PowerEnJoy
Design Document
Software Engineering 2 project

POLITECNICO
MILANO 1863

Authors:
Arcari Leonardo
Bertoglio Riccardo
Galimberti Andrea

Document version: 1.3.1

Contents

1 Introduction

1.1 Purpose

The purpose of this document is to provide an overall description of the architecture of the system to be. Design choices made and discussed in this paper should make possible the implementation of a system that fulfills the goals and the requirements set in the requirements analysis process and explained in the RASD. The content of the document is addressed to developers and technical team as a guidance and source of information about the key aspects of the system design. This includes but is not limited to:

- high level overview of the system
- various levels of abstraction to give a finer description of system components
- architectural choices rationale
- most relevant algorithms design
- user interface design

1.2 Scope

The required product is a digital management system for PowerEnJoy, a car sharing service employing exclusively electric vehicles.

The system has to provide functionalities to support both the users of the service and the employees of the company who interact with customers and cars.

Typical functions of a car-sharing service have to be supported, such as reserving a car and finding a parking area where to plug it in at the end of the rent.

A critical issue that has to be considered is the management of the electric vehicles, which have to be continuously recharged as they get used by customers and have to be distributed in a quite uniform way around the town; hence, the user has to be incentivized through discounts to recharge the car in power plugs-equipped parking areas, don't leave the car with a low battery level and park the car where there are few other available vehicles.

The product also has to provide a way to attract more potential customers to the service, therefore sharing the car with other passengers will be incentivized through appropriate discounts. To achieve such functionalities, the system shall provide applications to its users, both customers and employees.

1.3 Definitions, acronyms, abbreviations

RASD Requirements Analysis and Specification Document

DD Design Document

RPC Remote Procedure Call

EJB Enterprise Java Bean

MVC Model View Controller

ECU Engine Control Unit: is a type of electronic control unit that controls a series of actuators on an internal combustion engine to ensure optimal engine performance. It does this by reading values from a multitude of sensors within the engine bay, interpreting the data using multidimensional performance maps (called lookup tables), and adjusting the engine actuators accordingly.

OBD On-board diagnostics: is an automotive term referring to a vehicle's self-diagnostic and reporting capability. OBD systems give the vehicle owner or repair technician access to the status of the various vehicle subsystems.

1.4 Reference documents

- Requirements Analysis and Specification Document
- Project Assignment Document

1.5 Document structure

1. **Introduction**: a description of this document, its purpose, its intended audience and definitions of acronyms, technologies used and abbreviations
2. **Architectural design**: section containing material to communicate the architecture of the system to be. It is composed of the following:
   (a) *Overview*: a high-level description of the system, logical and physical layers.
   (b) *Component view*: a description of the components the system is made of. Particular emphasis is given to the relation between the façade sub-components every high-level component contains and to the interfaces they communicate through.
   (c) *Deployment view*: a description of the topology of system hardware and how components are distributed on it.
   (d) *Runtime view*: a description system execution flow and components interaction, in the sense of message/call exchanges.
   (e) *Component interfaces*: a description of the internal interfaces of the system, in the sense of the application protocols which two components communicate with.
   (f) *Selected architectural styles and patterns*: rationale about the choices taken to build the system architecture.
   (g) *Other design decisions.*
3. **Algorithm design**: section presenting a set of meaningful algorithms that characterize the system written in pseudo-code.
4. **User interface design**: mockups of the applications user interface.
5. **Requirements traceability**: section showing the mapping of RASD goals and requirements to the elements described in the DD.
6. **Effort spent**: number of hours each group member has worked towards the fulfillment of this deadline.
7. **References**: sources examined for knowledge building.

2 Architectural Design

2.1 Overview

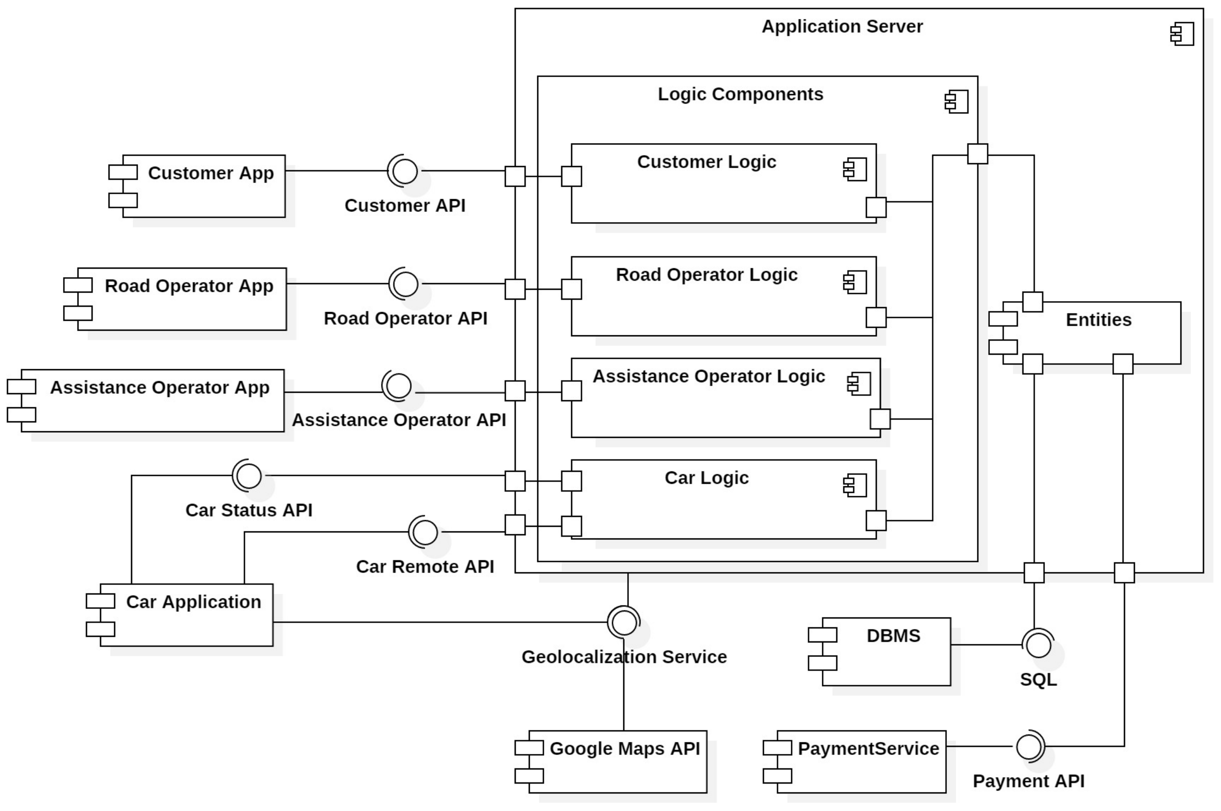


Figure 1: High-level system Component Diagram

The PowerEnJoy system architecture is designed following a 3-tier architecture: presentation, application processing and data management are physically separated. Before proceeding in the description, stating what components belong to which logical layer is due:

Presentation Customer, Road Operator, Assistance Operator and Car Applications

Business Logic Customer Logic, Road Operator Logic, Assistance Operator Logic and Car Logic

Data management DBMS

Independently from what each component actually does, such a heterogeneous system has to be properly orchestrated to let the components interaction. A client-server architecture suits. The actor in the play are the presentation components on the client side and the application server on, of course, the server side. Each presentation component interacts with the server only. It will be an application server task to forward a message to another presentation component in case it is needed.

Since most of the time each presentation component requires some logic by the server, strictly related to that component domain, also the application server is composed of sub-components, or modules, that perform the server side logic.

Each server logic component, then, communicates with other server logic components and with the system entities which are across-the-board to the whole system.

An entity is a class, or a façade class of a more complex set of classes, that is made persistent to the DBMS.

Because of the distributed nature of the system, communication between presentation and business logic layers happens over the Internet exploiting a set of interfaces exposed by the application server.

Technologies adopted In order to support the implementation of such architecture the following technologies have been chosen:

HTML + CSS + Javascript for implementing the Customer, the Road Operator and Assistance Operator web-applications. The frameworks available for JavaScript makes moving some logic to the client possible.

RESTful API architectural style for defining the set of interfaces exposed by logic components. This allows cross-language, cross-platform RPC.

WebSocket communication protocol to implement *push notification*. It provides full-duplex communication channels over a single TCP connection.

Java Enterprise Edition platform for building the application server components. More comments on this choice will follow.

JSON format for message and data exchange. Its well-known structure and interoperability among different languages and platforms make it a perfect choice in pair with REST interfaces.

JSON Schema for JSON documents specification. Provides a way of defining the structure of well-formed JSON messages.

Technologies adoption rationale The choice of adopting a *JEE* platform has been done for the sake of ease of development and components scalability. A *EJB* container match perfectly the concept of logic components inside the application server, making the logic distributable over multiple machines running an instance of the application server. The *Java Persistence API* allows instances of system entities to persist on a DBMS and be queried by logic components when needed, through the *Entity Manager*. REST API and WebSocket support is also provided by *JAX-RS* and *Java API for WebSocket* respectively.

