Politecnico di Milano

A.A 2016-2017

Design Document

Version 1.0

PowerEnjoy

Instructor : Prof. Di Nitto

Authors:
Amico Simone
Chianella Claudia Beatrice
Giovanakis Yannick

CONTENTS

1. INTRODUCTION

1.1 Purpose

The presented document is the Software Design document (SDD) for the PowerEnJoy platform project. The main purpose of this document is to be a guideline for the concrete implementation of the platform, provide developers with high level descriptions of the main algorithms, describe the architectural styles and patter, and generally establish the design standards for the development phase.
This document is intended for stakeholders, software engineers, and programmers and must be used as reference throughout the whole development of the system. The secondary audience for this document includes system maintainers and developers who wish to integrate the platforms services within their own software. The design choices listed here must be respected throughout the whole development of the platform and any other further expansion of the codebase.

1.2 Scope

The descriptions listed in this document define the design language that must be used by design stakeholders, and implement the Design leads to code philosophy. These design standards represent constraints to the development of the codebase, with all unspecified design decisions left to the developers. Key reasons for the design language described in this document are:

1. To facilitate the development, integration, expansion and maintenance of the platform.

2. To define a business identity (kept consistent in UX and marketing design).

3. To implement the requirements listed in the RASD of the project in a consistent way.

1.3 Definitions,Acronyms,Abbreviations

Throughout this document, the following definitions will be applied without further explanations:

- **Platform**: the set of software applications and hardware infrastructure that are part of the PowerEnjoy service. The platform includes:
  - Back-End Server application
  - Web Application
  - Mobile Application
  - On-board Display
  - MySQL Database

  Other third party software may be necessary to interface different components or support the listed applications.
- **System**: any individual component of the platform.
- **Back-End**:the software run on the back-end server of the platform which is used to handle the communication between the user applications. The term also addresses all the necessary software components that are needed to store data, perform calculations and manage the hardware (e.g. an operating system).
- **User Application**: set of applications that are used by a user which are the Mobile Application ,Web application and On-Board display application. [1]
- **User**: any person registered and authorized to use the above mentioned systems.
- **MySQL**:is an open-source relational database management system (RDBMS).

In addition this document will contain the following acronyms:

- **RASD**: Requirements and Analysis Specifications Document

[1] For more informations check Section 2.4.2 on the *RASD*

- **DD**: Design Document
- **DBMS** : Data Base Management Systems
- **UX**: User Experience
- **API**: Application Programming Interface
- **CRUD**: Create, Read, Update, Delete
- **UI**: User Interface
- **RESTful or REST**: REpresentational State Transfer
- **HTTPS**: HyperText Transfer Protocol over Secure Socket Layer

1.4 Reference Documents

- IEEE 1016-2009: "Software design description"
- Project description: Assignments AA 2016-2017.pdf
- UML Language Reference : `https://www.utdallas.edu/œchung/Fujitsu/UML_2.0/Rumbaugh--UML_2.0_Reference_CD.pdf`
- Google Maps API:`https://developers.google.com/maps/`
- GlassFish : `https://glassfish.java.net/docs/4.0/quick-start-guide.pdf`

1.5 Document Structure

The presented DD is divided in sections and structured as follows:

- Section 1 - Introduction: contains support information to better understand the presented document.
- Section 2 - Architectural design: contains a description of the architectural styles and patterns selected for the platform, which serve as an implementation guideline for developers.
- Section 3 - Algorithms design: contains a high-level description of the core algorithms of the back-end.

- Section 4- User interface design: contains a description and a conceptual preview of the user interface and UX.
- Section 5 - Requirements traceability: links the decisions described in this document to the requirements specified in the RASD.
- Section 6 - Effort spent: contains a summary of the hours spent in producing the document.

2. Architectural Design

2.1 Overview

An essential part of the platform design is to satisfy not only functional requirements but non-functional requirements as well.This means that architectural choices are crucial at this point of the development for the sake of achieving the desired output in terms of requirements ,performance ,scalability and user experience.
An important focus will be set on how the different systems interact with each other and the back-end to obtain the events as required by sec. 3 of the RASD.Furthermore choices regarding up scaling and redundancy will be discussed.

2.2 Component View

This section focuses on the component overview giving an insight on their core functionality and various interfaces.For more detailed information about component interfaces see *section 2.5*
A first classification of components can be made at a high level:

- **Server** provides the core functionality of the platform. The server incorporates the biggest part of the *business logic* and stores some *data* locally.

- **User Client** provides a high level representation of the real user clients. It is considered a *thin client* as it leaves most of the functionalities to the Server.
- **On-Board Client** provides navigation functionalities done exclusively client-side , whilst other functionalities are left to the server side (like authentication).

As mentioned above the platform is designed using the *client-server*paradigm. Component interactions are handled by the server which is able to receive calls from the clients . Client interaction is never done peer to peer but via server.

2.2.1 Server

The server is composed of :

Back-End Application

The Back-End Application is the component that handles most of the business logic. The application is written in Java EE and to fulfil its tasks (see section 3 of the RASD) it needs to interface with the Internet network using the HTTPS protocol and the JAVA API for RESTful Web Service, with a MySQL database and GooleMaps API.

Back-End internal components

- **User manager**
  This component handles user data, registration and authentication.The user manager has direct access to the DBMS and receives read and write requests from the RESTful API.
- **Notification manager**
  This component is responsible for implementing the push notification service towards the user-side applications. It is necessary to have this type of module running in the back-end because there are

cases in which a communication must happen between the clients and the back-end, but no direct request is made to the RESTful API by the clients: for example when the reserved car is nearby, the system must send the user a 'ready to unlock' notification.

- **Vehicle manager**
  The car manager component's job is to manage all the vehicles throughout the city.It can access the DBMS to query vehicle information but stores essential vehicle information locally such as location ,battery level and number of seats.Car positions are managed through the Position manager.

- **Position manager**
  Keeps track of car positions around the city. When the Search manager gets triggered it forwards requests to the Position manager to get all available cars in compliance to the user input. It updates car positions through the vehicle manager at the end of each ride through the Vehicle manager. Moreover it handles the fair car distribution scenario when triggered by the on-board display.

- **Search manager**
  The search manager component handles all incoming search requests forwarded by the user applications and interfaced through the RESTful API. The main functionality is to handle the user input and query the vehicle manager accordingly, returning the desired output to the user.
  In the event of a booking request the search manager forwards the demand to the Ride Manager.

- **Ride manager**
  The Ride manager component examines all incoming reservation request and creates *Ride objects* according to the user selection and user data.The ride manager is connected to the following components:

  - *Vehicle manager*: need to communicate and receive communications about car status changes in terms of availability, battery level and location.
  - *Notification manager*: need to inform the user about remaining time to unlock the car or send the user 'ready to unlock ' notifications. Moreover it sends messages to the on-board display to

inform the user about the response to a *end ride request*.

Data Base

The MySQL database fulfills the task of storing and granting access to all the data generated and used by the service.
The connection between the Java EE application and the Data Base is supported by the *JDBC connector* [2].For more information about the data base see *section 2.7*.

2.2.2 User Client

As stated in section 2.4.2 of the RASD a native mobile application is developed for Android, iOS and Windows Phone as well as Web application for the main browsers. To fulfil the requirements expressed in section 3 of the RASD, all the clients need to communicate with the Server making calls to the REST API using platform specific API for REST HTTP calls.

2.2.3 On-Board Client

The on-board client needs to provide navigation functionality without interacting with the PowerEnjoy Server but directly through GoogleMaps API. At the beginning of each ride an authentication via PinCode is requested and verified server side.
Safe Areas are stored locally on the client and updated only when the changes on the DB occur.On the other hand charging stations need to be updated in real-time as it can happen that charging stations are currently out of capacity.

