# PowerEnJoy

DESIGN DOCUMENT

Flavio Primo, Hootan Haji Manoochehri

POLITECNICO DI MILANO | SOFTWARE ENGINEERING 2

## Index

# 1 Introduction

## 1.1 Purpose

This is the Design Document, it shows the architecture underlying PowerEnJoy, starting from the specifications and requirements described in the RASD document. It is addressed to developers and maintainers primarily. The focus is to show the major decisions about software and hardware, through several UML and UX diagrams at different level of details: the goal is to explain how each component interact with each other using a standard language.

## 1.2 Scope

This system, from now on called PowerEnJoy, is about a digital management system for a car-sharing service that exclusively employs electric cars. PowerEnJoy provides classical functionality found in similar services such as: user registration, search for an available car and renting a car.

Since PowerEnJoy is about electric cars, it will manage facilities to park and recharge the cars and special discounts for users with virtuous behavior in respect to the environment and other users.

## 1.3 Definitions, Acronyms, Abbreviations

- **RASD:** Requirement Analysis and Specification Document
- **DD:** Design Document
- **RDBMS:** Relational Database Management System
- **DB:** Database managed by a RDBMS
- **JDBC:** Java Database Connectivity
- **Application server:** component which provides the application logic that interacts with the DB, the MobileApp and the PowerEnJoy Car
- **MobileApp:** PowerEnJoy mobile application installed on a smartphone
- **Back-end:** term used to identify the Application Server
- **MVC:** Model View Controller
- **JAX-RS:** Java API for RESTful Web Services
- **JPA:** Java Persistence API
- **REST:** REpresentational State Transfer
- **LTS:** Long Term Support
- **AWS:** Amazon Web Services
- **Amazon EC2:** Elastic Compute Cloud
- **Amazon VPC:** Virtual Private Cloud
- **Amazon RDS:** Relational Database Service
- **Client:** PowerEnJoy App and PowerEnJoy Box
- **MobileApp:** PowerEnJoy App
- **parking spots:** the collection of both normal and special SafeAreas

## 1.4 Reference Documents

[1] *PowerEnJoy – Requirement Analysis and Specification Document* Flavio Primo, Hootan Haji Manoochehri – Politecnico di Milano

[2] Haversine formula – https://rosettacode.org/wiki/Haversine_formula#Java

# 2 Architectural Design

## 2.1 Overview: High level components and their interactions

The architecture, from a high perspective, is implemented as in Figure 1. Considered system features and design decisions comes from the requirements specified in [1] and the common needs of an enterprise application.

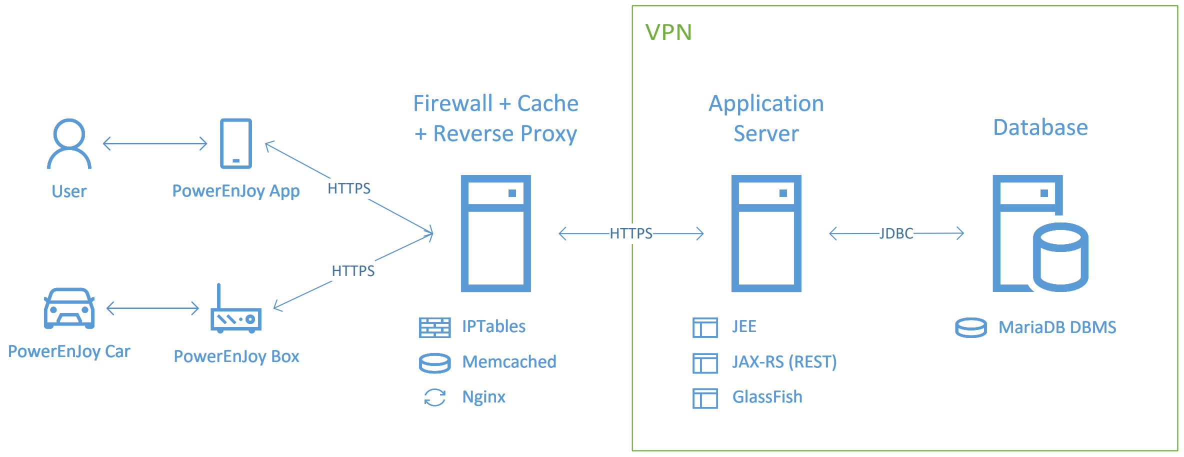


*Figure 1:* Architecture Draft diagram

The system components are presented in Figure 2. The emphasis in this diagram has been given to "vertical slices" of components comprehending each mainly a controller, a service and a linked entity.

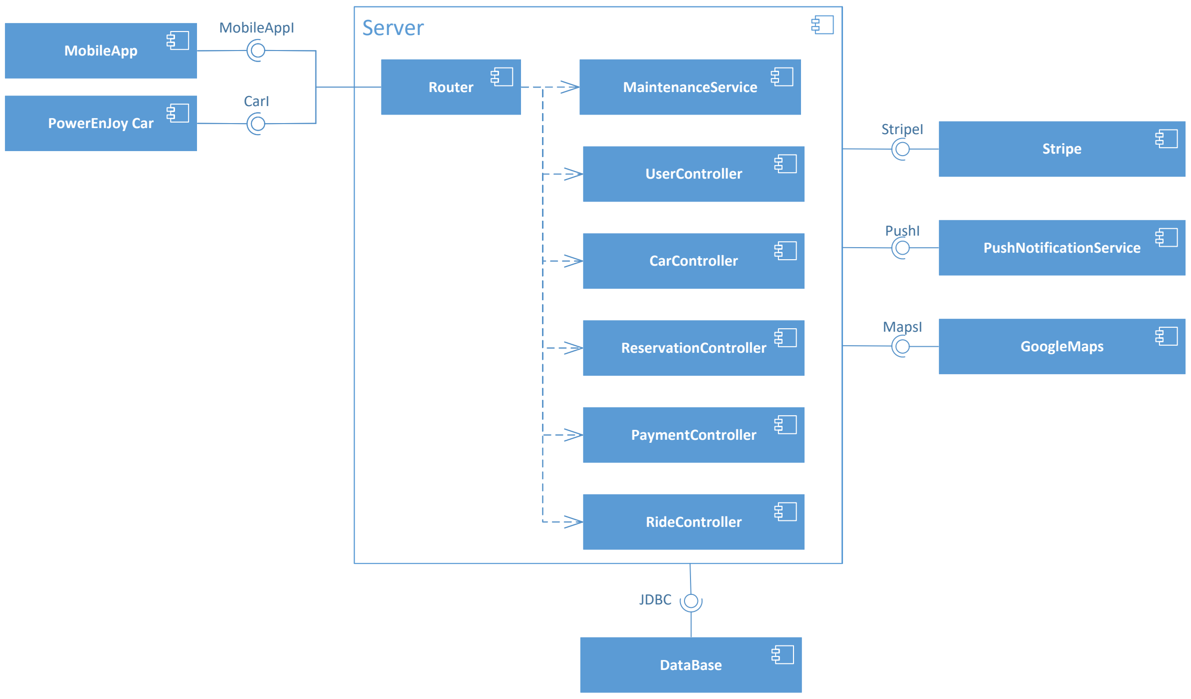


*Figure 2: Main Components diagram*

A better detailed view of the interaction of the components is given below with the composite structure diagram:

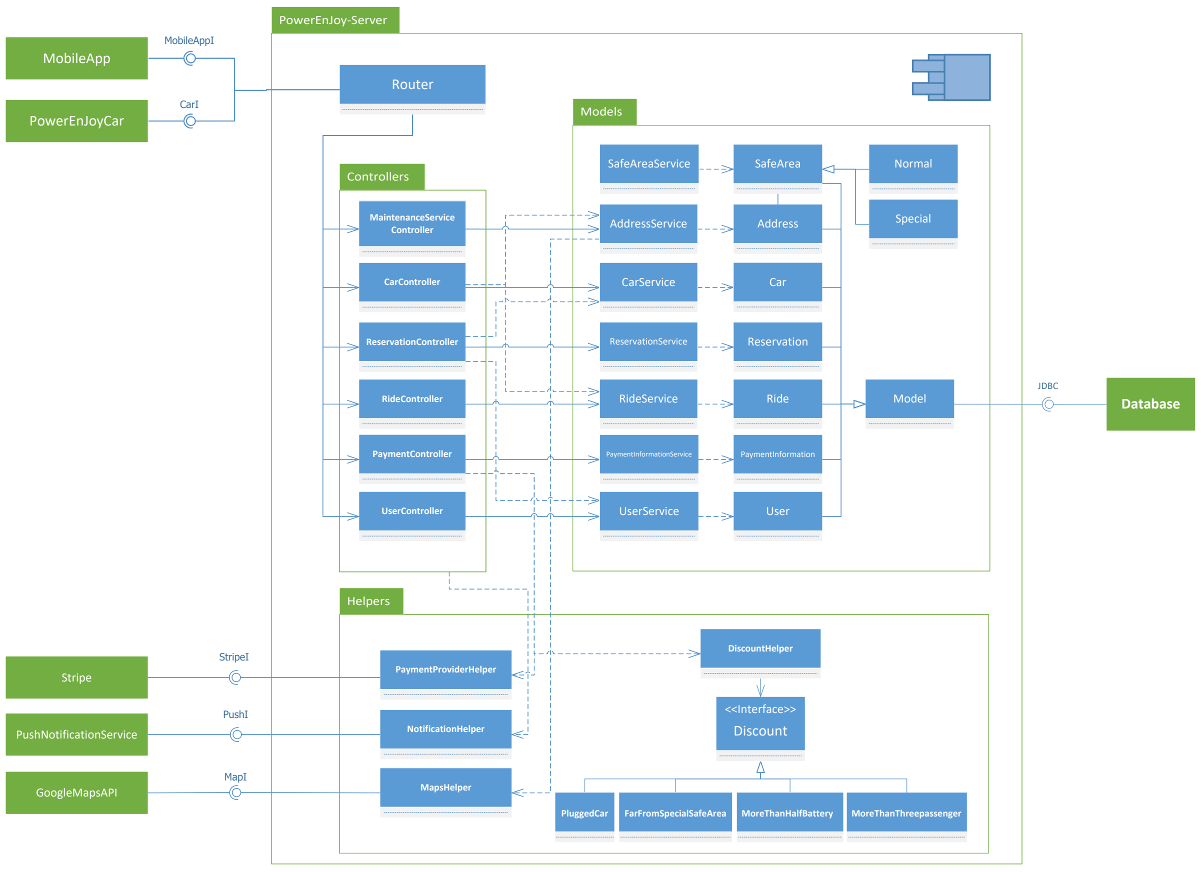


*Figure 3: Composite Structure Diagram*

To have a better idea of the application domain, an ER Diagram is provided in Figure 3.