So, basically all the required server-side technologies are supported.

As for the RESTful API style adoption, it seemed the most reasonable choice as:

- allows RPC by binding a callback function on the server to be fired upon a request on a REST endpoint
- it is cross-language as it is implemented as a standard HTTP connection
- exposes a unique set of URLs that can be load-balanced on multiple application servers

Although, REST is not suitable for everything. Due to its nature, it is necessary that a client performs a request to the server exposing a REST endpoint and only after it the server can send data to the client. This does not make *push notification* possible, where it is the server that must begin the communication. *Push notification* is particularly necessary to interact with the Car Application. Remote control of the Car, for example to unlock it remotely when the Customer asks for it through its application.

Thus, *WebSocket protocol* has been chosen. It does not only provide a full-duplex channel but it is also implemented over HTTP and it is a W3C standard since 2011. Long-term and quality support should be expected.

2.2 Component view

2.2.1 Car Application

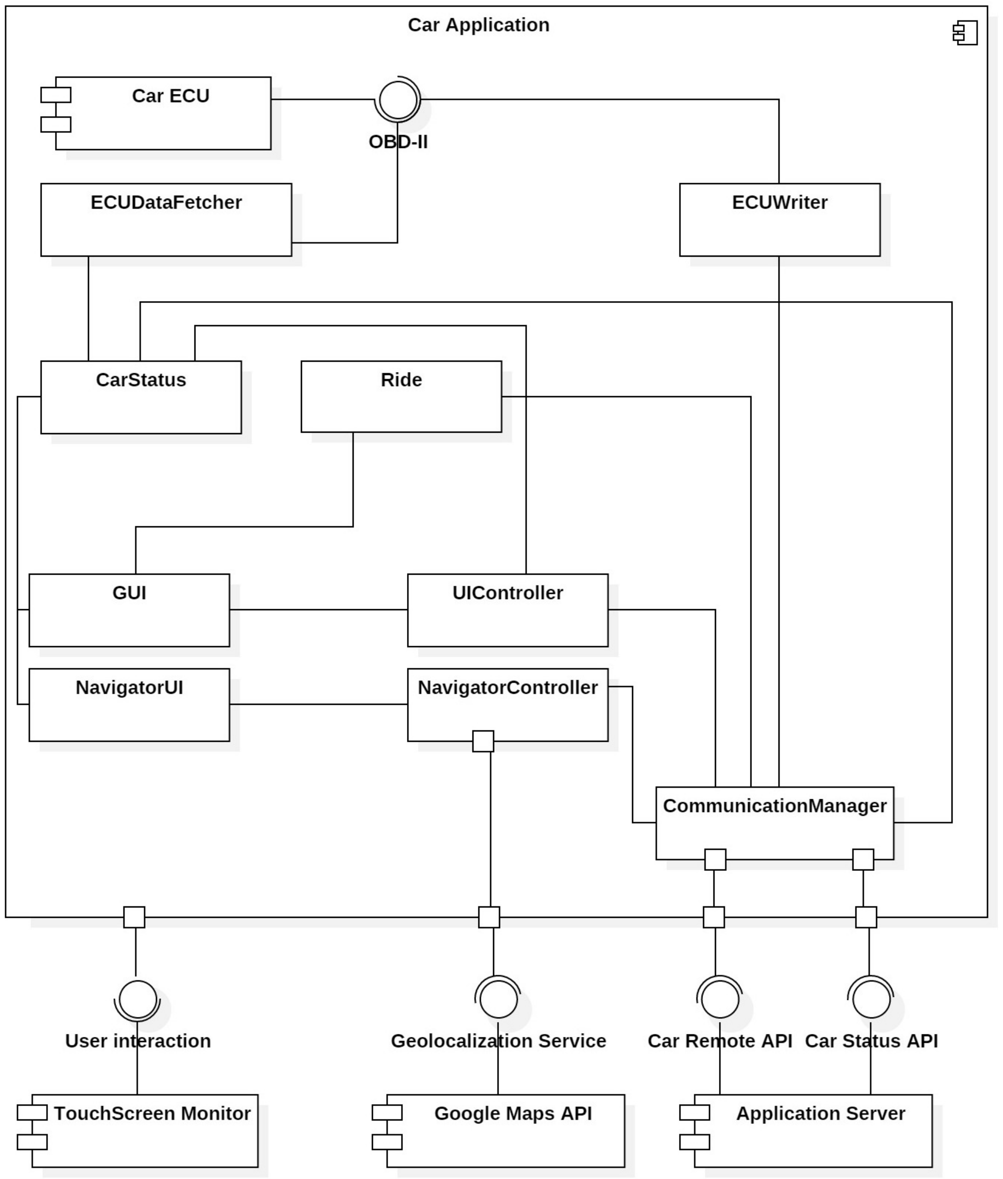


Figure 2: Car Application Composite Structure Diagram

Car Application component is responsible for managing the Car's information system. It is designed around a MVC architecture:

Model *CarStatus* and *Rental* compose the model part of the architecture. *CarStatus* collects data representing the Car state, such as GPS location, Car service status, car speed, number of people inside, internal healthy values and so on. *Rental* carries data related to the current ride, from the unlock performed by the Customer to the rental ending; this includes, but not limits to, driving time and discounts and overcharges applied. Data in the model is updated by user interaction, application server messages and internal *Car ECU*. *Car ECU* is read and written through a standard *OBD-II* interface.

View *GUI* and *NavigatorUI* compose the view part of the architecture. *GUI* defines the whole user interface except for the Navigator user interface which is provided *Google Maps API* that Car Application interfaces with. The Customer interacts with the UI through a touch screen monitor whose input is handled by the underlying operating system that Car Application is running on.

Controller *UIController, Navigator Controller, CommunicationManager, ECUDataFetcher and ECUWriter* compose the controller part of the architecture. While the first two are pretty straightforward controllers, some more details are due about the other sub-components. *CommunicationManager* is responsible for handling the communication between Car Application and Application Server to communicate car status updates and executing remote control operations sent by the server. *ECUDataFetcher* and *ECUWriter* allow interaction with the internal car ECU to physically apply remote commands and retrieving internal status values of the car.

2.2.2 Car Logic

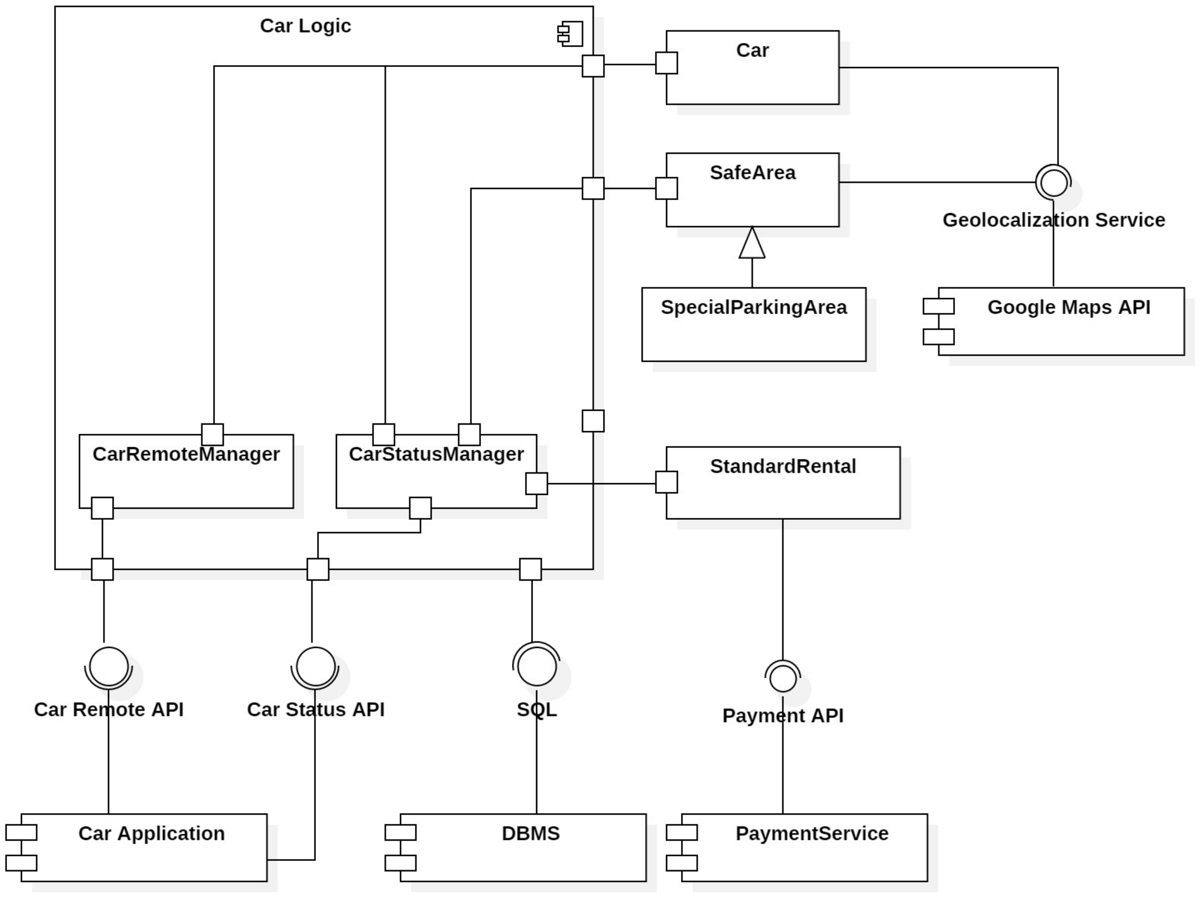


Figure 3: Car Logic Composite Structure Diagram

Car Logic component is the Application Server sub-component responsible for the server-side logic related to the Car-Server interaction.