[2] `http://dev.mysql.com/downloads/connector/j/`

2.2.4 Diagrams

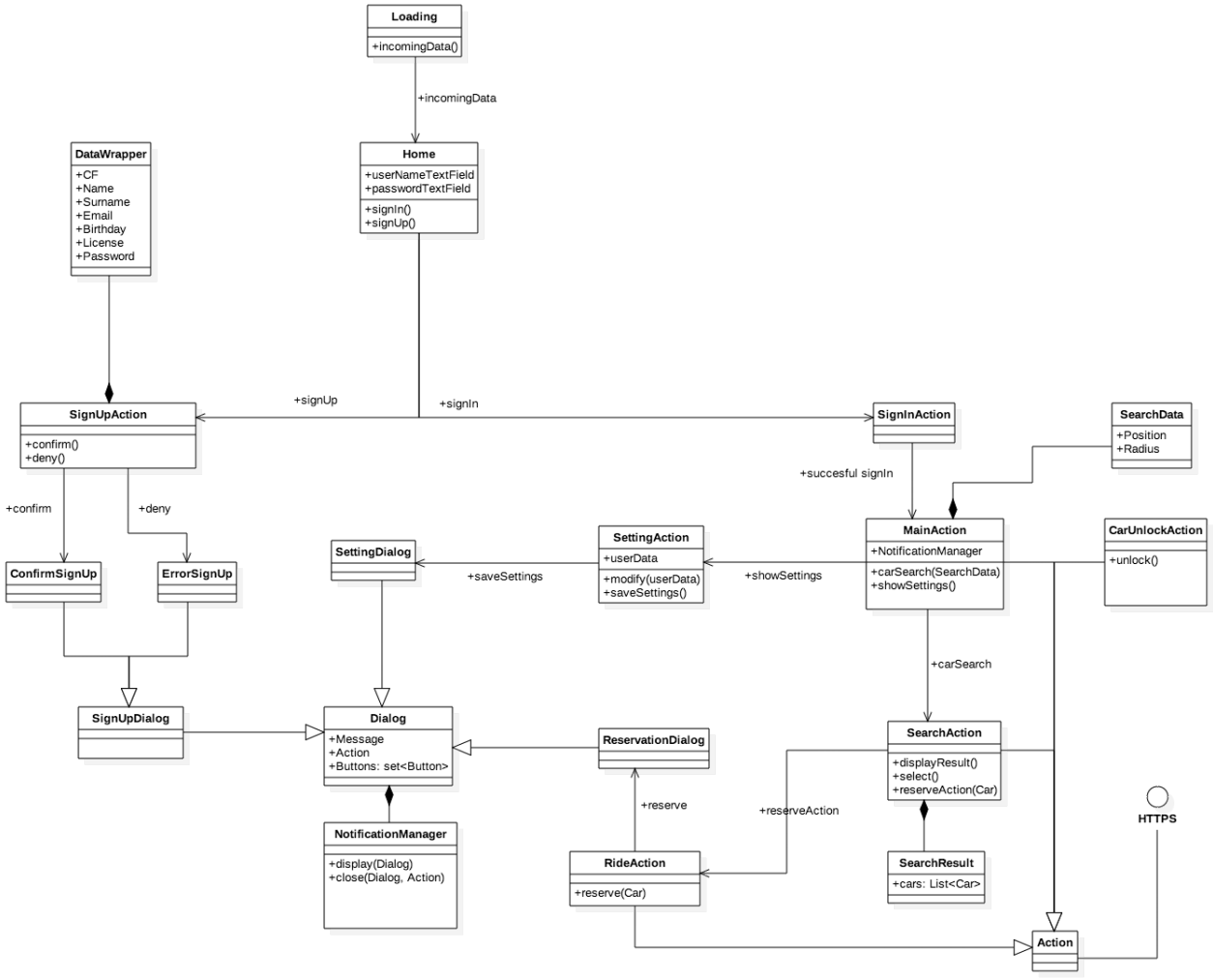


Figure 1: User Application Class Diagram

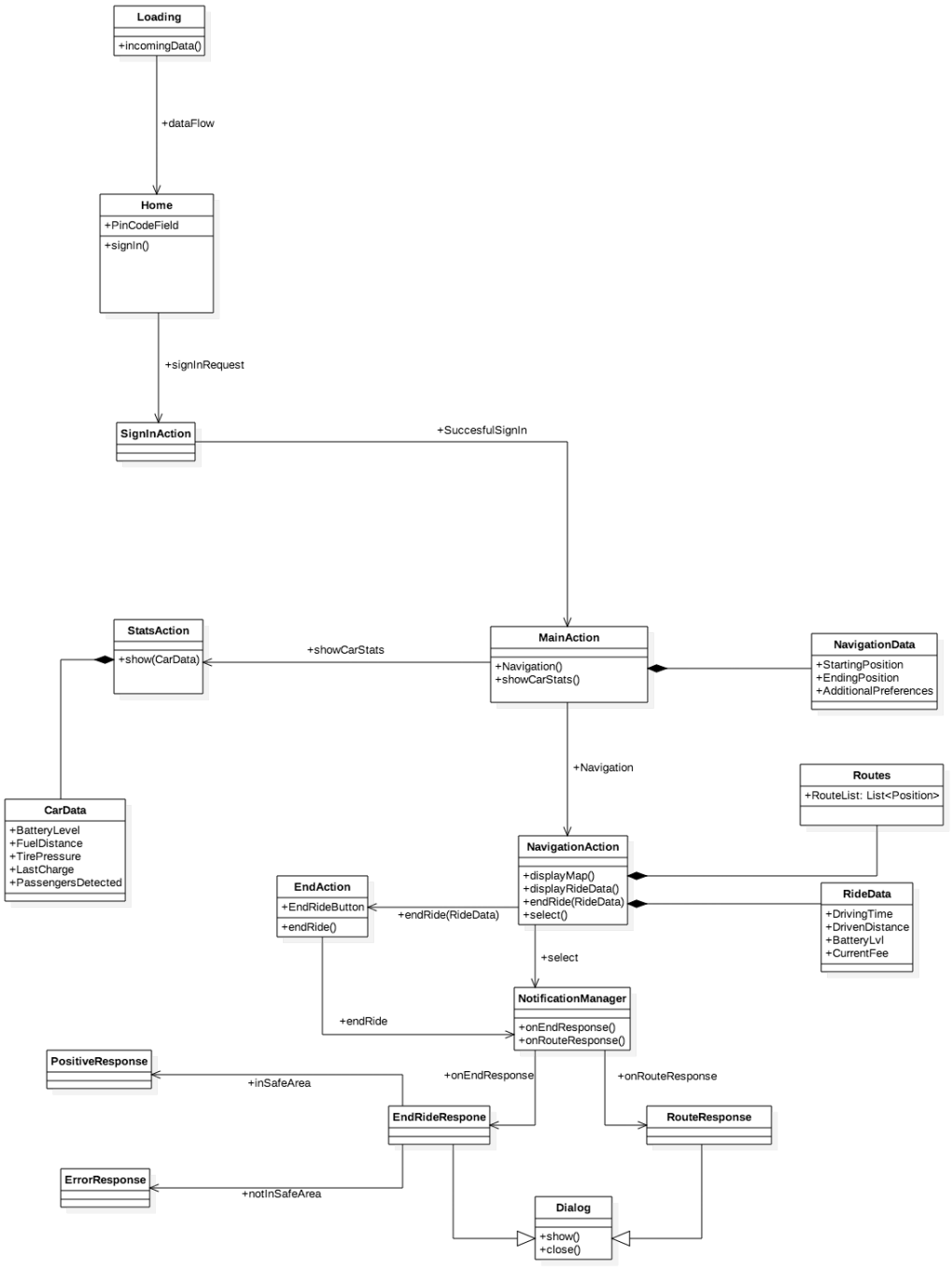


Figure 2: On-board Class Diagram

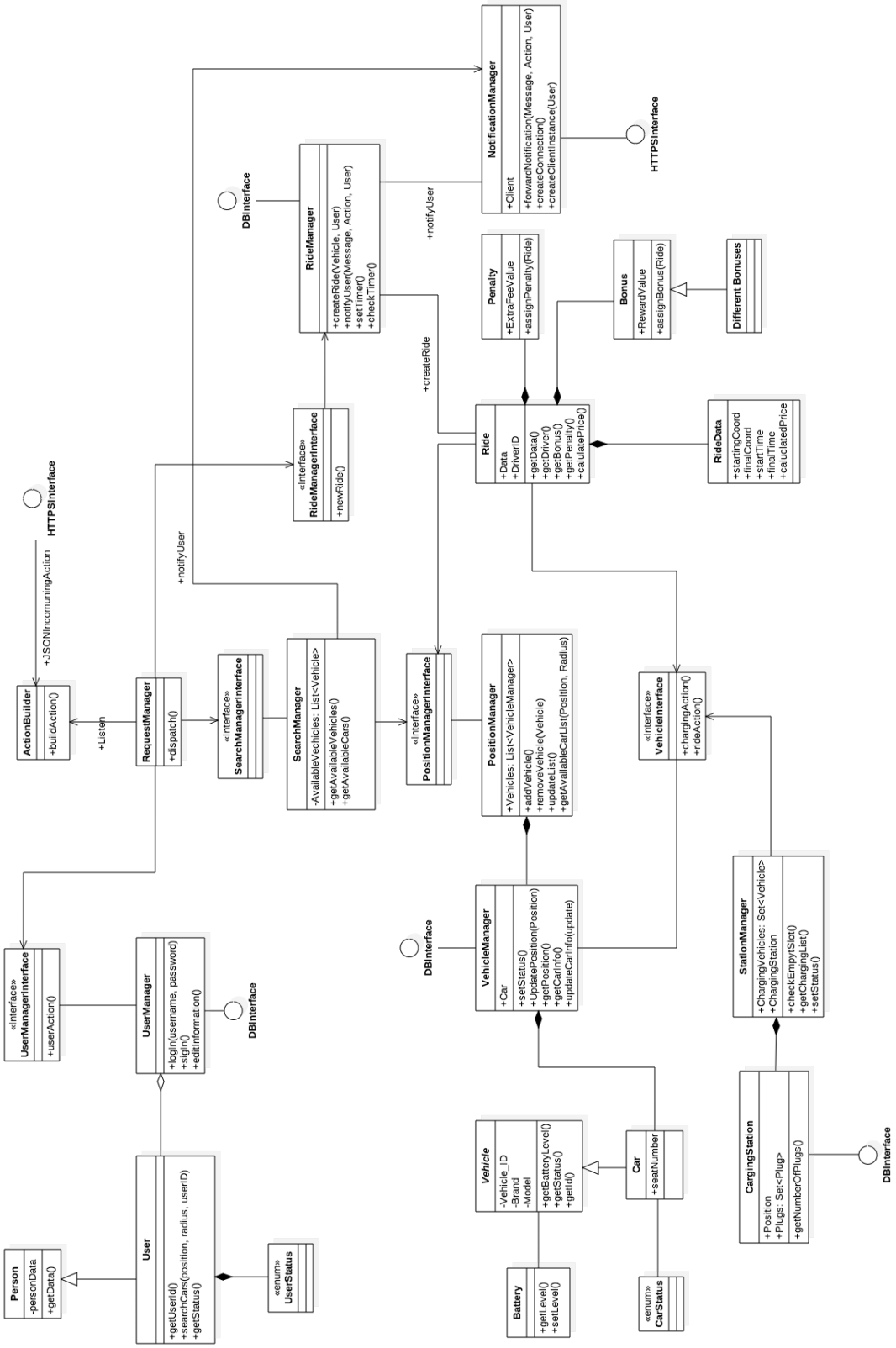


Figure 3: Back-End Class Diagram

2.3 Deployment View

The following section contains a schema of the hardware infrastructure. It was decided to omit the representation of the purely network-related hardware (routers, switches, etc.). The chosen architecture is a typical web-based client-server architecture, with the following components:

1. **Primary and BackUp Storage**: the Primary DB is a long term storage system on which the operational database is saved. This node provides the highest Read/Write speeds and is used in normal operational regime.The BackUP DB is a long term storage system with high reliability property. It is mainly used to store data backups and becomes fully operational in the event of total failure of the Primary RDBMS or DBMS nodes.

2. **Primary RDBMS Server**: server dedicated to run the DBMS software on account of the main server and to manage the communication with the secondary storage node. This node is connected to the main server, from which it receives requests, and the storage nodes (either primary or secondary - the connection is transparent to the node and is hot-swappable in the event of a failure).

