Software Engineering 2: PowerEnJoy

Design Document

Nardo Loris, Osio Alberto

Politecnico di Milano

1 Introduction

1.1 Purpose

This is the Design Document for the PowerEnJoy system. Its aim is to completely describe the logical and physical architecture of the system, along with some guidelines for most critical algorithms and some dynamic views of the system. This document will present content using mainly UML standards, along with textual descriptions.This document does not include mock ups of the application, as they were already included in the Requirements Analysis and Specification Document. This document is written for project managers, developers, testers, and Quality Assurance people.

1.2 Scope

The system is an electrical car sharing management system. Its main goal is to provide easy access to the service for the end user and to incetivize virtuous behaviours of the same users.

The system addresses to two types of users:

- PowerEnjoy employee (Employee)
- Generic people (User)

Users must be allowed to reserve and use cars, employees must be allowed to manage the parameters of the service (geographical regions and safe areas).

This document is written in the context defined in the RASD, and aims at mapping requirements defined in the same RASD on software components which are to be deployed on real machines. Components must be designed as to have the highest possible cohesion and the lowest possible coupling, so they must encapsulate a single functionality. This also leads to high reusability of the same modules.

Design patterns and architectural styles will be used for solving architectural problems in order to achieve good performances of the system and to build a highly maintanable and modular system.

1.3 Definitions, acronyms and abbreviations

- **System**: the system to be developed for PowerEnJoy
- **User**: a generic person interacting with the system
- **Employee**: a person who works for PowerEnJoy
- **Actor**: can refer to both users and employees
- **Car**: an electric vehicle owned by the company
- **Bill**: an amount of money a user has to pay. It is related to only a single ride
- **Pending bill**: a bill that the user has not paid yet
- **Paid bill**: a bill that the user has already paid for
- **Area**: a space delimited by a polygonal line whose vertices are a set of geographical coordinates
- **Geographical coordinates**: a tuple of latitute and longitude describing a location on Earth
- **Geographical region**: an area where a user can reserve at most one car. They do not overlap
- **Safe area**: an area where it is possible to park a car and optionally to recharge it, in order to make it available for another user
- **Safe parking area**: a special safe area where it is not possible to recharge the car
- **Recharging station area**: a special safe area where it is possible to plug the car for recharging its battery
- **Registered user**: a user who has completed the sign up process
- **Logged user**: a user who has completed the log in process and has not yet started the log out process
- **Banned user**: a registered user who cannot reserve a car until all his pending bills are estinguished
- **Reservor user**: the user who has made a reservation for the specific car. A user is considered the reservor user of a car until the reservation expires or the user is charged with the bill
- **Available car**: a locked car for which no reservation exists
- **Reserved car**: a locked car for which it exists a user who has reserved it
- **Becoming available car**: an unlocked car is said to be "becoming available" as soon as all the passengers and the driver of this car exits the car, the doors of the car are closed and it is parked in a safe area
- **In maintenance car**: a locked car is said to be "in maintenance" as soon as its battery level is below 20%, the car cannot be reserved

- **In use car**: an unlocked car which is not becoming available
- **GPS**: A system capable of providing the location of a receiver device with a good precision (5 meters)
- **Overlapping areas**: Two areas are said to be overlapping if there exists at least one geographical coordinates which is contained inside the two areas
- **Expiration of a car reservation**: when a reservation expires, the car becomes available again, the reservor user loses his reservation and he is charged a fee of 1€
- **Percentage delta**: a discount or a raise based on percentage
- **Applying a raise** or **a discount**: The operation of increasing or reducing the amount of a bill for a specific reason. The amount is computed just before the system charges the user of a bill, and then all those amounts (each one related to a specific reason) are algebraically added to the same bill.
- **RASD**: Requirements Analysis and Specification Document
- **DBMS**: DataBase Management System

1.4 Reference Documents

1. Project rules of the Software Engineering 2 project
2. Template for the Design Document
3. Requirement Analysis and Specification Document (previous deliverable)

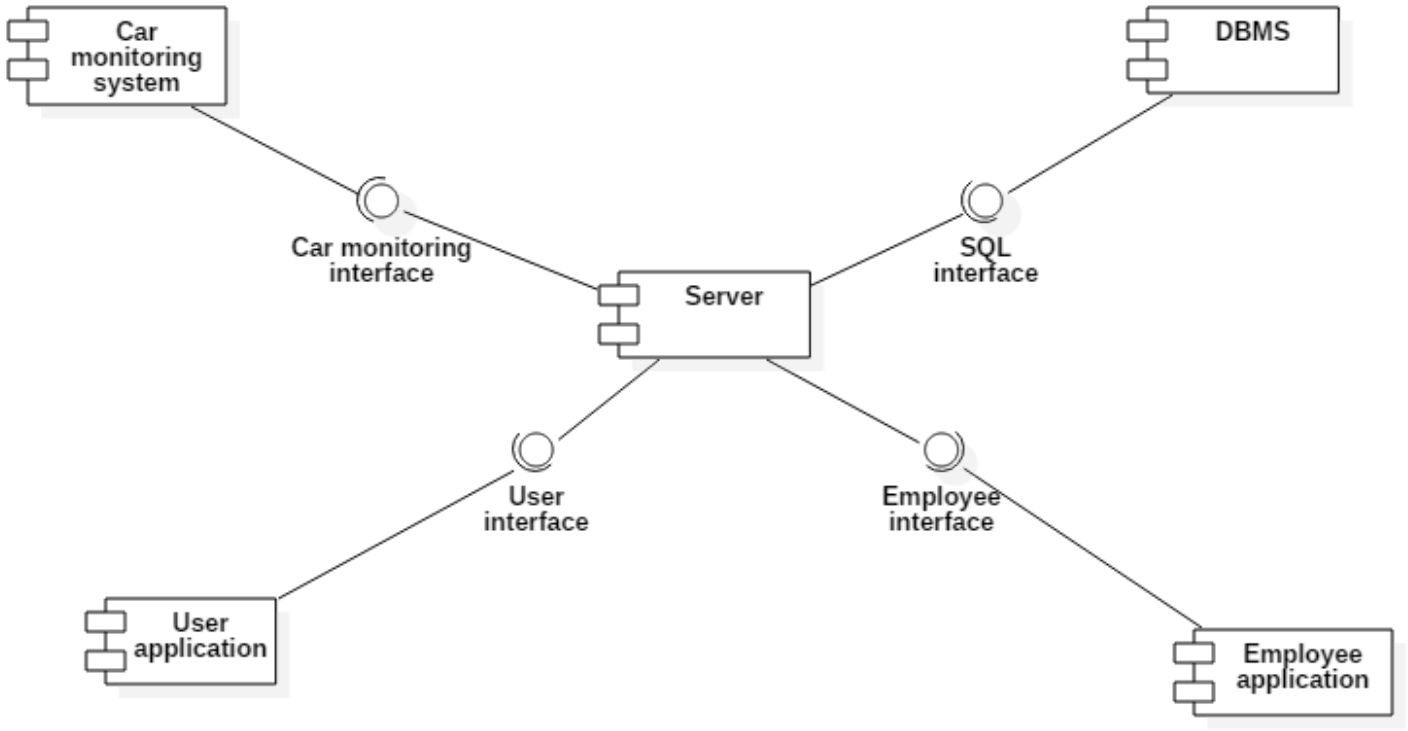
1.5 Document structure

The document is organized in six sections:

1. **Introduction**: describes the overall structure of the whole document
2. **Architectural design**: describes both the logical architecture and the deployment of the system, along with some dynamical views of the presented components
3. **Algorithm design**: describes some guidelines for the implementation of the most critical algorithms
4. **User interface design**: describes a UX model for the user interface (both for users and employees)
5. **Requirements traceability**: presents a mapping of functional and non functional requirements on described components and architectural decisions
6. **Appendix**: contains informations about the effort spent in writing this document and about the references from which further documentation can be obtained.

2 Architectural design

2.1 Overview



The picture shows a representation of the proposed logical architecture. The system is composed of five main components:

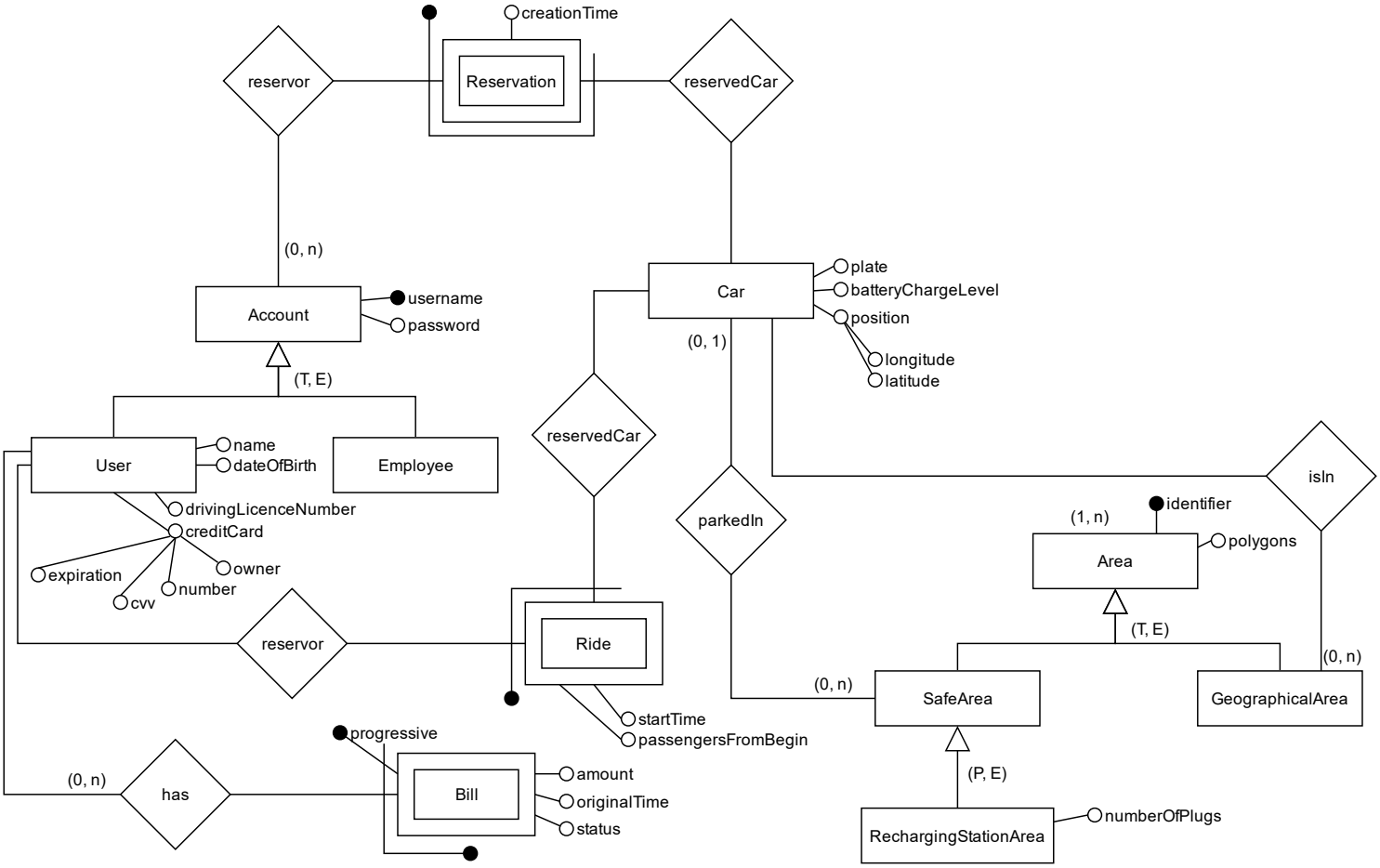
- A *DataBase Management System (DBMS)* used to access data related to cars, reservations, payments, geographical regions and safe areas in a reliable and efficient way
- A *Server* component that implements the application logic both for user-related and employee related features
- A *User Application* component, that takes care of user-related functionalities, making them available to the same users
- A *Employee Application* component, that takes care of employee-related functionalities, making them available to the employees
- A *Car Monitoring System* which models the monitoring system actually installed on cars