CarRemoteManager handles the Car / Application Server communication according to *Car remote control* protocol. The server sends a *car remote control* message and receives an *acknowledgment* from the Car. This could be the case of a remote command to unlock the Car after the Customer has correctly inserted their *PIN*.

CarStatusManager handles the Car / Application Server communication according to *Car status update* protocol. This component reacts to remote procedure calls performed by the Car through the RESTful API exposed by the component. This could be the case of the Car publishing its new GPS location.

CarStatusManager updates *Car* fields, produces the (possibly) new set of the nearby *Safe Areas* and sends it back to the Car according to the *Safe area set* message protocol.

Other components are not specific property of the *Car Logic* component. They have been added to the diagram to express that they are involved in the component's business, hence they are in relation with it. As *Entities* in *JEE* platform they are made persistent in a *DBMS*, thus an interaction with it is made explicit.

2.2.3 Road Operator Logic

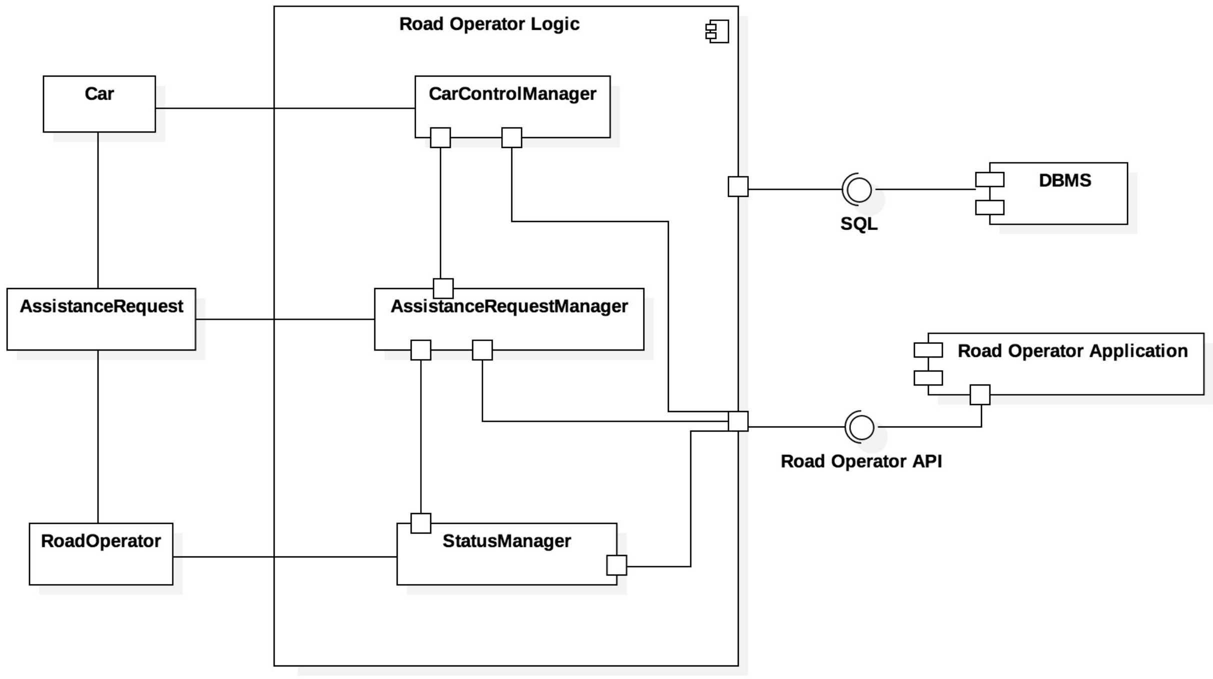


Figure 4: Road Operator Logic Composite Structure Diagram

The Road Operator Logic component is the Application Server subcomponent responsible for the server-side logic related to the Road Operator.

CarControlManager handles the communication between the Road Operator and the Car, enabling the Road Operator to remotely control a Car in order to unlock or lock it as needed (e.g. when he has to intervene on a faulty car); it interfaces with the Road Operator Application in order to respond to the needs of the on-field Road Operator, and it can inquire about or change the status of the Car by making requests to its corresponding Car entity.

AssistanceRequestManager enables the Road operator to be notified of new pending Assistance Requests as they are added, that he can then accept or not by interfacing to the AssistanceRequestManager via the Road Operator Application.

StatusManager handles the management of the availability of the Road Operator to serve a new Assistance Request; the Road Operator can update his availability status via the Road Operator Application, which interfaces to the StatusManager through the Road Operator API.

Other components Car, AssistanceRequest and RoadOperator are not specific property of the *Road Operator Logic* component. They have been added to the diagram to express that they are involved in the component business, hence in relation with it; as *Entities* in *JEE* platform, they are made persistent in a *DBMS*, thus an interaction between *the Road Operator Logic* component and the DBMS via SQL interface is made explicit.

2.2.4 Assistance Service Operator Logic

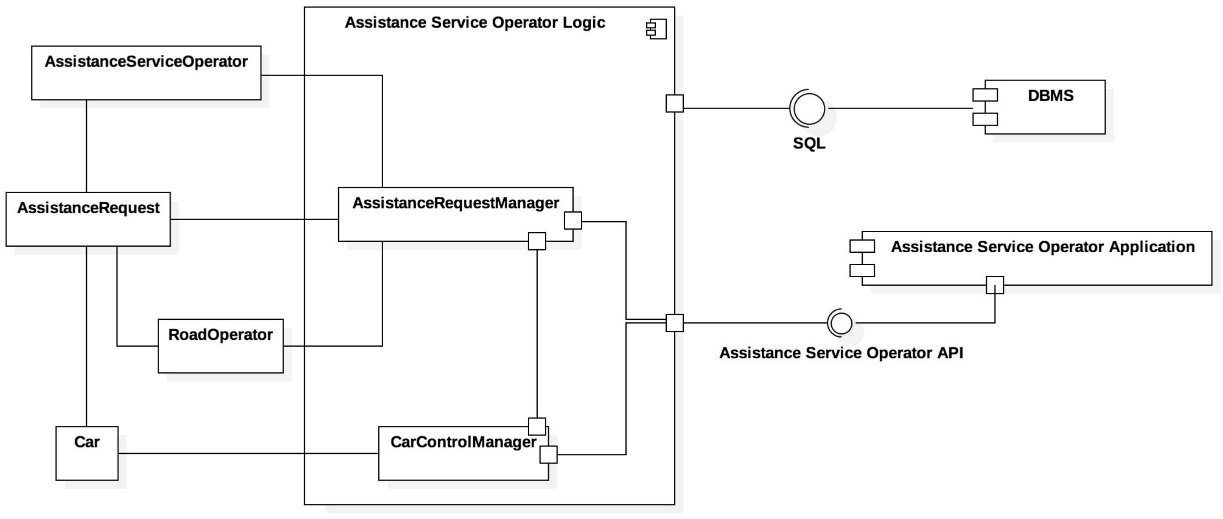

Assistance Service Operator Logic
AssistanceServiceOperator
DBMS
SQL
AssistanceRequest
AssistanceRequestManager
Assistance Service Operator Application
RoadOperator
Assistance Service Operator API
Car
CarControlManager

Figure 5: Assistance Service Operator Logic Composite Structure Diagram

The Assistance Service Operator Logic component is the Application Server subcomponent responsible for the server-side logic related to the Assistance Service Operator.

CarControlManager handles the communication between the Assistance Service Operator and the Car, enabling the Assistance Service Operator to remotely control a Car in order to unlock or lock it; it interfaces with the Assistance Service Operator Application to allow the Assistance Service Operator to inquire about or change the status of the Car by making requests to its corresponding Car entity.

AssistanceRequestManager enables the Assistance Service Operator to open a new Assistance Request, assigning it to a currently available Road Operator, who will get notified; to make this possible, AssistanceRequestManager calls methods from the RoadOperator and AssistanceRequest entities.

Other components Car, AssistanceRequest, AssistanceServiceOperator and RoadOperator are not specific property of the *Assistance Service Operator Logic* component. They have been added to the diagram to express that they are involved in the component business, hence in relation with it; as *Entities* in *JEE* platform, they are made persistent in a *DBMS*, thus an interaction between the *Assistance Service Operator Logic* component and the DBMS via SQL interface is made explicit.