3. **Secondary RDBMS Server**:server dedicated to run the DBMS software and manage the backup requests from the primary DBMS server. The node is connected to the secondary storage and to the primary DBMS server, for the reasons described above.

4. **Main Server**: a typical rack of servers dedicated to run the main back-end application.The rack is composed by independent nodes managed by usual load balancing technology and is able to be fully functional under average load of the system.The redundancy of the rack has the dual function of improving fault tolerance and performance under above-average load conditions.

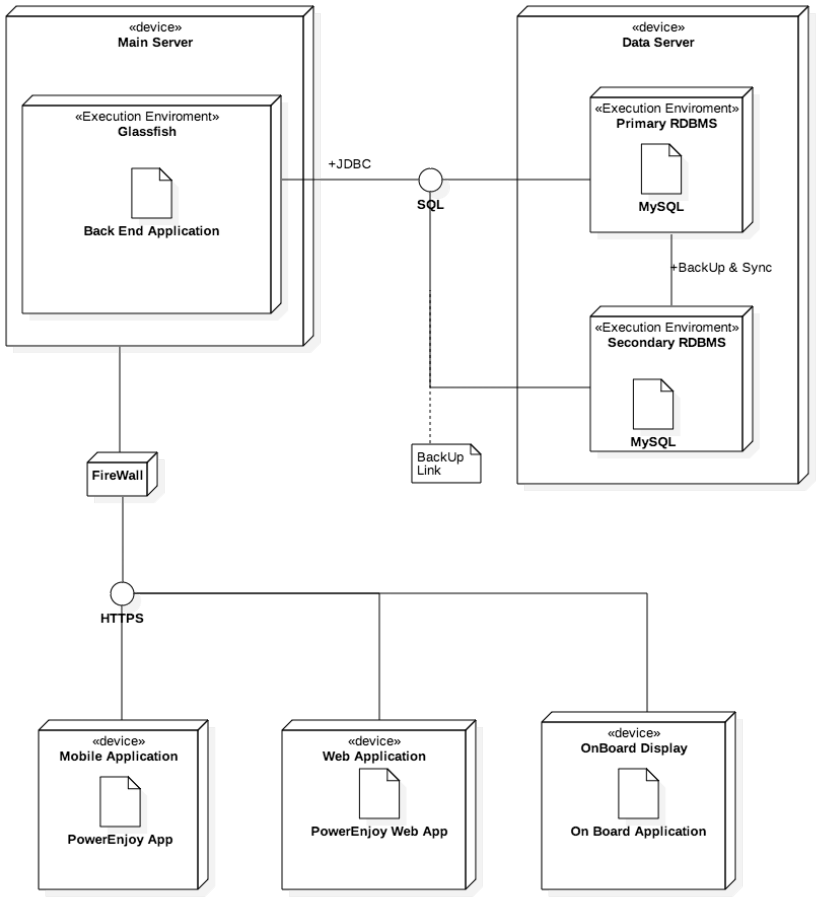


Figure 4: Deployment Diagram

2.4 Runtime View

2.4.1 Server View

This section focuses on the Runtime View of the system. While the system is up and running, the *Server* receives many *HTTP requests* from different clients that are handled by a *Load Balancing* component that distributes the calls evenly to every real machine client. Every search request is not stored in the DB but is stored locally for processing purposes. Ride request on the other hand are stored on the MySQL databases as it is necessary to keep track of user rides. From the internal point of view of the Back-End application, user requests (search, ride or data request) are at first parsed by the Request Manager and the dispatched to Java object that is in charge of computing the result of that request.

