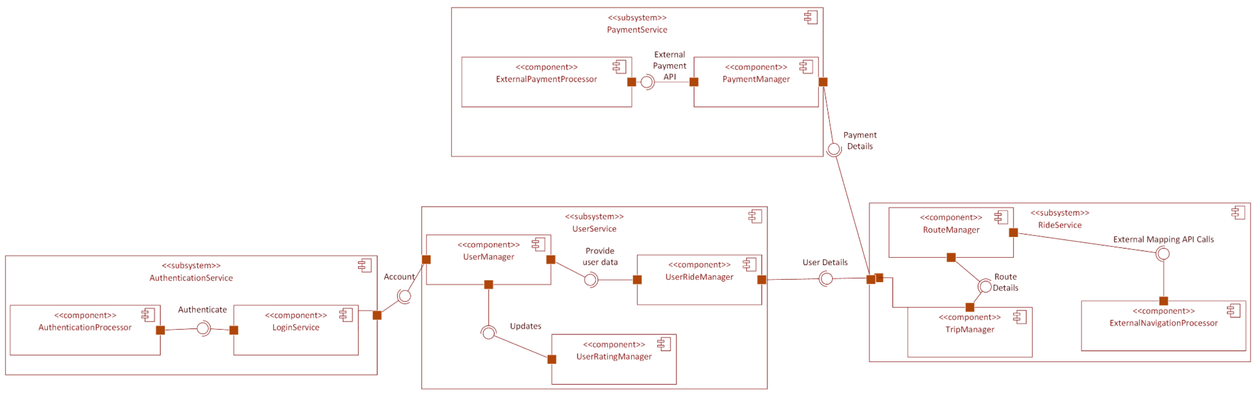



Navigate Trip
:User
:System
:NavigationSystem
MatchingSystem
new TripManager()
TripManager.createTrip(userRiderManager)
Gets drivers and passengers
new Trip()
Trip ID and data is instantiated
new RouteManager()
routeManager.createRoute(pickupLocation,destination)
Trip route is instantiated
new PaymentManager()
PaymentManager.calculatePrice(tripid,route)
Price is calculated
createRideTrip(trip. payment)
Trip Starts
Trip data displayed to driver and passengers
tripCompleted()
Trip ends
chargeCustomer(Payment)
payDriver(Payment)
<<extend>>
Rate Ride
rateDriver()
ratePassengers
loop
updateRating(userId,rating)