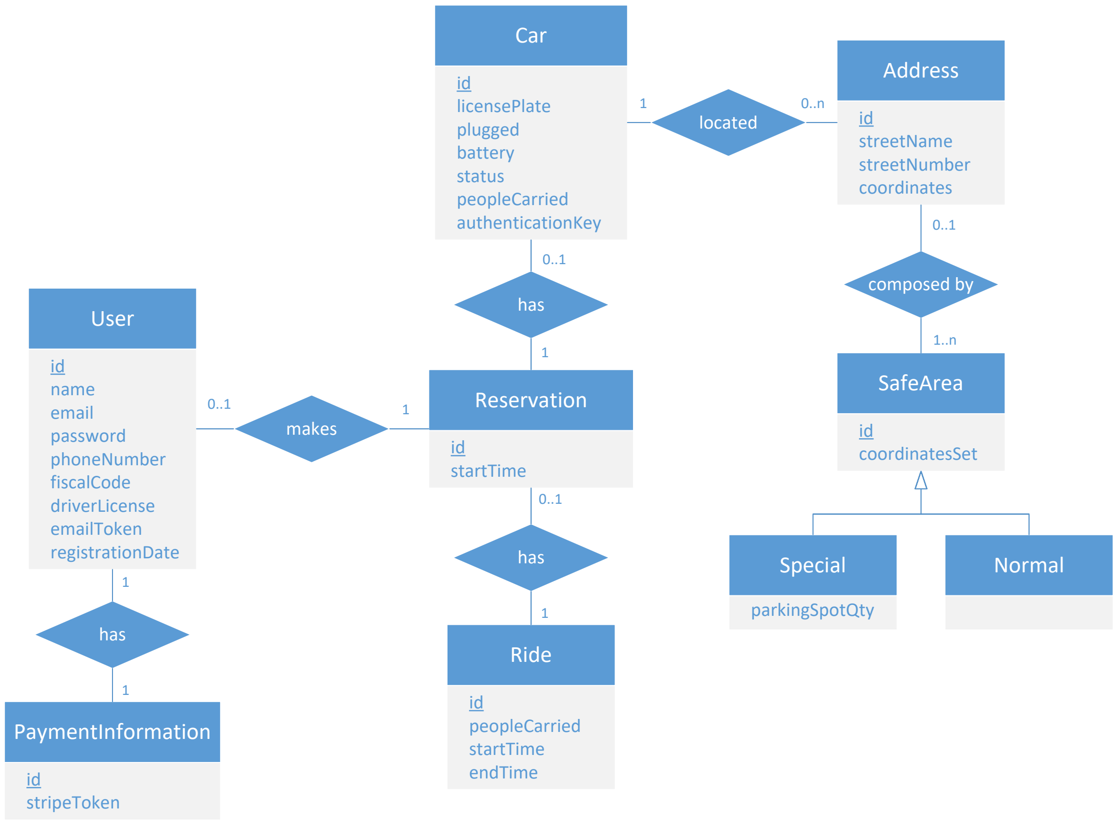
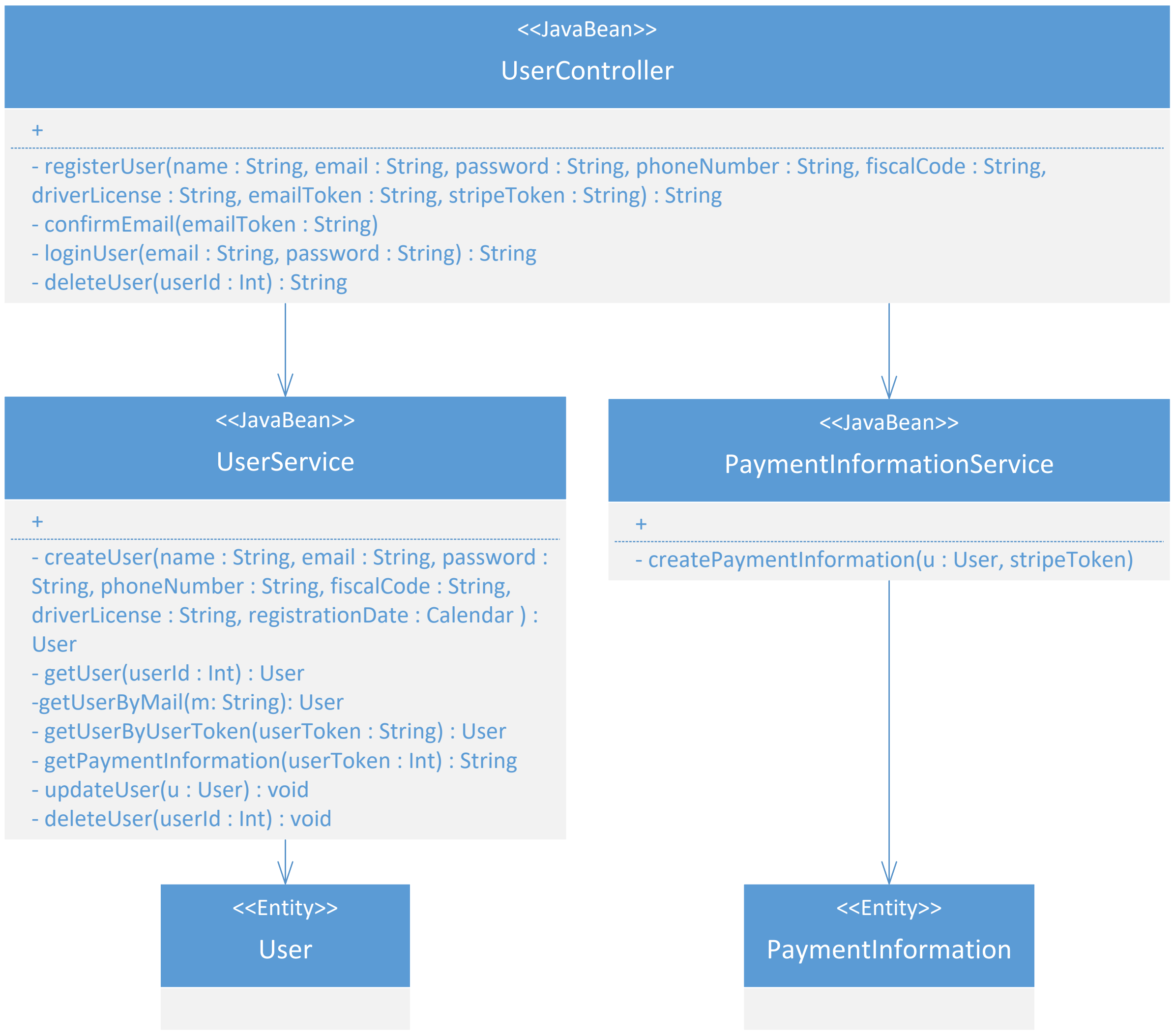


*Figure 4: ER diagram*

## 2.2 Component View

### 2.2.1 UserController

UserController manages the User and the Guest (User to be). It provides functionalities such as: guest registration, user login, user deletion and confirmation of the email.