2.2.5 Customer Logic

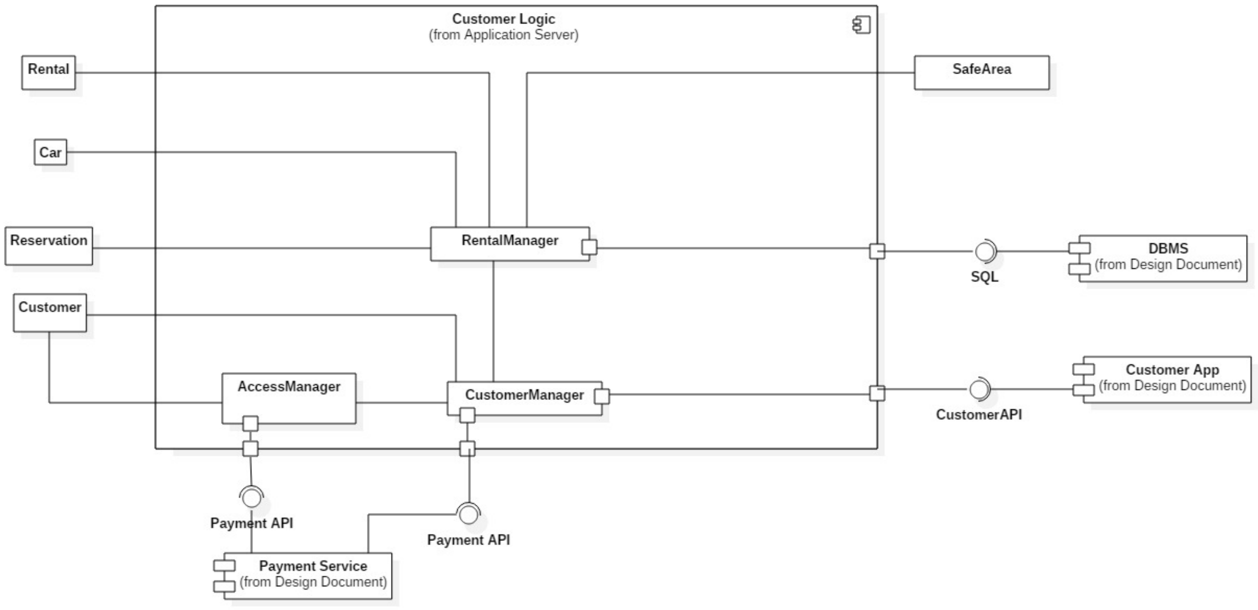


Figure 6: Customer Logic composite structure diagram

Customer Logic component is the Application Server sub-component responsible for the server-side logic related to the Customer-Server interaction.

Customer Manager handles all the requests from the customer app dispatching them to other sub-components. It's the core sub-component of the *Customer Logic* because it has a full view on all the functionalities available to the customer. It has direct access to the customer's personal information and interact with the *Payment service* to charge the customer for the rent. It also communicates with the *Rental Manager* and the *Access Manager*.

Rental Manger manages the entire rental process. It returns to the customer safe areas positions near to them and the list of parked cars in a given safe area. In doing so it interacts with *Safe Areas* and make use of the *Google Maps API*. It handles reservation requests and allows the customer to unlock the car when they press the dedicated button. It also provides rental information to the customer, such as the duration of the rent, the remaining battery of the car and the traveled kilometers.

Access Manger manages the login and logout procedures and allows the customer to register to the service.

It interacts with the *Payment Service* to check the validity of the inserted payment data and has direct to the Customer's information to verify the login data.

2.2.6 Logic Components Composite Structure

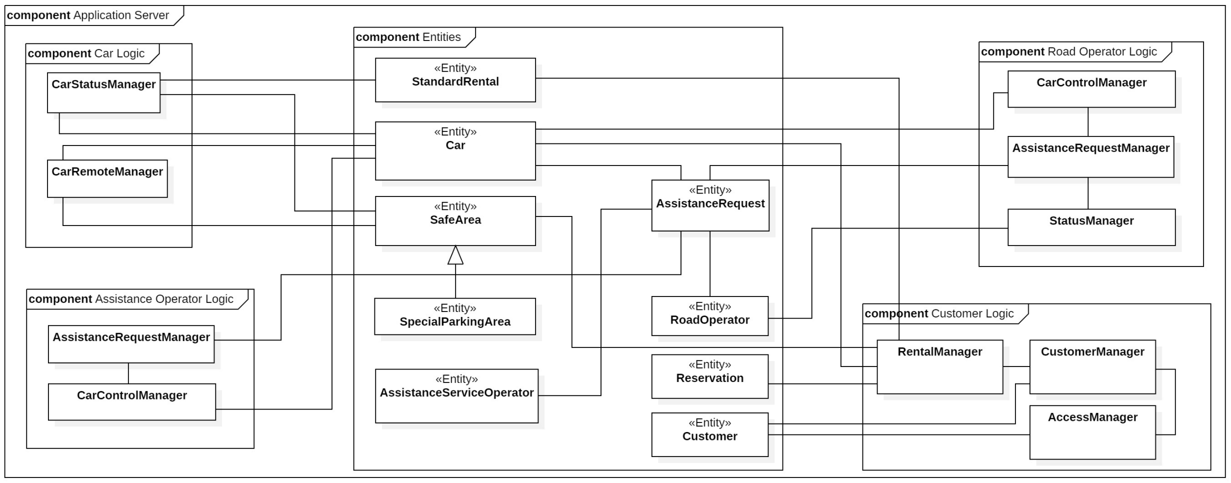


Figure 7: Customer Logic composite structure diagram

The above diagram highlights the interaction between different *Logic Components* on the *Application Server*. The main property of such architecture is how different business logic components are loosely coupled. Each Logic Component only deal with *Entities* of the system, updating their status. The Publish-Subscribe (Observer) pattern underlying the architecture will notify the other logic components of changes related some Entity and *those components **only*** will execute an operation in response of that event or not. This way Logic Components can be implemented and tested separately, as they only need to be integrated with the system Entities. Another benefit is that each logic component can run on its own thread or machine, while the Java Persistence API will take care of making data persistent to the DBMS. The Entity Manager of other instances of the Application Server will be notified by the DBMS of an update of the data, causing Java to update the Entity instance and publishing the new value to the subscribers running on that instance.

2.3 Deployment view

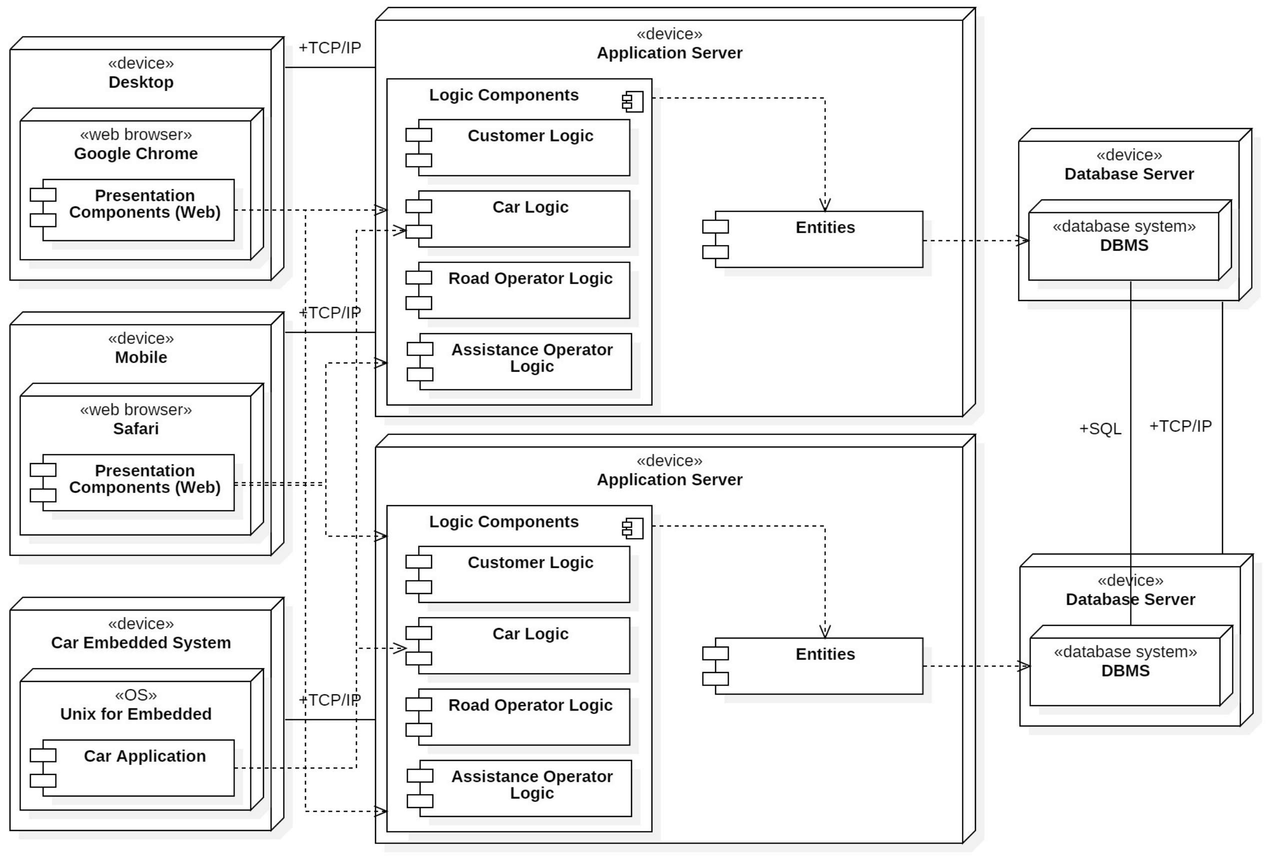


Figure 8: A realistic components Deployment Diagram

According to the 3-tier architectural style, the system deployment takes into account three set of machines on which components are distributed.