Each component is connected only to the *server* component. In this way it is possibile to keep clearly distinguished three logical levels:

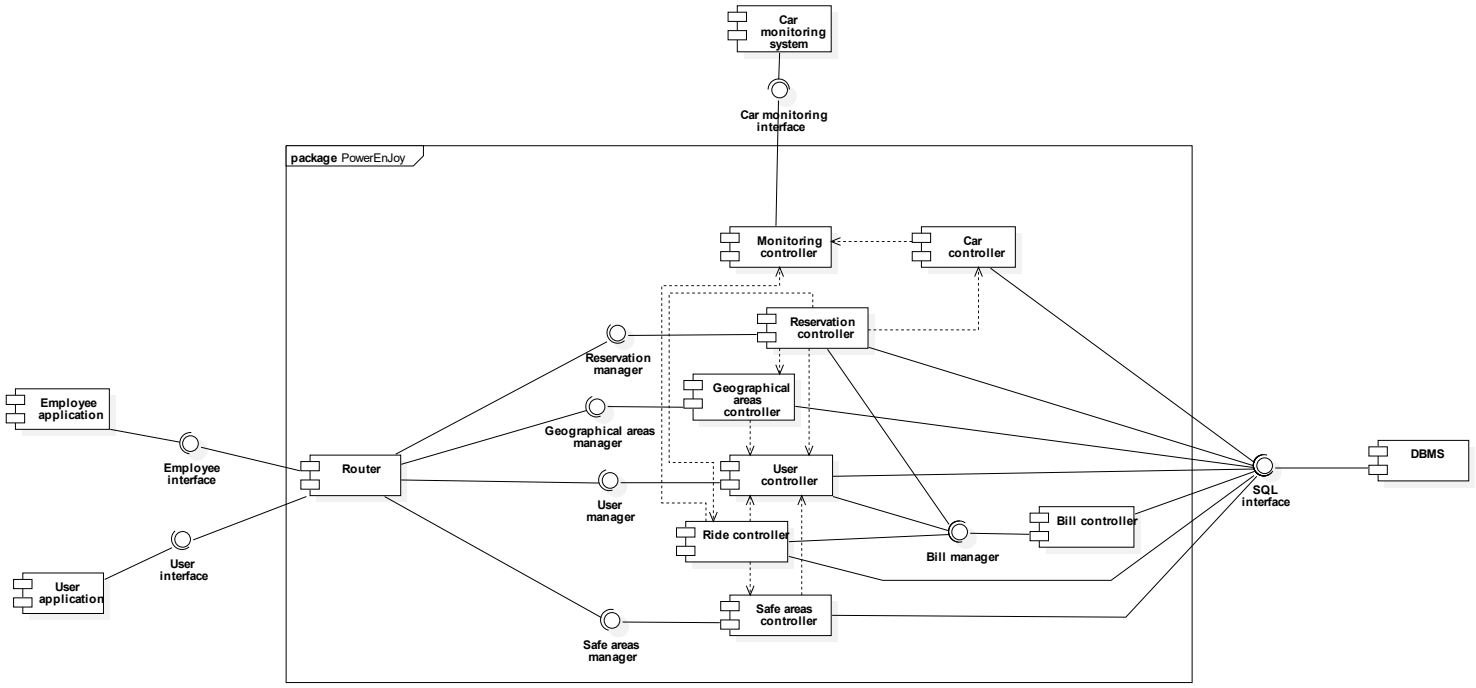
1. The *DBMS*, which represents the *data layer* of the system
2. The *server*, which represents the *application logic layer* of the system. It is the only component to interact with the *DBMS*, as to grant data security and privacy.
3. The *user application* and the *employee application*, which represent the *presentation layer* of the system. They carry out all the tasks related with interaction with end users (both users of the service and employees). They are connected only with the *server* as to preserve data security. Moreover, they are modeled as thin clients, as no application logic function is delegated to them. This was chosen for security reasons, as it is very easy to attack functions delegated to a client living in a browser.

The *monitoring system* actually is part neither of the *presentation layer* nor of the *application logic layer*, as it represents only a kind of *sensor layer*. By the way, it only interacts with the *server* component in a event-driven architectural style.

2.2 DataBase structure



2.3 Component view



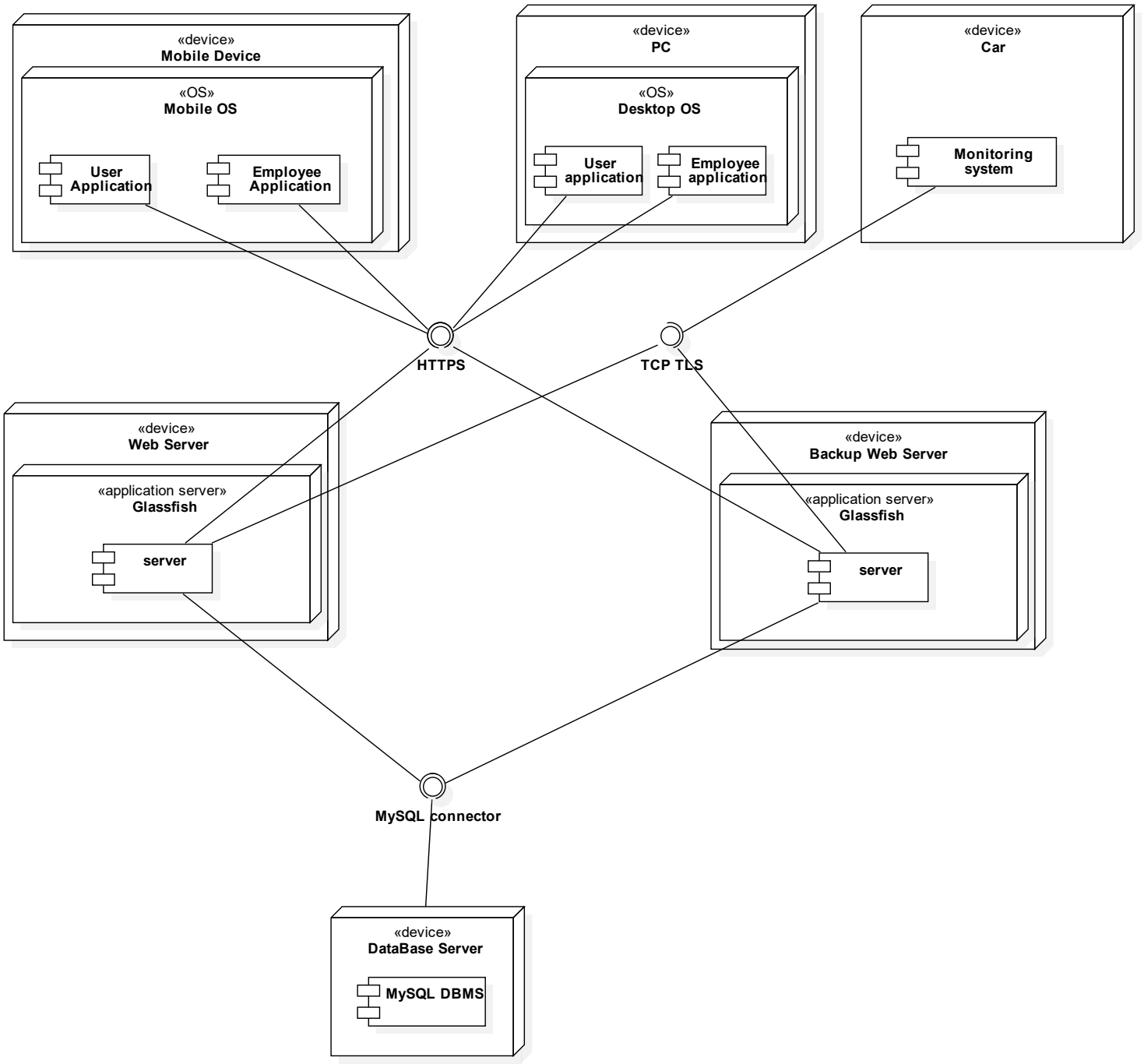
The picture shows the logical architecture for the *Server* component. The architecture is made up of several different components, each one devoted to a specific task:

- **Router** component dispatches all the requests coming from the *User Application* and *Employee application* to

the business component that can handle it. It multiplexes the parts of the interfaces of the different business components in two "virtual" interfaces, the former dedicated to user interaction, the latter to employee interaction.

- **Monitoring controller** represents the broker of the event-driven architectural style used to collect data from cars
- **Car controller** controls the state of each car and to trigger transitions between car states according to the state chart defined in the Requirements Analysis and Specification Document.
- **Bill controller** handles tasks related to the storage of pending bills and to the interaction with the external system for payments processing
- **User controller** controls the state of each user according to the state chart defined in the RASD. In particular, it provides all the functionalities needed for authenticating users and employees.
- **Geographical areas controller** provides an API that can be used to retrive and manipulate data related to geographical areas boundaries
- **Safe areas controller** provides an API with methods used to retrive and manipulate data about safe areas (both safe parking areas and recharging stations)
- **Reservations controller** provides the functionalites needed for creating and canceling a reservation and for "using" a reservation to actually take the reserved car
- **Ride controller** controls each ride in progress in the given instant of time, monitors the state of the car during the ride (in particular, the number of passengers and the battery level) and computes the bill at the end of the ride, applying proper discounts or raises on the calculated fee.