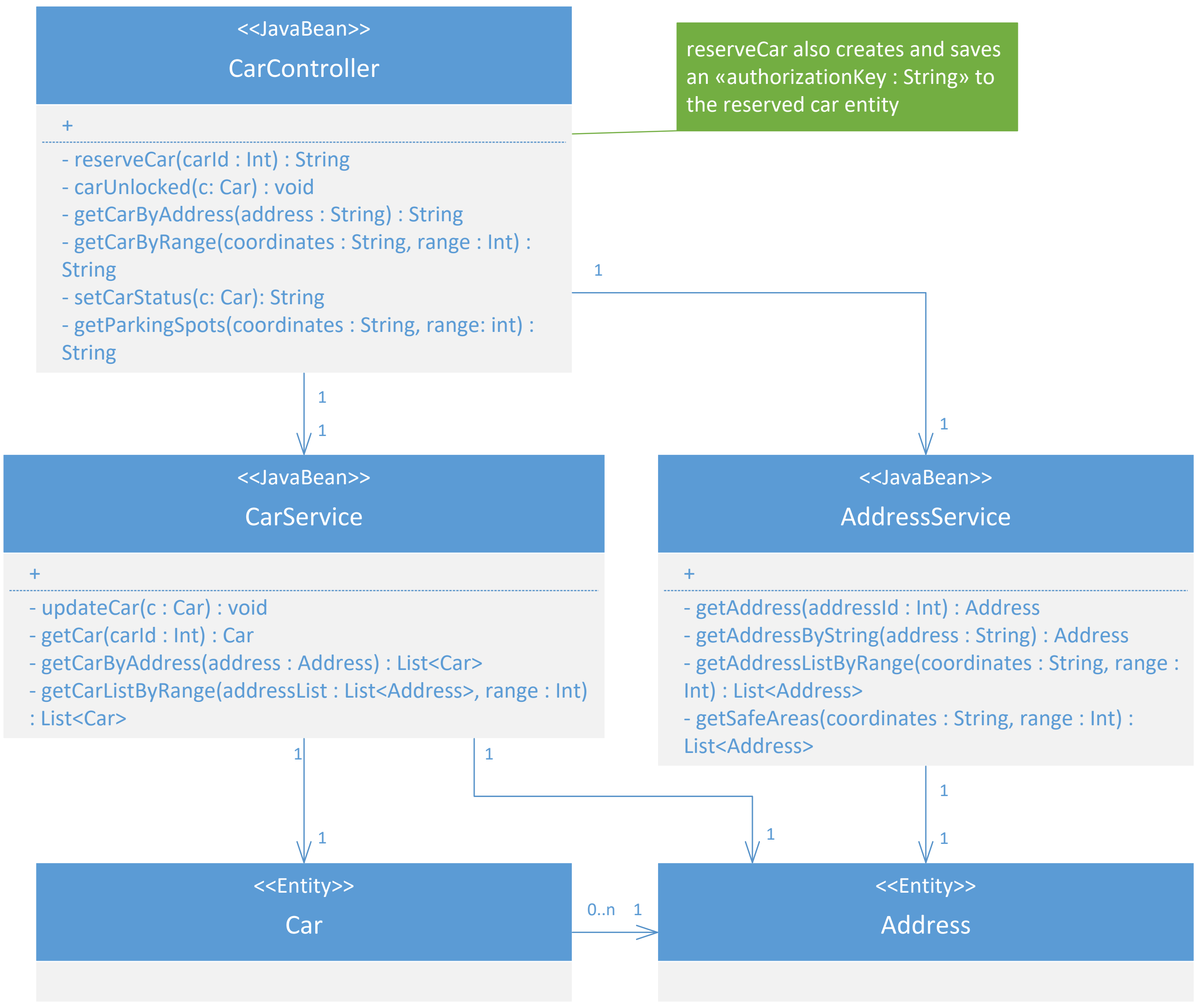


<<JavaBean>>
UserController
+
- registerUser(name : String, email : String, password : String, phoneNumber : String, fiscalCode : String, driverLicense : String, emailToken : String, stripeToken : String) : String
- confirmEmail(emailToken : String)
- loginUser(email : String, password : String) : String
- deleteUser(userId : Int) : String
<<JavaBean>>
UserService
+
- createUser(name : String, email : String, password : String, phoneNumber : String, fiscalCode : String, driverLicense : String, registrationDate : Calendar ) : User
- getUser(userId : Int) : User
-getUserByMail(m: String): User
- getUserByUserToken(userToken : String) : User
- getPaymentInformation(userToken : Int) : String
- updateUser(u : User) : void
- deleteUser(userId : Int) : void
<<JavaBean>>
PaymentInformationService
+
- createPaymentInformation(u : User, stripeToken)
<<Entity>>
User
<<Entity>>
PaymentInformation

*Figure 5: CarController component diagram*

### 2.2.2 CarController

CarController manages the car and its related information. It provides functionalities such as: car reservation, car unlocking, get cars by range or address and set the car status. It is responsible to give the system an updated status of the managed cars by the system. Since this is the controller that is responsible to communicate with the car, another duty of this controller is that to provide safeAreas to the PowerEnJoy Box to be shown on the screen.

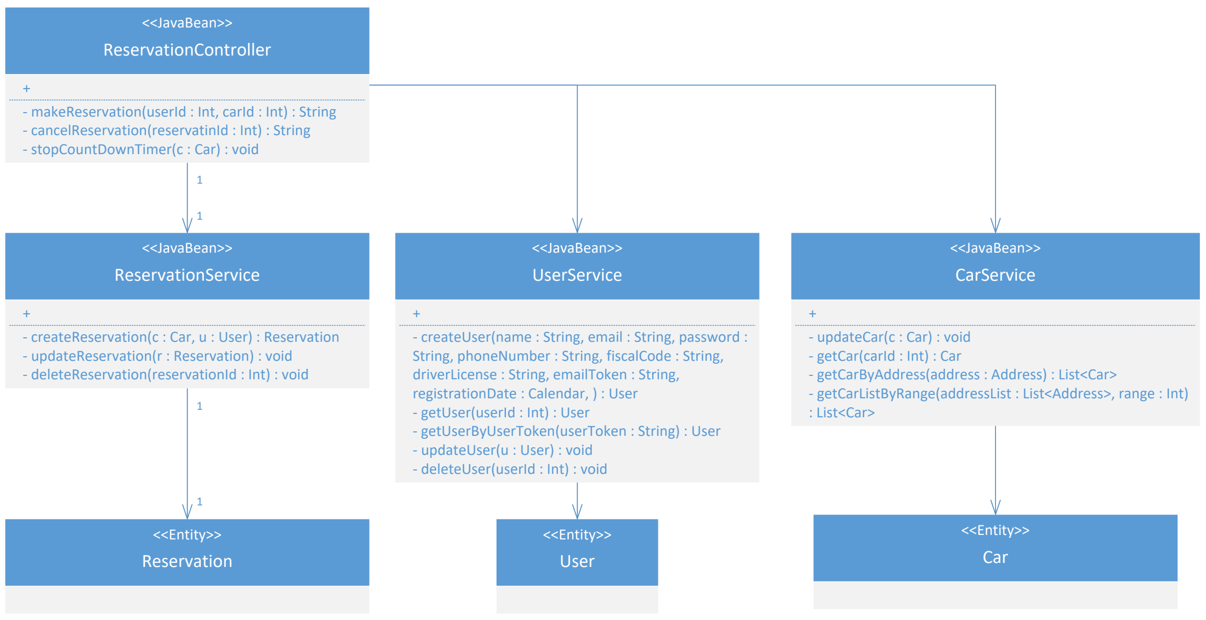


<<JavaBean>>
CarController
+
- reserveCar(carId : Int) : String
- carUnlocked(c: Car) : void
- getCarByAddress(address : String) : String
- getCarByRange(coordinates : String, range : Int) : String
- setCarStatus(c: Car): String
- getParkingSpots(coordinates : String, range: int) : String
reserveCar also creates and saves an «authorizationKey : String» to the reserved car entity
1
1
1
1
<<JavaBean>>
CarService
+
- updateCar(c : Car) : void
- getCar(carId : Int) : Car
- getCarByAddress(address : Address) : List<Car>
- getCarListByRange(addressList : List<Address>, range : Int) : List<Car>
<<JavaBean>>
AddressService
+
- getAddress(addressId : Int) : Address
- getAddressByString(address : String) : Address
- getAddressListByRange(coordinates : String, range : Int) : List<Address>
- getSafeAreas(coordinates : String, range : Int) : List<Address>
1
1
1
1
1
1
<<Entity>>
Car
0..n
1
<<Entity>>
Address

*Figure 6: CarController component diagram*

### 2.2.3 ReservationController

ReservationController manages the reservation and its related information. It provides functionalities such as: making and cancelling a reservation and to start the ride by stopping the countdown to the automatic reservation cancellation.

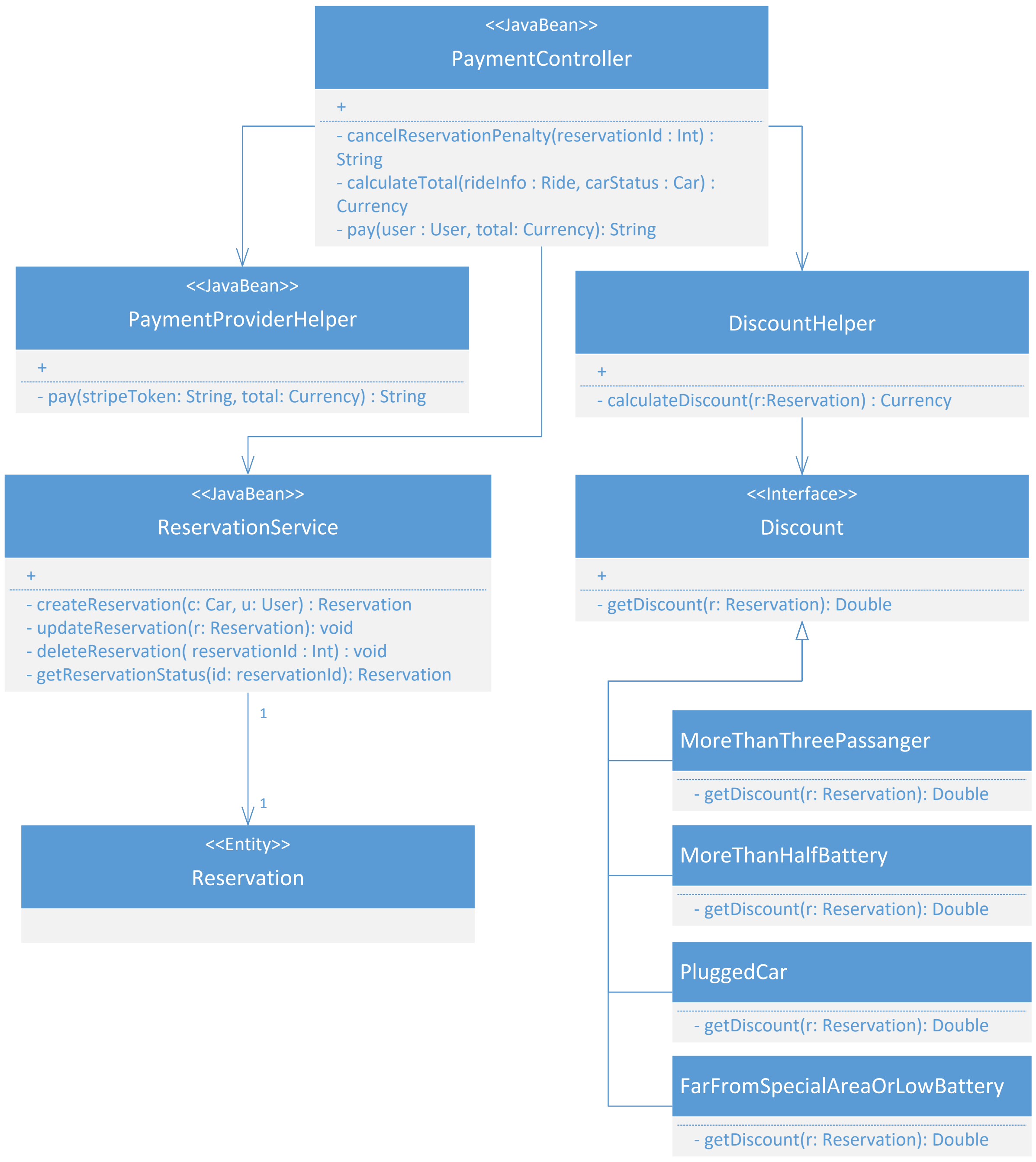


<<JavaBean>>
ReservationController
+
- makeReservation(userId : Int, carId : Int) : String
- cancelReservation(reservatinId : Int) : String
- stopCountDownTimer(c : Car) : void
1
1
<<JavaBean>>
ReservationService
+
- createReservation(c : Car, u : User) : Reservation
- updateReservation(r : Reservation) : void
- deleteReservation(reservationId : Int) : void
1
1
<<Entity>>
Reservation
<<JavaBean>>
UserService
+
- createUser(name : String, email : String, password : String, phoneNumber : String, fiscalCode : String, driverLicense : String, emailToken : String, registrationDate : Calendar, ) : User
- getUser(userId : Int) : User
- getUserByUserToken(userToken : String) : User
- updateUser(u : User) : void
- deleteUser(userId : Int) : void
<<Entity>>
User
<<JavaBean>>
CarService
+
- updateCar(c : Car) : void
- getCar(carId : Int) : Car
- getCarByAddress(address : Address) : List<Car>
- getCarListByRange(addressList : List<Address>, range : Int) : List<Car>
<<Entity>>
Car