The **presentation tier** could theoretically be made of an infinite number of machines; they divide themselves into three categories: desktop machines, mobile devices and the embedded system shipped inside the Car.

The first two are dedicated to users of the system, Customer, Road Operator and Assistance operator, who interface with the web applications to exploit the features offered by the system.

The latter comes as a finite set of devices, equal to the number of Cars, owned by the company, and host the Car Application.

The **application logic tier** contains a set of machines aimed to handle the load generated by the *presentation tier* devices. A *JEE-compliant* platform, which necessary logic components are deployed on, shall run on them. The number of application server machines should be sized to respect the constraints of reliability, availability and response time. A load-balancer between the presentation and business logic tiers could help in this sense.

The **data management tier** carries similar considerations, with a difference: in the case of application servers, the machines work independently serving their clients; on the other hand the database servers need to be connected together to keep the data consistent in case two components of different application servers are working on the same object persistent on two different database servers.

2.4 Runtime view

2.4.1 Car Application - Application Server interaction

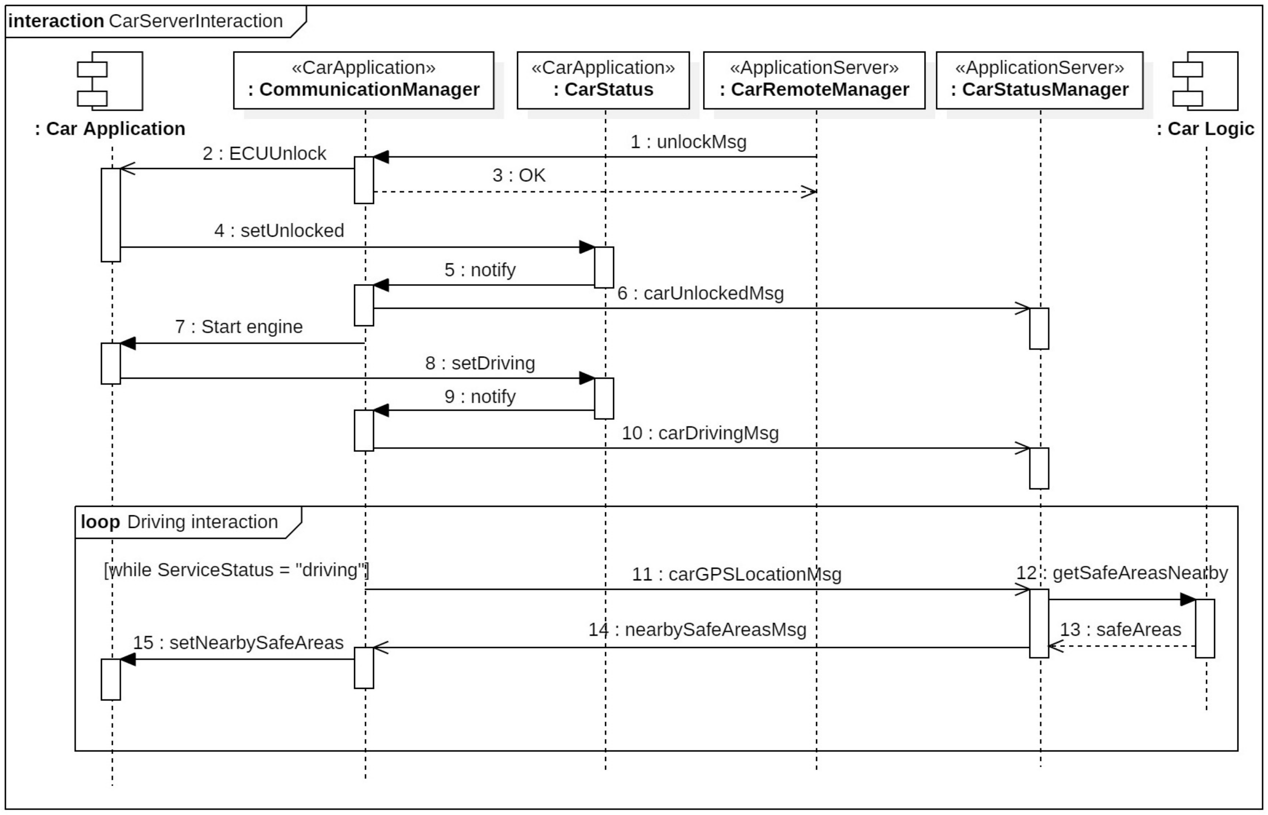


Figure 9: A Sequence Diagram showing interaction of *Car Application* and *Application Server* after the *Customer* has unlocked the Car through the *App*

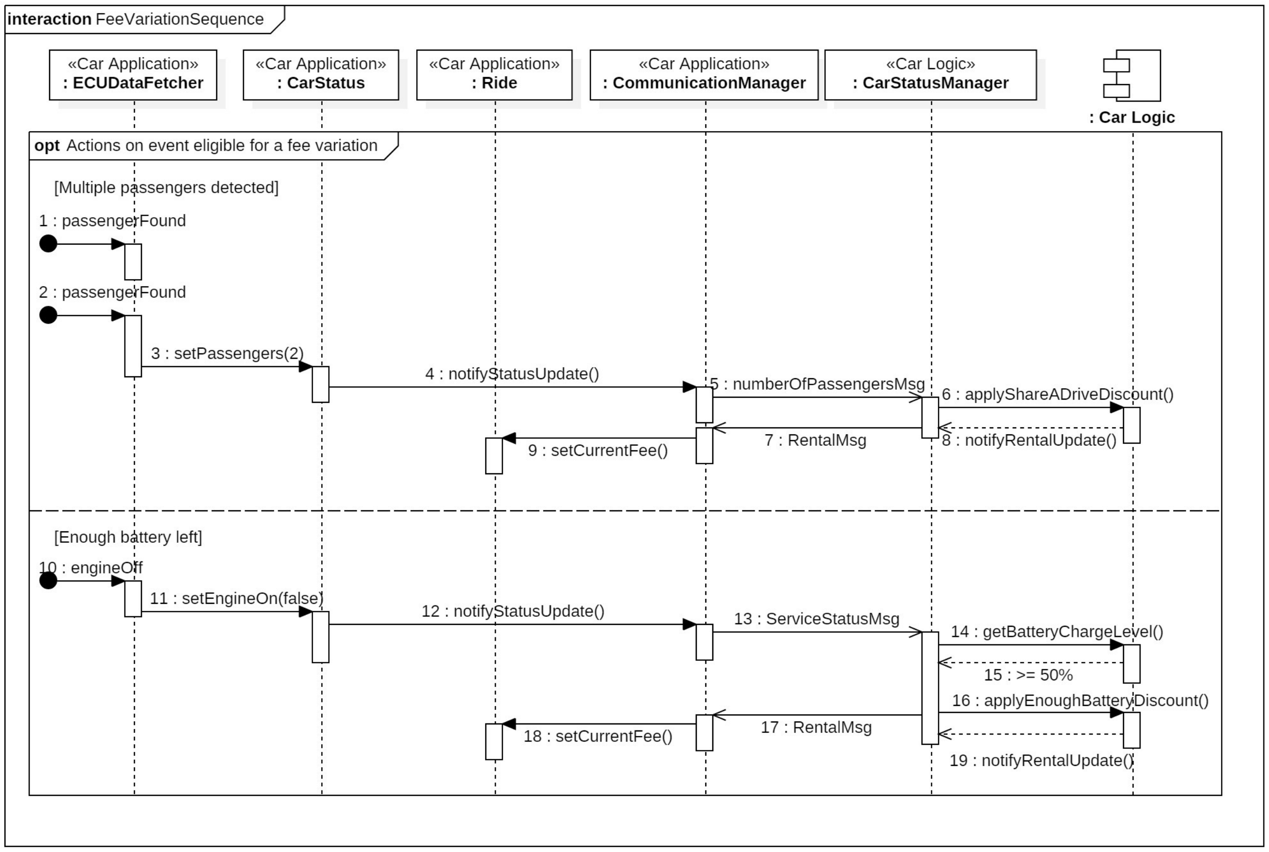

interaction FeeVariationSequence
«Car Application» : ECUDataFetcher
«Car Application» : CarStatus
«Car Application» : Ride
«Car Application» : CommunicationManager
«Car Logic» : CarStatusManager
: Car Logic
opt Actions on event eligible for a fee variation
[Multiple passengers detected]
1 : passengerFound
2 : passengerFound
3 : setPassengers(2)
4 : notifyStatusUpdate()
5 : numberOfPassengersMsg
6 : applyShareADriveDiscount()
7 : RentalMsg
8 : notifyRentalUpdate()
9 : setCurrentFee()
[Enough battery left]
10 : engineOff
11 : setEngineOn(false)
12 : notifyStatusUpdate()
13 : ServiceStatusMsg
14 : getBatteryChargeLevel()
15 : >= 50%
16 : applyEnoughBatteryDiscount()
17 : RentalMsg
18 : setCurrentFee()
19 : notifyRentalUpdate()

Figure 10: A Sequence Diagram showing two use cases for applying a fee variation.