2.4 Deployment view

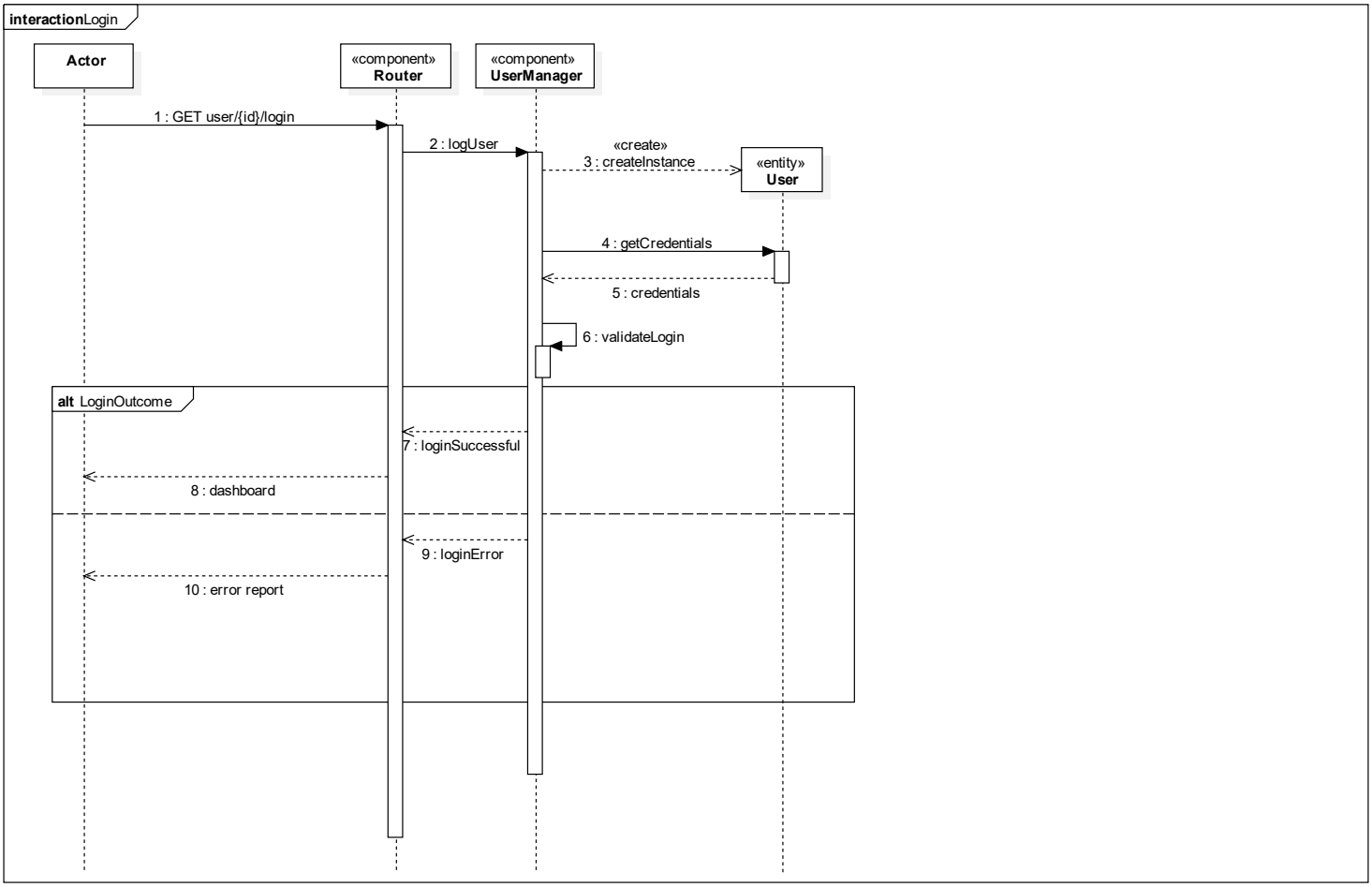


This picture shows how the system should be deployed. The database server and the application server are deployed on two different physical machines, in order to have more security for data and to achieve a decoupled architecture that can be replicated for reliability reasons. This way we can have a main application server, and a backup application server, that are almost identical as of deployed components. The HTTPS protocol, and the routing IP protocol beside it, are configured so as to send all the request to the main application server, and if that server is not available, to the backup application server. This also gives the possibility to carry out maintenance tasks on the system without bringing it completely down, just working on a server at a time. Finally, the DBMS server is not replicated for the money saving reasons. It is not so likely that an attacker can gain access to the DBMS server, and so there is no need of invest money in a distributed DBMS.

2.5 Runtime view

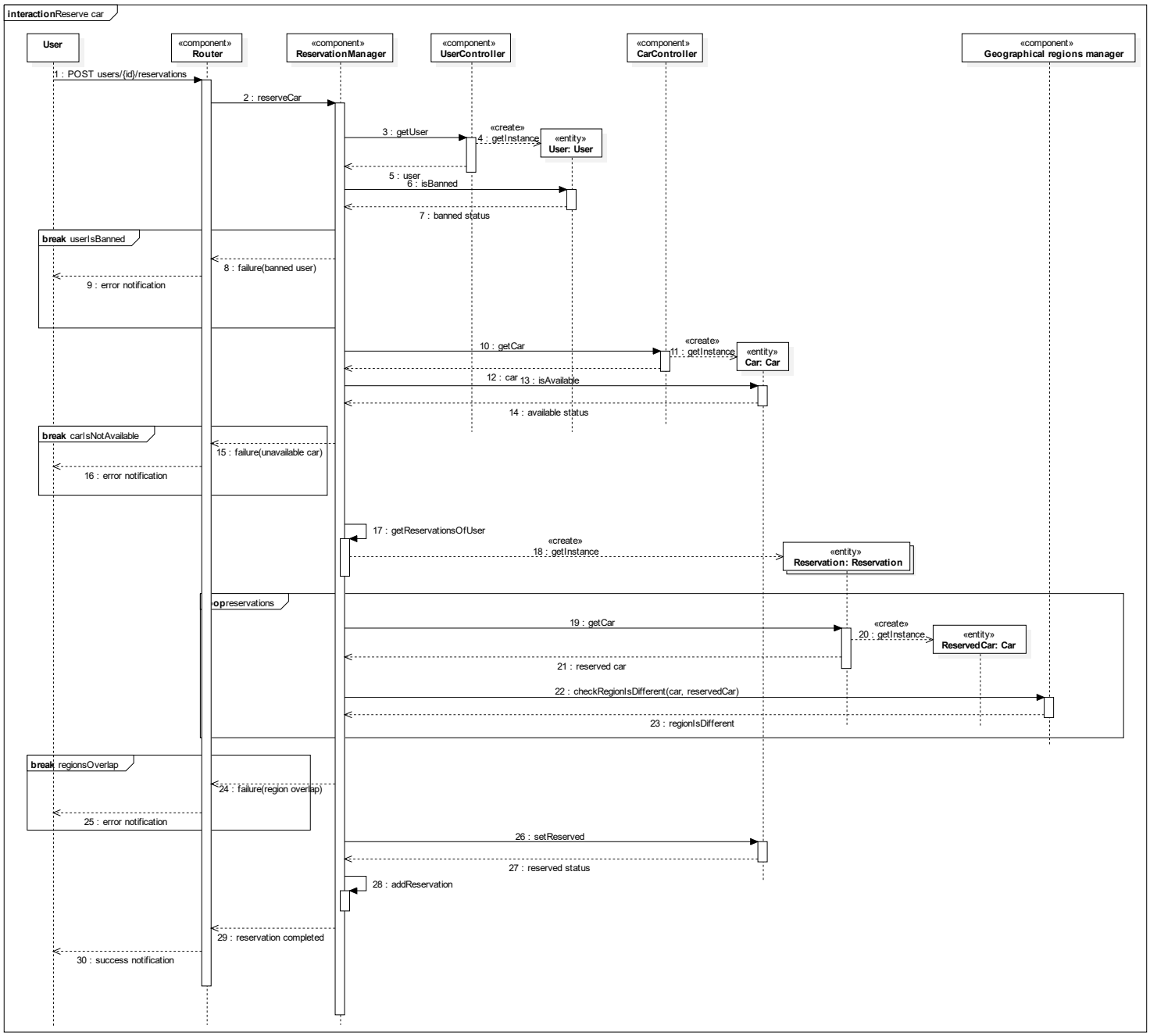
This section describes several interesting dynamic behaviours of the system.

2.5.1 Login



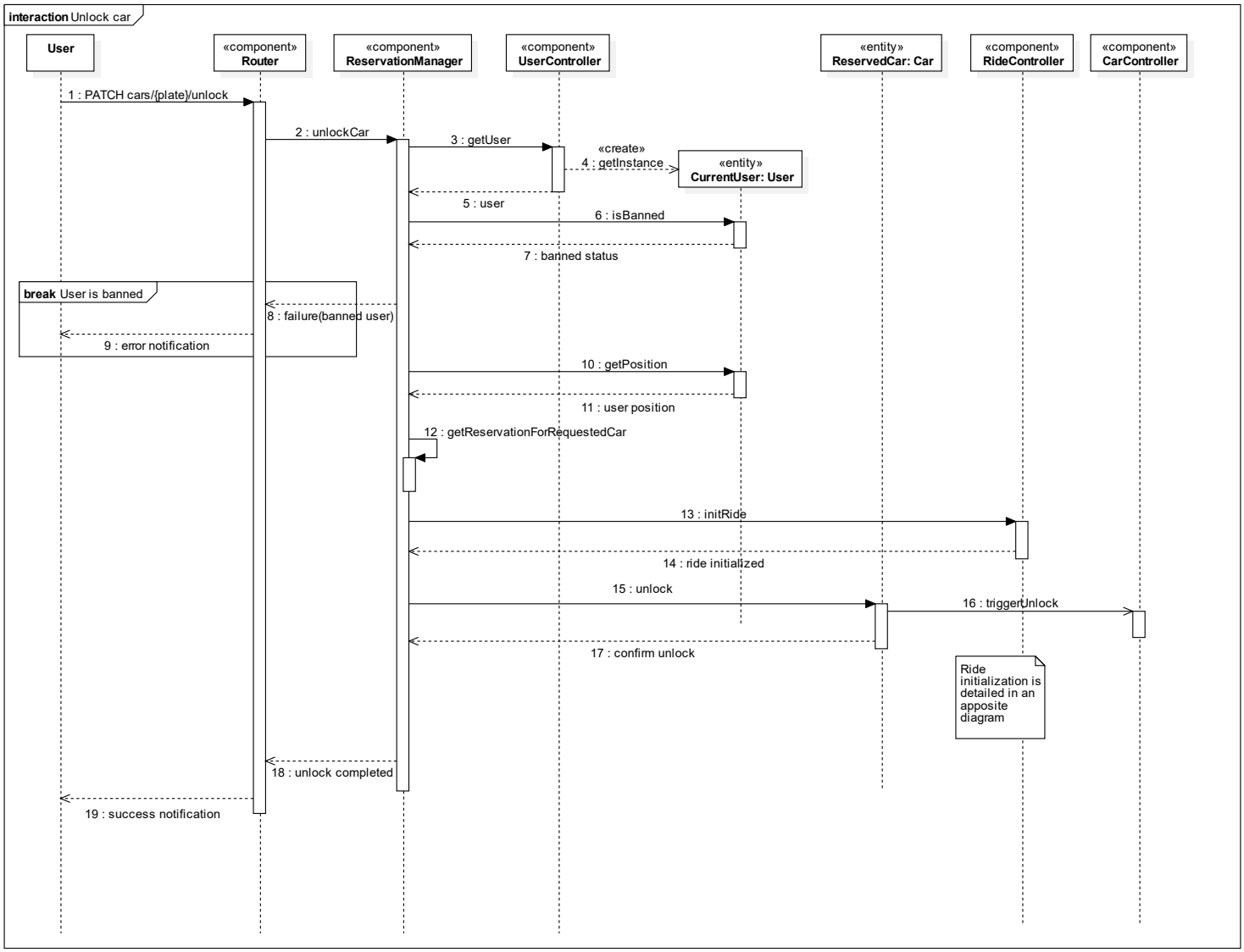
This diagram describes the interaction between the components involved in user login. The "actor" lifeline models the human actor (both employee and final user) trying to login to the system.