*Figure 7: ReservationController component diagram*

### 2.2.4 PaymentController

*PaymentController is responsible to communicate with the* ***Stripe*** *calculating the total cost of the ride, discount etc. it achieves these goals by using some helpers like DiscountHelper and PaymentProviderHelper.*

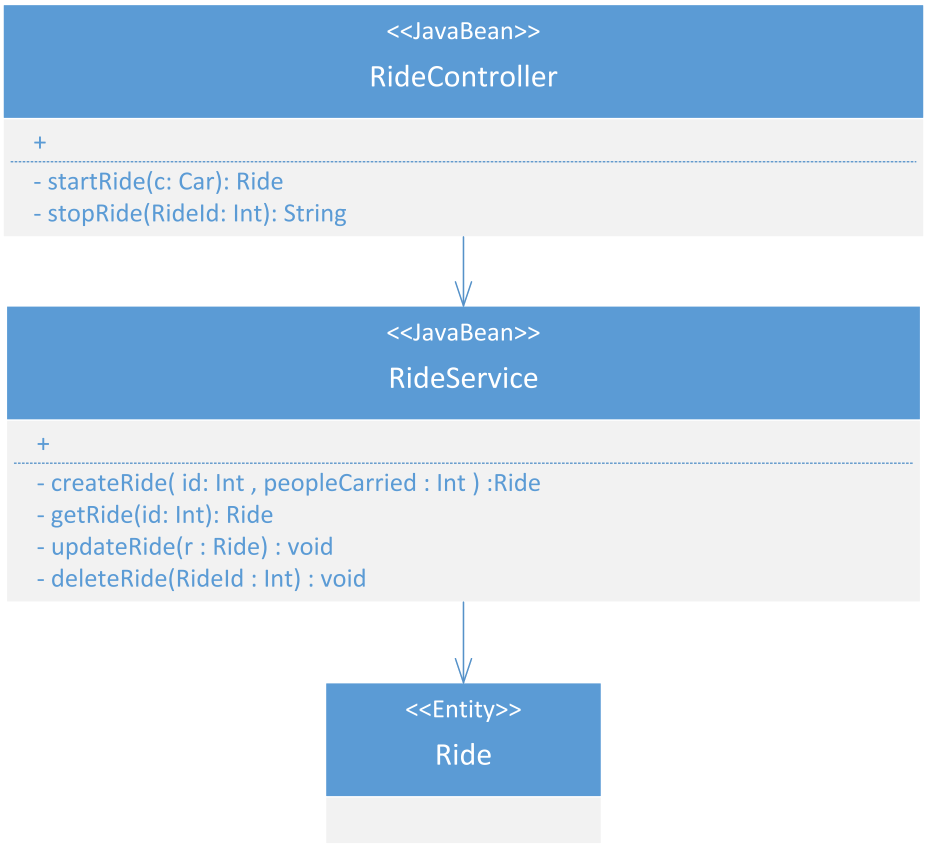


<<JavaBean>>
PaymentController
+
- cancelReservationPenalty(reservationId : Int) : String
- calculateTotal(rideInfo : Ride, carStatus : Car) : Currency
- pay(user : User, total: Currency): String
<<JavaBean>>
PaymentProviderHelper
+
- pay(stripeToken: String, total: Currency) : String
DiscountHelper
+
- calculateDiscount(r:Reservation) : Currency
<<JavaBean>>
ReservationService
+
- createReservation(c: Car, u: User) : Reservation
- updateReservation(r: Reservation): void
- deleteReservation( reservationId : Int) : void
- getReservationStatus(id: reservationId): Reservation
<<Interface>>
Discount
+
- getDiscount(r: Reservation): Double
1
1
<<Entity>>
Reservation
MoreThanThreePassanger
- getDiscount(r: Reservation): Double
MoreThanHalfBattery
- getDiscount(r: Reservation): Double
PluggedCar
- getDiscount(r: Reservation): Double
FarFromSpecialAreaOrLowBattery
- getDiscount(r: Reservation): Double

*Figure 8: ReservationController component diagram*

### 2.2.5 RideController

RideController is responsible for start and stop the ride when the unlocking and payment takes place respectively.



<<JavaBean>>
RideController
+
- startRide(c: Car): Ride
- stopRide(RideId: Int): String
<<JavaBean>>
RideService
+
- createRide( id: Int , peopleCarried : Int ) :Ride
- getRide(id: Int): Ride
- updateRide(r : Ride) : void
- deleteRide(RideId : Int) : void
<<Entity>>
Ride

*Figure 9: RideController component diagram*

## 2.3 Deployment View

In this section, it is described the deployment of the system. In Figure 4 is depicted the physical structure of the system. In the following paragraphs, it is explained in more detail the composing elements.

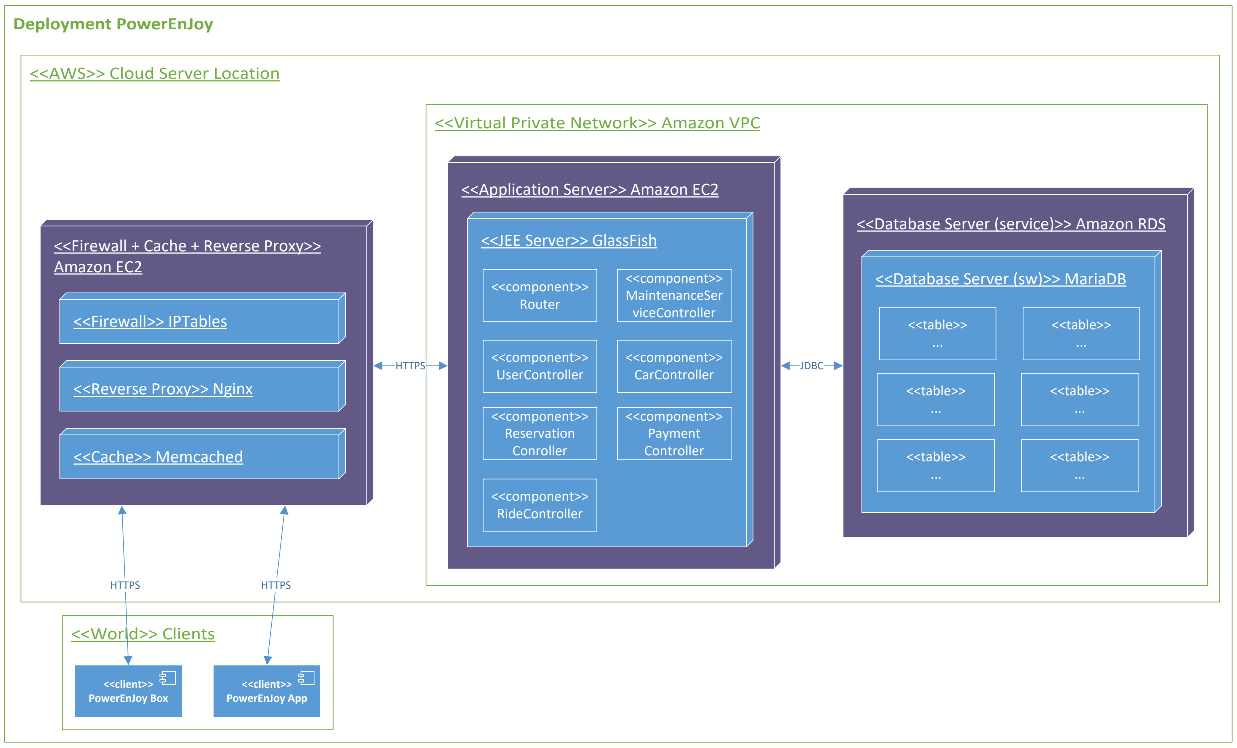

Deployment PowerEnJoy
<<AWS>> Cloud Server Location
<<Virtual Private Network>> Amazon VPC
<<Firewall + Cache + Reverse Proxy>> Amazon EC2
<<Firewall>> IPTables
<<Reverse Proxy>> Nginx
<<Cache>> Memcached
HTTPS
<<Application Server>> Amazon EC2
<<JEE Server>> GlassFish
<<component>> Router
<<component>> MaintenanceServiceController
<<component>> UserController
<<component>> CarController
<<component>> Reservation Conroller
<<component>> Payment Controller
<<component>> RideController
JDBC
<<Database Server (service)>> Amazon RDS
<<Database Server (sw)>> MariaDB
<<table>> ...
HTTPS
HTTPS
<<World>> Clients
<<client>> PowerEnJoy Box
<<client>> PowerEnJoy App

*Figure 10: Deployment Diagram*

The components in diagram Figure 3 are:

- **PowerEnJoy App:** is any smartphone with the PowerEnJoy application installed that interacts with the PowerEnJoy system.
- **PowerEnJoy Box:** is any PowerEnJoy Box mounted on PowerEnJoy Cars that interacts with the PowerEnJoy system.
- **Firewall + Cache + Reverse Proxy:** this server tasks are to filter, cache and forwarding the requests between the application server and the clients, namely the MobileApp and the PowerEnJoy Box.
  - **IPTables (Firewall):** IPTables is a generic table structure for the definition of rulesets. Each rule within an IP table consists of several classifiers (matches) and one connected action (target). Its features stateless and stateful packet filtering (IPv4 and IPv6).
  - **Memcached (Cache):** high-performance, distributed memory object caching system, that speeds up web applications by alleviating database load. Memcached is an in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, or page rendering.
  - **Nginx (Reverse Proxy):** reverse proxy server that supports various protocols including HTTP and HTTPS. It forwards requests between the application server and the clients. It also helps hiding the implementation of the offered web services, thus simplifying client interaction and improving security. If necessary, it can also act as a load balancer as the number of requests to the system increase.
- **Application Server:** contains the business logic behind the PowerEnJoy system.
  The application is served by the application server GlassFish, the reference J2EE implementation.
  Database communication is made possible by the Java Persistence API (JPA), which wraps all the database functionalities.