2.4.2 Road Operator interaction

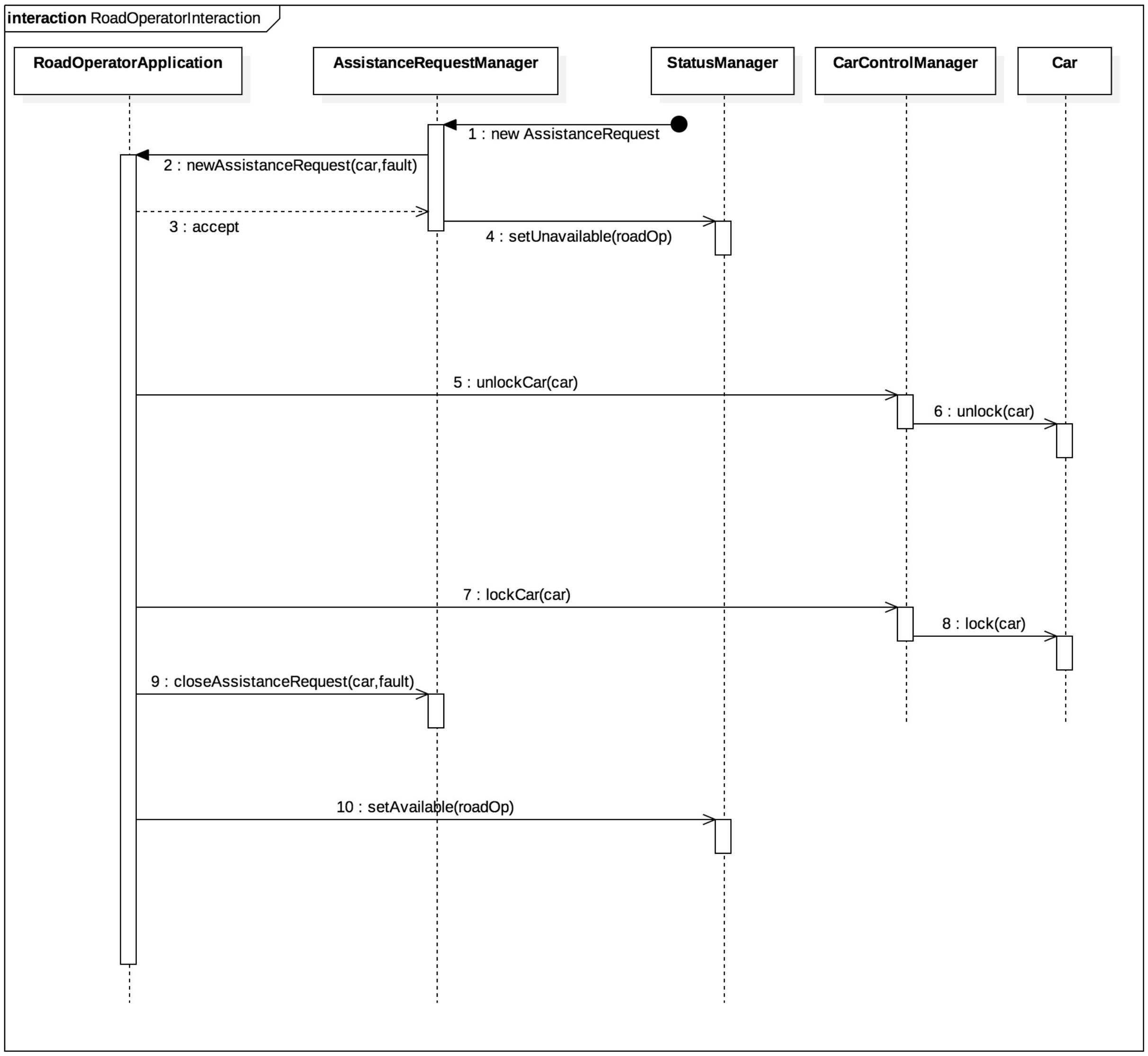

interaction RoadOperatorInteraction
RoadOperatorApplication
AssistanceRequestManager
StatusManager
CarControlManager
Car
1 : new AssistanceRequest
2 : newAssistanceRequest(car,fault)
3 : accept
4 : setUnavailable(roadOp)
5 : unlockCar(car)
6 : unlock(car)
7 : lockCar(car)
8 : lock(car)
9 : closeAssistanceRequest(car,fault)
10 : setAvailable(roadOp)

Figure 11: A Sequence diagram showing the interaction of a Road Operator with the other components

2.4.3 Customer interaction

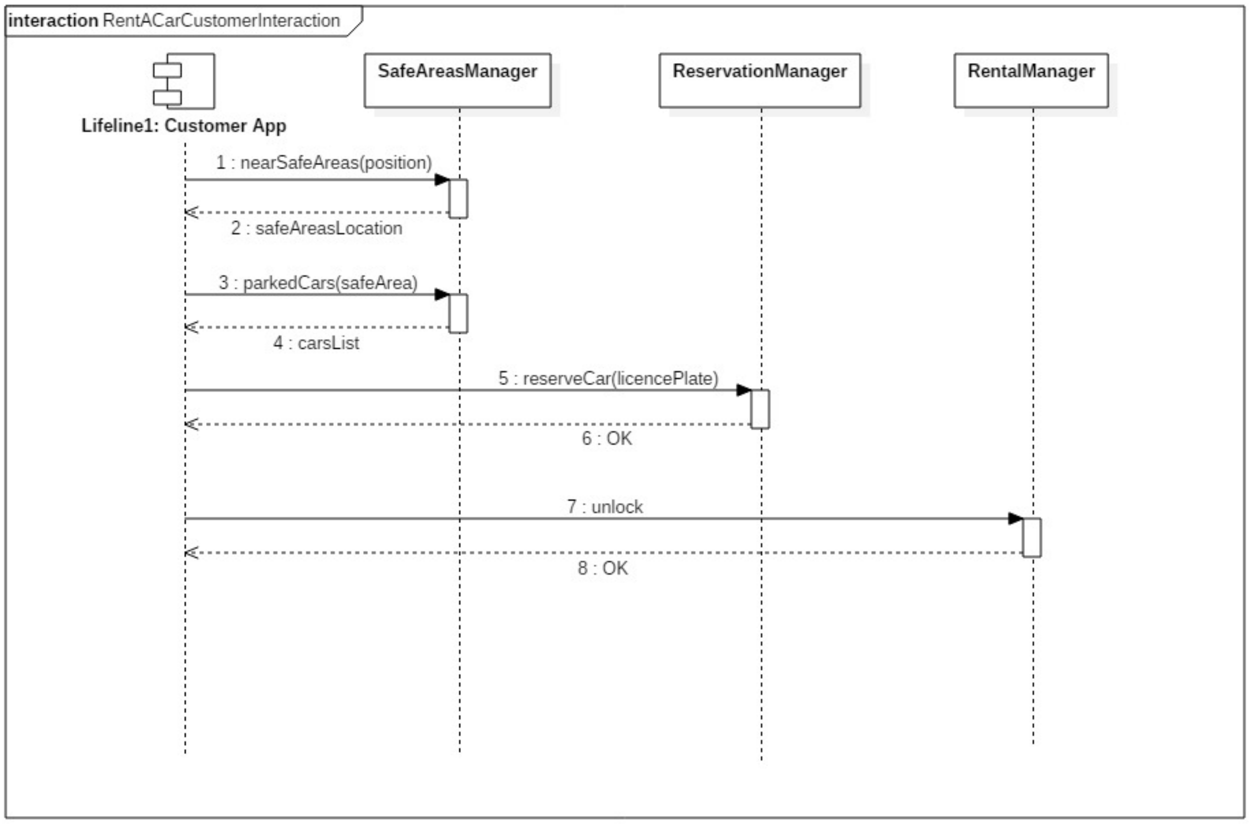


Figure 12: A Sequence diagram showing the interaction of a Customer with other components in the case of a rental procedure

2.5 Component interfaces

2.5.1 Car APIs

Car Remote API and *Car Status API* are the interfaces exposed by the Application Server to the Car Application. The API is implemented according to the Restful paradigm allowing Car Application to access and update data needed to support its features. In order to make the Application Server capable of sending message in a *push* fashion, a *WebSocket* communication is established between the two parts. Below the communication protocols that compose the *Car Remote API* and *Car Status API* are described.