2.4.2 Client View

From the client point of view, when a User opens his application, the client starts a first ”handshake” to check for basic authentication data and if it’s successful, the client can proceed with requests. The flow of a request start from a User Interface component (like a button) and is finally handled by the class that implements the HTTP interface.
When a User enters a car a similar process is used but a User pin code has to be entered for authentication purposes.

2.4.3 Diagrams

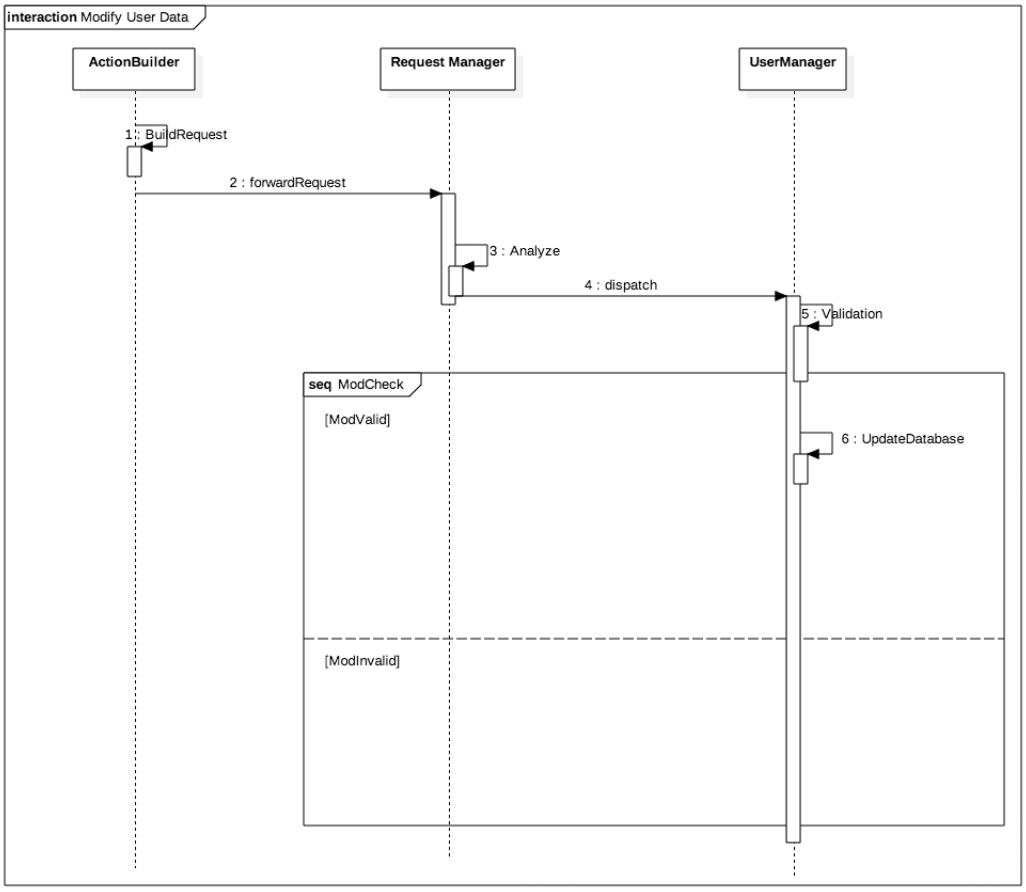


Figure 5: Data modification sequence diagram

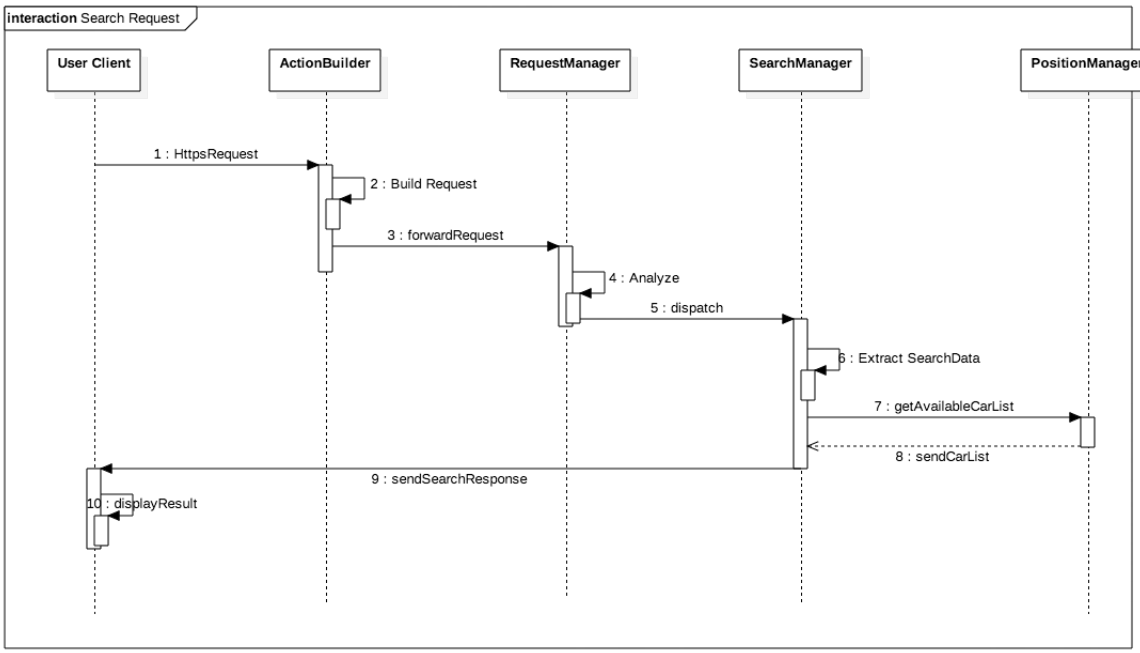


Figure 6: Search action sequence diagram

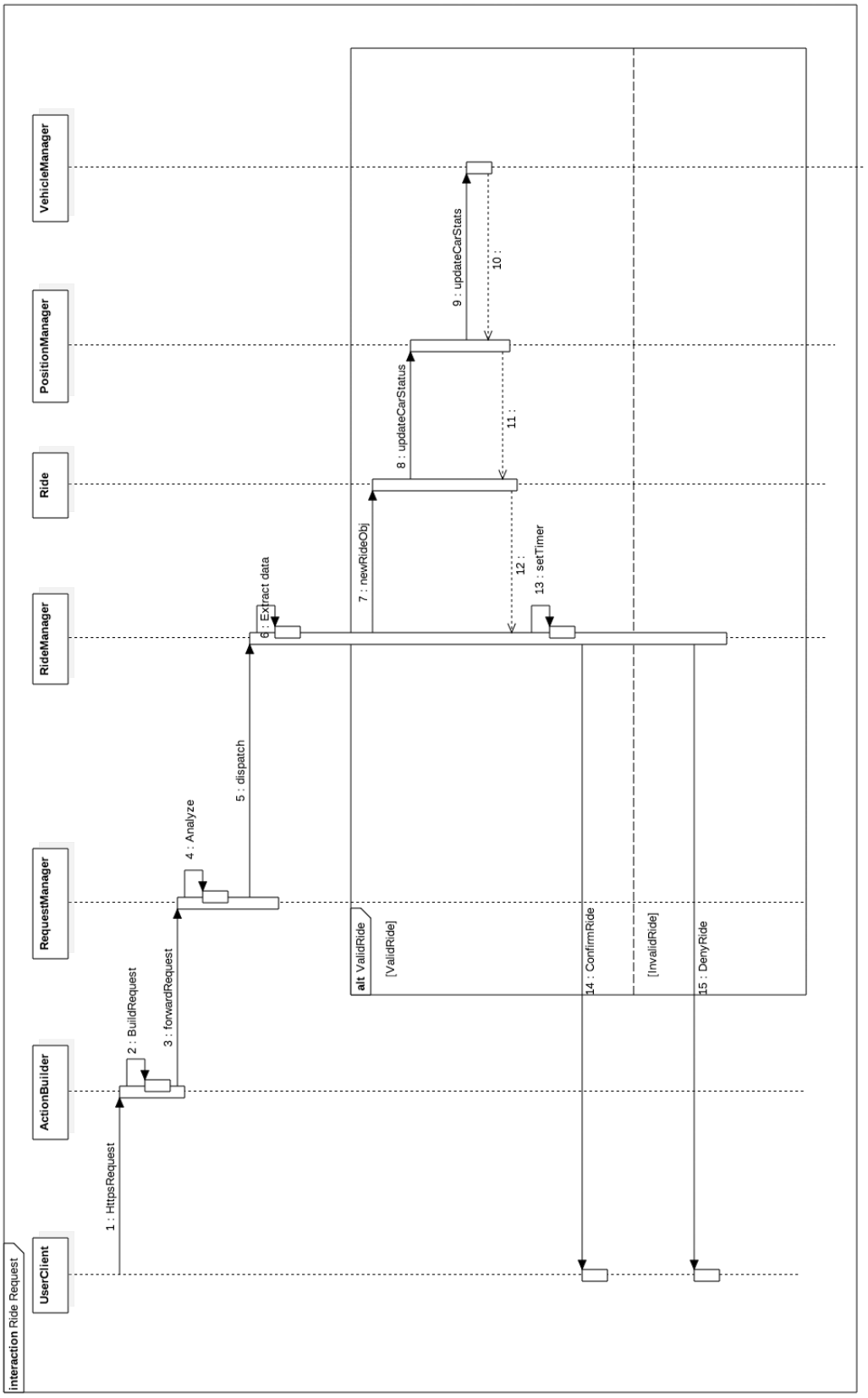


Figure 7: Ride action sequence diagram

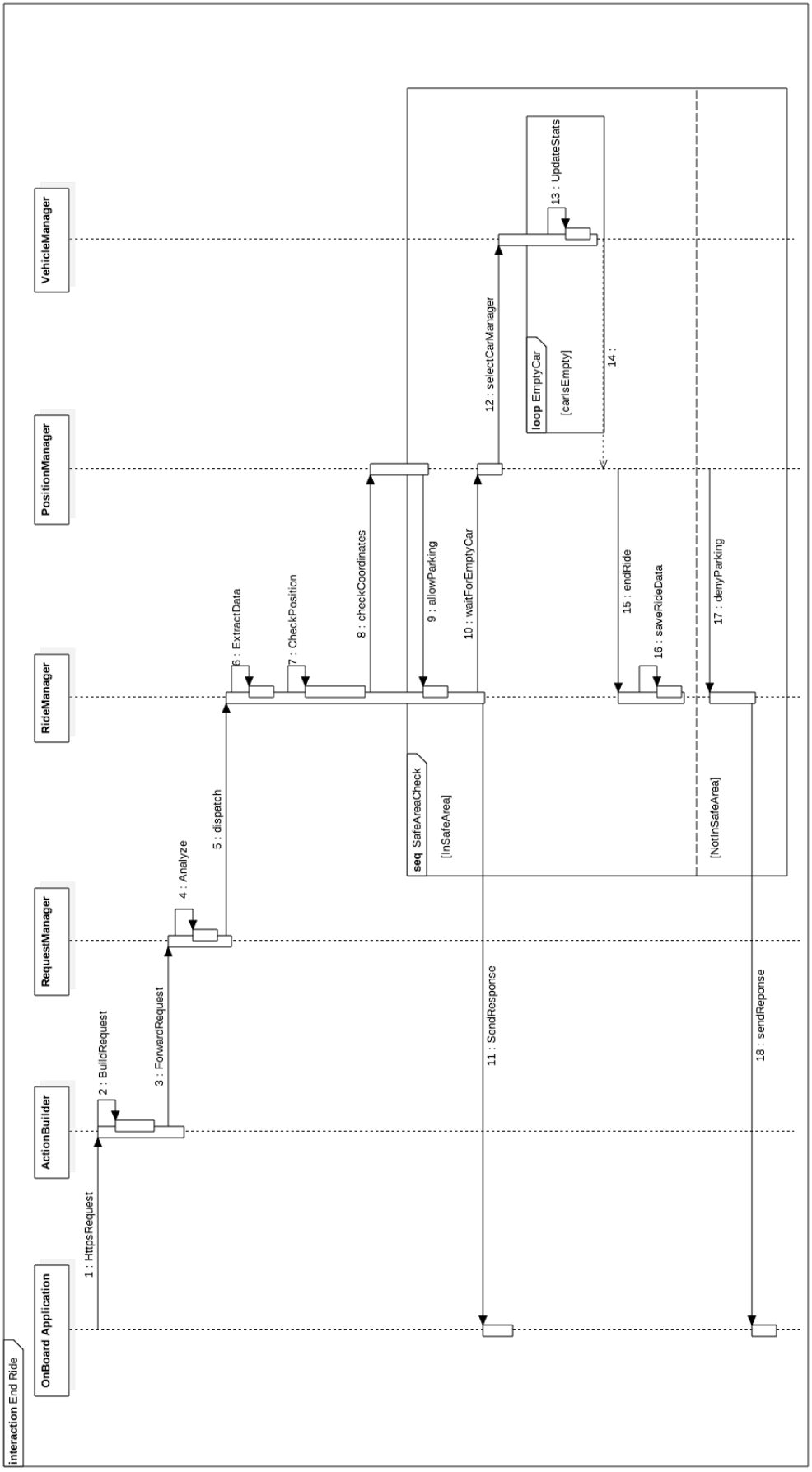


Figure 8: End ride sequence diagram

2.5 Component Interfaces

This section provides a description of the interfaces between the main components of the system.

Server interaction

- **Back-End application → Database**
  The *back-end application* uses SQL language to query the *database*. Queries from Java are supported by the *Java Database Connectivity (JDBC) API* [3].JDBC comes extremely handy thanks to it powerful *"Write once, Run Everywhere "*feature.

- **Back-End→ Client Application**
  The connection between the *Back-End* and the *Client application* is managed by the internet network based on a HTTPS protocol and supported by the RESTful API service.To add further security to the data flow an intermediate Firewall is deployed.

- **Back-End → On-Board Application**
  The connection between the *Back-End* and the *On-board application* is managed by the internet network based on a HTTPS protocol and supported by the RESTful API service.To add further security to the data flow an intermediate Firewall is deployed.

RESTful API
The RESTful API is a gateway communicator between clients and the back-end system: it's a stateless service which provides methods for data submission or requests returning the requested computations as a result.
RESTful API are an optimal solution for an application that must handle a vast number of users on a different number of platforms as it allows to guarantee the same user experience on all platforms. Furthermore the architectural properties positively affected by the constraints of the REST architectural include many important quality requirements such as performance, scalability and reliability.

[3] `http://www.oracle.com/technetwork/java/javase/jdbc/index.html`

JDBC
JDBC is a Java API that manages the access of a generic client to a database. In our platform, it acts as interface between the *back-end application* and the DBMS component (on both the primary and secondary nodes).

2.6 Selected architectural styles and patterns

This section highlights both hardware and software selected styles and design patterns.The following patterns are meant to be guidelines for developers and may not reflect entirely the actual pattern adopted as different choices can be made during development phase.