2.5.2 Car reservation



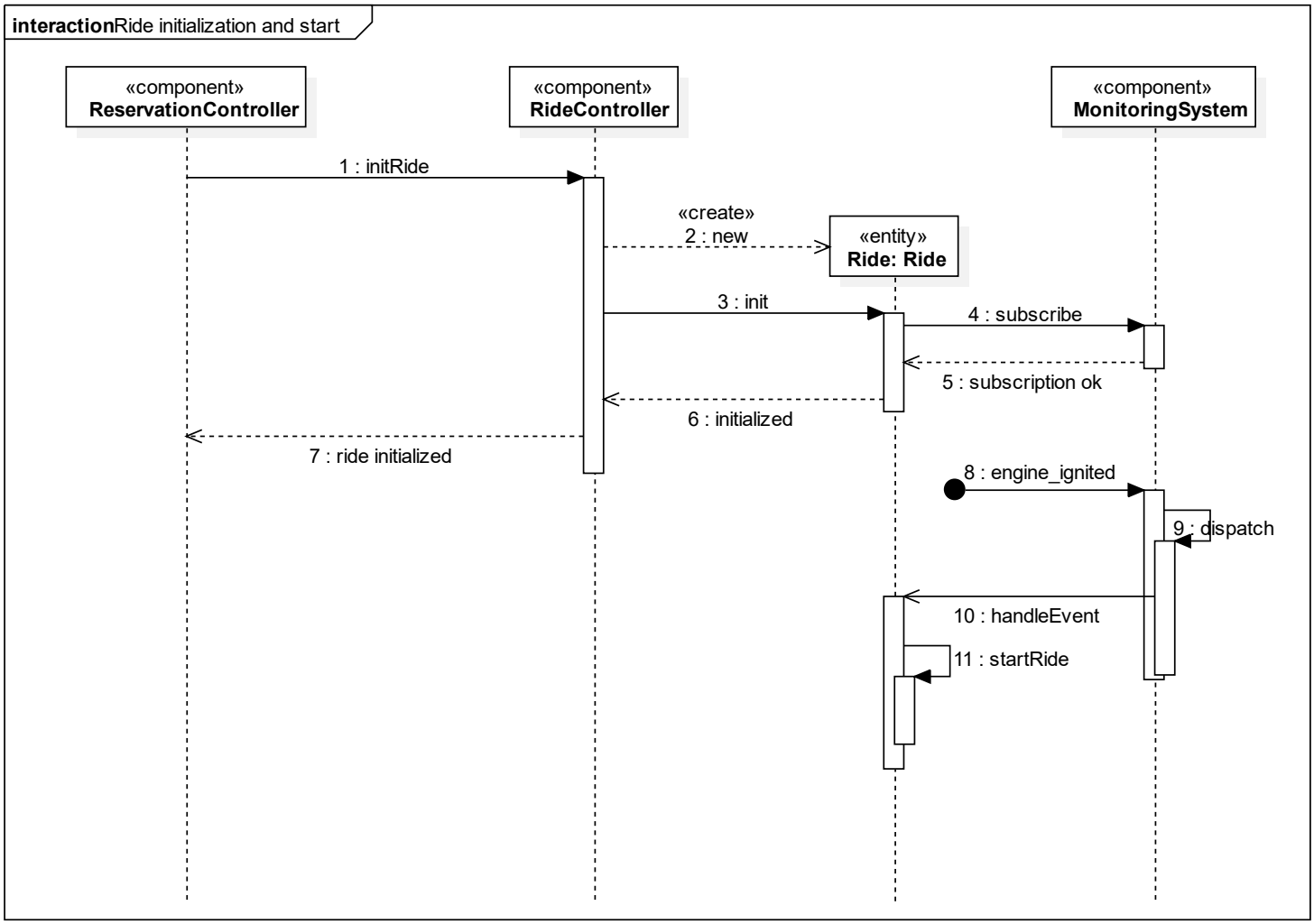
This diagram describes the interaction between the components involved in the reservation of a car. The "user" lifeline models a human user trying to reserve a car. Error conditions are modeled as *break* frames

2.5.3 Car unlock



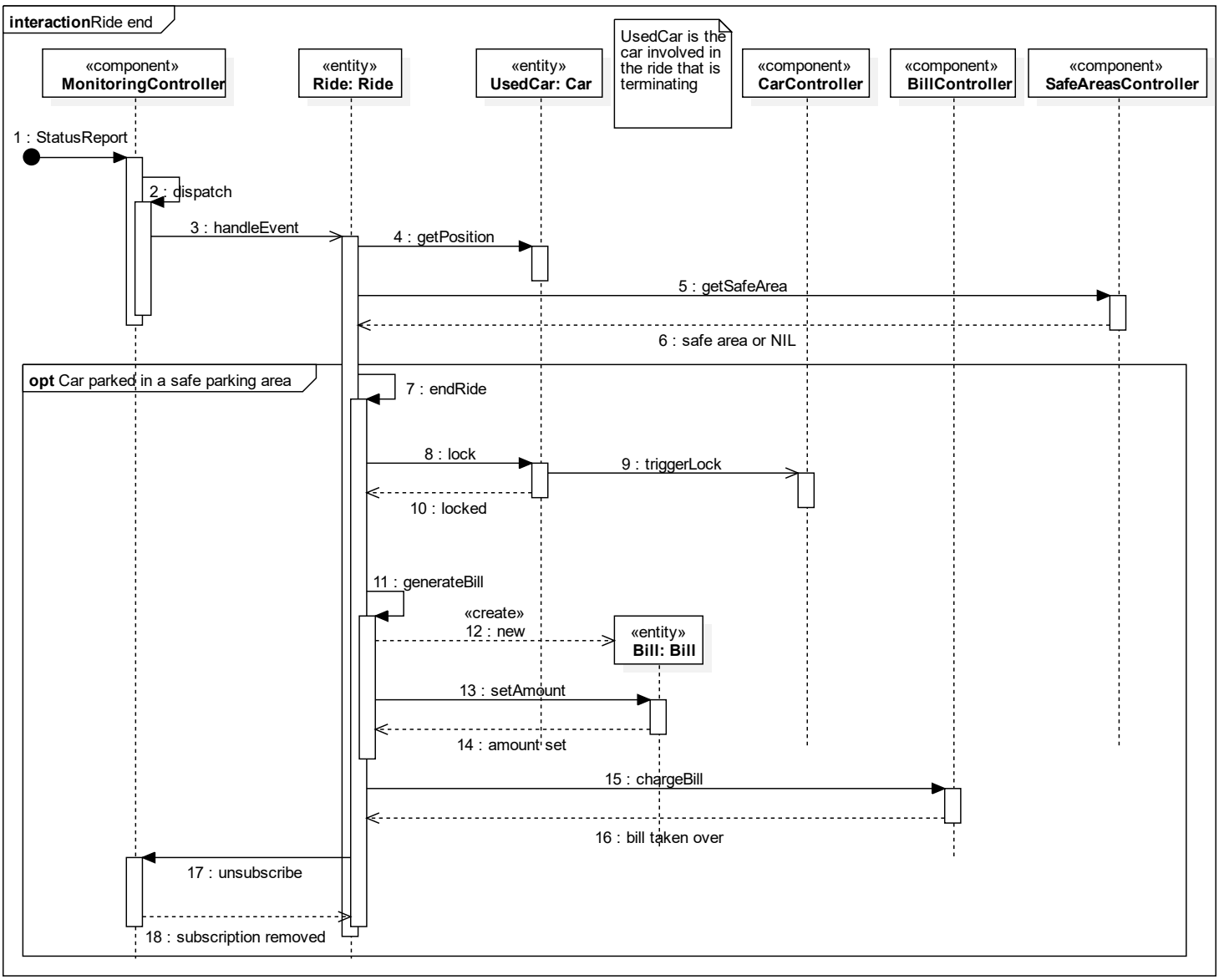
This diagram describes the interaction between components involved in the procedure of unlocking a car. The "user" lifeline models the human user trying to unlock a car. The procedures searches for a reservation which maps to the car requested by the user (this is the meaning of the "getReservationForRequestedCar" message), and then sends to the car a *Unlock* event throw the event-driven monitoring system connection (invoked by the car controller). During the procedure a new ride entity is created to model the incipent ride. For details about the creation of this ride entity refer to the next sequence diagram.

