PowerEnJoy
Software Engineering II

Design Document

Giovanni Scotti, Marco Trabucchi

Document version: 1
December 4, 2016

Contents

Section 1

Introduction

1.1 Purpose

The Design Document is intended to provide a deeper functional description of the *PowerEnJoy* system-to-be by giving technical details and describing the main architectural components as well as their interfaces and their interactions. The relations among the different modules are pointed out using UML standards and other useful diagrams showing the structure of the system.

The document aims to guide the software development team to the architecture of the project providing a stable reference and a single vision of all parts of the software itself and clearly defining how they work.

1.2 Scope

The system aims to support a car-sharing service that exclusively employs electric cars.

The system is structured in a four-layered fashion, which will be thoroughly described in this document, that adapts to several forms of clients: various types of actors that interact with the system-to-be by generating a client-server dualism, hence a flow of requests-responses.

The architecture must be designed with the intent of being maintainable and extensible, also forseeing future changes.

This document aims to drive the implentation phase so that cohesion and decoupling are increased as much as possible. In order to do so, individual components must not include too many unrelated functionalities and reduce interdependency between one another.

Specific architectural styles and design patterns will be followed in this document and used for future implementation, as well as common design paradigms that combine useful features of said concepts.

1.3 Definitions, Acronyms, Abbreviations

ACID: Atomicity, Consistency, Isolation and Durability. This is the set of properties of database transactions.

DD: Design Document.

JAX-RS: Java API that provides support in creating web services according to the REST pattern.

JPA: Java Persistence API.

MVC: Model-View-Controller.

RASD: Requirements Analysis and Specification Document.

REST: REpresentational State Transfer. It is an architectural style and an approach to communications often used in the development of Web services.

RESTful: a REST-compliant system.

UX: User eXperience.

1.4 Reference Documents

This document follows the guidelines provided by ISO/IEC/IEEE 1016:2009 [3] related to system desing and software design descriptions for complex software systems.

The indications provided in this document are also based on the ones stated in the previous deliverable for the project, the RASD document [1].

Moreover it is strictly based on the specifications concerning the RASD assignment [2] for the Software Engineering II project, part of the course held by professors Luca Mottola and Elisabetta Di Nitto at the Politecnico di Milano, A.Y. 2016/17.

1.5 Document Structure

This document consists of five sections:

Section 1: Introduction. This section provides a general introduction and overview of the Design Document and the covered topics not previously taken into account by the RASD [1].

Section 2: Architectural Design. It shows the main system components together with sub-components and their relationship. This section is divided into different parts whose focus is mainly on design choices, interactions, architectural styles and patterns.

Section 3: Algorithm Design. This section provides a high-level description and details about some of the most crucial and critical algorithms to be implemented by the system-to-be.

Section 4: User Interface Design. It provides an overview on how the user interface will look like and behave giving further information with respect to those contained in the RASD [1].

Section 5: Requirements Traceability. This section describes how the requirements defined in the RASD [1] are mapped to the design elements defined in this document.

At the end of the document are an **Appendix** and a **Bibliography**, providing additional information about the sections listed above.

Section 2

Architectural Design

2.1 Overview

In this section is a detailed view of the physical and logical infrastructure of the system-to-be, as well as the description of the main components and their interactions.

A top down approach will be adopted for the description of components:

Section 2.2 A description of high level components and their interactions.

Section 2.3 A detailed insight of the components described in the previous section.

Section 2.4 A set of indications on how to deploy the illustrated components on physical tiers.

Section 2.5 A thorough description of the dynamic behaviour of the software, complete with diagrams for the key functionalities.

Section 2.6 A description of the different type of interfaces among the various described components.

Section 2.7 A list of the architectural styles, design patterns and paradigms adopted in the design phase.

Section 2.8 A list of all other relevant design decisions not mentioned before.

2.2 High Level Components

The main high level components of the system are:

Database: The system data layer; it includes all structures and entities responsible for data storage and management. No application logic is found at this level, apart from the DBMS one that must guarantee the correct functioning of the data structures while assuring the ACID properties of transactional databases.

Application Server: This layer encloses all the logic for the system applications, including the logic needed to interface with external systems and the key algorithms.

Web Server: This layer is in charge of providing web pages for the web-based application, and does not include any logic besides the basic request-response interaction one.

Mobile Application: The presentation layer dedicated to mobile devices; it communicates directly with the application server and only includes presentation logic.

Web Browser: The presentation layer dedicated to web browsers; it relies on the connection with the Web Server to obtain the pages to be rendered on the client.

On-Board Application: The presentation layer dedicated to the on-board computers applications; it communicates with the Application Server for the most part of the logic; however it also includes the logic needed to interface with the physical car systems and to perform ride-related actions/computations.

The described components are structured in four layers, as shown in Figure 2.1. Said figure also includes the interaction with external systems, that is intended to happen at the level of the Application Server.

The choice of separating the Application and Web Server layers allows greater scalability, since it allows the deployment on distinct physical tiers that can individually be optimized to perform their respective task.

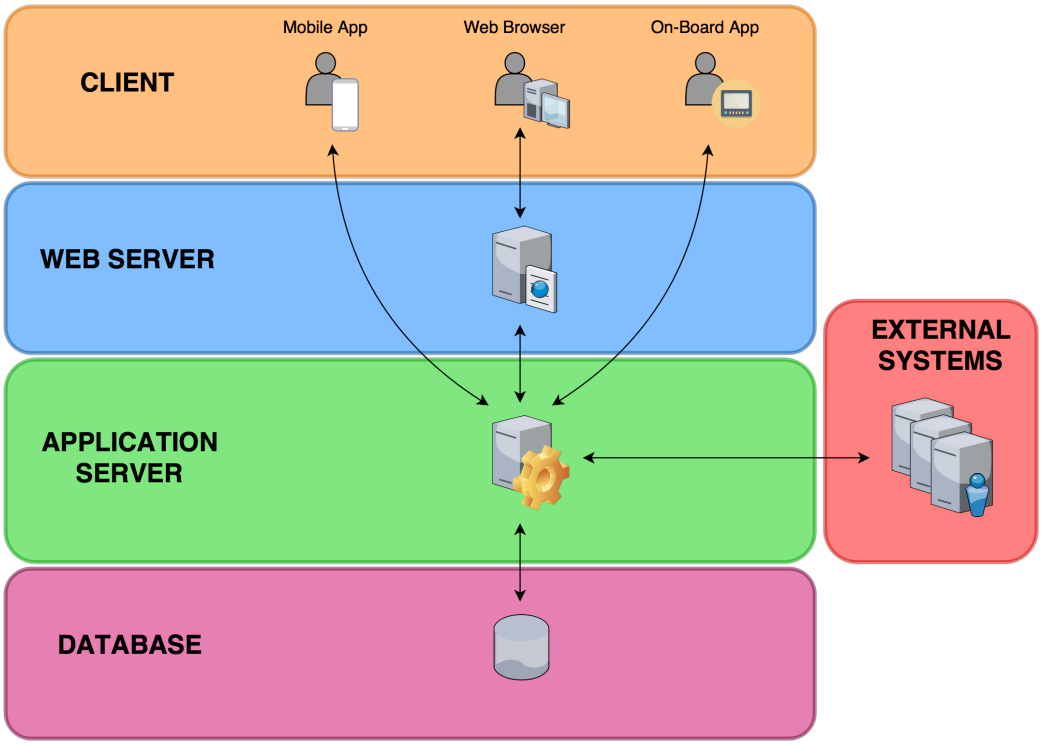

Figure 2.1: Layered structure of the system.

The interaction among the main system components is shown in Figure 2.2. The diagram noticeably points out the interaction with the payment handler and the maintenance systems, meant to support the *PowerEnJoy* service as stated in the RASD document [1]. Note that the web server and the application server are multi-threaded.

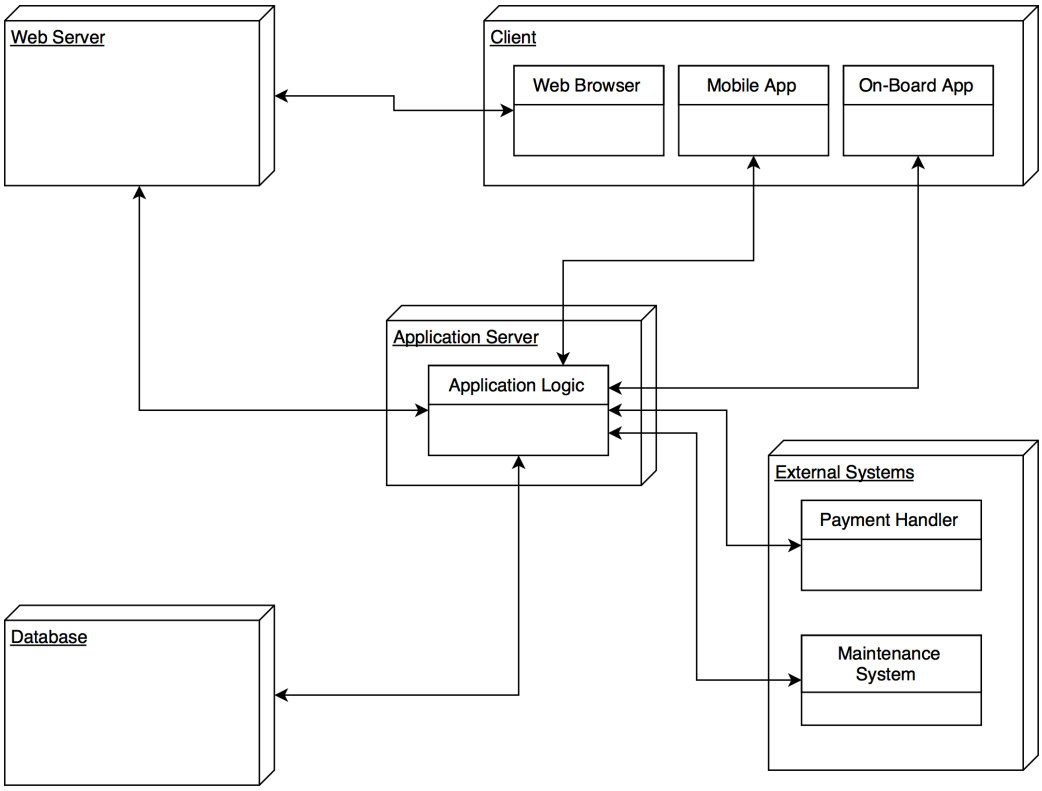

Figure 2.2: The high level components of the system.

2.3 Component View

In this section the individual components will be discussed in terms of the needed high level sub-parts and their functioning, as well as how those sub-elements interface with one another within the overlaying component and which of them is in charge of interfacing with other components.

2.3.1 Database

The database layer must include a DBMS component, in order to manage the insertion, modification, deletion and logging of transactions on data inside the storage memory.

Regardless of the implementation, the DBMS must guarantee the correct functioning of concurrent transactions and the ACID properties; it also must be a relational DBMS, since the application needs in terms of data storage do not require a more complex structure than the simple one provided by

the relational data structure.

The data layer must only be accessible through the Application Server via a dedicated interface. With respect to this, the Application Server must provide a persistence unit to handle the dynamic behaviour of all of the persistent application data.

The Database must be tuned during the implementation phase in order to ensure security by granting data access according to the privilege level of the requester. Sensible data such as passwords and personal information must be encrypted properly before being stored. Users must be granted access only upon provision of correct and valid credentials.

The E-R diagram in Figure 2.3 illustrates a concept of the Database schema.