2.6.1 Software patterns

The overall software system must follow the *Object Oriented Paradigm*. In particular, developers should follow the principles of *encapsulation, composition, inheritance, delegation and polymorphism*.
Developers should promote code reuse and should try to solve programming problems using common OO Design Patterns[4] Code produced by developers, must be fully commented and documented in order to promote simple refactoring and maintenance.
Some patterns used in the description of the architecture are considered essential:

- **Publish/Subscribe**:The publish/subscribe design pattern is used to interface the Ride Manager with the Notification manager, where the latter is the subscriber and must react to events suitably generated by the former, which is the publisher.

- **Push**: Push technology is an implementation of the publish/subscribe design pattern, which is widely used in modern mobile applications to manage notification services.
  In our platform, requests are generated by the Notification manager and picked up by the user-side mobile applications, which react to them by displaying the suitable notifications on the users devices.

[4] *Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides*

2.6.2 Hardware patterns

As outlined in the *deployment view* 2.3 the platform is obviously designed as a client-server architecture, in which the back-end nodes are the server and the various users' devices are the clients.
The system is based on a *three tier architecture*:

- **First Tier**: composed of client devices (user application and on-board application).
- **Second Tier**:composed of a rack of servers.
- **Third Tier**: composed of database servers controlled by load balancing and storage units.

We chose to deploy each logical layer on a physical tier :

- **Presentation Layer**: is the layer responsible for displaying data to the users and for transmitting input data to the business logic layer. The presentation layer is deployed in the First Tier.
- **Logic Layer**:This layer is responsible for receiving data from the presentation layer, for computing and transmitting a response(using also data provided by the data store layer) to the presentation layer.This layer implements most of the business logic and is deployed in the second tier.
- **Data Storage Layer**:This layer is responsible for storing all the data meaningful to the system. This layer is deployed in the third tier.

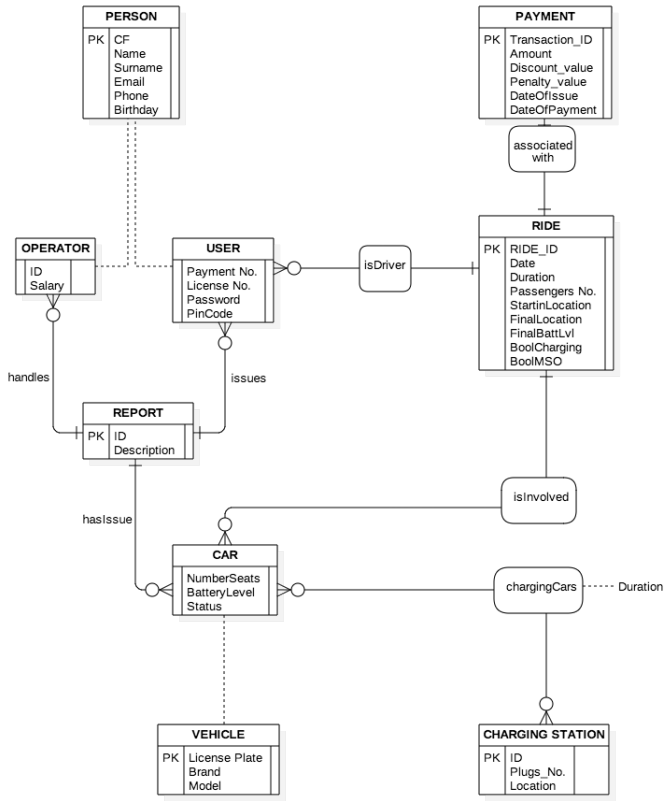
2.7 Data Management view

This section focuses on how the data is stored and structured.

2.7.1 Storing policy

Data storage about people or physical belongings to PowerEnjoy are never automatically eliminated from the DB. On the other hand *Rides* older than 2 years are automatically eliminated from the DB to save storage space and speed up search queries.
In order to reduce the load on the Server and to speed up users query response, all the data that does not change frequently (like the users profile data) is saved locally on the device and reloaded only when a modification of the profile occurs.

2.7.2 Entity-Relation Diagram



The above shown figure represents the structure of the database. For better readability only vital entities are represented while others (like employees or managers) are omitted.

- *Person*: is the generalisation of the operator and user entities. It holds common information and is identified by a *Codice Fisicale*. All attributes must be different from null.

- *Operator*: is a person with a *Salary* and an *ID*. Operators can handle 0 or more Reports.

- *User*: have an encoded *password* and a valid *license number* which must be provided together with other personal data during the registration process.The *payment number* attribute can hold the null value initially. The *PinCode* is a 4 digit number generated randomly during the registration process. Users can *issue* reports (0 ore more) and can be drivers of cars (0 or more)

- *Report*: identified by an *ID* and provided with a description attribute. Each report is associated with one Operator and one User.If the report is about a car it can hold a reference to said car (*hasIssue* has cardinality 0,1).

- *Vehicle*: identified by a *License Plate*. Each vehicle has also a *Brand* and *Model* attribute.It is the generalisation of the *Car* entity. A generalisation is useful as future implementations can include other means of transport.

- *Car*: *status* holds a boolean value to show if the car is available or under maintenance.Available cars can be involved in *Rides* or can be *charging*.A car can be involved in 0 or more rides , and can be listed in 0 or more *chargingCar* relations.

- *Charging Station*: identified by an *ID*. Each charging station can be listed in 0 or more *chargingCars*relations.

- *Ride*: identified by an *RideID*. Important attributes are *Passenger No.* which determine eventual bonuses or penalties. Each Ride has exactly one car, one user and one payment. *Payment*: holds an *Transaction ID* attribute as identifier.The amount, discount and penalty val-

ues are saved. The value of *Amount* must be greater than 1 , while the others greater than 0. A payment is involved in exactly one ride.

3. ALGORITHM DESIGN

3.1 Car Search

The Search and Position Manager provide all the methods to handle incoming search queries. The incoming HTTPS request is dispatched by the request manager which has to decide the manager it needs to forward the request.

- **carSearch(SearchData)**: the user is asked to input a desired location and radius within which the search takes place.The resulting data will be collected inside a *Search Data object* which will be forwarded inside a *SearchAction* to the *Back-End* through the *carSearch()* method.
- **getAvailableVehicles()**: after the *back-end* identifies correctly the request through the *request manager*, the *Search Manager* extracts the data inside the request and queries the *Position Manager* for the available cars.
  Through the *Notification Manager* the available car positions will be send back to the *client-side* who will then take car to display the results.

3.2 Car Reservation

After a *Search* has been made the *User* selects the *vehicle* he desires to reserve. The *selected car* will be encapsulated inside a *Ride Action object* which will be send to the *back-end*.Once the *Ride request* has been dispatched correctly the *Ride Manager* creates a *Ride Object* and starts the *Timer*.
The *Ride Manager* listens in the background to check to *user position*. Once the user is near the desired car , a *unlock-notification* will be send to the *user-client*.

- **reserveCar(Car car)**: encapsulates the relevant *car data* inside a *RideAction* and forwards it to the *back-end*.

- **unlockCar()**: once a notification has been received *client-side* , this method trigged through an UI Component (like a button) unlocks the car and stops the timer.

3.3 Ride & Park

The ride and park action are handled mainly by the *Ride Manager*.The following actions can be performed only after a successful *unlock-action*

- **authenticate(User user, int pinCode)** : the *On-Board application*requires to input the user code to begin the ride.If the *PinCode* corresponds to the one assigned to the User,the authentication process is completed and the user can start to use the *on-board application*.
- **startRide(NavigationData,Route)**:once the user has selected one of the *routes* suggested by the system through *Google Maps API* queries, a *RideData Object*is created to keep track of time, distance, battery level and bonuses. During the route selection process the *Money Saving Option* can be enable through the toggle of an UI Component Button.Car sensors check for passengers in car which will be added as *statistic* and later considered during bonus assignment.
- **parkCar()**: if the car is *standing still* and in *parking modus* , the *on-board display* allows to *end the ride* via an UI Component (like a button).
  Pressing the button triggers the *parkCar()*method which will send a request to the *back-end application* to check if the car is parked in a *Safe Area*.
- **lockCar(RideData rideData)**:once the *parkCar()* method has successfully send a positive response to the *on-board application* the system waits for the user to leave the car and close the doors.
  If the car is empty and the doors are closed a signal-message is send to the *back-end application* to *lock the doors* and *update the car status*.
  Subsequently the *RideData* will be forwarded to the *back-end application* that updates the *Ride object* with the final data and calculates the bonuses and final fee.