2.5.4 Ride entity creation and ride start



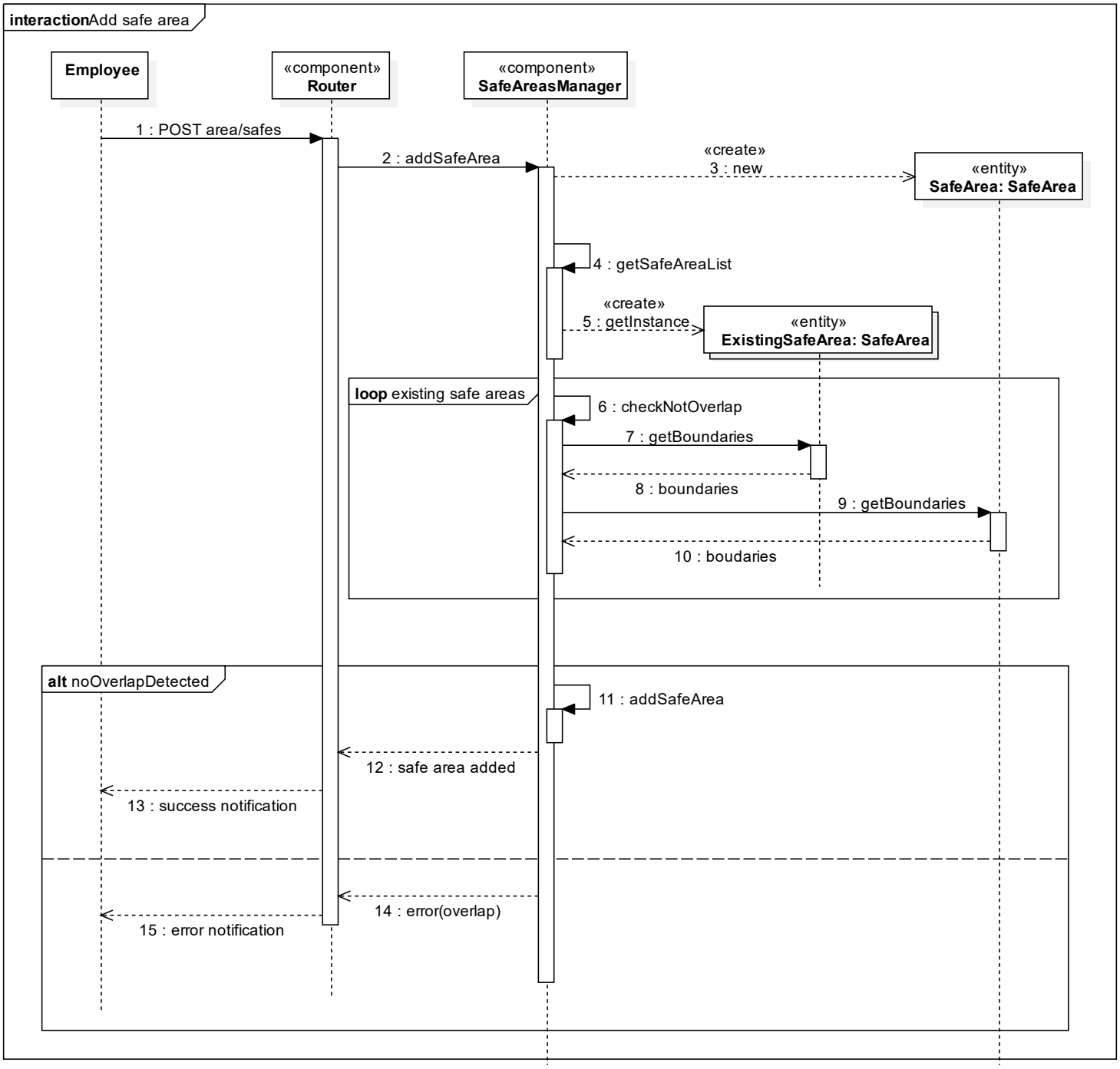
This diagram shows the interaction between components involved in the creation of a new ride entity and in the starting procedure of that same ride. No human actor appears here, just because this is an interaction that is fully carried out inside the *server* component. The first part of the diagram is aimed at specifying the steps involved in the entity creation and consequent subscription to events sent by the car. The second part specifies the interaction aimed at handling the event relative to the engine ignition (and, as of the RASD, to the ride start).

2.5.5 Ride conclusion and bill computation



This diagram refers to the situation in which the car is parked in a safe area and not in a recharching station (otherwise, a timeout of 5 minutes should be added before locking the car). This diagram shows the interaction between components involved in the conclusion of a ride. The interaction is triggered by an event sent from the car, that signals that all passengers have got off the car. The *ride* entity models the ride that is listening on that event, and that will end. At ride end, a new bill is created and charged to the user whom the ride is referred to, and then the ride unsubscribes from the events sent by the car, as they are of no interest anymore.

2.5.6 Insertion of a new safe area



This diagram shows the interaction between components involved in the insertion of a new safe area. It can be seen as a general model for all the interaction involving an employee, as all of them follow the same message pattern (adapted to the operation being carried out).

2.6 Component interfaces

There are two different kinds of interfaces: RESTful APIs are used with client-server architectural style, and Messages are used in event-driven interactions

2.6.1 RESTful APIs

The token mentioned in the /users/{id}/login must be have the following features:

- It must provide an expiration time
- It must univocally identify a user without quering the database

- It must provide the user type (employee, user) without quering the database
- It must be secure and thus it cannot be easily changed or guessed

GET `/users/{id}/login`

It allows a user or an employee to login, the returned token must be used to access all the other api except for `/users/{id}/register`. It is subject to expiration after a predefined amount of time

Parameters

Field	Type	Description
id	String	The username preceded by `user:`

Fields

Field	Type	Description
password	String	The password associated to the account

Success 200

Field	Type	Description
token	String	The token which grant access to the other api

Error 4xx

Field	Description
UserOrPasswordInvalid	The username or the password is invalid

POST `/users/{id}/register`

It allows an unregistered user to register

Parameters

Field	Type	Description
id	String	The username preceded by `user:`

Fields

Field	Type	Description
password	String	The password associated to the account
name	String	The real first and last name
dateOfBirth	String	The date of birth in `dd-MM-yyyy` format
creditCard	Object	The credit card information
number	String	The credit card number
owner	String	The credit card owner
cvv	String	The credit card cvv
expiration	String	The credit card expiration in `dd-MM-yyyy` format
drivingLicenceNumber	String	The driving licence number

Error 4xx

Field	Description
UsernameAlreadyExists	There is already a user with such a username
InvalidDrivingLicence	The driving licence inserted is invalid or it is not associated to this user
InvalidCreditCard	The credit card inserted is invalid or expired

GET `/users/{id}/bills`

It retrives all the pending bills a user has still to pay

Parameters

Field	Type	Description
id	String	The username preceded by `user:` or `me` for the current user

Success 200

Field	Type	Description
bills	Object[]	An array of pending bills
dateTime	String	The datetime in `dd-MM-yyyyThh:mm:ss` format
amount	String	The amount in Euro the user must pay, the fractional part is preceded by a `.`

DELETE `/users/me/bills`

It allows a user to pay the pending bills in a non defined order, if the user cannot pay all the pending bills there is no rollback

POST `/users/{id}/reservations`

It allows a user or an employee to reserve a car

Parameters

Field	Type	Description
id	String	The username preceded by `user:` or `me`

Fields

Field	Type	Description
car	String	The plate associated with the car the user wants to reserve

Error 4xx

Field	Description
CarUnavailable	The car for some reason is not available, or the plate is not valid
QuotaExceeded	The user has already reserved a car in the same geographical area
UserBanned	The user is banned and thus he cannot reserve any car

GET `/users/{id}/reservations`

It allows a user or an employee to obtain the list of the reservation made by the user

Parameters

Field	Type	Description
id	String	The username preceded by `user:` or `me` or `all`

Fields

Field	Type	Description
position optional	Object	The center to search for reservations
latitude	Number	The latitude of the position
longitude	Number	The longitude of the position
radius optional	Number	The maximum distance in meters from the position to search for reservations

Success 200

Field	Type	Description
reservations	Object[]	An array of all the reservations in the search area for the specified user
car	String	The plate of the car
position	Object	The position where the car is located
latitude	Number	The latitude of the position
longitude	Number	The longitude of the position
creationTime	String	The datetime in `dd-MM-yyyyThh:mm:ss` format

GET `/users/{id}/reservations/{plate}`

It allows a user or an employee to obtain information about a specific reservation

Parameters

Field	Type	Description
id	String	The username preceded by `user:` or `me` or `all`
plate	String	The plate of the car for which the user expressed by the id parameter has made a reservation

Success 200

Field	Type	Description
reservations	Object	The information for the specific reservation
car	String	The plate of the car
position	Object	The position where the car is located
latitude	Number	The latitude of the position
longitude	Number	The longitude of the position
creationTime	String	The datetime in `dd-MM-yyyyThh:mm:ss` format

Error 4xx

Field	Description
NoReservationFound	There is no reservation for this tuple of parameters

GET /cars

It allows a user or an employee to obtain a list of all cars in an area.In the case of a user, the `position` or the `location` and `radius` field are mandatory. The `position` and `location` are mutually exclusive

Fields

Field	Type	Description
position optional	Object	The center to search for cars
latitude	Number	The latitude of the position
longitude	Number	The longitude of the position
location optional	String	The location expressed as a string suitable for geocoding
radius optional	Number	The maximum distance in meters from the position to search for cars
status	String[]	The admissible status of the cars returned

Success 200

Field	Type	Description
cars	Object[]	An array of cars
position	Object	The position where the car is located
latitude	Number	The latitude of the position
longitude	Number	The longitude of the position
status	String	The status of the car
plate	String	The plate of the car
batteryChargeLevel	Number	The normalized percentage (0-1), of the battery charge
geographicalLocation	Number	The identifier of a geographical region
parkingLocation optional	Number	The identifier of a safe area where the car is parked

GET `/cars/{plate}`

It allows a user or an employee to obtain information about a specific car

Parameters

Field	Type	Description
plate	String	The plate of the car to search for

Success 200

Field	Type	Description
cars	Object	The car whoose plate is the same as the parameter
position	Object	The position where the car is located
latitude	Number	The latitude of the position
longitude	Number	The longitude of the position
status	String	The status of the car
plate	String	The plate of the car
batteryChargeLevel	Number	The normalized percentage (0-1), of the battery charge
geographicalLocation	Number	The identifier of a geographical region
parkingLocation optional	Number	The identifier of a safe area where the car is parked

Error 4xx

Field	Description
NoCarFound	The plate is not associated with a car

PATCH `/cars/{plate}/unlock`

It allows a user to unlock the specific car

Parameters

Field	Type	Description
plate	String	The plate of the car to unlock

Fields

Field	Type	Description
position	Object	The position where the user is located
latitude	Number	The latitude of the position
longitude	Number	The longitude of the position

Error 4xx

Field	Description
NoCarFound	The plate is not associated with a car
FarUser	The user is too far from the car

GET /area/geographicals

It allows an employee to obtain the list of all the geographical regions

Success 200

Field	Type	Description
areas	Object[]	An array of all the defined geographical region
id	Number	The identifier of this region
path	Object[]	The path of this geographical region
latitude	Number	The latitude of a single position in the path
longitude	Number	The longitude of a single position in the path

GET /area/geographicals/{id}

It allows an employee to obtain information about a specific geographical region

Parameters

Field	Type	Description
id	Number	The identifier of a geographical region

Success 200

Field	Type	Description
area	Object	The geographical region whoose id matches the one in the parameters
id	Number	The identifier of this region
path	Object[]	The path of this geographical region
latitude	Number	The latitude of a single position in the path
longitude	Number	The longitude of a single position in the path

Error 4xx

Field	Description
NoRegionFound	The identifier is not associated with a valid geographical region