Communication with the MobileApp and the PowerEnJoy box relies on the Representational State Transfer (REST) through the JAX-RS library.
The application logging is delegated to the widely used and supported library Log4j 2.

- **Database:** contains the RDBMS application. This database contains all the persistence needs of the Application Server.

The "Firewall + Cache + Reverse Proxy" and "Application Server" servers run the Ubuntu Linux distribution version 16.04. It is a Long-Term Support (LTS) edition which means that it is supported with hardware and maintenance updates for 5 years.

Scalability for heavier traffic can be achieved by duplicating one or multiple times the Application Server and the Database. The databases are mirrored so they contain the same tables and tuples. The Nginx reverse proxy must then be configured as a load balancer and equally distribute the traffic between all the available Application Servers.

The physical implementation relies on the cloud services offered by AWS. It has been chosen a cloud physical implementation because it's easier to implement, secure and can be scaled at will with minimal effort. The system configuration illustrated in Figure 1 uses the following Amazon services:

- **Elastic Compute Cloud (EC2):** virtual computing environment, allows to create instances with a variety of operating systems, load them with a custom application environment, manage the network's access permissions, and run a custom image using as many or few systems as desired.
- **Relational Database Service (RDS):** managed relational database service that provides various database engines to choose from (including MariaDB). Amazon RDS handles routine database tasks such as provisioning, patching, backup, recovery, failure detection and repair.
- **Virtual Private Cloud (VPC):** lets provision a logically isolated section of the Amazon Web Services (AWS) cloud where can be launched AWS resources in a custom virtual network. It is given complete control over the virtual networking environment, including selection of your own IP address range, creation of subnets, and configuration of route tables and network gateways.

The "Firewall + Cache + Reverse Proxy" and "Application Server" servers are run in 2 separate EC2 instances.

The "Database" is configured as a MariaDB instance of RDS.

A VPC is used to protect and isolate the "Application Server" and the "Database" servers from unauthorized web access. The only server that accesses the "Application Server" is the "Firewall + Cache + Reverse Proxy".

## 2.4 Runtime View

The behavior of the system is presented with improved sequence diagrams, better explaining how system components and classes interacts together.