Car remote control

JSON Schema

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "Car Remote Control",
    "description": "Remote Control operations",
    "type": "object",
    "properties": {
        "id": {
            "description": "ID of the car",
            "type": "string"
        },
        "operation": {
            "type": "string",
            "enum": [
                "unlock",
                "lock",
                "ok"
            ]
        }
    },
    "required": ["id", "operation"],
    "additionalProperties": false
}
```

Description A *Car remote control* message is sent in general from the Application Server to the Car Application to control its internal ECU system. This is particularly useful to unlock and lock the Car remotely, either because the Customer is unlocking the Car through the Customer App or because of communication issues the Customer could not perform this action on his own and an Assistance Request is required. This protocol can be expanded in future allowing a deeper degree of remote control of the Car.

The protocol is made of two fields:

- *id*: Car identification code
- *operation*: operation to remotely execute on the Car. It can be an *OK* message as well, to acknowledge the sender that message has been received.

Messages following this protocol are sent through the WebSocket.

Car status update

JSON Schema

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "Car Status Update",
    "description": "A Car status update",
    "type": "object",
    "properties": {
        "id": {
            "description": "ID of the car communicating an update",
            "type": "string"
        },
        "timestamp": {
            "description": "Date and time of creation of this update",
            "type": "object",
            "properties": {
                "date": {
                    "type": "object"
                },
                "time": {
                    "type": "object"
                }
            }
        },
        "status": {
            "description": "Updated value for a particular status",
            "type": "object",
            "oneOf": [
                {
                    "$ref": "#/definitions/GPSLocationStatus"
                },
                {
                    "$ref": "#/definitions/LockStatus"
                },
                {
                    "$ref": "#/definitions/ServiceStatus"
                },
                {
                    "$ref": "#/definitions/NumberOfPassengers"
                }
            ]
        }
```

```
},
"definitions": {
    "GPSLocationStatus": {
        "description": "Coordinates of the Car",
        "properties": {
            "type": {
                "enum": [
                    "GPSLocationStatus"
                ]
            },
            "location": {
                "$ref": "http://json-schema.org/geo"
            }
        },
        "required": [
            "type",
            "latitude",
            "longitude"
        ],
        "additionalProperties": false
    },
    "LockStatus": {
        "description": "Car lock status",
        "properties": {
            "type": {
                "enum": [
                    "LockStatus"
                ]
            },
            "state": {
                "type": "boolean"
            }
        },
        "required": [
            "type",
            "state"
        ],
        "additionalProperties": false
    },
    "ServiceStatus": {
        "description": "Service state of the Car",
        "properties": {
            "type": {
                "enum": [
                    "ServiceStatus"
                ]
            },
            "state": {
                "enum": [
                    "available",
                    "reserved",
                    "driving",
                    "stopped",
                    "toRepair"
                ]
            }
        },
        "required": [
            "type",
            "state"
        ],
        "additionalProperties": false
    },
    "NumberOfPassengers": {
        "description": "Number of passengers identified on the Car",
        "properties": {
            "type": {
                "enum": [
                    "NumberOfPassengers"
                ]
            },
            "passengers": {
                "type": "integer",
                "minimum": 0
            }
        },
        "required": [
            "type",
            "passengers"
        ]
    }
},
"required": [
    "id",
    "timestamp",
    "status"
]
}
```

Description A *Car status update* message is, in general, the payload of a REST method call on some endpoint exposed by the *Car Logic* EJB. A common use case is the Car notifying the Application Server of

some status change, such as GPS location, lock status and so on.
The protocol is made of the following fields:

- *id*: Car identification code
- *timestamp*: date and time of message creation
- *status*: type of the status changed and its updated value

Safe area set

JSON Schema

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "Safe Area Set",
    "description": "Remote Control operations",
    "type": "array",
    "items": {
       "$ref": "#/definitions/SafeArea"
    },
    "definitions": {
        "SafeArea": {
            "properties": {
                "location": {
                    "$ref": "http://json-schema.org/geo"
                },
                "available_slots": {
                    "type": "integer",
                    "minimum": 0
                },
                "available_plugs": {
                    "type": "integer",
                    "minimum": 0
                }
            },
            "required": [
                "location",
                "available_slots"
            ],
            "additionalProperties": false
        }
    }
}
```

Description A *Safe area set* message is, in general, returned by the Application Server to the Car Application after receiving a *GPSLocationStatus* update. The message contains the set of *SafeAreas* nearby the current Car location. This information is then temporarily stored and displayed on the Car touchscreen monitor.

RESTful API end-point URIs

HTTP Method	URI	Protocol
POST	/CarLogic/GPSLocation/{ID}	Car Status Update
POST	/CarLogic/LockingStatus/{ID}	Car Status Update
POST	/CarLogic/ServiceStatus/{ID}	Car Status Update
GET	/CarLogic/NearbySafeArea/{ID}	Safe Area Set

2.5.2 Customer API

Customer API is the interface exposed by the Application Server to the Customer Application. The API is implemented according to the Restful paradigm allowing Customer Application to access and update data needed to support its features.

Cars in a safe area

JSON schema

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "CarsInASafeArea",
    "description": "A set of cars that are parked in a given safe area",
    "type": "object",
    "properties": {
        "carsSet": {
            "type": "array",
            "items": {
                "type": "object",
                "properties": {
                    "licencePlate": {
                        "type": "string",
                        "pattern": "^([0-9]|[A-Z]){7}$"
                    },
                    "batteryLevel": {
                        "type": "number",
                        "minimum": 0,
                        "maximum": 100
                    }
                },
                "required": ["licencePlate", "batteryLevel"]
            },
            "uniqueItems": true
        }
    },
    "required": ["carsSet"]
}
```

Description A *Cars in a safe area* message is returned by the Application Server to the Customer Application after receiving a position of a *Safe Area*. The message contains the set of cars in the given *Safe Area*.

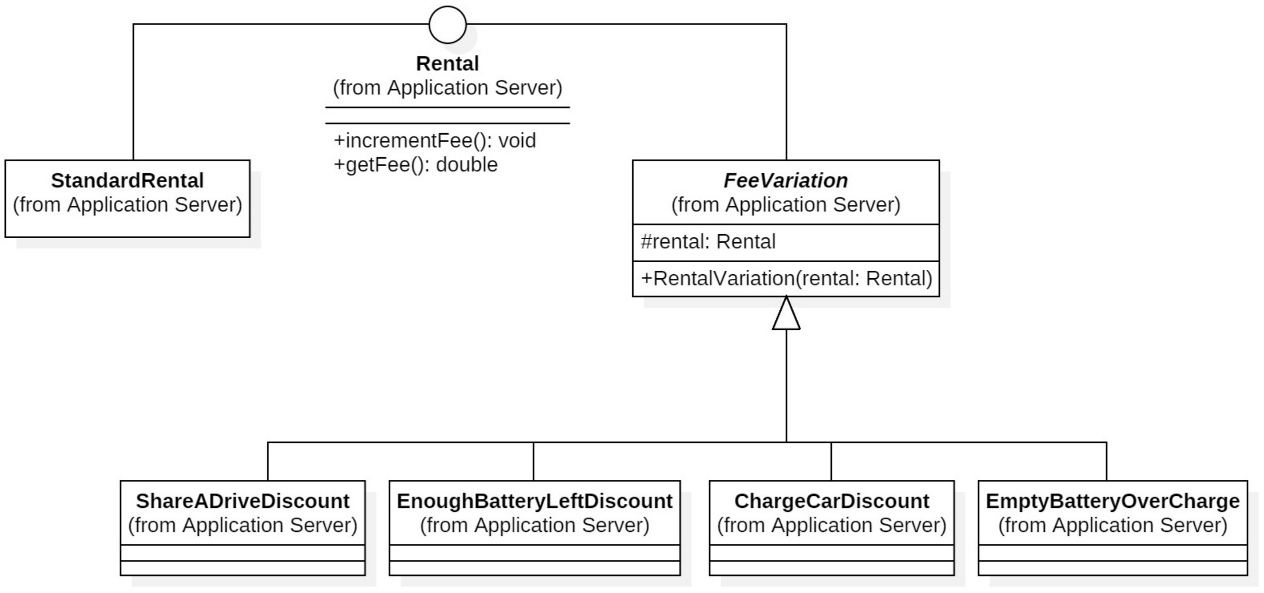
2.6 Selected architectural styles and patterns

Client - Server architecture has been adopted to split the presentation layer from the business logic and data layers. This holds in principle, although some logic exists on the client side to accomplish some tasks specifically related to the presentation layer only. This main reason for choosing this architectural style is to ensure data consistency over all the system components. Thus, a client application making modification to the local copy of some data immediately notifies the server, so that the main data is kept in sync. The same happens the other way: when the server modifies some data, all the clients subscribed to that resource receive a copy of the published update.