PATCH `/area/geographicals/{id}/split`

It allows an employee to split a geographical region

Parameters

Field	Type	Description
id	Number	The identifier of a geographical region

Fields

Field	Type	Description
path	Object[]	The path used to split the region
latitude	Number	The latitude of a single position in the path
longitude	Number	The longitude of a single position in the path

Success 200

Field	Type	Description
areas	Object[]	The new geographical regions replacing the one splitted
id	Number	The identifier of this region
path	Object[]	The path of this geographical region
latitude	Number	The latitude of a single position in the path
longitude	Number	The longitude of a single position in the path

Error 4xx

Field	Description
NoRegionFound	The identifier is not associated with a valid geographical region

PATCH /area/geographicals/{id}/merge

It allows an employee to merge two geographical regions

Parameters

Field	Type	Description
id	Number	The identifier of a geographical region

Fields

Field	Type	Description
id	Number	The identifier of the region to be merged with this one

Success 200

Field	Type	Description
area	Object	The new geographical region replacing the two merged
id	Number	The identifier of this region
path	Object[]	The path of this geographical region
latitude	Number	The latitude of a single position in the path
longitude	Number	The longitude of a single position in the path

Error 4xx

Field	Description
NoRegionFound	The identifier is not associated with a valid geographical region

GET /area/safes

It allows an employee to obtain the list of all safe areas

Success 200

Field	Type	Description
areas	Object[]	An array of all the defined safe areas
id	Number	The identifier of this area
path	Object[]	The path of this safe area
latitude	Number	The latitude of a single position in the path
longitude	Number	The longitude of a single position in the path
numberOfPlugs optional	Number	The number of plugs if it is a recharging station area
availablePlugs optional	Number	The number of plugs if it is a recharging station area that are not in use

GET /area/safes/{id}

It allows an employee to obtain information about a specific safe area

Parameters

Field	Type	Description
id	Number	The identifier of the safe area

Success 200

Field	Type	Description
areas	Object[]	An array of all the defined safe areas
id	Number	The identifier of this area
path	Object[]	The path of this safe area
latitude	Number	The latitude of a single position in the path
longitude	Number	The longitude of a single position in the path
numberOfPlugs optional	Number	The number of plugs if it is a recharging station area
availablePlugs optional	Number	The number of plugs if it is a recharging station area that are not in use

Error 4xx

Field	Description
NoSafeAreaFound	The identifier is not associated with a valid safe area

DELETE /area/safes/{id}

It allows an employee to remove a safe area

Parameters

Field	Type	Description
id	Number	The identifier of the safe area

Error 4xx

Field	Description
NoSafeAreaFound	The identifier is not associated with a valid safe area

POST `/area/safes`

It allows an employee to insert a new safe area

Fields

Field	Type	Description
path	Object[]	The path of this safe area
latitude	Number	The latitude of a single position in the path
longitude	Number	The longitude of a single position in the path
numberOfPlugs optional	Number	The number of plugs if it is a recharging station area

Success 200

Field	Type	Description
id	Number	The identifier of the newly created safe area

Error 4xx

Field	Description
SafeAreaOverlap	The safe area overlaps with an already present safe area

PATCH `/area/safes/{id}`

It allows an employee to modify a safe area

Parameters

Field	Type	Description
id	Number	The identifier of the safe area

Fields

Field	Type	Description
path	Object[]	The path of this safe area
latitude	Number	The latitude of a single position in the path
longitude	Number	The longitude of a single position in the path
numberOfPlugs optional	Number	The number of plugs if it is a recharging station area

Error 4xx

Field	Description
SafeAreaOverlap	The safe area overlaps with an already present safe area

2.6.2 Messages

Messages are exchanged using a protocol based on TCP TLS.

2.6.2.1 StatusReport

This message is sent by the car system, to communicate variations of the car status. It is sent immediatly after the status or the number of passengers change or the batteryLevel drops more than 5% or the batteryLevel reaches its minimum or maximum value, or every 15 minutes if the batteryLevel changes by less than 5% and nothing else happens in the meantime.

Field	Type	Description
batteryLevel	float	The normalized percentage of the battery charge level
status	CarStatus	The status of the car
passengers	integer	The number of persons that the car has detected inside
position	Position	The current position of the car

2.6.2.2 Unlock

This message is sent by management system to the car system in order to unlock the car. This message has no parameters

2.7 Selected architectural styles and patterns

There are two different architectural styles used to build the architucture of the system:

- **Client - Server** style is used in the interaction between *user application* and *employee application* and the *server* component. This architectural style supports the request - response pattern, that is the one that mostly fits the way actors interact with the system: they make a request invoking some services provided by the *server*, and the *server* itself provides a response according to the received request.

- **Event - driven** style is used in the interaction between *monitoring system* built on the cars and the *server* component. This architectural style was selected due to two main reasons:

  - *Monitoring system* collects data about the car status on board, and sends them to the server without waiting for a server response, it justs *"sends and forgets"*
  - Different objects living in the server might want to react to events coming from cars

  Both these can be reliably achieved with a event-driven architectural style

- **RESTful APIs**

  - RESTful API are implemented using JAX-RS.
  - RESTful APIs are implemented over the HTTPS protocol used in client-server interaction. This was chosen in order to have a clean and neat API, that is easier to understand, extend and mantain.
  - The content could have been formatted using XML or JSON; for this application the two standards provide the same capability but we opted for JSON since it is easier to be used on client side
  - To allow future changes of the API that will be incompatible with the one defined, all the url of the RESTful API are relative to a version qualifier path "v1/"

2.8 Other design decisions

A possible implementation of the shape of an Area was to use a table with the following columns (*areaid*,*order*,latitude,longitude), but then all the query requesting in which area was a point would have required the scan of the whole table and this was not acceptable, so we opted to use a column named **polygons** in the entity **Area** of type MULTIPOLYGON (http://dev.mysql.com/doc/refman/5.7/en/gis-class-multipolygon.html) as such it can store any geometrical shape composed of segments, and can also be indexed.

2.8.1 Framework selection

Java Enterprise Edition was selected for the implementation of the server components, because we can easily build reliable and scalable application modeling the components as Enterprise Java Beans, and using Java Server Pages for building dynamical user interfaces. Moreover, Java Persistence APIs can be used for the interaction with the DBMS.

2.8.2 DBMS selection

MySQL DBMS was selected because it grants good performance along with no licence cost, in order to reduce system total cost.

2.8.3 Security

- Passwords are not stored in plain text, but are hashed and salted with strong cryptographic functions.
- Payments security is granted by the external system for payments processing.
- Remote communications are carried out using TLS.

2.8.4 Service providers

2.8.4.1 Maps generation and address translation

The system uses *Google Maps (https://maps.google.com)* to carry out map rendering and address translation (into geographical coordinates) in a reliable, well-known and well-tested way. Moreover, this can save the huge cost of the implementation of a new system and of the collection of data.

2.8.4.2 Driving licence validation

The system uses *Il portale dell'automobilista (www.ilportaledellautomobilista.it)* for validating the driving licence numbers. This was the only feasible solution to have access to a updated data source.

2.8.4.3 Payment information validation and payment processing

The system uses *Paypal (www.paypal.com)* to carry out tasks related to payment processing. *Paypal* was chosen because it is a very well-known and well-tested platform, which provides a lot of guarantees about payments and that is very used, so the majority of users of the PowerEnJoy system already has a Paypal account. Moreover, it provides a well-defined set of APIs to carry out all the required tasks.

3 Algorithm design

The algorithms listed here can be useful, but it is not mandatory to implement exactly the following algorithms as long as the result is equivalent

3.1 Computing the bill amount

Assuming time subtraction yields a time offset and PercentageDelta.delta are normalized in the interval [-1, 1], this function computes the bill amount from a ride and a list of all percentageDeltas implemented in the system

```
function computeBillAmount(ride: Ride, discounts: PercentageDelta[]):
     let multiplier = 1;
     let elapsedMinutes = (Time.now() - ride.startTime).totalMinutes
     if ride.car.parkedIn is a RechargingStationArea then:
         wait first of (new Timer(5, "minute") or ride.car.status == "charging")
     for discount in discounts:
        if discount.canBeApplyed(ride) then:
            multiplier = multiplier + discount.delta
     return elapsedMinutes * multiplier * COST_PER_MINUTE
```

3.2 Getting the MULTIPOLYGON representation of a Path

It is assumed that a path is an ordered set of segments, as such it cannot have duplicate items. The entry point for this algorithm is **getMultiPolyRepresentation**.

getMultiPolyRepresentation(*path*: Path): Path[][] This algorithm converts a generic path to a multipolygon representation suitable for storage in a dbms or for application of simpler algorithm

1. Let *polygons* be an empty sequence
2. Iterate over the **complex polygons of *path***, at each time
    1. Let *simplePolygons* be the **simple polygons of the *current polygon***
    2. Iterate over *simplePolygons*, at each time
        1. Let *point* be one random interior point of the *current simple polygon*
        2. Apply the even-odd algorithm to the *current complex polygon* and *point*
        3. If the *point* is outside the *complex polygon*, then mark the *current simple polygon* as "hole"
        4. Otherwise, mark the *current simple polygon* as "fill"
        5. **Insert the *current simple polygon* inside *polygons***
3. Iterate over *polygons*, at each time
    1. Iterate over the remaining part of *polygons*, at each time
        1. If the *two polygons* have at least a segment in common and all the common segments are contiguos, then
            1. Remove from *polygons* the *two polygons*
            2. **Insert inside *polygons* the sequence containing only *merged polygon***
            3. Repeat 3.1
        2. Otherwise, do nothing
4. Iterate over *polygons*, at each time
    1. Let *fill* be the current polygon
    2. If *fill* is marked as "hole", then skip this polygon
    3. Let holes be an empty sequence
    4. Iterate over the remaining part of *polygons*, at each time
        1. Let *newHole* be the *current polygon*
        2. If *newHole* is not contained inside *fill* or it is marked as "fill", then skip this polygon
        3. Iterate over *holes*, at each time
            1. If *newHole* is contained by the *current hole polygon*, then go to the next iteration of 4.4
            2. If *newHole* contains the *current hole polygon*, then remove the *current hole polygon* from *holes*
        4. Add *newHole* to *holes*
    5. Prepend *fill* to *holes*
    6. Append *holes* to *result*
5. Return *result*

getComplexPolygons(*path*: Path): Path[]

This algorithm returns a sequence of complex polygons from a single path object

1. Iterate over the point sequence of *path*, at each time
    1. If the *current point* is not marked yet, then mark it
    2. Otherwise
        1. Collect all marked points in the sequence from the previous occurrence of *current point* (the one that caused that point to be marked) up to *current point* (but do not include this occurrence of *current point*), and name this sequence *currentPath*
        2. Remove the marking for each point in the *currentPath* sequence
        3. Mark the *current point*
        4. **Remove the leaf segments from *currentPath***
        5. If *currentPath* is not empty, then add *currentPath* to the *result* sequence
2. Return the *result*

removeLeafSegments(*path*: Path): Path

This algorithm returns a path where all the trailing segment not connected will be removed from path