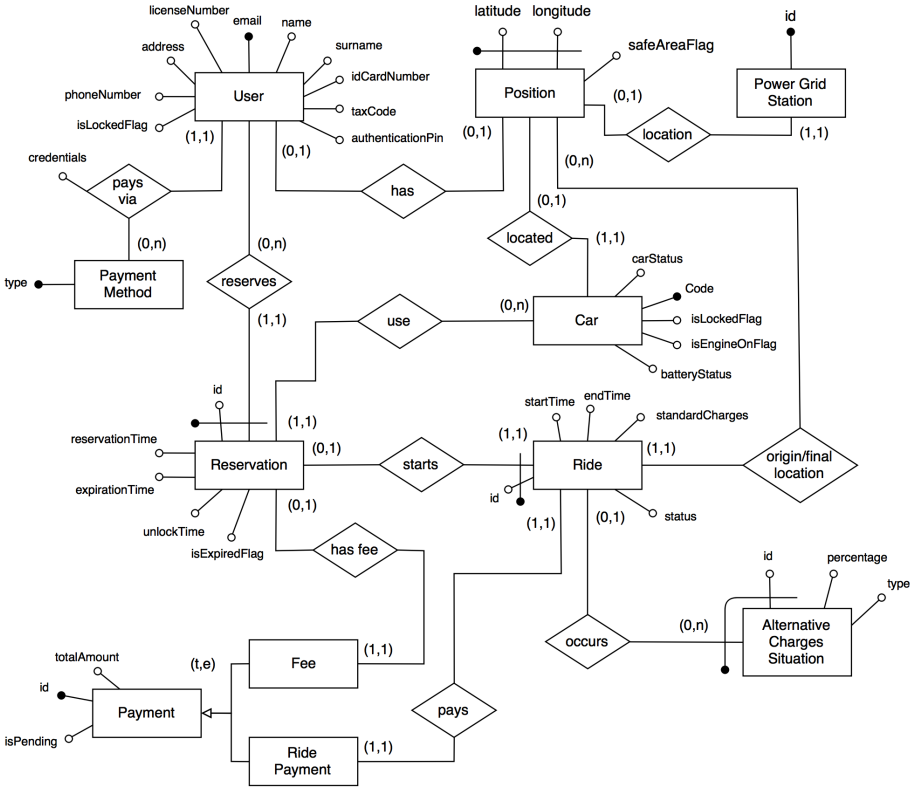

Figure 2.3: The E-R diagram of the database schema. Note that the relation named "origin/final location" must be considered as a short-hand notation to indicate two distinct relations: one for the starting location of the ride and one for the ending location.

2.3.2 Application Server

This layer must handle the business logic as a whole and the connections with the data layer and the multiple ways of accessing the application.

The main feature of the Application Server are the specific modules of business logic, which describe business rules and work-flows for each of the functionalities provided by the application itself.

The interface with the data layer must be handled, as stated in the previous section, by a dedicated persistence unit, that will be in charge of the object-relation mapping and dynamic data access and management; this ensures the fact that only the Application Server can access the Database.

The Application Server must provide a means to interface with the Web Server and the mobile and on-board clients via specific APIs in order to decouple the different layers with respect to their individual implementation. Moreover, it must provide a way to communicate with external systems by adapting the application to the existing external infrastructures.

The main business logic modules must include:

UserManager: This module will manage all the logic involved with user account management, login, registration, profile customization and management, as well as the generation and provision of user credentials.

ReservationManager: This module provides the logic behind the reservation management, with particular focus on the timing restrictions, car status updates (via the CarStatusManager) and reservation release conditions. It is also in charge of the controls to be performed in order to avoid multiple reservations by a user; lastly it must handle concurrent reservation issues such as pseudo-simultaneous requests for the same car.

RideManager: The logic included in this module is in charge of record all useful information about the rides as provided by the on-board application, including ride time, car battery levels and ride charges. It also manages the ride status updates.

MapManager: This module contains the logic used to locate cars and users, as well as defining the Safe Area boundaries and the power grids locations. It must provide useful data to the ReservationManager and the RideManager logic units, since both of them need localization information to perform their functionalities.

CarStatusManager: This module includes the logic needed by other components to set the car status of a car. It must also serve as an interface

with the external maintenance system by providing an automatic way to signal out-of-service cars.

SecurityAuthenticator: All the logic in charge of performing user-reservation-car matches is included in this module: this includes the control over cars unlocking, the control over reservation timer stopping upon unlock and the logic behind the on-board authentication method.

DiscountProvider: This module is in charge of recording virtuous and bad situations as parameters of rides, so that the PaymentGateway can gather all the needed information to compute the corresponding net total charges.

PaymentGateway: The logic involved in the computation of final charges is included in this module; moreover, this unit must stand as an interface with the payment handlers upon the act of the automatic payments.

NotificationManager: Serves as a gateway from all the modules that need to send an email to the clients by managing the logic behind the email notification services.

2.3.3 Web Server

The Web Server layer connects clients with the business logic layer in case the access to the application services is performed via web browser.

That being said, it is clear that the main functions to be implemented by this layer will essentially consist of interfaces, since - as stated in Section 2.2 - there will not be any logic implemented within the Web Server besides the presentation of pages.

The presentation must be structured in a clean and simple way, such that the components providing the client with web pages are decoupled as possible from the components that communicates with the business logic subsiding the Application Server in order to fetch relevant data to be shown. The communication must be thought in a way that allows quick and efficient data transfer through textual data files over HTTPS, e.g. XML or JSON.

Adequate APIs must be designed in order to separate in the most efficient way the design of the Web and Application Servers.

2.3.4 Mobile Application Client

The Mobile Client must be designed in a way that makes communications with the Application Server easy and independent from the implementation

of both sides. In order to do so, adequate APIs must be defined and used similarly to what has been described for the interactions between the two server layers.

The mobile application UI must be designed following the guidelines provided by the Android and iOS producers.

The application must provide a software module that manages the GPS connection of the device and keeps track of locations data, providing it to the Application Server to be processed.

2.3.5 On-Board Application Client

The On-Board Application consists of an application designed to run on pre-existing embedded devices on every car. The devices come together with the rest of the car equipment directly from the manufacturer. For this reason, the application must be designed following the guidelines provided by the manufacturer and based on given APIs.

The software module to be included in the application must manage the GPS locations of cars and the connection with the car sensors, as well as the transmission of all car-related data to the Application Server for the processing needed by the various business logic functions.

As far as the interface with the Application Server, the same considerations made with respect to the mobile application must be taken into account also for the on-board application.

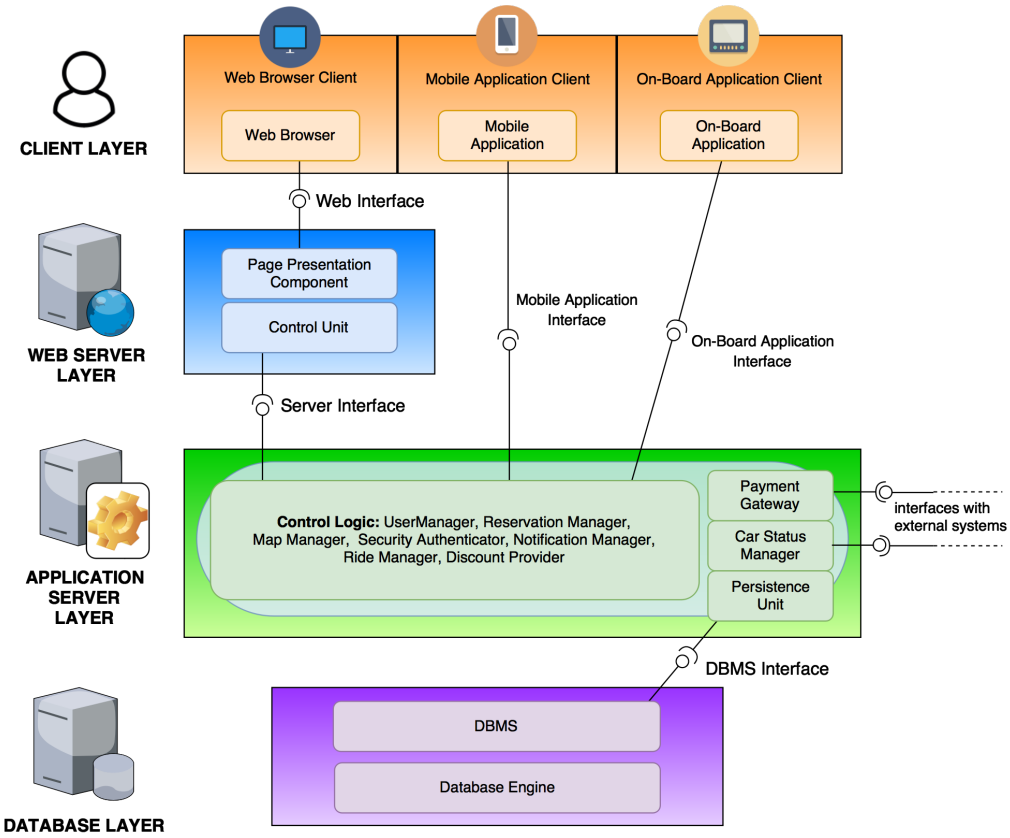

Figure 2.4: The global component view. Note that in the application server layer, most modules have been collapsed and only those that are relevant to particular interfaces have been highlighted. The modules interacting with the clients are more than one, so the connection has been specified with a macro-module containing all the access points.

2.3.6 Implementation Choices

Database Implementation

The implementation choices for the Database layer are the following:

- MySQL 5.7 as the relational DBMS;
- InnoDB as the subsiding database engine; InnoDB is a good choice for this application, because it manages concurrent access to the same tables in a very clean and quick way;

- The Java Persistence API (JPA) within the Application Server will serve as an interface with the Database.

Application Server Implementation

The main choice for this layer is the use of Java Enterprise Edition 7 (JEE). This was the most reasonable option for several reasons: the final product is a large-scope application, and thus needs distribution to great numbers of clients simultaneously; for the same reason, it needs to satisfy continuously evolving functional requirements and customer demands; JEE also allows the developers to focus on the logic behind the main functionalities while being supported by a series of reliable APIs and tools that, among other features, can guarantee the main non-functional requirements of the case (e.g. security, reliability, availability...); lastly, it can reduce the complexity of the application by using mechanisms and models that easily adapt to a large-scale project.

The specific implementation choices are:

- GlassFish Server as the Application Server implementation;
- Enterprise JavaBeans (EJB) to implement the single business logic modules described in the sections above. These will be appropriately subdivided into EJB containers as specified by the JEE documentation;
- Java Persistence API (JPA) as the persitence unit to perform the object-relation mapping and the Database access. JPA Entities will be used to implement the object representation of the data entities;
- JAX-RS to implement proper RESTful APIs to interface with clients and the Web Server;
- To interface with external systems, existing RESTful APIs defined by the partner (payment handlers, maintenance system) will be used.

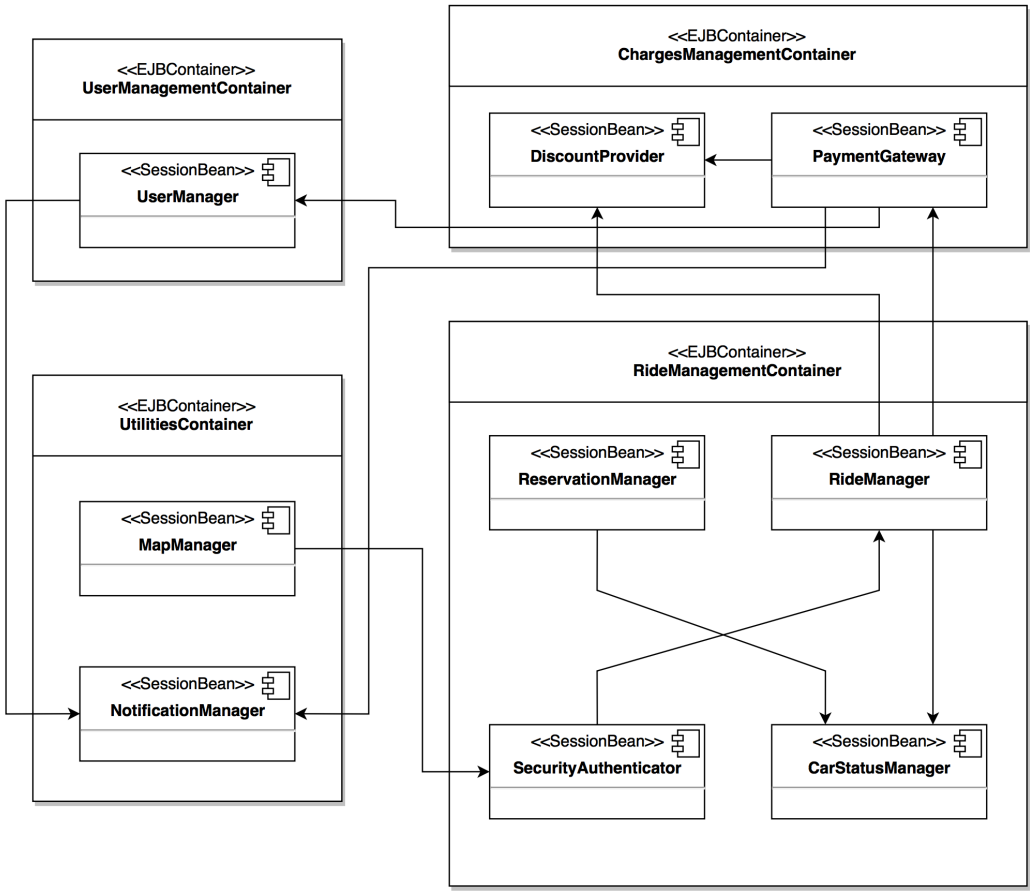

Figure 2.5: The components of the Application Server implemented as session beans to develop the business logic. The detailed interface of each bean is thoroughly described in section 2.6.5.