### 2.4.1 Sign-In

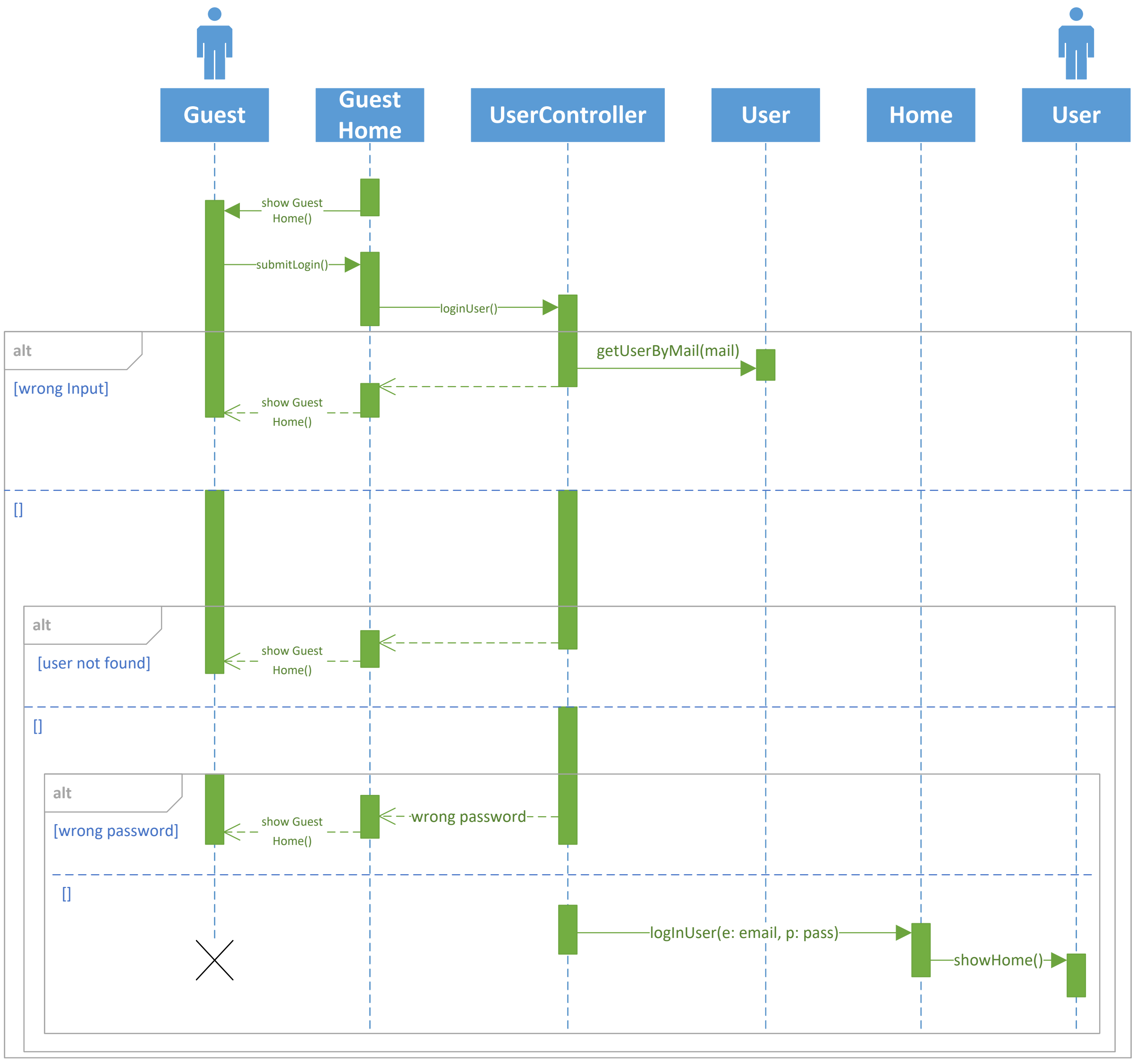


*Figure 11: Sign-In sequence diagram*

## 2.4.2 Registration

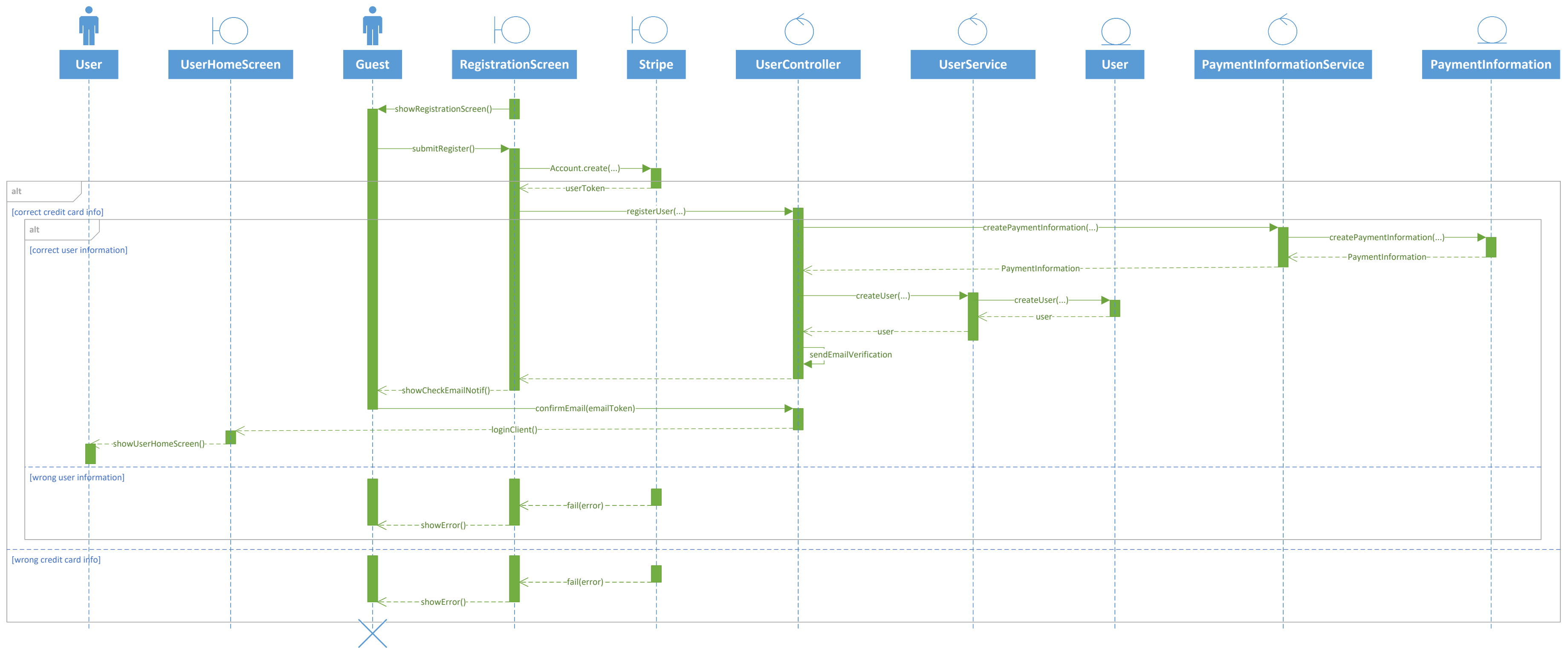


*Figure 12: Registration sequence diagram*

### 2.4.3 Reservation

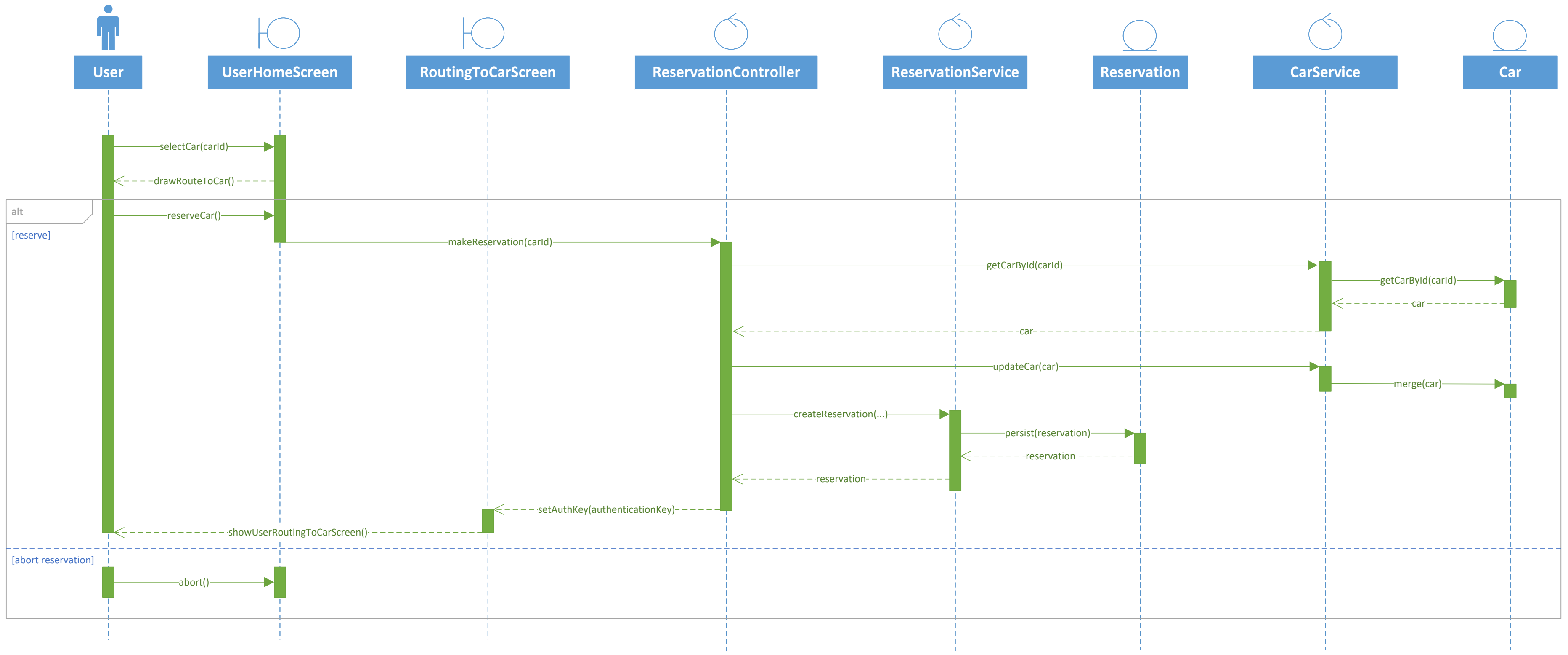


*Figure 13: Reservation sequence diagram*

## 2.4.4 Unlocking

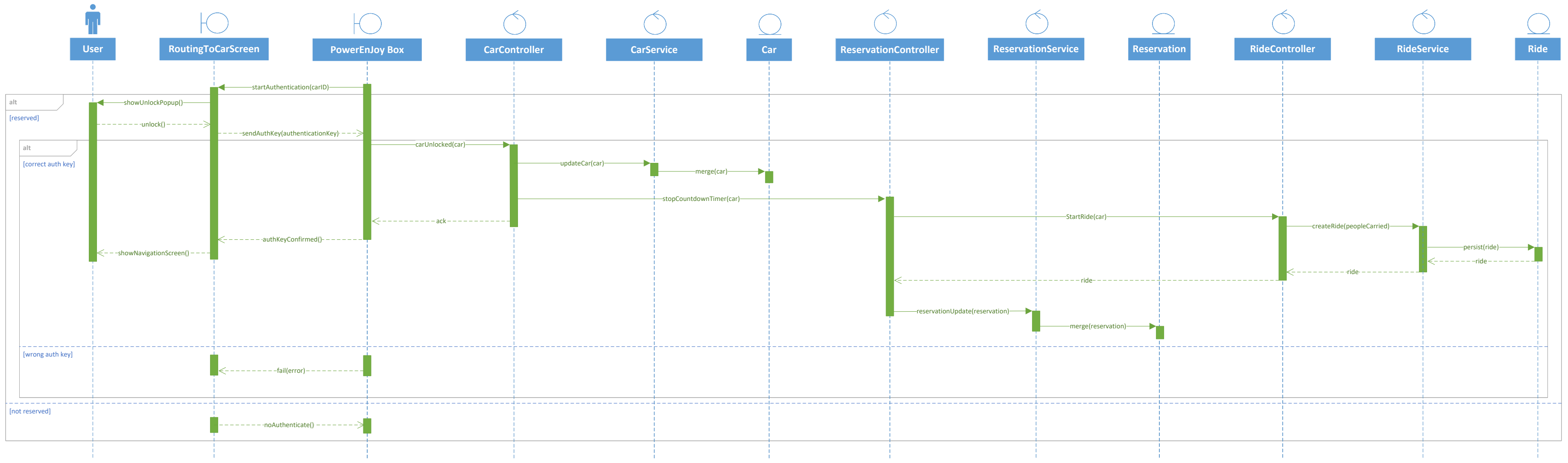


*Figure 14: Unlocking sequence diagram*

## 2.4.5 Payment

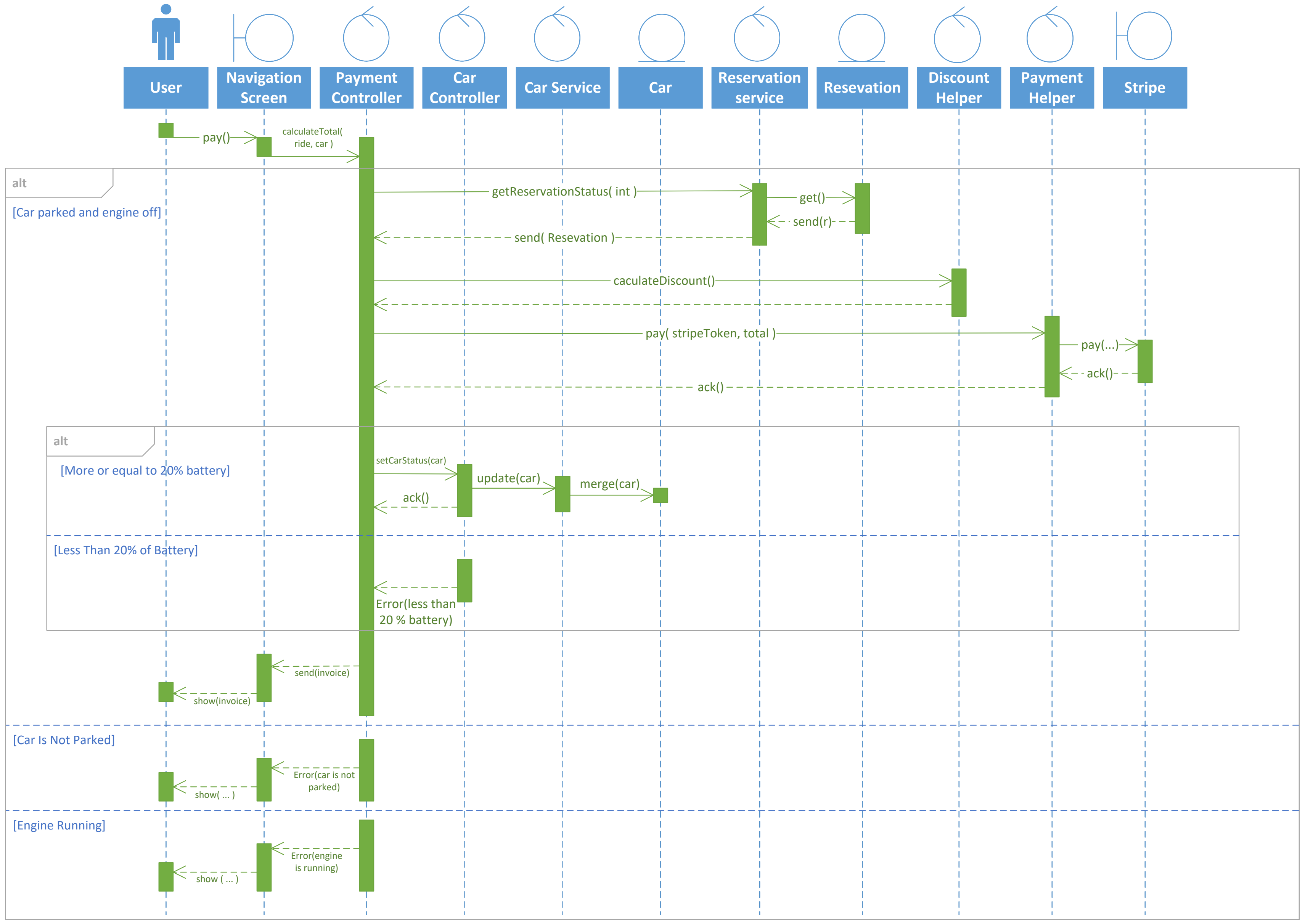


*Figure 15: Payment sequence diagram*

## 2.5 Component Interfaces

### 2.5.1 Connection application server – Database

The application server uses EntityBeans to map relations into Java objects, which themselves refer to the underlying DB through JDBC calls. These connection types (connection, query, handling results, …) are standard and won't be discussed here.

### 2.5.2 Web Service – JAX-RS

The following public methods are available through a Web service, which is handled by JAX-RS on the server side through REST APIs. This means that each client that knows the address of the service can use it according to the functions described in detail below. Authorization for specific operations is controlled by tokens, as it is required in the parameter list.