1. Let *points* be the point sequence of *path*
2. Iterate over *points* except for the last point, at each time
    1. If the *current point* is equal to the last point in points, then return a path constructed from *points*
3. Remove the last item from *points*
4. If *points* is empty, then return an empty path
5. Repeat the steps from 2

getSimplePolygons(*path*: Path): Path[]

This algorithm returns a sequence of simple polygons starting from a complex polygon

1. Let *segments* be the segment sequence of *path*
2. Iterate over *segments*, at each time
    1. Let *first* be the *current segment*
    2. Iterate over the remaining portion of *segments*, at each time
        1. Let *second* be the *current segment*
        2. If *first* and *second* intersects in the point *intersectionPoint*, then
            1. Let *firstSplit* be the split of first in two segments by *intersectionPoint*
            2. Let *secondSplit* be the split of second in two segments by *intersectionPoint*
            3. Replace *first* with *firstSplit* in *segments*
            4. Replace *second* with *secondSplit* in *segments*
3. Let *newPath* be a path constructed from *segments*
4. Return the **complex polygons from *newPath***, these are not complex anymore

insertSimplePolygonToSequence(*polygon*: Path, *sequence*: Path[])

This algorithm adds the polygon to the sequence preventing any violation of the property of the multipolygon representation

1. If the same point sequence of *polygon* is already in *sequence*, then
    1. Let *presentPolygon* be the polygon whose point sequence is the same as *polygon*
    2. **Update the mark of *presentPolygon* with respect to *polygon***
    3. Return
2. Let *polySequence* a sequence containing only *polygon*
3. **Compute the simple polygon intersection of *sequence* with *polySequence***
4. Iterate over *polySequence*, at each time
    1. **Insert the *current polygon* inside *sequence***

computeSimpleIntersection(*sequence*: Path[], *simple*: Path[])

This algorithm computes the intersection polygons and updates the two sequence inserting these intermediate polygons

1. Iterate over *simple*, at each time:

2. Let *currentSimple* be the *current polygon*
    1. Iterate over *sequence*, at each time:
        1. Let *currentSequence* be the *current polygon*
        2. If *currentSequence* intersects with *currentSimple*, then
            1. Remove *currentSimple* from *simple*
            2. Remove *currentSequence* from *sequence*
            3. Let *segments* be the union of the segments of *currentSimple* with *currentSequence*
            4. Let *polySequence* be the sequence of **simple polygons of the path constructed by *segments***
            5. Iterate over *polySequence*, at each time
                1. call the **addSimpleIntersection**(*simple*, *currentSimple*, *current polygon*)
                2. call the **addSimpleIntersection**(*sequence*, *currentSequence*, *current polygon*)
        3. Otherwise, do nothing

addSimpleIntersection(*sequence*: Path[], *originalPolygon*: Path, *newPolygon*: Path):

This procedure adds a copy of newPolygon to sequence if the originalPolygon intersects with the newPolygon

1. If *originalPolygon* intersects with *newPolygon*, then
    1. Let *copy* be the copy of *newPolygon*
    2. Set the mark of *copy* to the same of *originalPolygon*
    3. **Insert *copy* inside *sequence***
2. Otherwise, do nothing

updateMark(*target*: Path, *source*: Path)

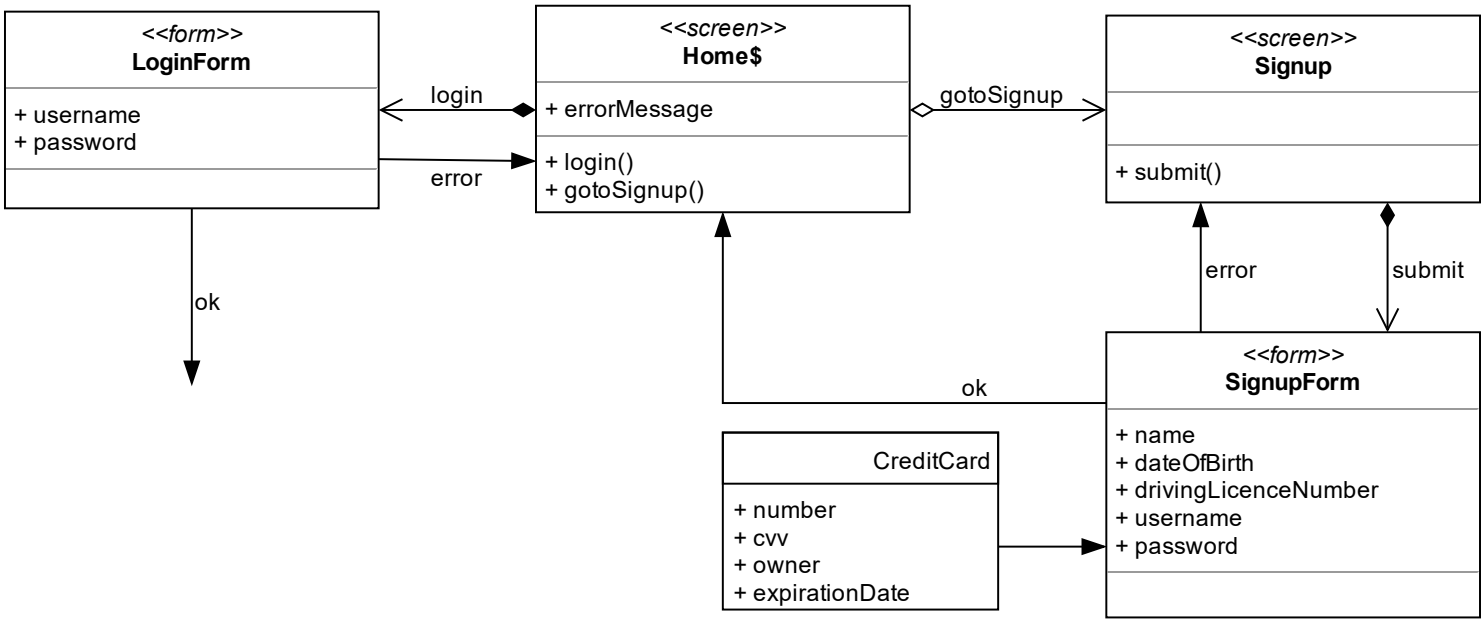
This procedure will assign the correct mark to target

1. If *target* is marked as "fill" and *source* is marked as "fill", then mark *target* as "hole"
2. If *target* is marked as "hole" and *source* is marked as "fill", then mark *target* as "fill"
3. Otherwise, do nothing

4 User Interface Design

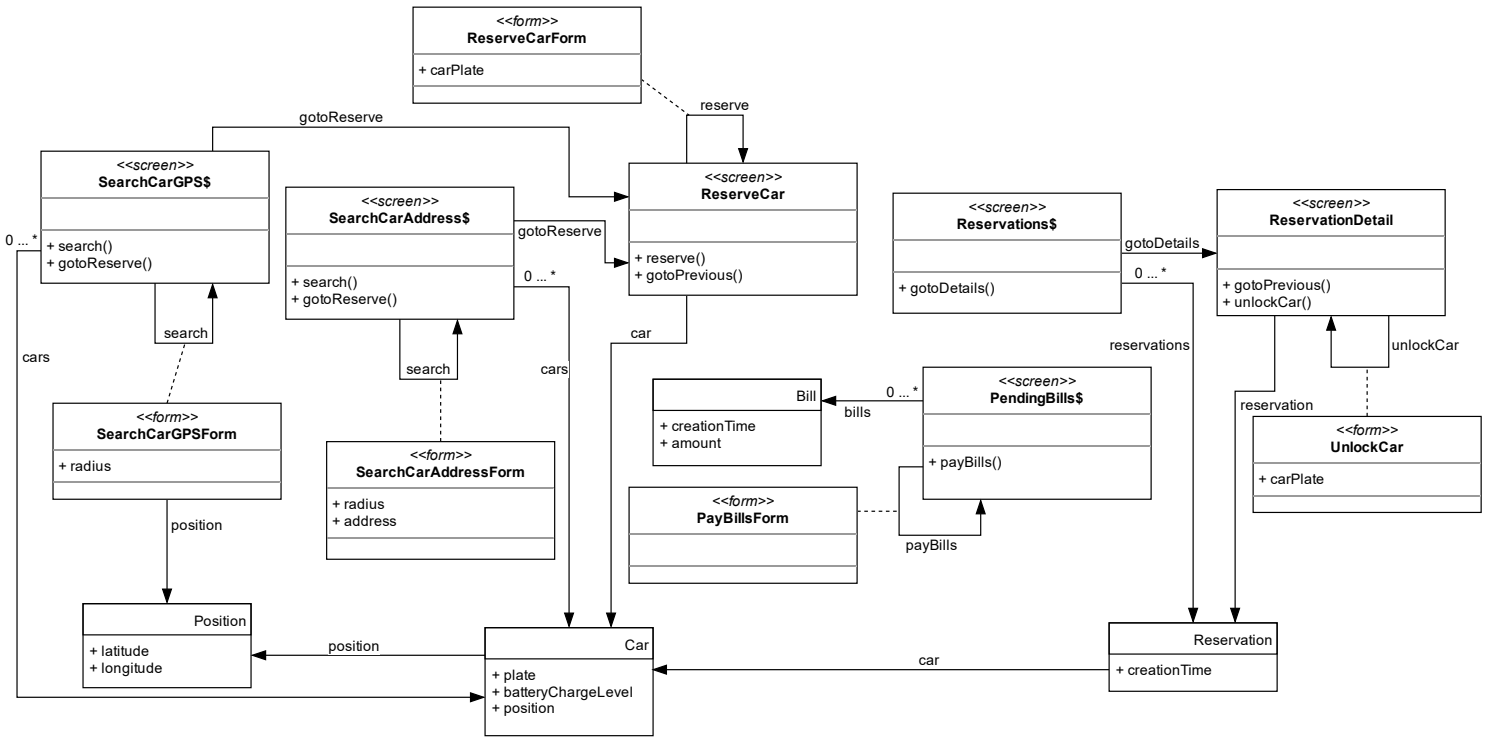
This section describes with a standard UX diagram the user interface to be implemented both in *User Application* and *Employee Application*. This section does not include any mockup of the final application, as they were already included in the RASD. Three diagrams are provided, the first describes login and registration interfaces, the second describes the user interface for the *User Application*, and the third describes the user interface for the *Employee Application*. According to RASD, both applications provide breadcrumbs navigation.