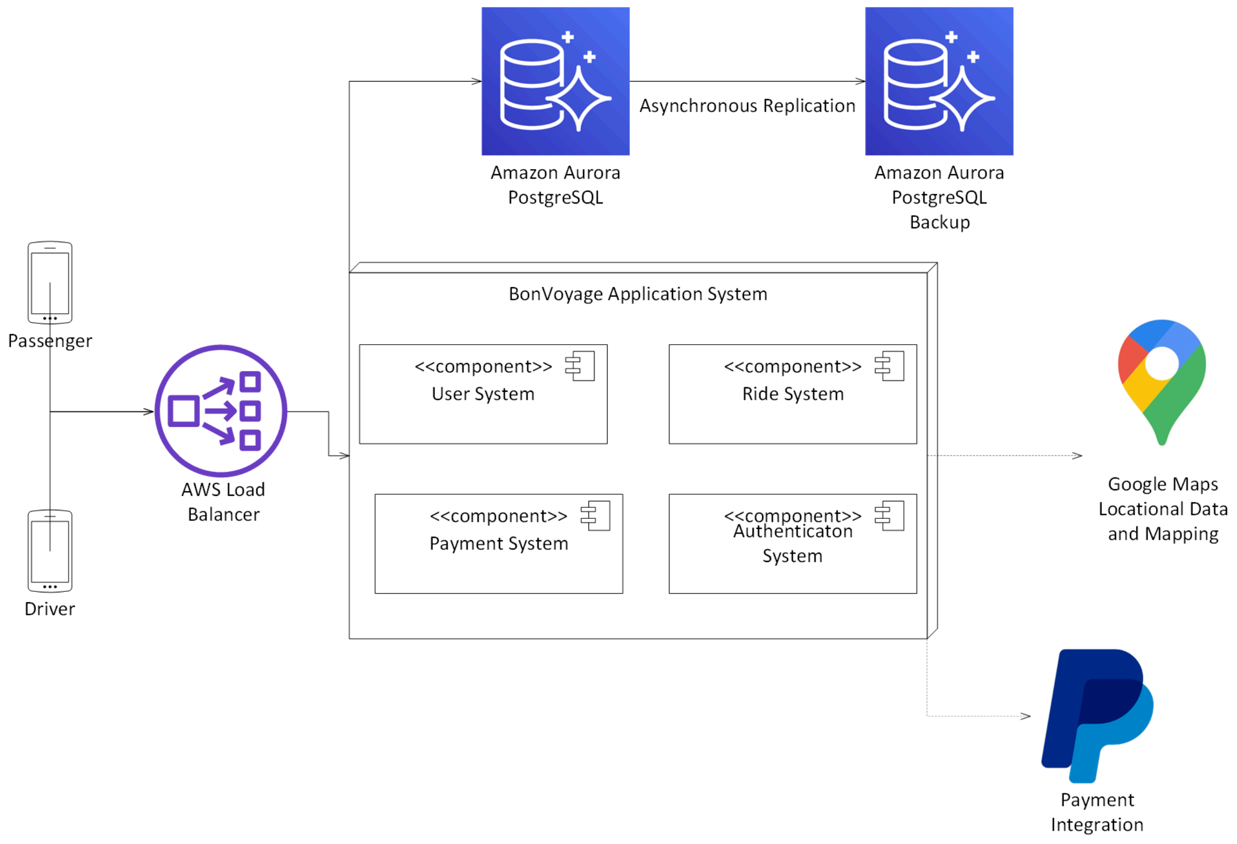
State Diagram



Component Diagram



Deployment Diagram



Class Outlines

User

- Attributes:
  - userId: string (Unique identifier for the user)
  - username: string
  - password: string (hashed)
  - role: string (passenger, driver, administrator)
  - email: string
  - phoneNumber: string
  - rating: float
- Methods:
  - updateProfile(): void
  - getRole(): string
  - viewRating(): float

Passenger (extends User)

- Attributes:
  - paymentMethod: string (credit card details)
  - passengerId : string

Driver (extends User)

- Attributes:
  - vehicleId: string
  - driverId : string (Unique identifier for the Driver)
- Methods:
  - acceptRide(): void
  - updateLocation(): void
  - viewEarnings(): float
  - getRoute(): Route
  - getDriver(driverId): Driver
  - getDriverLocation(): Address

Trip

- Attributes:
  - tripId: int
  - Status: String
  - startTime: Date
  - endTime: Date
  - driver: Driver
  - passengers: Passengers [ ]
- Methods:
  - getTripInfo(tripId) : Trip
  - getTripId(): int

Location

- Attributes:
  - latitude: float
  - longitude: float
- Methods:
  - calculateDistance(): float

Rating

- Attributes:
  - ratingId: string
  - userId: string
  - ratedUserId: string
  - score: int (1 to 5)
  - comment: string
- Methods:
  - submitRating(): void
  - getAverageRating(): float