Web Server Implementation

Based on the choices made for the Application Server implementation, an easy way to avoid interface issues between the two server layers is to use the GlassFish functionalities for the Web Server as well.

This layer will also heavily rely on JEE: the core components of the Web Server will be implemented with JEE web components. In particular, the application will use JavaServer Pages (JSP) as the technology for developing the web presentation logic based on the MVC pattern. The choice of JEE also allows the Web Server to make use of Servlets to manage specific interactions when necessary.

The interface with the Application Server will be provided by the same

RESTful API specified in the previous subsection.

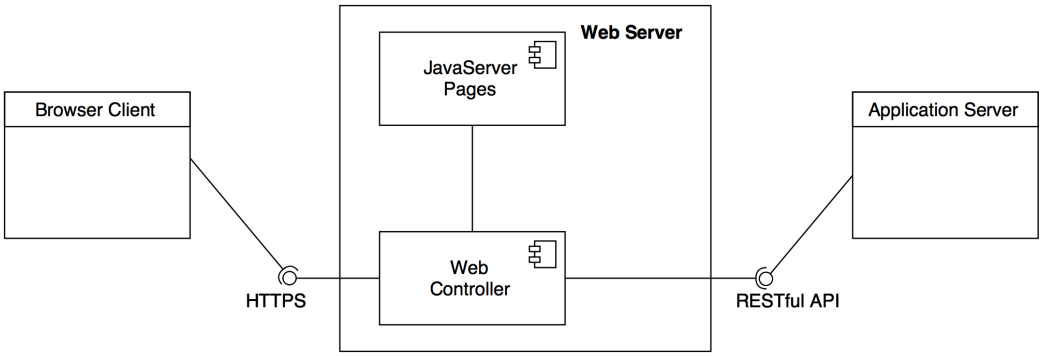

Figure 2.6: The components of the web server and their interfaces with other layers.

Mobile Application Client Implementation

The mobile application UI must be designed following the design guidelines provided by the devices' manufacturers. Two architectures must be supported: iOS and Android. The iOS application must be written in Swift, while the Android one must be implemented in Java.

The core of both application must be a Controller that communicates user inputs (after translating them from the UI input) to the Application Server via RESTful APIs. The access to the GPS of the devices must be performed through the default frameworks of the respective systems.

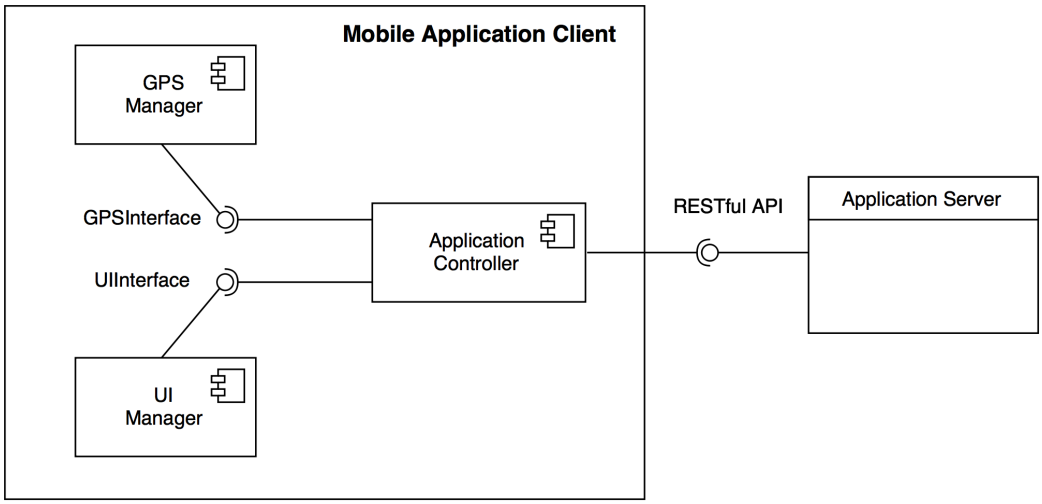

Figure 2.7: The components of the mobile application.

On-Board Application Client Implementation

The on-board application must strictly follow the car manufacturer guidelines for the development. The given APIs must be used to properly connect with the car equipment and sensors via the car CAN bus.

Since the car embedded devices have limited hardware specifications, the software must be written in C/C++. The user interface must be managed by a dedicated unit. The core of the application must include a GPS manager and a mobile connectivity unit. The communication with the Application Server must be performed by the connectivity unit via an adequate RESTful API. Data-flows from the car sensors must be handled by a separated data collection unit and properly evaluated by a data processing unit. The single elements will be coordinated by a Controller.

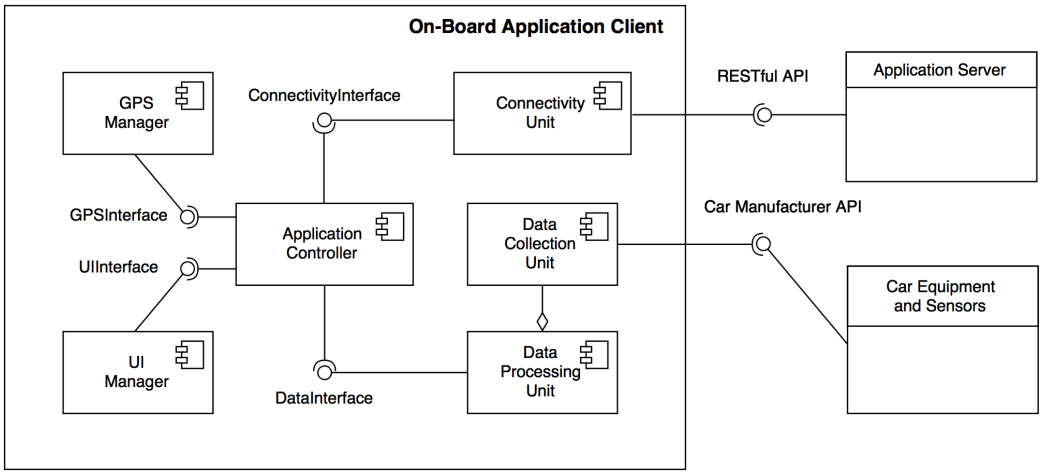

Figure 2.8: The components of the on-board application.

Figure 2.9: The overall decision tree for the first phase of the architectural design.

2.4 Deployment View

The deployment diagram for the system is shown in the following Figure 2.10.

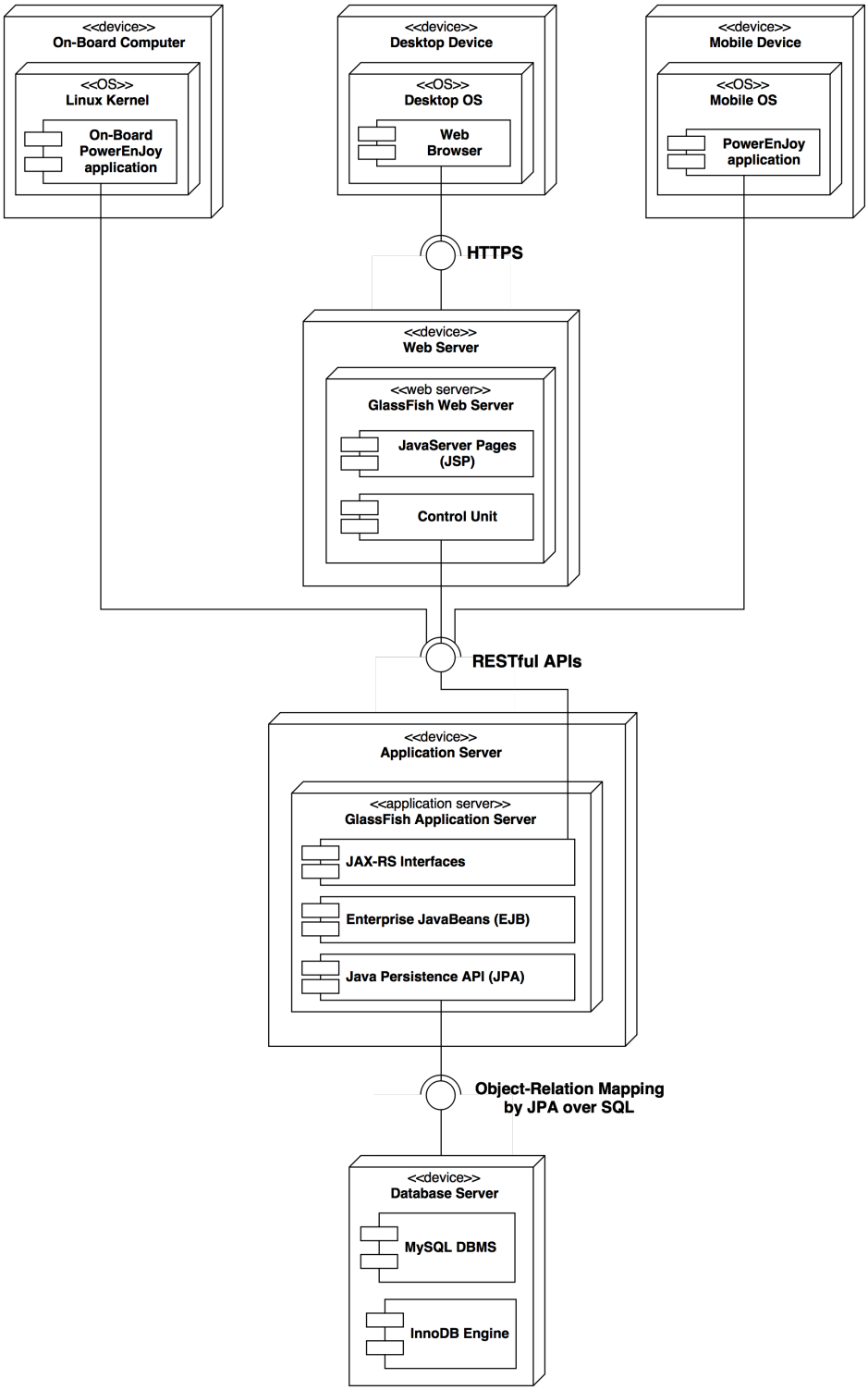

Figure 2.10: The global deployment view diagram for the system.

2.5 Runtime View

This section describes the dynamic behaviour of the system in the most relevant cases.

The following sequence diagrams highlight the runtime interactions between clients, servers and the database; in the case of internal interaction between sub-components of the Application Server, these interactions are expanded and detailed. This is not the case with the client applications, since the internal structure is simpler and there is no need to further detail said relations.

Direct interactions between the Application Server and the Database are not explicitly represented, since such interactions are abstracted by the persistence unit of the Application Server.