4.1 Login and registration

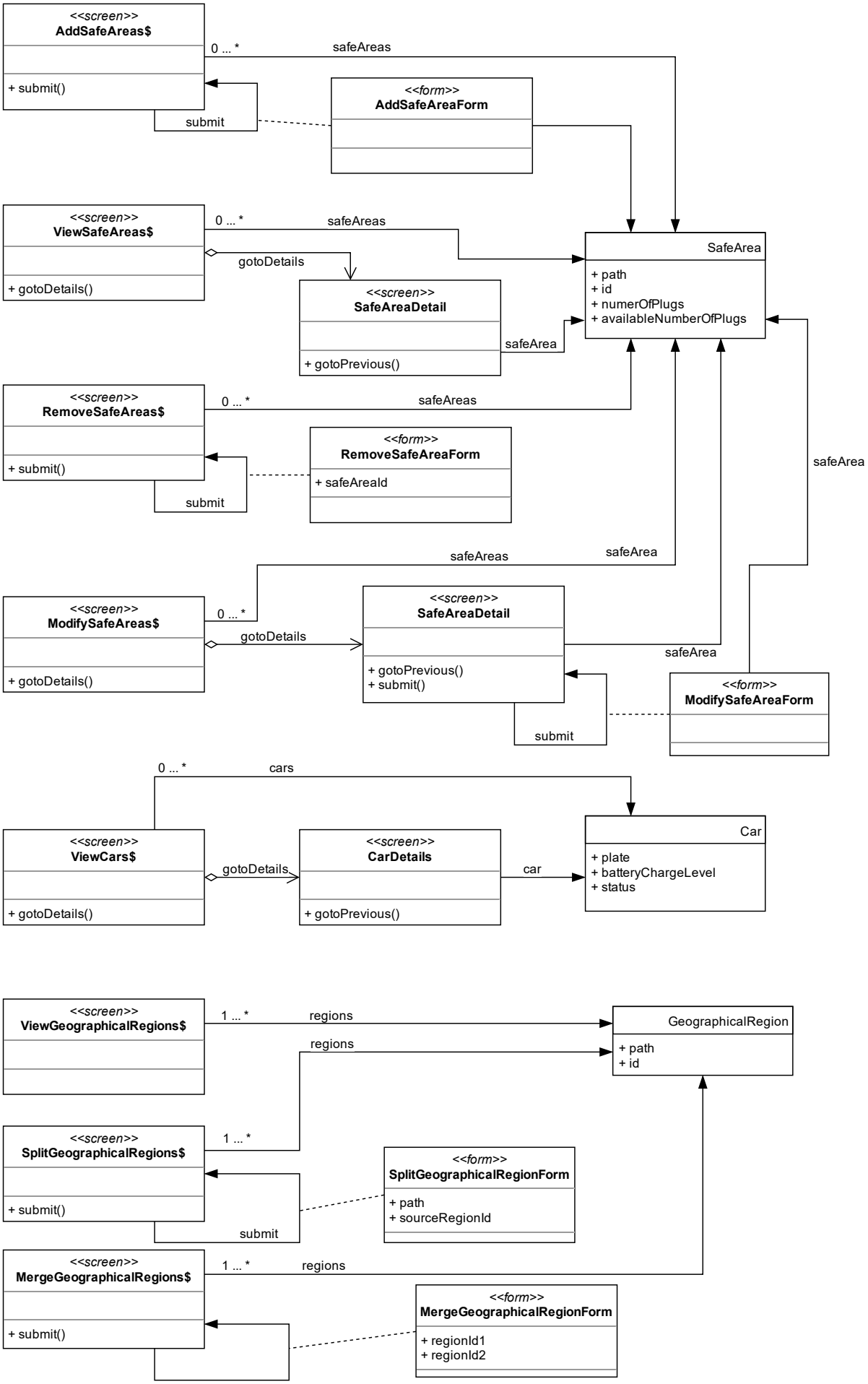


The "ok" action of the login form triggers the loading of the appropriate start page, that is modeled with the screen named "SearchCarGPS" for a user, and with the screen named "ViewCars" for an employee

4.2 User application



4.3 Employee application



5 Requirements traceability

All the decisions written in this document have been taken following functional and non-functional requirements presented in the RASD. Here is a mapping between requirements and design decisions.

5.1 Functional requirements

Functional requirements have been mapped on different components in the overall architecture. Here is a mapping table.

Requirement (RASD)	Component (DD)
[R1.1]: The system can acquire user information for the registration (name, surname, address, birth date, driving licence number, credit card number and CVV)	User Controller
[R1.2]: The system validates the driving licence number using the external service for driving licence validation	User Controller
[R1.3]: The system validates the payment information using the external service for payment information validation	User Controller
[R1.4]: The system is able to verify that no other registered user exists with the same username or driving licence number or payment information	User Controller
[R1.5]: The system registers this new user only if given information are valid	User Controller
[R2.1]: The system can acquire user information for login (username and password)	User Controller
[R2.2]: System is able to check whether a tuple (username, password) is correct, that is whether that tuple matches the information of a registered user or of an employee or not	User Controller
[R2.3]: The user or the employee logs in if and only if username and password costitute a correct tuple	User Controller
[R3.1]: The system only allows the logged in user who is not banned to insert the search radius	User Application
[R3.2]: The system is capable of finding all available cars within the inserted distance from the user’s position	Car Controller
[R3.3]: The system is able to show to a user a list of cars with their position and battery level	User Application
[R4.1]: The system only allows the logged in user who is not banned to insert the address on which the search area will be centered	User Application
[R4.2]: The system is capable of finding all available cars within a distance range from the geographical coordinate of the address	Car Controller
[R5.1]: The system only allows the logged in user who is not banned to reserve an available car	Reservation Controller
[R5.2]: The system is able to get the geographical region from the car geographical coordinates	Geographical Areas Controller
[R5.3]: The system only reserves a car if the logged in user that requests it has no other reservation for the same geographical area in which the car is located	Reservation Controller
[R6.1]: The system keeps track of the time elapsed since a reservation is made	Reservation Controller
[R6.2]: If the elapsed time is greater than one hour, then the reservation expires	Reservation Controller
[R7.1] The system unlocks a car only if distance between the car and the reservor user is less then 8 meters and the reservor user has requested the unlocking	Reservation Controller
[R8.1] The system can create an empty bill for the reservor user only when the engine is ignited	Ride Controller

[R9.1]: The system knows whether a car is in a safe area or not	Safe Areas Controller
[R9.2]: The system knows the time elapsed since the engine ignition	Ride Controller
[R9.3]: The system updates the reservor user bill according to the elapsed time	Ride Controller
[R10.1] The system knows the reservor user bill	Ride controller
[R12.1]: When a car is becoming available and this car is parked in a recharging station area there is a 5 minute window before the system charges the bill to the reservor user	Ride Controller
[R12.2]: When a car is becoming available and this car is parked in a safe parking area the system charges immediatly the reservor user with the bill	Ride Controller
[R11.1]: When a car is becoming available, if the system detects that this car has had at least two passengers (not including the driver) for all the time of the ride when the engine was ignited, then the system applies a discount of 10% on the bill of the reservor user	Ride Controller
[R11.2]: When a car is becoming available, if its battery level is greater or equal to the 50% of the total battery level, the system applies a discount of 20% on the bill of the reservor user	Ride Controller
[R11.3]: When a car is becoming available, if it is parked in a recharging station area and it is plugged to the power grid, the system applies a discount of 30% on the bill of the reservor user and the 5 minute window terminates	Ride Controller
[R11.4]: When a car is becoming available, if the battery level is less than 20% of the total battery level, the system applies a raise of 30% on the bill of the reservor user	Ride Controller
[R11.5]: When a car is becoming available, if it is left more than 3Km away from the nearest recharging station, the system applies a raise of 30% on the bill of the reservor user	Ride Controller
[R13.1]: The system knows whether a payment attempt was made for the current car	Bill Controller
[R13.2]: The system locks the car and makes it available only if the payment attempt was processed, either if the payment is successful or not	Ride Controller
[R14.1]: The system knows the result of the payment operation	Bill Controller
[R14.2]: The user is banned only if there exists a pending bill bound to him	User Controller
[R14.3]: The system marks a bill as paid if and only if a payment operation associated to this bill is successful	Bill controller
[R14.4]: A user can ask the system to try again to extinguish his pending bills	User Application
[R14.5]: Only pending bills can be required to be paid	Bill Controller
[R15.1]: The system only allows a logged in user who is not banned to view his reservations	Reservation Controller
[R15.2]: The system is capable of finding all the reservation made by the user	Reservation Controller
[R15.3]: The system is able to show to a user a list of cars with their position, battery level and the expiration time	User Application
[R16.1]: The system only allows a logged in employee to manage geographical regions	Geographical Areas Controller
[R16.2]: The system is able to find all the geographical regions already defined	Geographical Areas Controller
[R16.3]: The system is able to display the geographical regions to the employee	Employee Application
[R17.1]: The system allows the employee to select a geographical region	Employee Application
[R17.2]: The system allows the employee to draw a line inside the selected geographical region	Employee application

[R17.3]: The system is able to compute the new geographical region and store them	Geographical Areas Controller
[R18.1]: The system allows the employee to select two geographical regions	Employee Application
[R18.2]: The system merges the two region into one single region and store it, removing the two sources region	Geographical Areas Controller
[R19.1]: The system only allow a logged in employee to manage safe areas	Safe Areas Controller
[R19.2]: The system is able to find all the safe areas already defined	Safe Areas Controller
[R19.3]: The system is able to display the safe areas to the employee	Employee Application
[R20.1]: The system allows the employee to select a safe area	Employee application
[R20.2]: The system removes a selected safe area	Safe Areas Controller
[R22.1]: The system allows the employee to select a safe area	Employee application
[R21.1]: The system can acquire from the employee the type of safe area, its shape as a sequence of coordinate, and eventually the number of plugs of the recharging station area	Employee application
[R22.2]: The system is able to check if the defined safe area will overlap with the already defined safe areas except for the selected one	Safe Areas Controller
[R22.3]: The system updates the selected safe area only if it is not overlapping	Safe Areas Controller
[R23.1]: The system only allows a logged in employee to view the list of in maintenance cars	Car Controller
[R23.2]: The system is capable of finding all the in maintenance cars	Car Controller

5.2 Non functional requirements

Requirement (RASD)	Design decision (DD)
Reliability	Presence of a backup server
Availability	Presence of a backup server
Security	Strong cryptographic functions
Portability	Web Standards and J2EE framework

6 Appendix

6.1 Effort spent

- Nardo Loris: 18 hours of work
- Osio Alberto: 22 hours of work

6.2 References

- http://www.ibm.com/developerworks/rational/library/3101.html

  For "break" frame semantics in sequence diagrams

- http://dev.mysql.com/doc/refman/5.7/en/gis-class-multipolygon.html

  For *multipolygon* data type definition

6.3 Software and tools used

- Github (https://github.com) for version control
- StarUML (http://staruml.io/) for UML diagrams
- Draw.io (http://www.draw.io/) for UX and ER diagrams