Database Outlines

```
CREATE TABLE Users (
   user_id VARCHAR(255) PRIMARY KEY,
   username VARCHAR(255) NOT NULL,
   password VARCHAR(255) NOT NULL,
   role ENUM('passenger', 'driver', 'administrator') NOT NULL,
   email VARCHAR(255) NOT NULL UNIQUE,
   phone_number VARCHAR(20) NOT NULL
);
```

```
CREATE TABLE Passengers ( passenger_id VARCHAR(255) PRIMARY KEY,
payment_method VARCHAR(255), rating FLOAT, FOREIGN KEY (passenger_id)
REFERENCES Users(user_id) );

CREATE TABLE Drivers (
   driver_id VARCHAR(255) PRIMARY KEY,
   vehicle_id VARCHAR(255) NOT NULL,
   rating FLOAT,
   FOREIGN KEY (driver_id) REFERENCES Users(user_id)
);

CREATE TABLE Trip (
   trip_id VARCHAR(255) PRIMARY KEY,
   passenger_id VARCHAR(255),
   driver_id VARCHAR(255),
   pickup_location_latitude FLOAT NOT NULL,
   pickup_location_longitude FLOAT NOT NULL,
   dropoff_location_latitude FLOAT NOT NULL,
   dropoff_location_longitude FLOAT NOT NULL,
   status ENUM('pending', 'in-progress', 'completed', 'canceled') NOT NULL,
   price DECIMAL(10, 2) NOT NULL,
   timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   FOREIGN KEY (passenger_id) REFERENCES Passengers(passenger_id),
   FOREIGN KEY (driver_id) REFERENCES Drivers(driver_id)
);

CREATE TABLE Ratings ( rating_id VARCHAR(255) PRIMARY KEY, user_id
VARCHAR(255), rated_user_id VARCHAR(255), score INT CHECK (score BETWEEN 1
AND 5), comment TEXT, FOREIGN KEY (user_id) REFERENCES Users(user_id), FOREIGN
KEY (rated_user_id) REFERENCES Users(user_id) );
```

Design Patterns/Best Practices

GRASP:

1. Expert: most of the classes designed within the application utilize classes that perform responsibility of actions to the same class that holds that information for it. This is utilized through classes such as

the Trip, Driver, Passenger, that hold all of the attribute information needed in order to perform actions.

2. Creator: the TripManager creates and manages the use of the Trip objects that it creates. This allows it to create and handle many instances at the same time and decouples the responsibility of the Trip class.
3. Controller: the TripManager and UserRideManager manage the workflow of creating, updating, and managing trips and matching riders together. The coordinated interactions of both these systems separate the control while also being organized control flows.

SOLID:

1. Single Responsibility Principle: classes like the RouteManager, TripManager, and PaymentManager are responsible for only one aspect of the system, handling different interactions specific to them. This allows the system to be more modular and easier to maintain.
2. Liskov Substitution Principle: the Driver and Passenger subclasses can be used interchangeably for some actions where a User may be expected.

GOF (Gang of Four):

1. Factory Pattern: The TripManager and PaymentManager act as the factory pattern to create instances of Trip and Payment objects. This encapsulates the object creation and makes it more flexible.

Justification for external tooling and APIs:

Loading Balancing: AWS Load Balancer allows distribution to instances of the application instances like the Route, Trip, User, and Payment service. This implementation allows no instance of the application to become an application, improving application performance and availability. It also allows for the requirement to scale up based on the needs of the system given a large increase of traffic.

Database: Amazon Aurora PostgreSQL Database and redundant backups implementation allows the system to have high performance and scalability. It handles high transaction loads and large amounts of concurrent users. The backup database operates as an automatic failover system and can detect and recover from instance failures dynamically. Lastly, there are security benefits to using this like the encryption of data on transit and fine grained-access control.

Routing and Mapping APIs: Google Maps was chosen as the option for the locational data and mapping tool because of its many features. It has accurate and reliable mapping, up to date imaging and street information, as well as additional geographical data. It also offers data related to real-time traffic and can optimize routes based on current traffic conditions that would otherwise strain the application system.

Payments: PayPal as the payment integration was chosen because it is a highly accessible payment processing platform that is able to facilitate secure transactions online with almost all major credit card companies. PayPal features robust encryption, making sure that any user payment information is secured on transactions and that it is completely successful. They also have the capability to work on a global scale, with their multi-currency support and availability in most countries around the world. This solves any dependency issues if the BonVoyage applications ever needed to operate in regions outside the United States. Lastly, PayPal supports various payment methods, allowing you to use your PayPal balance or other digital wallets if you don’t have access to a credit or debit card.