3.4 Low Level Algorithm Description

3.4.1 Search

```
/* performed inside the Search Manager */
getAvailableCars(SearchData data){
   List<VehicleManager> nearbyCars = new ArrayList<VehicleManager>;
   Position p = data.getPosition();
   Radius r= data.getRadius();
   nearbyCars=queryPosManger(p,r);
   notificationMangager.send(data.getUserDevice);

}

/* performed by the Position Manager */
getAvailableCarList(Position p,Radius r){
   List<VehicleManager> nearbyCars= new ArrayList<VehicleManager>;
   for(VehicleManager v: Vehicles){
      if (distance(v.position,p)<r)
      nerabyCars.add(v);
   }
}

/* performed client-side after receiving the notification */
...
navigationAction.displayMap(cars)
...

displayMap(List<VehicleManager> cars){
   Map map = GoogleMaps.getMap();
   map.render();
   for(VehicleManager c: cars){
      Position p=c.getPosition();
      mapIndicator = new MapIndicator(p);
      mapIndicator.render();
   }
}
```

3.4.2 Reserve

```
/* performed client-side */
reserveCar(Car car){
   action=new RideAction(encapusulateCarData(car));
   sendRequest(action);
}


/* performed server-side inside the Ride Manager*/

createRide(RideData data){
   if(data.isConsistent()){
      ride= new Ride();
      /* all ride data added battery status,position,car...*/
      posManager.updateStatus(data.getCar(),newBusyCarStatus));
      ride.startTimer();
      notificationManager.send(Message,data.getUserDevice());
   }
   else notificationManager.send(Message,data.getUserDevice());
}
```

3.4.3 Unlock

```
/* Server-Side inside Ride Manager*/
checkIfReadyToUnlock(Ride r, User u){
   while(!(isTimeUp())){
     if(distance(ride.getCar().getPosition(),u.currentPosition())<0.002){
        notificationManager.send(new
            UnlockAction(...),u.getDevice);
     }
   }
}
```

3.4.4 Authenticate & StarRide

```
/* On-board authentication request */

authenticate(User u,int pinCode){
   request =new AuthenticationRequest(pinCode,u);
   request.encode();
   sendRequest(u.getDevice, request);
   reponse= getInputStream();
   if(response.valid()){
    notificationManager(display(response.message));
    updateCarStatus(new DrivingCarStatus);
   }
   else{
      notificationManager(display(response.message));
   }

}

startRide(Position destination, Preferences pref){
   List<Route> routes = new ArrayList<>();
   if(pref.getMSOFlag()){
      destination = getNewMSOPosition(destination));
   }
   routes.add(GoogleMaps.getPathes(destination,carPosition));
   displayMap();
   displayRoutes(routes);
}

/* The getMSOPostion method is inside navigationAction class and
    forwards data to the back-end.
 */

 getMSOPosition(Position destination){
   sendMSORequest(destination);
   /* the back-end checks within the database for the closest
       charging station and return the position to the on-board
       application */
   reponse=getInputStream();
   return response.getNewDestination();
```

```
}
```

3.4.5 Park & Lock

```
/* on-board side*/
parkCar(){
   sendParkingRequest(rideData);
   response=getInputStream();
   if(response.isPositive());
      {
        notificationManager.display(Message);
          updateCarStatus(new ParkedCarStatus);
      }
   else{
      notificationManager.display(Message);
   }
}

/*Server-side inside the Ride Manager class*/

checkSafeArea(RideData r){
    position=r.getCurrentPostion();
    if(isSafAreaPostion(position);
    /* create response object */
    notificationManager.send(response,r.getCarDevice());
}

/* Car locking is done server-side.As soon as the sensors detect
    an empty car in parked status ,the lockCar mehtod gets
    triggered server-side. */

lockCar(RideData r){
   ride.status = end;
   ride.timer.stop();
   car.lock();
   float fee = ride.calculateFee();
   if (ride.numberPassenger >=2)
      totalFee = ride.fee - 0,1*ride.fee;
```

```
   if (car.batteryLevel> 0,5)
      totalFee = ride.fee - 0,2*ride.fee;
   if (car.isInCharge == true)
      totalFee = ride.fee - 0,3*ride.fee;
   if (car.position is not in powerGrid or car.batteryLevel<0,2)
      totalFee = ride.fee + 0,3*ride.fee;
   car.status = available;
   ride.fee = totalFee;
   return true;
}
```

4. User Interface Design

This section focuses on the user interface point of view using UX UML diagram to specify the possible interactions and the flow of events. Layouts for the applications are described and outlined in section 3.1 of the RASD.