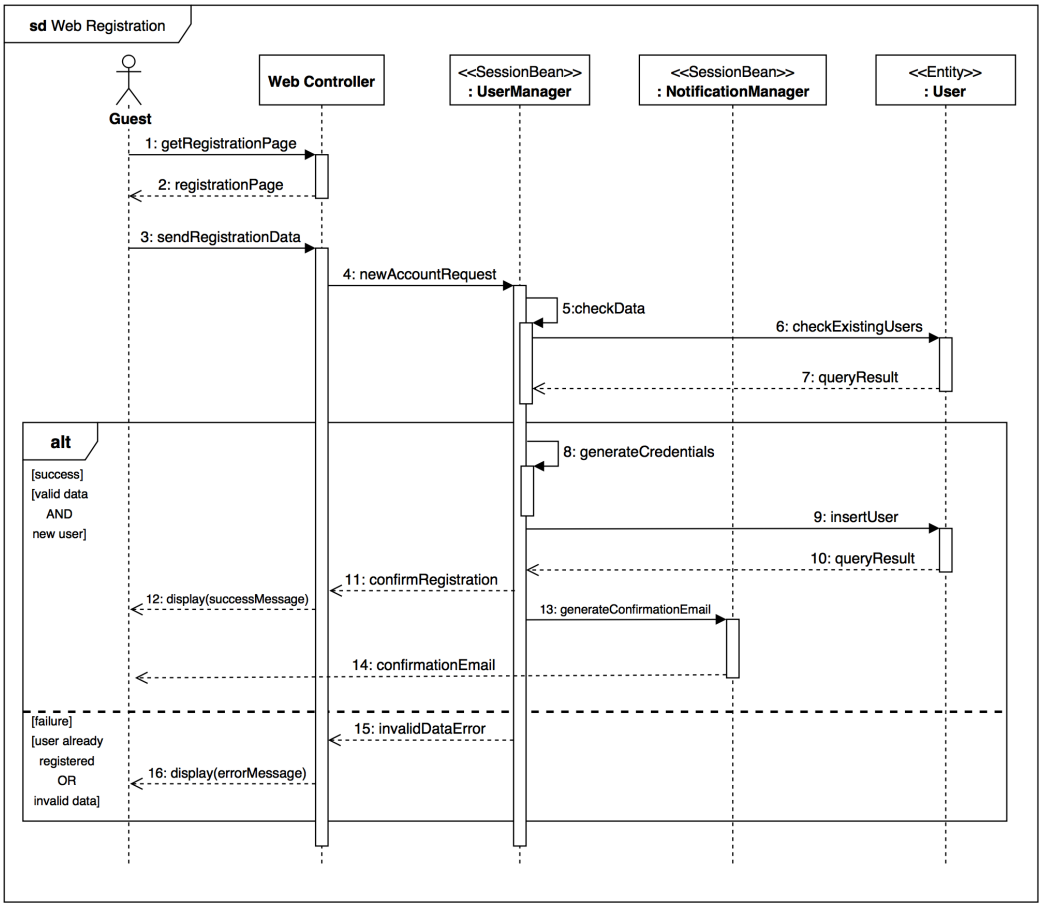

Figure 2.11: Sequence diagram of the registration process via the web browser client.

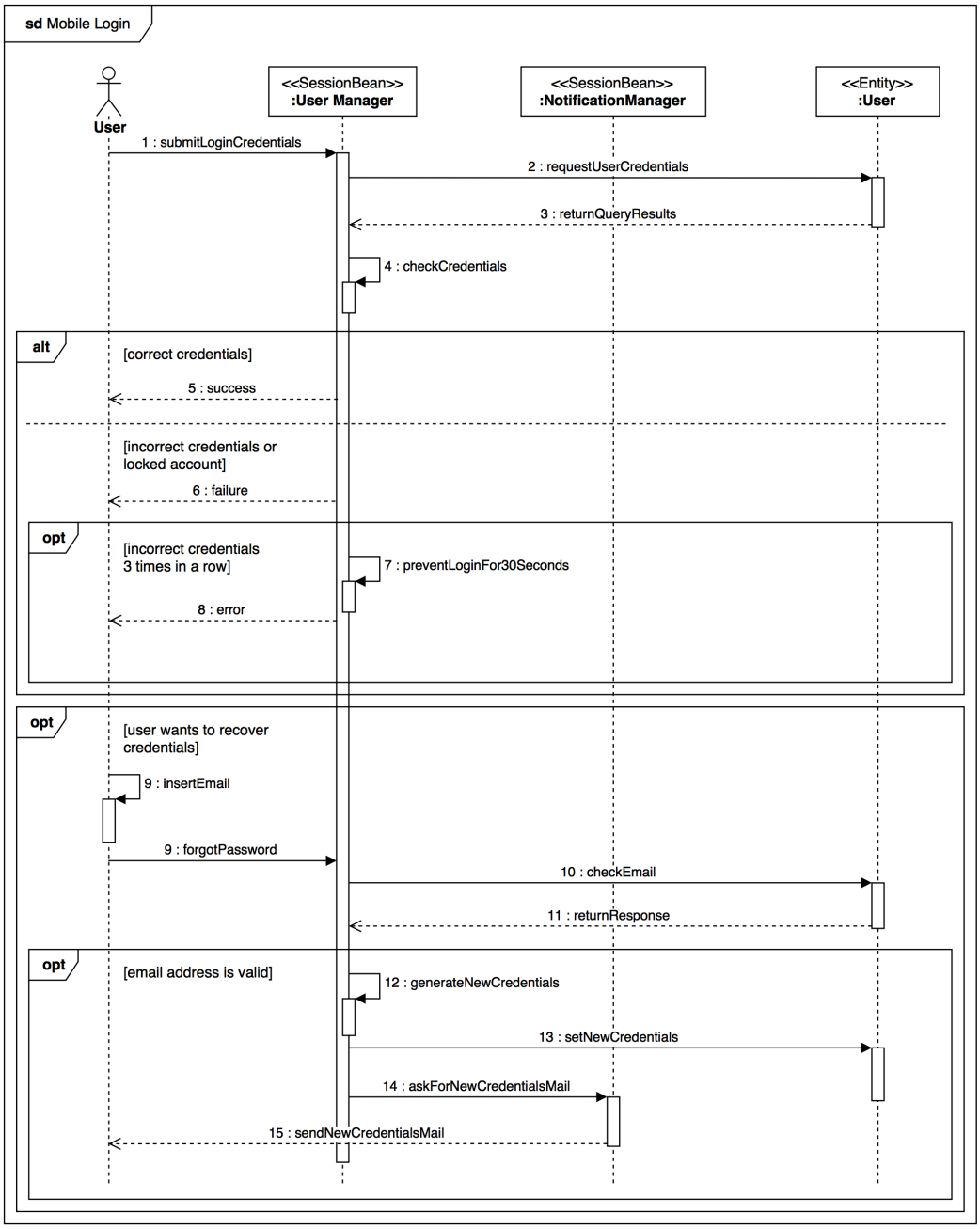

Figure 2.12: Sequence diagram of the login process and password recovery via the mobile application client.

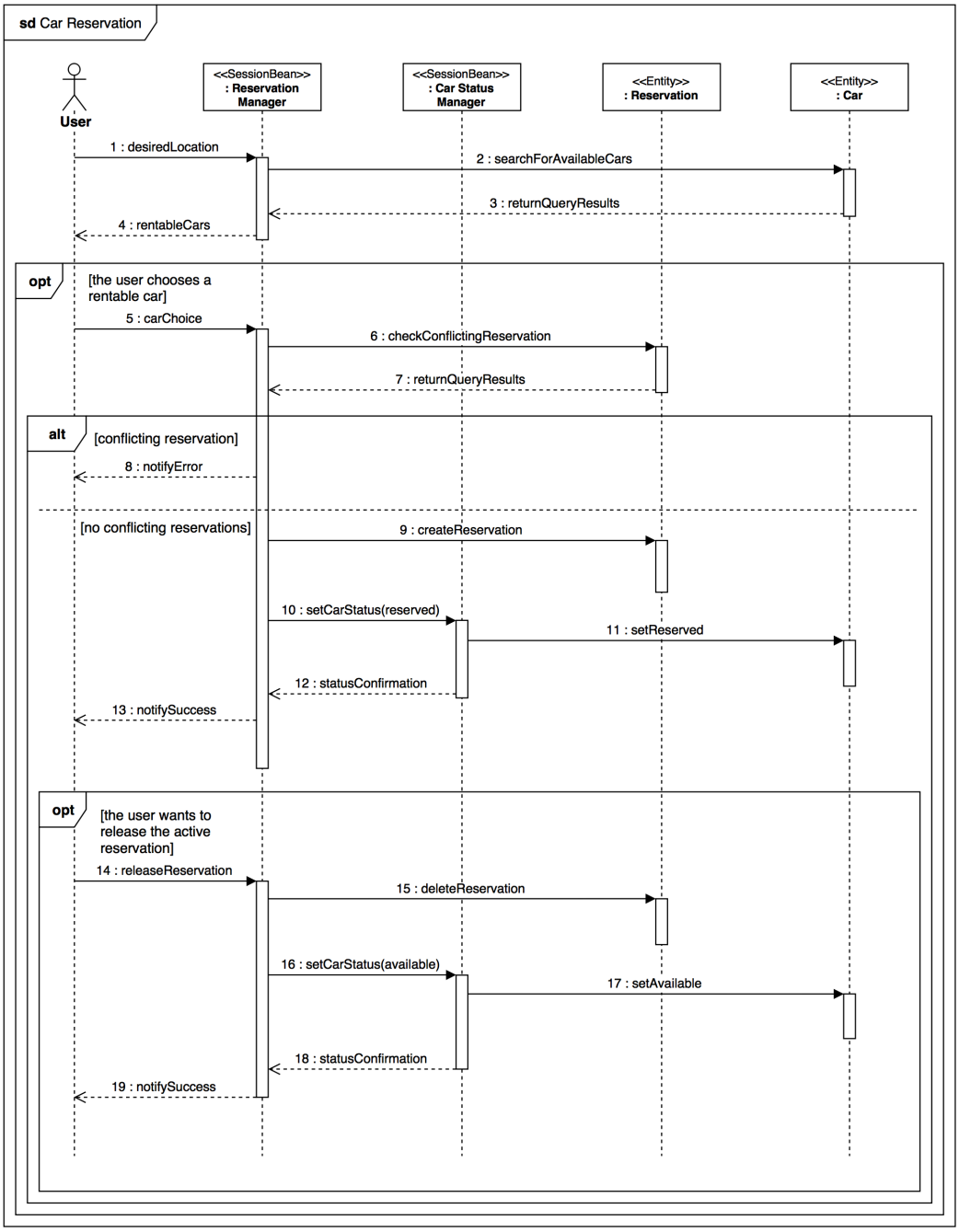

Figure 2.13: Sequence diagram of the reservation and releasing reservation processes via the mobile application client.

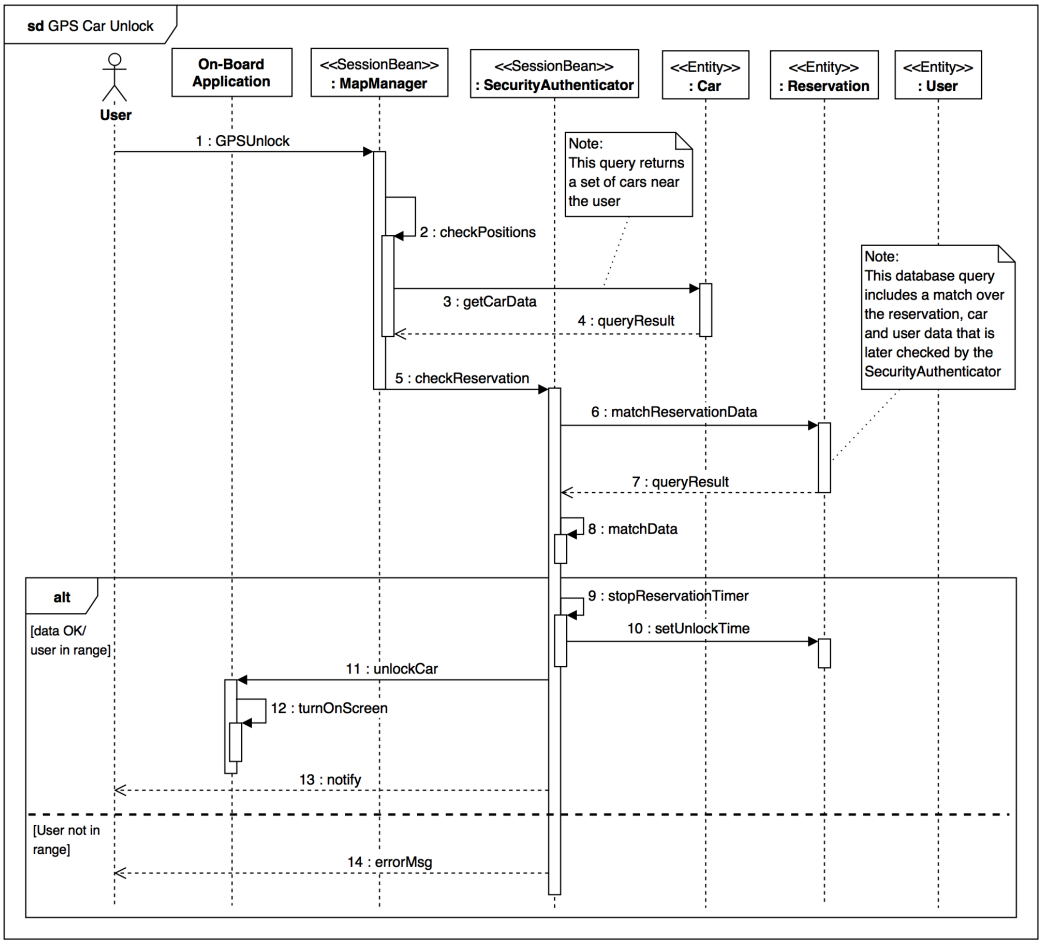

Figure 2.14: Sequence diagram of the car unlocking process using the GPS method.