#### *2.5.2.1 PaymentController*

The paymentController bean is the most important controller of our service, it does the last operation of the ride. In other word, it is responsible to calculate the cost of the ride and according to the circumstances apply some discounts for the user and at the end charges the user with total amount that is calculated.

| Method name | Token | User | Parameter name | Parameter description | Returns |
|---|---|---|---|---|---|
| **cancelReservationPenalty** | Yes | System | Reservation id | The ID of the reservation | String |
| **calculateTotal** | Yes | User | RideInfo<br>CarStatus | an instance of the ride<br>an instance of the Car | Currency |
| **pay** | Yes | System | User<br>total | It gets user information to extract the stripe token<br>amount of the payment | String |

#### *2.5.2.2 UserController*

The UserController is responsible to gives the guest of the system ability to create an account on the system, do some validation like email verification and provide the login procedure to the users of the system. It also provides some account management functionality like delete account and update user information.

| Method name | Token | User | Parameter name | Parameter description | Returns |
|---|---|---|---|---|---|
| **registerUser** | No | Guest | Name<br>Email<br>Password<br>Phone number | Full name of the user<br>email address of the user<br>Secret code of the user<br>cellphone number | String |

| | | | | | |
|---|---|---|---|---|---|
| | | | Fiscal code<br><br>Driver license<br><br>Stripe Token | The tax code of the user<br>Code of the driving license<br>It's a token that user get from his stripe account | |
| **confirmEmail** | Yes | User | Email Token | It's a token that generated for the confirmation of validity of the user's email address | String |
| **loginUser** | No | Guest | Email<br>Password | Email address<br>password of user | String |
| **deleteUser** | Yes | User/System | Id<br>Password | Id of the user<br>password of the user | String |

### *2.5.2.3 ReservationController*

The ReservationController is responsible to manage reservations. It let a reservation to be created and canceled by a User. It also provide a way to let start the ride.

| Method name | Token | User | Parameter name | Parameter description | Returns |
|---|---|---|---|---|---|
| **makeReservation** | Yes | User | carId | Id of the car | String |
| **cancelReservation** | Yes | User, System | reservationId | Id of the reservation | String |
| **stopCountDownTimer** | Yes | User | reservationId | Id of the reservation | void |

### *2.5.2.4 CarController*

The CarController is responsible to manage cars. It let a car to be reserved, unlocked and set its status. The PowerEnJoy Box token is the authorization key given upon car reservation by the system. It let also the user know the location of the cars giving an address and a range or the car in an address.

| Method name | Token | User | Parameter name | Parameter description | Returns |
|---|---|---|---|---|---|
| **reserveCar** | Yes | System | carId | Id of the car | String |
| **carUnlocked** | Yes | PowerEnJoy Box | car | Car object | void |
| **getCarByAddress** | Yes | User | address | String of the address where to | String |

| | | | | | |
|---|---|---|---|---|---|
| | | | | search for an available car | |
| **getCarByRange** | Yes | User | coordinates | String of coordinates that locate the center of the range in which to search for cars | String |
| | | | range | Range where to search for cars | |
| **setCarStatus** | Yes | System | car | Car object | String |
| **getParkingSpots** | Yes | PowerEnJoy Box | coordinates | String of coordinates that locate the center of the range in which to search for SafeAreas | String |
| | | | range | Range where to search for SafeAreas | |

##### *2.5.2.5 RideController*

The RideController is responsible to manage rides. It let a ride to be started and stopped. It is used exclusively by the system.

| Method name | Token | User | Parameter name | Parameter description | Returns |
|---|---|---|---|---|---|
| **startRide** | No | System | car | Car object | Ride |
| **stopRide** | No | System | rideId | Id of the ride | String |

## 2.6 Selected Architectural Styles and Patterns

This section describes high level patterns used in the PowerEnJoy system.

### 2.6.1 Deployment

3-tier architecture composed by:

- **Client tier:** it comprehends the PowerEnJoy app and the PowerEnJoy box.
- **Logic Tier:** it controls application functionality by performing detailed processing: calculations, logical decisions, data and model manipulation.
- **Data Tier:** houses database servers where information is stored and retrieved. Data in this tier is kept independent of the Logic Tier.

3-tier architecture has been chosen because it is widely used and proven to be effective. The design principles it adheres are:

- **Divide & Conquer:** software is divided in 3 independent tiers that putted together offer the PowerEnJoy application services.
- **Cohesion:** each module has his specific purpose; Client tier handles the system clients; Logic tier handles the application processing and the Data layer that handles and manages the persistence of data.
- **Decoupling:** software is divided in 3 independent tiers that communicates through interfaces.
- **Max Abstraction:** every tier hides its implementation with an API interface.
- **Max reusability design:** if a tier is substituted by another one, the other tier can still be used.
- **Flexibility:** because of decoupling, abstraction and reusability.
- **Anticipate obsolescence:** if the technologies that compose a tier become obsolete, a tier can be substituted without affecting the others.
- **Portability:** each tier can be run on different platforms.
- **Testability:** each tier can be tested on its own simplifying error detection and testing.
- **Defensive Design:** each tier can be accessed only by predefined API limiting the misuse of tiers.

### 2.6.2 Communication

A **Service-Oriented Architecture** has been chosen. It provides application functionalities as a set of consumable services by other applications. Services are loosely coupled self-contained units.

- It logically represents a business activity with a specified outcome.
- It is self-contained.
- It is a black box for its consumers.
- It may consist of other underlying services

### 2.6.3 Structure

An **Object-Oriented Architectural Style** has been chosen. It supports the division of responsibilities for a complex system into small and reusable parts called "Objects". They communicate with each other through interfaces, by sending and receiving messages or by calling methods in other objects.

Main benefits are:

- **Extensible:** change of implementation does not imply an interface change
- **Reusable:** objects are developed as small reusable piece of software
- **Testable:** encapsulation improves testability
- **Understandable:** it maps the application with real world entities

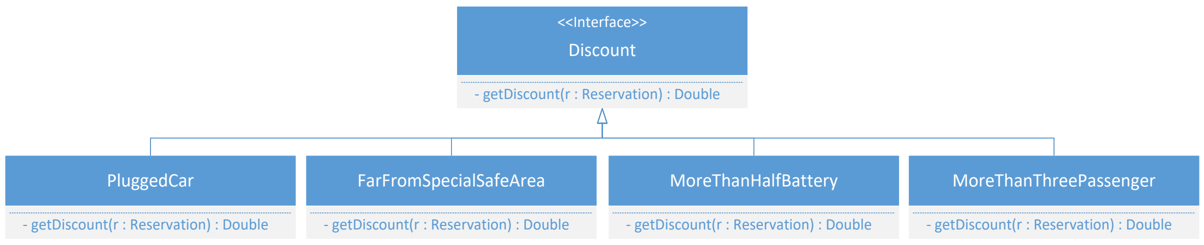
### 2.6.4 Design Patterns

#### *2.6.4.1 Strategy pattern*

Strategy pattern is implemented for Discount calculation.

**Problem:** how to design an ensemble of variable policies and algorithms but correlated? How to design to let adding or removing such policies and algorithms?

**Solution:** define any algorithm and policy in a separate class providing a common interface.



<<Interface>>
Discount
- getDiscount(r : Reservation) : Double
PluggedCar
- getDiscount(r : Reservation) : Double
FarFromSpecialSafeArea
- getDiscount(r : Reservation) : Double
MoreThanHalfBattery
- getDiscount(r : Reservation) : Double
MoreThanThreePassenger
- getDiscount(r : Reservation) : Double

*Figure 16: Strategy Pattern applied to Discount*

#### 2.6.4.2 Singleton pattern

Singleton pattern is implemented in the "MaintenanceServiceController".

**Problem:** it is permitted to run exactly one instance of a class. Other objects have the need for a single global point of access to this object.

**Solution:** define a static method that returns the singleton object. In JEE is implemented and managed by the EJB container with the "@Singleton" annotation through "javax.ejb.Singleton".

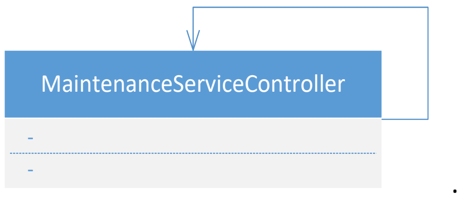


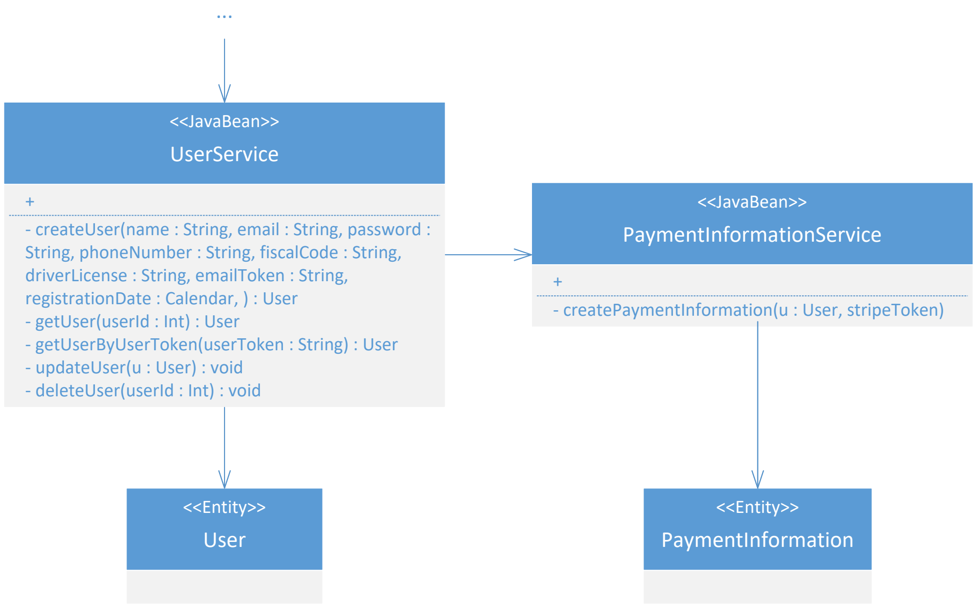
*Figure 17: Singleton pattern applied to MaintenanceServiceController in JEE*

#### 2.6.4.3 Service Layer pattern

Service pattern is implemented to simplify model access and manipulation.

**Problem:** how to provide an interface that let manipulate a set of model objects? How to decouple the model implementation with the business logic that handles the process?

**Solution:** define a separate class from the model that provides a higher-level interface, implements more complex model manipulation and manages model instances.



...
<<JavaBean>>
UserService
+
- createUser(name : String, email : String, password : String, phoneNumber : String, fiscalCode : String, driverLicense : String, emailToken : String, registrationDate : Calendar, ) : User
- getUser(userId : Int) : User
- getUserByUserToken(userToken : String) : User
- updateUser(u : User) : void
- deleteUser(userId : Int) : void
<<JavaBean>>
PaymentInformationService
+
- createPaymentInformation(u : User, stripeToken)
<<Entity>>
User
<<Entity>>
PaymentInformation

*Figure 18: Service Pattern example on the User entity*

## 2.7 Other Design Decisions

PowerEnJoy system relies on push notification to constantly update the user with smartphones, geolocalization to enable localization aware services.

### *Push notification*

We need to send instant notifications to PowerEnJoy App. The chosen technology is Push Notification. We decided to support both iOS, Android and Windows Phone smartphones.