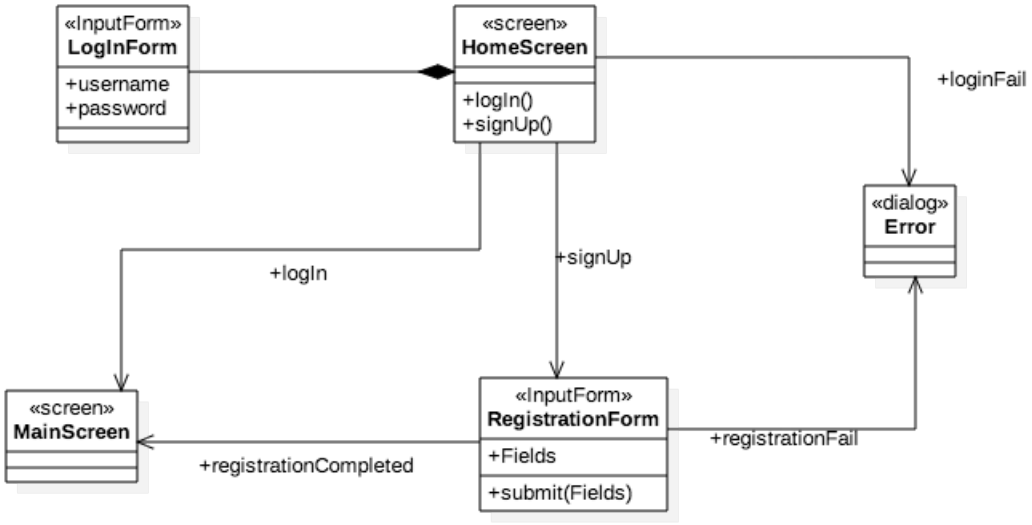


Figure 9: User login screen

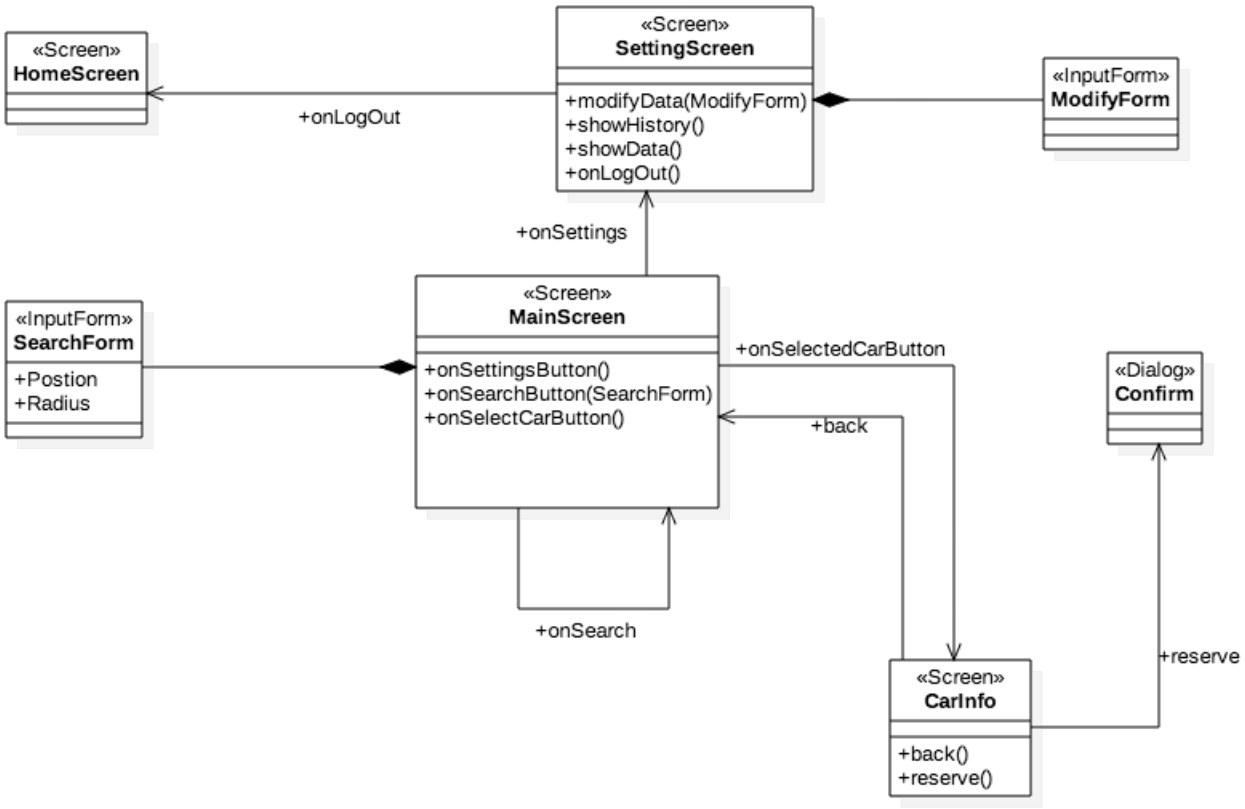


Figure 10: User main screen

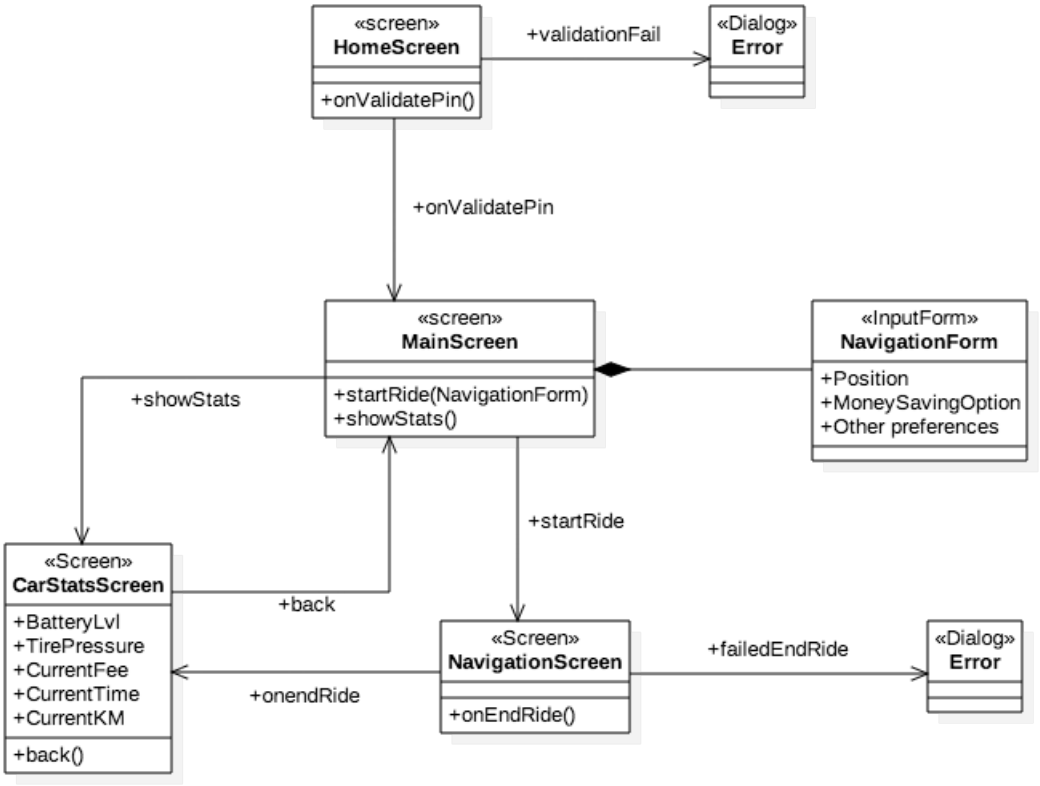


Figure 11: Display screen flow

5. Requirements Traceability

5.1 Mobile and Web application

Interaction between client-server requires always the notification messenger which is omitted.

- **[R1.1]**: Allow a guest to register entering all relevant personal data and a valid driving license whose validity will be check on the spot.
  - SignUpAction()
- **[R1.2]**: Allow a registered user to log-in.
  - SignInAction()
- **[R1.3]**: Allow a registered user to update his personal information.
  - SettingAction()
- **[R1.4]**: Allow a registered user to search for an available vehicle given his position within a selected distance.
  - SearchAction()
- **[R1.5]**: Allow a registered user to search for an available vehicle given an input position within a selected distance.
  - SearchAction()
- **[R1.6]**: Allow a registered user to reserve a selected vehicle.
  - ReserveAction()
- **[R1.7]**: Allow a registered user to check his payment history.
  - SettingAction()

- **[R1.8]**: Allow a registered user to recover his account log-in information in a secure way.
  - SettingAction()
- **[R1.9]**: Allow a registered user to search for an available vehicle given an input position and a radius.
  - SettingAction()
- **[R1.10]**: Allow a registered user to unlock *his/her* reserved car if the user is nearby.
  - UnLockAction()

5.2 On-Board Application

Interaction between client-server requires always the notification messenger which is omitted.

- **[R2.1]**: Enable a registered user to keep track of the distance covered, time spent using the car ,the current owed fee and the remaining battery charge.
  - MainAction -¿ showStats()
- **[R2.3]**: Provide a method to enable the money saving option.
  - MainAction
  - NavigationData
- **[R2.3]**: Enable the driver to end the ride if the car is parked in a safe area.
  - EndAction()

5.3 Back-End System

All client-side request are filtered by the *request manager* which dispatches each request to the corresponding class.

- **[R3.1]**: The back-end must provide an interface for the registration of users.
  - User Manager
  - DBMS
- **[R3.2]**: The back-end must be able to check payment methods and drivers licenes on the spot.
  - User Manager
- **[R3.3]**: The back-end must provide a password recovery system.
  - User Manager
- **[R3.4]**: The back-end must handle all search requests correctly.
  - Search Manager
  - User Manager
  - Position Manager
  - Request Manager
- **[R3.5]**: The back-end must calculate the current amount to pay.
  - User Manager
- **[R3.6]**: The back-end must handle discounts/penalties and recalculate the fee accordingly.
  - Ride Manager

- **[R3.7]**: The back-end must handle car status changes.
  - Position Manager
  - Vehicle Manager
- **[R3.8]**: The back-end must provide an interface to allow a support operator to perform all basics CRUD operations.
  - User Manager

6. APPENDICES

6.1 References

The following tools where used in the creation of this document:

- *TexMaker 4.5* as Editor
- *Git* for version control
- *StarUML* for UML Diagrams
- *IntelliJ*

6.2 Effort Spent

- Simone Amico 20 h
- Chianella Beatrice 20 h
- Giovanakis Yannick 20 h