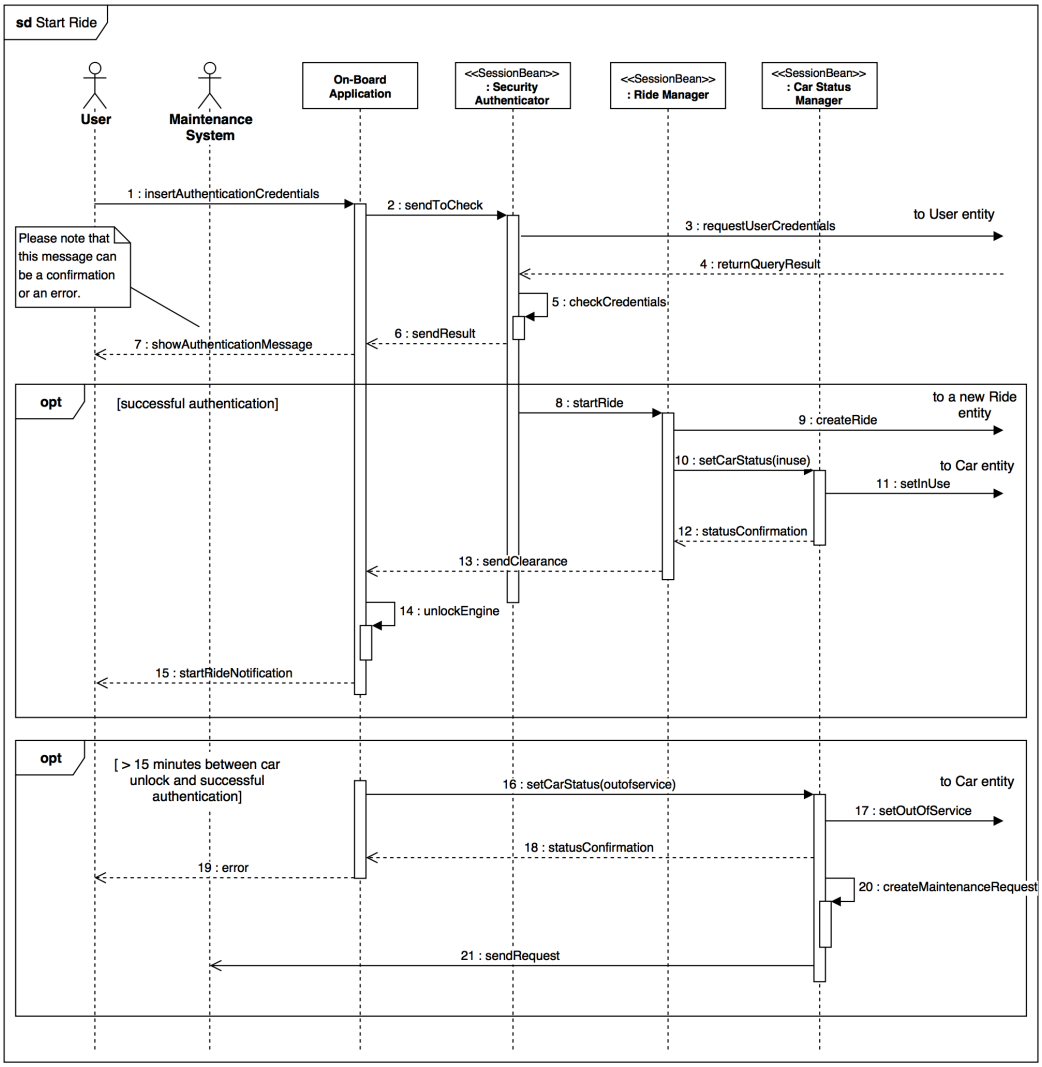

Figure 2.15: Sequence diagram of the start ride process.

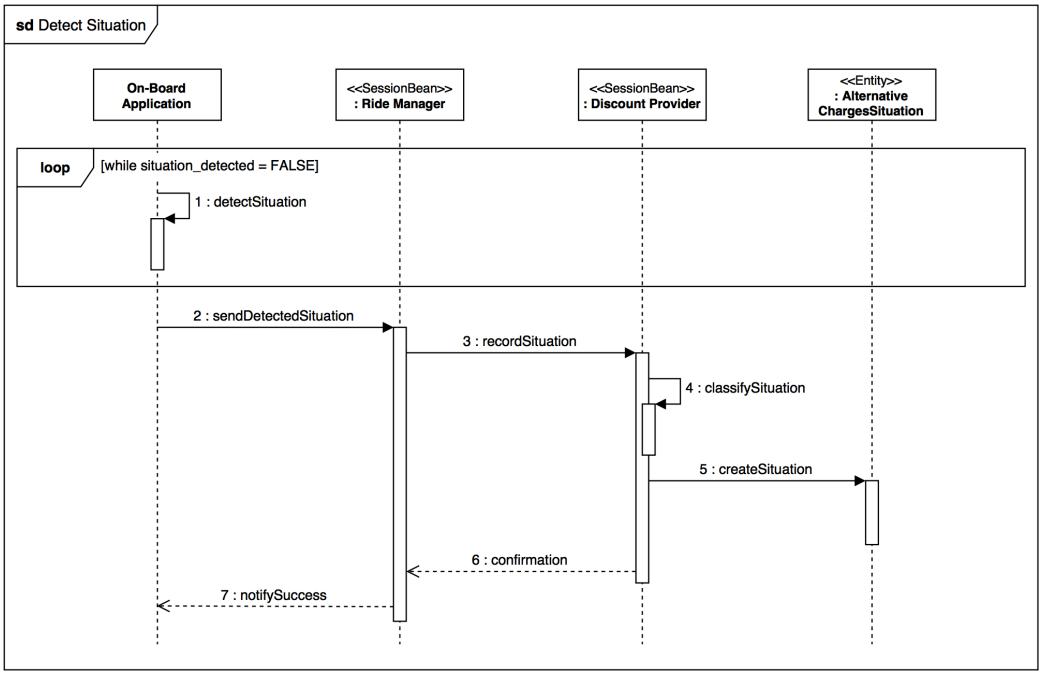

Figure 2.16: Sequence diagram of discount/additional charges situations detection and storage processes.

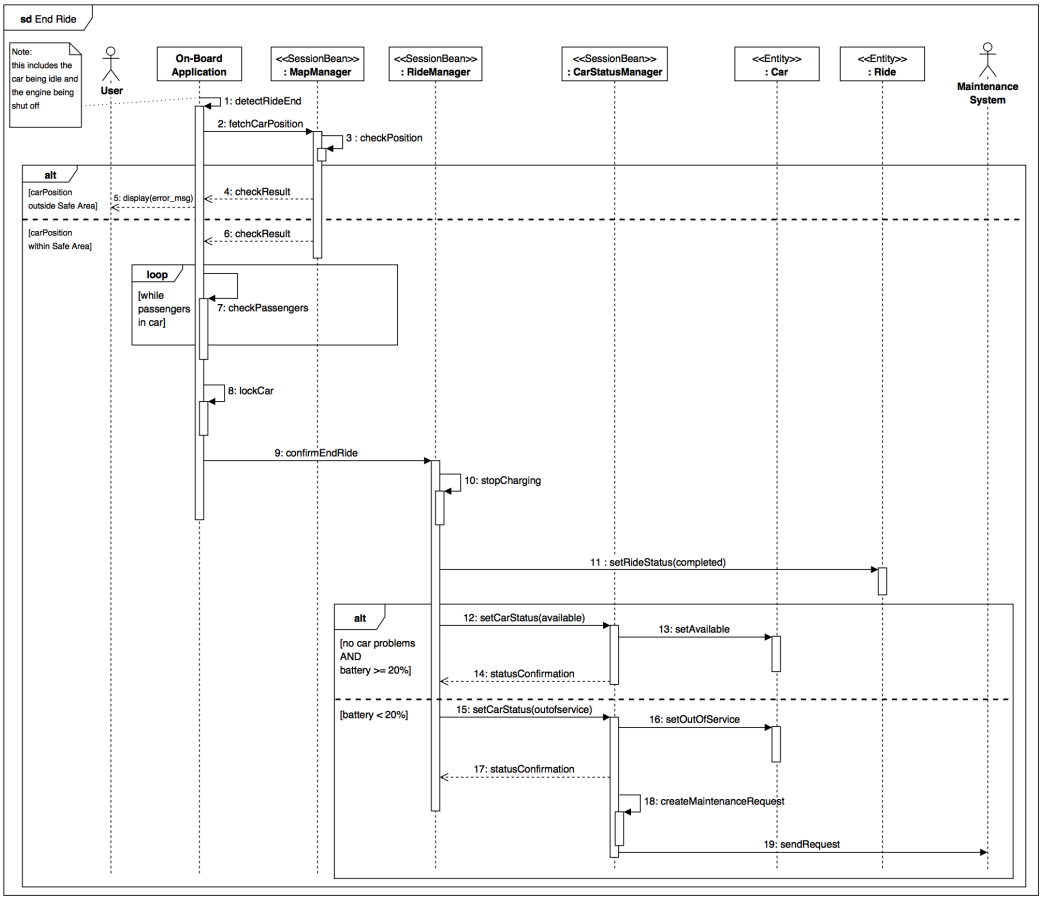

Figure 2.17: Sequence diagram of the end ride process in case the user parks the car within the Safe Area, highlighting the possibility to contact the Maintenance System in case of emergency after the end of the ride.

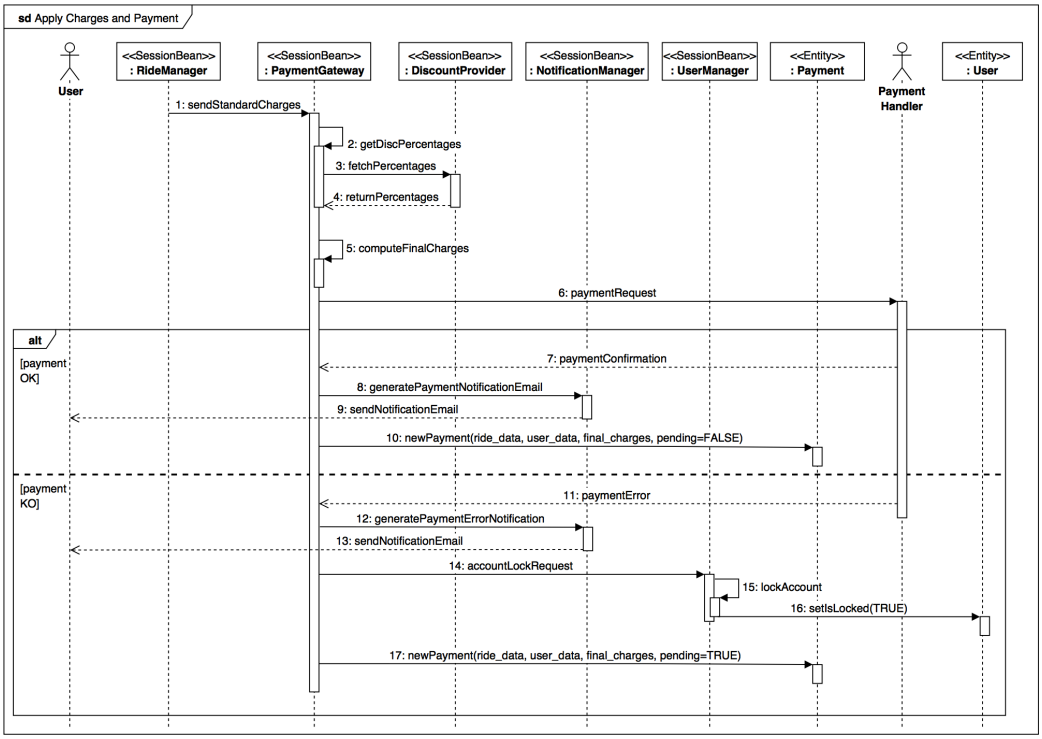

Figure 2.18: Sequence diagram of the charges application and payment process, including the account blocking in case of a failed payment.

2.6 Component Interfaces

This sections includes further details on the interfaces between different components of the system. Also, Subsection 2.6.5 is devoted to illustrate some relevant details about the interfaces needed to use and interact with each component of the Application Server, accessible both by other Application Server components and by other components of the system.