Different software API are available on these platforms to provide such technology:

- **Apple Push Notification Service** for iOS
- **Google Cloud Messaging** for Android
- **Windows Notification Service** for Windows Phone

### 2.7.1 Geolocalization

Google Maps is chosen as the geolocalization service because it is highly reliable, fast and it's cost effective. Also, it offers powerful APIs that will be used to provide map related functionalities such as calculate the best route among two points in the map.

# 3 Algorithm Design

To show both how the strategy pattern will be implemented and the algorithms relevant to the discount calculation, here is provided a snippet of code.

```
public interface Discount {
	public Double getDiscount(r: Reservation);
}

public class PluggedCar implements Discount {
	@Override
	public Double getDiscount(r: Reservation) {
		if (r.getCar().getPlugged())
			return 30;
		else
			return 0;
	}
}

public class FarFromSpecialSafeArea implements Discount {
	@Override
	public Double getDiscount(r: Reservation) {
		Address a = r.getCar().getAddress()
		if ( (!(a.getSafeZone() instanceof Special) and 
(AddressService.getNearestSpecialSafeAreaDistance(a) > 3)) ||
                r.getCar().getBattery() < 20))
                return -30;
            else
                return 0;
        }
}

public class MoreThanHalfBattery implements Discount {
        @Override
        public Double getDiscount(r: Reservation) {
            if (r.getCar().getBattery() > 50)
                return 20;
            else
                return 0;
        }
}

public class MoreThanThreePassenger implements Discount {
        @Override
        public Double getDiscount(r: Reservation) {
            if (r.getRide().getPeopleCarried() > 3)
                return 10;
            else
                return 0;
        }
}
```

Here follows the pseudocode of the function “Double getNearestSpecialSafeAreaDistance(Address address)” contained in AddressService class. It calculates the distance from the given address to the nearest special safe area.

```
get all the SpecialSafeZones
create tempList list
create specialSafeZoneDistances list

for each SpecialSafeZone {
        for each SpecialSafeZone coordinate {
            translate coordinates from String to Double
            calculate distance between address and current coordinate (haversine
formula)
```

```
            put calculated distance in tempList list
        }
        find the minimum distance contained in tempList
        put minimum distance contained in tempList in specialSafeZoneDistances list
        reset tempList list
    }
    return the minimum distance contained in specialSafeZoneDistances list
```

## 3.1 Haversine formula

To calculate the distance the Haversine formula is used. The haversine formula is an equation important in navigation, giving great-circle distances between two points on a sphere from their longitudes and latitudes.

$$d = 2 \cdot r \cdot \arcsin \sqrt{h}$$

Where:

- **d:** distance (spherical distance)
- **r:** radius of Earth
- **h:** haversine function:
  $$hav\left(\frac{d}{r}\right) = \sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + cos(\varphi_1) \cdot cos(\varphi_2) \cdot \sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)$$
  - $\boldsymbol{\varphi_1, \varphi_2}$: latitude of point 1 and latitude of point 2 (radians)
  - $\boldsymbol{\lambda_1, \lambda_2}$: longitude of point 1 and longitude of point 2 (radians)

Here follows an implementation in Java available in [2]:

```
public class Haversine {
    public static final double R = 6372.8; // In kilometers
    public static double haversine(double lat1, double lon1, double lat2, double
lon2) {
        double dLat = Math.toRadians(lat2 - lat1);
        double dLon = Math.toRadians(lon2 - lon1);
        lat1 = Math.toRadians(lat1);
        lat2 = Math.toRadians(lat2);

        double a = Math.pow(Math.sin(dLat / 2),2) + Math.pow(Math.sin(dLon / 2),2)
* Math.cos(lat1) * Math.cos(lat2);
        double c = 2 * Math.asin(Math.sqrt(a));
        return R * c;
    }
    // example of computation of a distance
    public static void main(String[] args) {
        System.out.println(haversine(36.12, -86.67, 33.94, -118.40));
        }
}
```

## 4 User Interface Design

Provide overview of how the user interface of the system will look like, if already in RASD, refer and possibly make extensions if applicable.

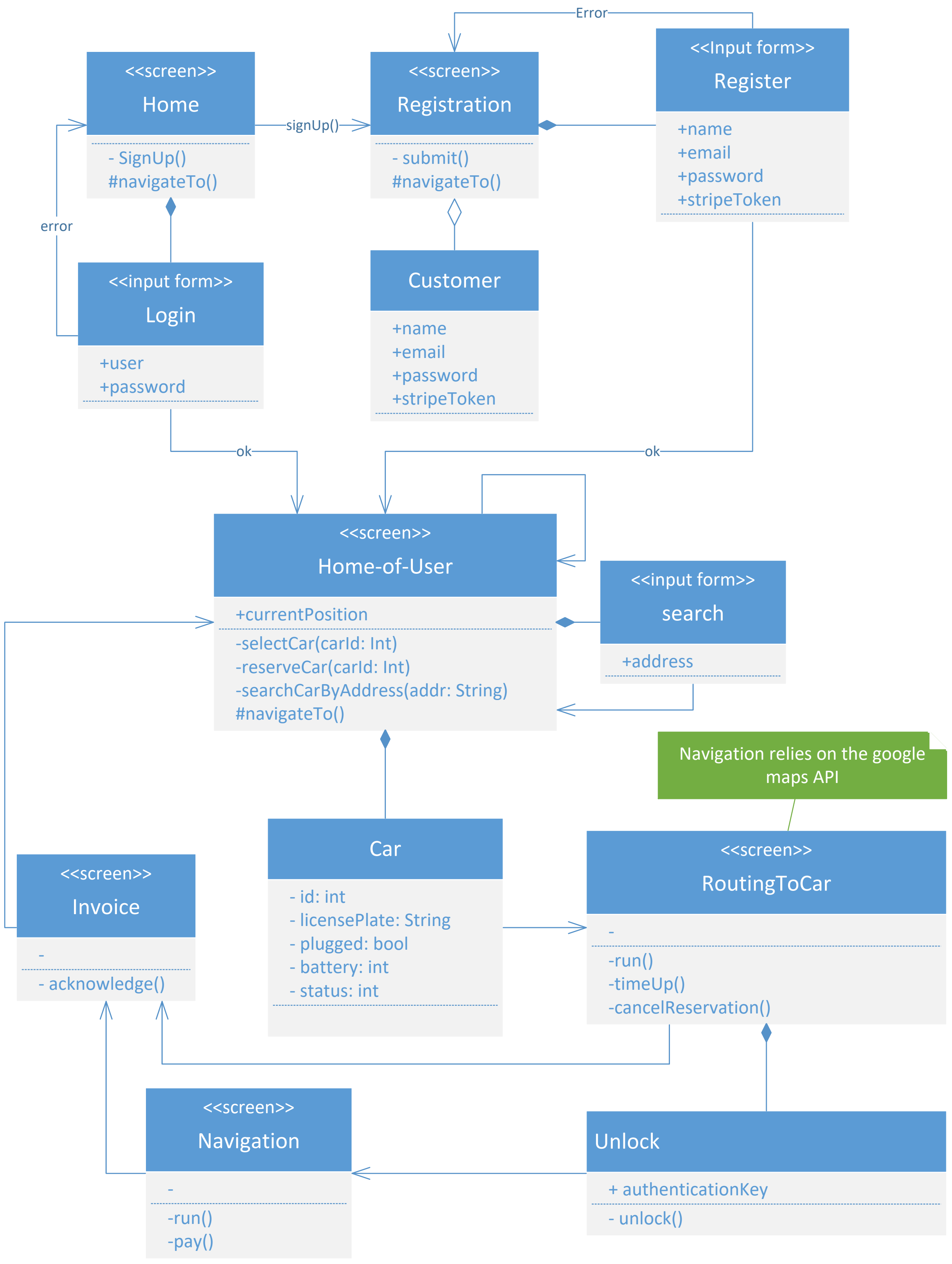


*Figure 19: User interface navigation diagram*

# 5 Requirements Traceability

## 5.1 Functional requirements

All functional requirements expressed in section 2 in the Requirements Document have been analyzed to comprehend which components were necessary and how they had to relate to each other.

In the following table, each requirement is backed up by some model (Entity) and business logic classes already presented in section 2.2. Please refer to each subsection for more information on each class. In addition to this, a sequence diagram is provided where possible.

| Functional requirement | Business logic | Entity | More info in section |
|---|---|---|---|
| **G1:** Allow a guest to register to the platform. | UserController | User | 2.2.1 |
| **G2:** Allow a user to access the platform by logging in. | User controller | User | 2.2.1 |
| **G3:** Allow a user to find an available car. | Car controller | Car | 2.2.2 |
| **G4:** Allow a user to reserve an available car. | Reservation controller | Car, Reservation, User | 2.2.3 |
| **G5:** Allow a user to cancel his reservation. | Reservation controller | Car, Reservation, User | 2.2.3 |
| **G6:** Allow a user to unlock a reserved car. | PowerEnJoy Box | Car, User, Reservation, Ride | 2.4.4 |
| **G7:** Allow a user to pick the car up. | Ride controller | Ride | 2.2.5 |
| **G8:** Allow a user to pay for the ride. | Payment controller | Reservation | 2.2.4 |
| **G9:** Allow a user to park in a safe area. | CarController AddressService | - | 2.2.2 |
| **G10:** Allow the user to plug the car into the grid. | -No Logic Required- | Car | RASD - Specific Requirement |

## 5.2 User Interface

Several mockups were already created and inserted into the RASD document; since no new mockups were added, please refer to *RASD* document *User interfaces* section. However, a UX diagrams have been created to describe in detail how a MobileApp works. For more information, see section 4.

# 6 Effort Spent

Table describing the time management for the team.

| Team member | Hours |
|---|---|
| **Flavio Primo** | 35 |
| **Hootan Haji Manoochehri** | 35 |
|  | 70 total |

# 7 References

- *Patterns of Enterprise Application Architecture* by Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, Randy Stafford
- *Stripe API* - https://stripe.com/docs/api

- *Service-Oriented Architecture Definition* - http://www.service-architecture.com/articles/web-services/service-oriented_architecture_soa_definition.html
- *Haversine formula implementation* - https://rosettacode.org/wiki/Haversine_formula#Java

# 8 Appendix

## 8.1 Software used

The following software were used to produce this document:

- **Microsoft Word 2016:** as the main word processor application
- **Microsoft Visio 2016:** for UML modelling and all the diagrams