2.6.1 Database - Application Server

The Application Server is the only one that can access the Database directly; this is done through the Java Persistence API mapping between objects and actual relations.

2.6.2 Web Server - Web Browsers

The interactions between the client's browsers and the Web Servers are based on the HTTPS protocol, as previously specified in the RASD [1].

2.6.3 Application Server - Web Server and Clients

The communication between Application Server and clients, both direct and via the Web Server, must happen via RESTful APIs provided by the Application Server itself and implemented using JAX-RS.

2.6.4 Application Server - External Systems

The Application Server must connect with two types of external systems:

- A Maintenance System, to which the Application Server must offer an interface API in order to provide access to data needed for maintenance interventions;
- One or more Payment Handlers, that provide the interface APIs to which the Application server itself must adapt in order to perform payments of any kind.

2.6.5 Internal Interfaces for Application Server Components

UserManager

The following are the procedures implemented by the UserManager component and called by other components:

`submitLoginCredentials` This procedure is used to submit the login credentials, hence accessing the application; this takes as *parameters* the user credentials: email and password; the *return value* for this function is a user-identifying token that is later going to be used as parameter by other procedures, in order to identify the requesting user; if the provided credentials are wrong or non-existing, the *return value* is an error.

`forgotPassword` This procedure is used when a user forgets his/her password for the service and wishes to recover it; this takes as *parameter* the user email; the *return value* for the procedure is not defined, as the procedure itself takes part in an asynchronous call.

`newAccountRequest` This procedure is used to request the creation of a new *PowerEnJoy* account; this takes as *parameters* all the personal user data required to register, noticeably: name, surname, license ID and email; the *return value* for the procedure is an error in case of invalid data submission or a confirmation message in case of success.

`deleteUser` This procedure is used to permanently delete a user account from the system; this takes no *parameters* apart from a user-identifying token to recognize the requester; the *return value* for this procedure is a confirmation message.

`editProfile` This procedure is used every time a user wishes to modify anything related to his personal account profile; this takes as *parameters* a set of modified items; the *return value* for this procedure is an error in case of invalid data insertion, and a confirmation message otherwise.

`accountLockRequest` This procedure is called every time a user account must be locked for any reason; the only *parameter* taken is a user-identifying token to get the correct target user; there is no defined *return value* since the procedure call is asynchronous.

`fetchHistory` This procedure is called to fetch the user's history from the database; the taken *parameters* include a user-identifying token and - possibly - some filters over the history request; the *return value* is an object containing a representation of the (possibly filtered) history of the user.

ReservationManager

The following are the procedures implemented by the ReservationManager component and called by other components:

`desiredLocation` This procedure is used to indicate the area in which a user wishes to look for available cars; this takes as *parameters* either an address or a GPS position; the *return value* for the procedure is a set of available cars based on the position parameter.

`carChoice` This procedure is used to indicate the vehicle chosen by a user to be reserved; this takes as *parameters* a car and a user-identifying token to associated the reservation with; the *return value* is an error if there is any issue during the reservation process, otherwise it is a confirmation message.

`releaseReservation` This procedure is called whenever a user decides to renounce to his/her current active reservation; the taken *parameter* is only a user-identifying token; the *return value* is a confirmation message.

RideManager

The following are the procedures implemented by the RideManager component and called by other components:

`startRide` This procedure is used to create a new ride after a successful authentication; this takes as *parameters* are a car and a user-identifying token to associate with the ride; the *return value* is undefined since the clearance to actually beginning the ride is sent directly to the car's application.

`sendDetectedSituation` This procedure is used to communicate an alternative payment situation to the DiscountProvider; the *parameters* needed by the procedure are a car and the detected situation; the *return value* is a confirmation message.

`confirmEndRide` This procedure is called when the system need to end a ride after detecting that this is really the case; the only needed *parameter* is a car to recover all necessary data; the *return value* is undefined, since the call is asynchronous.

MapManager

The following are the procedures implemented by the MapManager component and called by other components:

`GPSUnlock` This procedure is used to match the positions of a car and a user and unlock said car in case it was reserved by that user; this takes as *parameters* a user-identifying token and the user's position; the *return value* for the procedure is undefined, since its call is asynchronous.

`checkCarPosition` This procedure is used to check if a car is within the Safe Area once the system detected a possible ride ending situation; the needed *parameters* are a car and its position; the *return value* is the result of the performed check.

CarStatusManager

The following are the procedures implemented by the CarStatusManager component and called by other components:

`setCarStatus` The procedure is called whenever a component need to update the status of a car; this takes as *parameters* a car and the status to be set for said car; the *return value* for this procedure is a confirmation message.

SecurityAuthenticator

The following are the procedures implemented by the SecurityAuthenticator component and called by other components:

`codeUnlock` This procedure is used to unlock a car based on a vehicle-specific code found on the car itself; the procedure *parameters* are a user-identifying token and a code; the *return value* for the procedure is undefined, since its call is asynchronous.

`checkReservation` This procedure is used to perform the same action as the one listed before, but in the case of a GPS unlock attempt; the needed *parameters* are a user-identifying token and a list of cars detected to be near the GPS position of the identified user; the *return value* is undefined since the call is asynchronous: in fact, a confirmation/error message is directly sent to the user's application client.

`sendToCheck` This procedure processes an authentication PIN inserted by a user on the on-board application of a car to find out if the ride is being initiated by the user who reserved and unlocked that car; this takes as *parameters* a car and a PIN; the *return value* is the result of the check.

DiscountProvider

The following are the procedures implemented by the DiscountProvider component and called by other components:

`recordSituations` This procedure is used to record a detected alternative payment situation; this takes as *parameters* the detected situation and the corresponding ride; the *return value* for the procedure is a confirmation message.

`fetchPercentages` This procedure is called at the end of a ride to get all the alternative payment percentages in order to compute the final charges;

this takes as only *parameter* the ride for which the system needs to compute the final charges; the *return value* is a list of discount/additions percentages.

PaymentGateway

The following are the procedures implemented by the PaymentGateway component and called by other components:

`applyFee` This procedure is used to apply a fee after a reservation expires; this takes as *parameters* a reservation and a fee amount; the *return value* is undefined since the procedure includes a call to the NotificationManager that manages the response to the user directly based on the success of the fee payment.

`sendStandardCharges` This procedure is called to compute the final charges of a ride and carry out the corresponding payment; it takes as *parameters* a ride and its standard charges; the *return value* is undefined since the procedure includes a call to the NotificationManager that manages the response to the user directly based on the success of the fee payment.

NotificationManager

The following are the procedures implemented by the NotificationManager component and called by other components:

`generateConfirmationEmail` The procedure is used during the registration process to generate an email containing the provided user credentials; this takes as *parameters* the user credentials generated by UserManager and the addressee's email; the *return value* is undefined, since the confirmation email is immediately sent to the addressee, bypassing the caller.

`askForNewCredentialsMail` This procedure is called upon the request of new credentials by a user; the procuedure *parameter* are the user credentials generated by UserManager the user's email; the *return value* of the procedure is undefined, since the confirmation email is immediately sent to the addressee, bypassing the caller.

`generatePaymentNotificationEmail` The procedure is used to generate an email notification to a user as a consequence of a successful payment; this takes as *parameters* all the data related to the successful

payment and the user's email; the *return value* is undefined, since the confirmation email is immediately sent to the addressee, bypassing the caller.

`generatePaymentErrorNotification` The procedure is used to generate an email notification to a user as a consequence of a failed payment; this takes as *parameters* all the data related to the failed payment and the user's email; the *return value* is undefined, since the confirmation email is immediately sent to the addressee, bypassing the caller.

2.7 Architectural Styles and Patterns

The following architectural styles and patterns have been used:

Client and server

The client-server model is used at different levels in the *PowerEnJoy* system design:

- the mobile and the on-board applications are clients with respect to the Application Server that receives and elaborate requests.
- the user's browser communicates with the Web Server. The first one is the client while the latter is in charge of providing a service, that is the requested web page.
- the Web Server, as a client, communicates with the Application Server in order to process user's requests.
- the Application Server, in the role of the client, queries the Database that is responsible for getting results.

Multitier architecture

The client-server multilayered architecture allows to physically separate presentation, application processing and data management operations.

Moreover, despite the fact the separation of the Web Server from the Application Server increments the number of tiers in the system, it brings a huge improvement in service availability because in case of Web Server failure, the system remains still accessible from the mobile application.

Thin client

The think client approach has been used with respect to the interaction among user's machines and the system itself.

All the main logic is implemented by the Application Server that has a decent computing power and can manage concurrency issues in an efficient way. On the other hand, the mobile application and the user's browser are in charge of presentation only and they do not involve decision logic.

This choice is also reasonable because allows users to take advantage of the service via devices with limited computing power and makes software updates easier.

Thick client

The on-board application has been designed to involve a discrete amount of logic, but still requires a periodic connection to the central Application Server in order to work properly.

Its logic mainly deals with the control of the car systems, such as unlocking the doors, and gathering information from several sensors placed in the cabin. Furthermore the application has to detect different discount and additional charges situations, that affects the final charge the user has to pay, and communicates them to the Application Server.

Model-View-Controller

The web, the mobile and the on-board applications follow the Model-View-Controller software design pattern. The MVC allows to separate the application into three communicating and interconnected parts fulfilling the design principle of the separation of concerns.

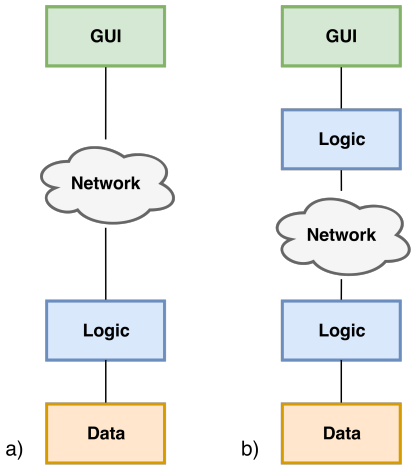

Figure 2.19: The remote presentation (a) and the distributed logic (b) architectural styles that inspired respectively the design of the web/mobile and the on-board application.

2.8 Other Decisions

2.8.1 Authentication Method

As soon as the user unlocks the car, he/she needs to authenticate him/herself again to start the ride. This is a remarkable step introduced for security purpose: the goal is to prevent someone who has somehow stolen the password of a registered user to take the advantage of the service provided by *PowerEnJoy*.

The authentication method will be implemented as a PIN provided by the system during the registration process and cannot be changed by the user as stated in the RASD document [1].

The above-mentioned PIN must be inserted via the on-board application and must be checked by the system before unlocking the engine of the car.

2.8.2 User's password storage

For security reasons, the user's password is stored using cryptographic hash functions. In addition to that, the password is not only hashed, but also salted. This is a common security choice since many users reuse passwords for multiple sites and a cyber attack could jeopardize their sensitive information.

2.8.3 Maps

In order to support the car finding functionality, the system will make use of an external maps service: *Google Maps*. This choice is motivated by the fact that manually developing a map service is not a viable solution with respect to the size of the project.

Google Maps's services will be used to translate addresses to coordinates and vice-versa during the process of finding a car: the translation from address to position will be performed when the location input by a user is an address; the inverse translation will happen when the system needs to display the address corresponding to the position of a reserved car.

Another case in which the services of *Google Maps* will be used is to graphically show the map during the same process.

Section 3

Algorithm Design

3.1 Discounts and Additional Charges Management

3.1.1 Situations Detection

As stated in the RASD [1], the system must be aware of different discount and additional charges situations that can occur during one user's ride and bring respectively a reduction or an increase of the final charges at the end of the ride.

The **discount situations** that the system must recognize are the following:

1. if the user takes at least two other passengers on the car, the system applies a discount of 10% on the final charges.
2. if a ride ends with no more than 50% of the car battery empty, the system applies a discount of 20%.
3. if the user plugs the car into the power grid at the end of the ride, the system applies a discount of 30%.

The **additional charges situations** that are detected by the system are the following:

1. if a car is left more than 3km away from the nearest power grid, the system charges 30% more on the last ride.
2. if a car is left with no more than 20% of battery level and the user does not plug it into the power grid, the system charges 30% more.

The on-board application is aware of all these situations and it is able to detect them thanks to the sensors the car is provided. As soon as a situation occur, the on-board application gets in touch with the Application Server. The business logic is in charge of:

1. understanding if the situation will bring a discount or additional charges;
2. storing it in the database and linking it to the current ride.

3.1.2 Final Charges Computation

After 5 minutes to the end of the ride, in order to be specifications-compliant, the system must calculate the final charges taking into account the previously saved situations:

1. the system queries the data and saves in a *List* of additional charges situations the events previously detected as additional charges situations;
2. the system stores in an *attribute* the discount situation with the highest percentage or NULL if no discount situations occurred during the ride.
3. *foreach* element of the additional charges List, the increment to the final charges is computed as: $standardcharges * situationpercentage$, where *standard charges* are the charges at the end of the ride without any discount/additional charges situation. *standard charges* are computed as $PRICE\_PER\_SECOND * (endTime - startTime)$ by the RideManager;
4. all the increments must be summed to the *standard charges*;
5. if at least one discount situation happened during the ride, this will lead to a decrement of the final charges that is equal to $standardcharges * situationpercentage$.

Note that *situation percentage* is the discount/additional charges percentage divided by 100.

Section 4

User Interface Design

4.1 UX Diagrams

The purpose of the UX diagram is to show the different screens provided by the user interface of a particular application and their dynamic contents. Moreover it points out the interactions among the screens themselves and the presence of input forms and required data in a specific screen.

The diagrams provided below follow the User Interface requirements stated in the RASD document [1].

The Web and the Mobile applications implement the same functionalities. This is why just a single UX diagram for both the applications has been attached to this document. Please notice that:

- *red* words describe functionalities accessible from the mobile application only (for example the provision of GPS data to the system);
- the possible choices of the user (e.g.: the car to be rented, the GPS coordinates...) have been depicted as *Input Forms* since they are data to be sent to the system and imply a screen update.

The UX diagram of the both the Web and Mobile applications is shown in Figure 4.1.
The UX diagram of the on-board application is shown in Figure 4.2.

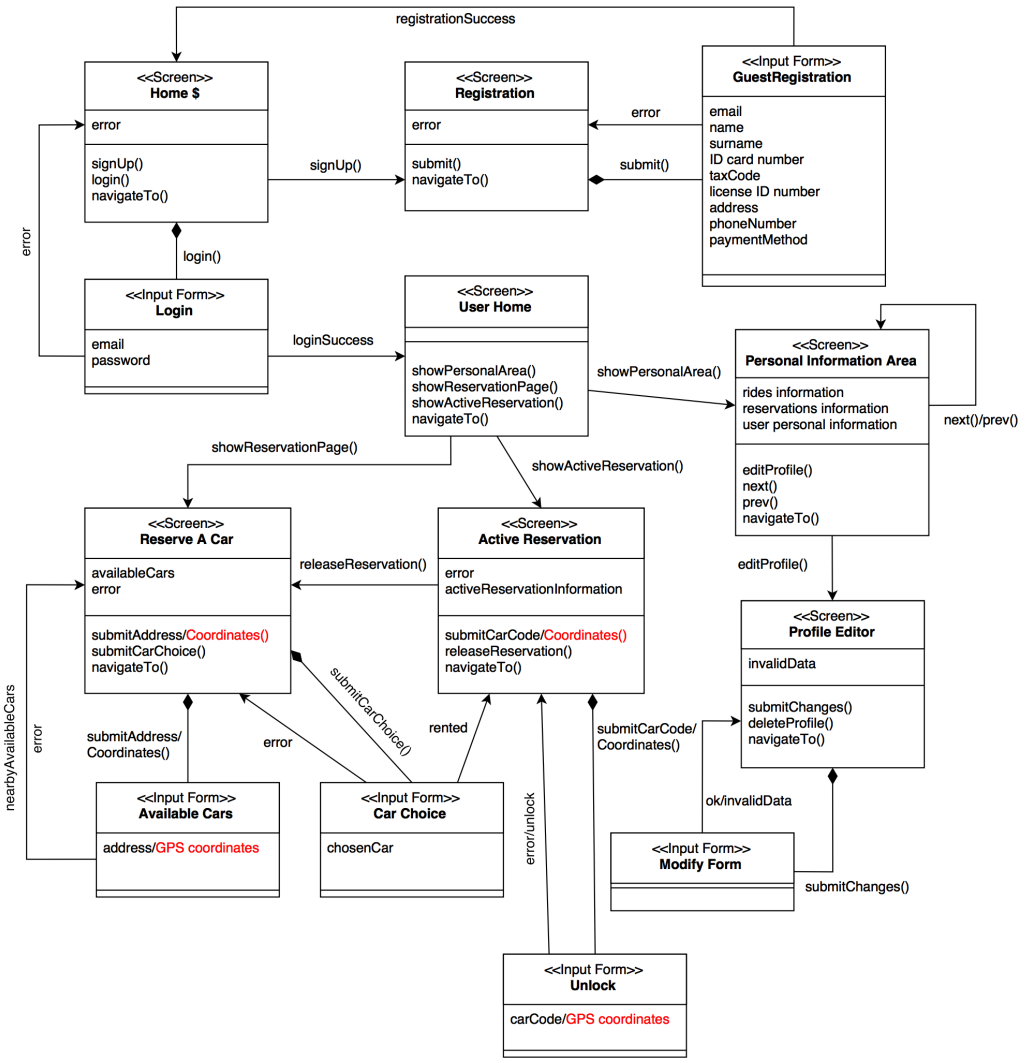

Figure 4.1: UX diagram of both the Web and the Mobile Applications. Note that the outgoing arcs exiting from "User Home" are mutually exclusive: they are shown based on the presence of an active reservation by the current user.

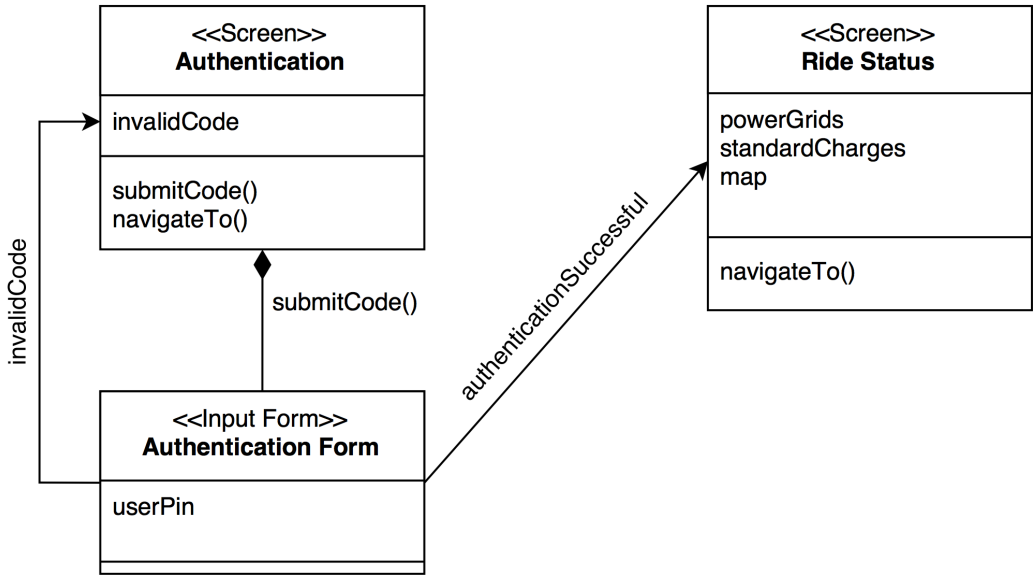

Figure 4.2: UX diagram of the On-Board Application. Notice that the attributes shown in the "Ride Status" screen are actually dynamic content. Note that there is no screen marked with the "$" symbol: this is because no page acts as a home page reachable from anywhere in the application; the first screen to be shown is of course the "Authentication" one.

4.2 User Interface

4.2.1 Web Interface

The following mockups show how the interface of the Web Application should look like on the user's browser.

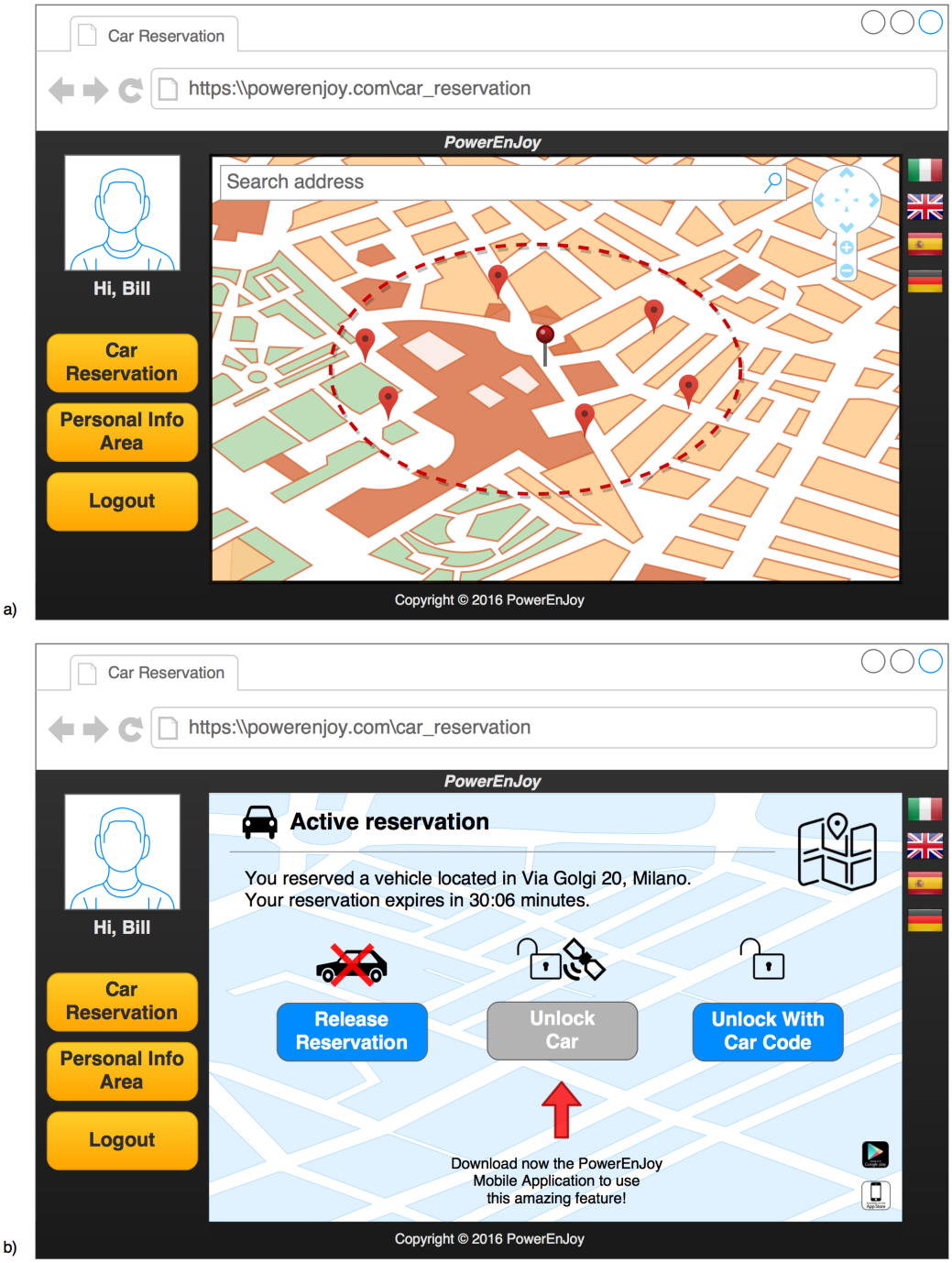

Figure 4.3: Look and feel of the "Reserve A Car" screen (a) and the Active Reservation screen (b). They are both accessible from the "Car Reservation" button in the toolbar: the first is loaded if the user has no active reservation while the second otherwise.

4.2.2 Mobile Interface

The following mockups show how the interface of the Mobile Application should look like on the user's smartphone.

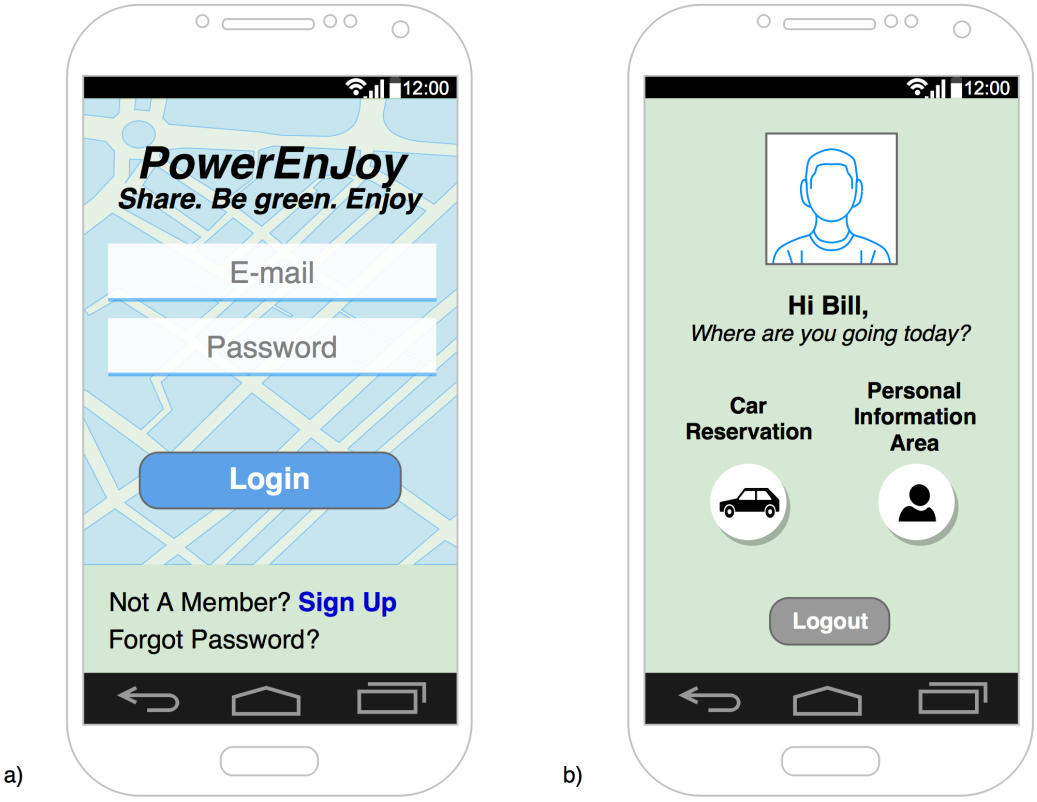

Figure 4.4: The "Login" screen (a) and the "User Home" screen (b) of the Mobile Application.

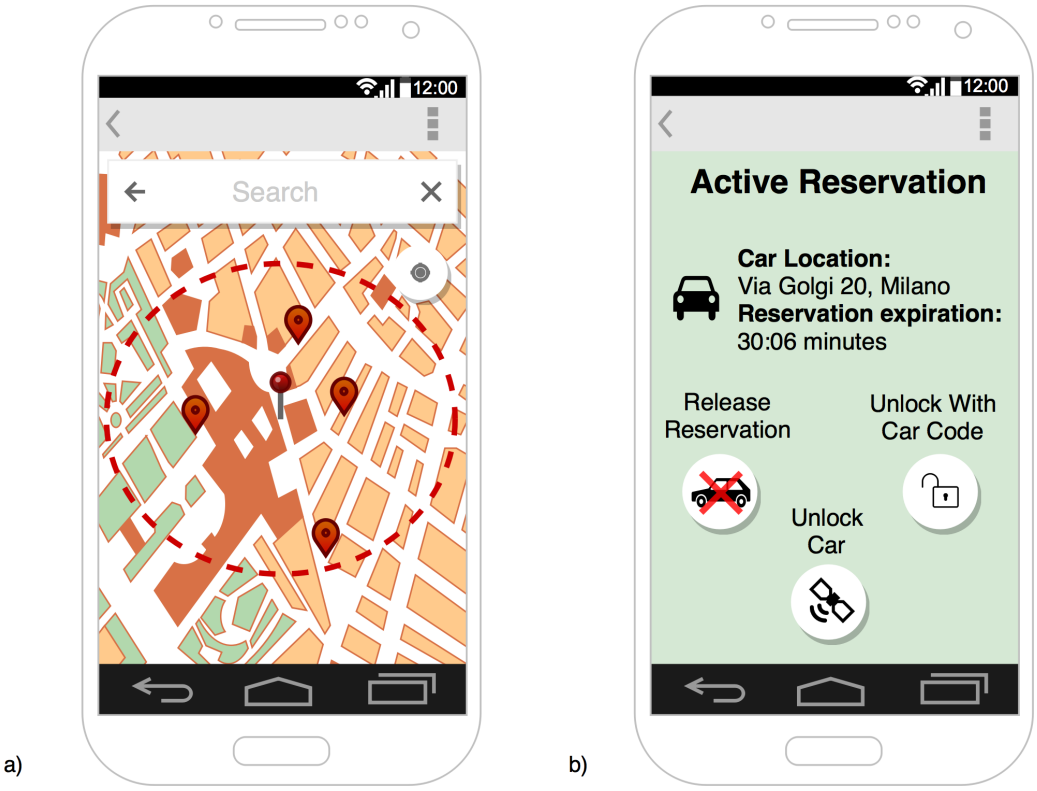

Figure 4.5: Look and feel of the "Reserve A Car" screen (a) and the Active Reservation screen (b). They are both accessible from the "Car Reservation" button in the "User Home" screen: the first is loaded if the user has no active reservation while the second otherwise.

4.2.3 On-Board Interface

The following mockups show how the interface of the On-Board Application should look like on the on-board computers of cars.

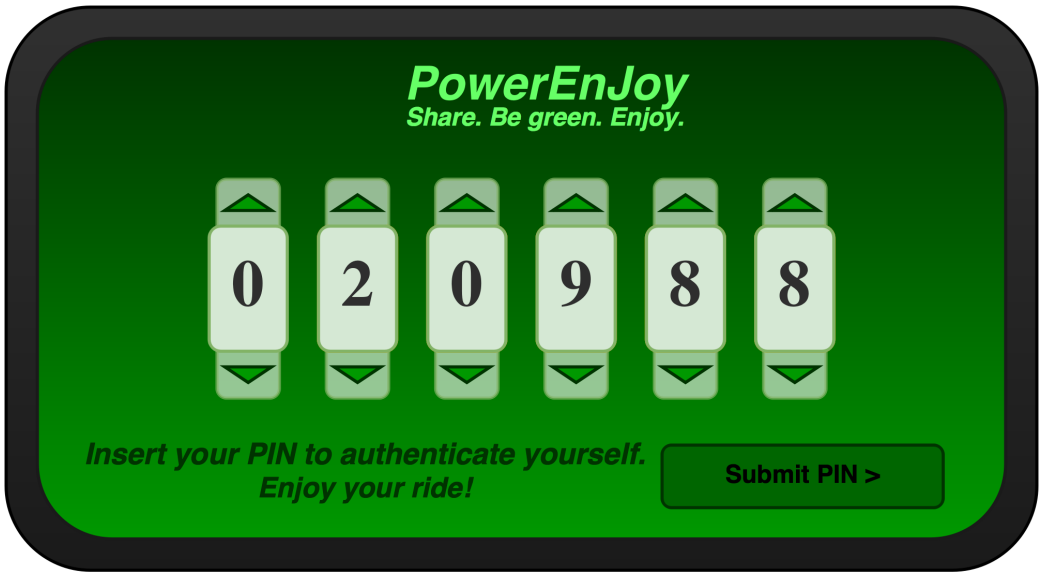

Figure 4.6: Concept of the looks of the "Authentication" screen on the on-board touch screen.

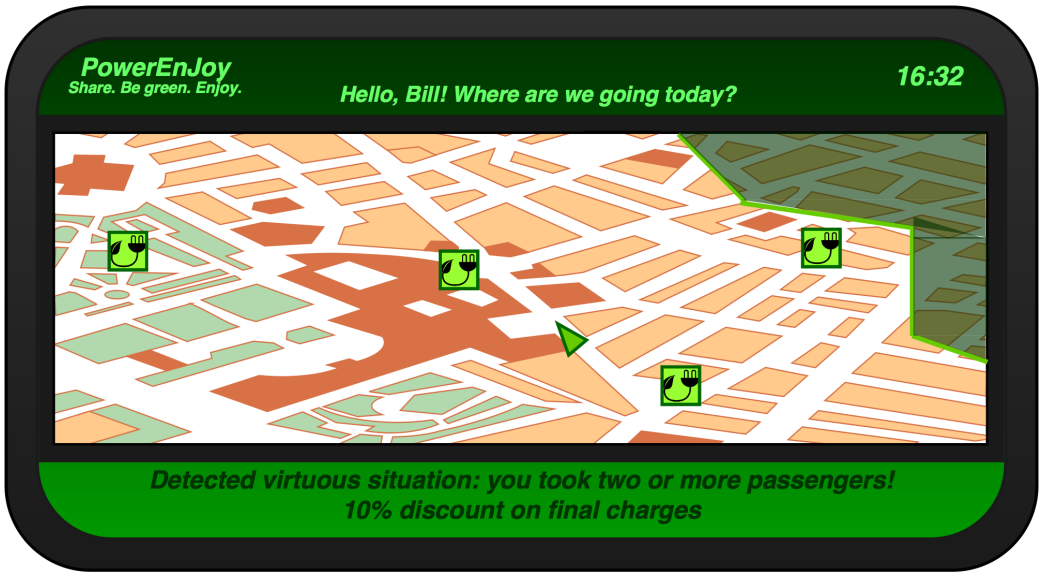

Figure 4.7: Concept of the looks of the "Ride Status" screen on the on-board touch screen.

Section 5

Requirements Traceability

5.1 Functional Requirements

Table 5.1 is aimed to map the functional requirements contained in the related sections of the RASD [1] to the components described in the Design Document. Note that the DD components in the table only include application logic components.

5.2 Non-Functional Requirements

Table 5.2 illustrates the sections of the Design Document that trace subjects related to the satisfaction of the corresponding non-functional requirements from the RASD [1].

Goal	DD Component	RASD Requirements
1	UserManager	3.2.1 Register
2		3.2.2 Login
		3.2.3 Manage Profile
3	ReservationManager	3.2.4 Reserve Car
4		3.2.11 Apply Fee
6	RideManager	3.2.5 Use Car
7		3.2.7 Start Ride
8		3.2.8 End Ride
		3.2.9 Apply Discounts/Additional Charges
		3.2.10 Apply Charges
5	MapManager	3.2.6 Unlock Car
7		3.2.8 End Ride
3	CarStatusManager	3.2.4 Reserve Car
4		3.2.5 Use Car
7		3.2.7 Start Ride
		3.2.11 Apply Fee
5	SecurityAuthenticator	3.2.6 Unlock Car
7		3.2.7 Start Ride (authentication method)
8	DiscountProvider	3.2.9 Apply Discounts/Additional Charges
6	PaymentGateway	3.2.10 Apply Charges (charges computation)
		3.2.12 Manage Payment
1	NotificationManager	3.2.1 Register (confirmation email)
		3.2.2 Login (forgot password)
5	On-Board Application	3.2.6 Unlock Car
7	Client	3.2.7 Start Ride
8		3.2.9 Apply Discount/Additional Charges

Table 5.1: Mapping between DD components and RASD functional requirements.

DD Section	RASD requirements
2.8 Other Decisions 2.6.2 Web Server - Web Browser	3.4 Security
2.6.2 Web Server - Web Browser	3.1.4 Communication Interfaces
2.2 High Level Components	3.4 Availability
2.3.6 Implementation Choices	3.4 Portability
2.4 Deployment View	3.4 Reliability
2.3.6 Implementation Choices	2.4.3 Parallel Operation
4 User Interface Design	3.1.1 User Interfaces

Table 5.2: Mapping between DD sections and RASD non-functional requirements.

Appendix A

Appendix

A.1 Software and tools used

- LaTeX, used as typesetting system to build this document.
- draw.io - `https://www.draw.io` - used to draw diagrams and mockups.
- StarUML Free - `http://staruml.io/` - used to produce some of the diagrams contained in the document.
- GitHub - `https://github.com` - used to manage the different versions of the document and to make the distributed work much easier.
- GitHub Desktop, the GitHub official application that offers a seamless way to contribute to projects.

A.2 Hours of work

The absolute major part of the document was produced in group work. The approximate number of hours of work for each member of the group is the following:

- Giovanni Scotti:
- Marco Trabucchi:

NOTE: indicated hours include the time spent in group work.

Bibliography

[1] AA 2016/2017 Software Engineering 2 - *Requirements Analysis and Specification Document* - Giovanni Scotti, Marco Trabucchi

[2] AA 2016/2017 Software Engineering 2 - *Project goal, schedule and rules*

[3] IEEE Standard 1016:2009 *System design - Software design descriptions*