3 - tier architecture is a natural consequence of the client-server paradigm adoption. The presentation layer is going to run on users web-browser and on Car's embedded system whereas the application logic runs on its own set of machines. Having the data management layer separated allows to replicate data on different machines, maybe spread over different geographical locations to improve service response time.

MVC organizes the system components interaction at different level of granularity. At the highest level, described in *Overview* section, the view part is represented by the presentation components, the model part is represented by the *entities* persistent on the DBMS, while the controller part is split between the presentation and the logic components: the logic necessary to handle user input event on the UI is contained within presentation components whereas the business logic to manipulate the model is part of the application server containers. At a finer granularity, MVC architecture builds the presentation layer applications, e.g., as described in *Car Application* component.

Decorator pattern is selected to handle multiple discount / overcharge application on a Car rental standard fee. The choice of such architecture is due to the flexibility it provides. In case of new options introduction, it's sufficient to introduce a new FeeVariation class without breaking existing production code.



Rental
(from Application Server)
+incrementFee(): void
+getFee(): double
StandardRental
(from Application Server)
FeeVariation
(from Application Server)
#rental: Rental
+RentalVariation(rental: Rental)
ShareADriveDiscount
(from Application Server)
EnoughBatteryLeftDiscount
(from Application Server)
ChargeCarDiscount
(from Application Server)
EmptyBatteryOverCharge
(from Application Server)

Figure 13: Fee variation options Class Diagram

3 Algorithm design

3.1 View nearby cars

The purpose of this algorithm to let the customer view only the closest cars (from their current location or from a specified address), also having a sufficient battery level (i.e. not less than a predefined threshold); furthermore, if the user has selected a destination, only show cars with enough remining battery to get there, proportionally to the distance between the starting point and the destination.

```
#define MIN_BATTERY_LEVEL    10     // percentage
#define MAX_DISTANCE         500    // meters
#define DISCHARGE_RATE       3      // Wh per meter

GeographicalPosition requested_position;

/** The user can look for a car close to his current position
or specify another address */

read(another_position);
if(another_position != NULL)
    requested_position = GeolocalizationAPI.getLocation(another_position);
else
    requested_position = user.getLocation();

/** The user can specify his destination to get better results
in terms of car that have enough battery for the whole trip */

read(user_destination);

List<SafeArea> all_safe_areas = getSafeAreas();
List<SafeArea> close_safe_areas;

forall(SafeArea sa : all_safe_areas){
    GeographicalPosition safe_area_position = sa.getLocation();
    if(safe_area_position.isDistant(requested_position) < MAX_DISTANCE)
        close_safe_areas.add(sa);
}

List<Car> available_cars;
forall(SafeArea s : close_safe_areas)
    available_cars.add(s.parkedCars());

List<Car> available_cars_enough_battery;
forall (Car c : available_cars)
    if(c.getBatteryLevel() >= MIN_BATTERY_LEVEL)
        available_cars_enough_battery.add(c);
```

```
/** Optional part, only if the user has previously inserted a destination */
if(user_destination != NULL){
    GeographicalPosition destination_position;
    destination_position = GeolocalizationAPI.getLocation(user_destination);

    int estimated_battery;
    estimated_battery_needed =
        destination_position.isDistant(requested_position) * DISCHARGE_RATE;

    forall (Car c : available_cars)
        if(c.getBatteryLevel() < estimated_battery_needed)
            available_cars_enough_battery.remove(c);
}

return available_cars_enough_battery;
```

3.2 Money saving option

When the money saving option is enabled, this algorithm enables the customer to find the best special parking area where to leave the car, such that it is close enough to the destination and it has an available plug, maintaining at the same time kind of a uniform distribution of the cars.

```
/** The user inserts his destination */
GeographicalPosition user_destination;
read(user_destination);

/** Zone containing the destination */
Zone destination_zone;
destination_zone = user_destination.getZone();

List<Zone> adjacent_zones;
adjacent_zones = destination_zone.getAdjacentZones();

/** The algorithm only checks for the special parking areas
inside the destination zone and its adjacent zones */
List<Zone> zones_to_check;
zones_to_check.add(destination_zone);
zones_to_check.add(adjacent_zones);

List<SpecialParkingArea> special_parking_areas_to_check;
forall(Zone z : zones_to_check)
special_parking_areas_to_check.add(z.containedSpecialParkingAreas());

/** Finding the special parking area with the lowest percentage
of occupied plugs (to guarantee an almost uniform distribution
of the cars in the city) */
```

```
SpecialParkingArea least_used_special_parking_area;
least_used_special_parking_area = special_parking_areas_to_check.get(0);
float least_used_ratio =
    least_used_special_parking_area.getPluggedCars() /
        least_used_special_parking_area.getTotalPlugs();

float spa_ratio;
forall(SpecialParkingArea spa : special_parking_areas_to_check){
    spa_ratio = spa.getPluggedCars() / spa.getTotalPlugs();
    if(spa_ratio < least_used_ratio){
        least_used_special_parking_area = spa;
        least_used_ratio = spa_ratio;
    }
}

/** Error returned if no parking area in the destination zone
or the adjacent ones has at least an available plug */
if(least_used_special_parking_area.getAvailablePlugs() > 0)
    return least_used_special_parking_area;
else
    error;
```

4 User interface design

4.1 Customer app

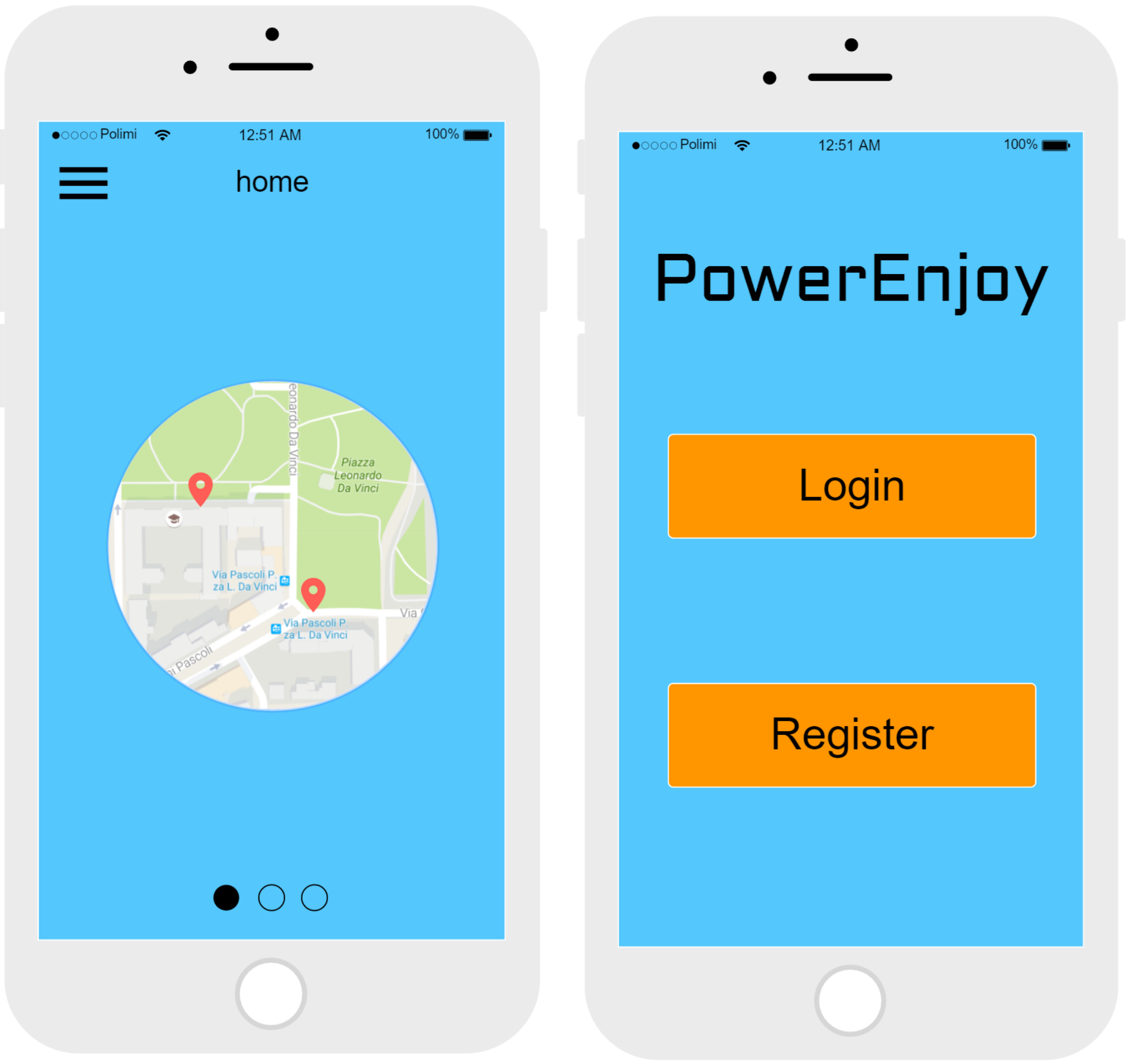


Figure 14: Home and Login-Register views

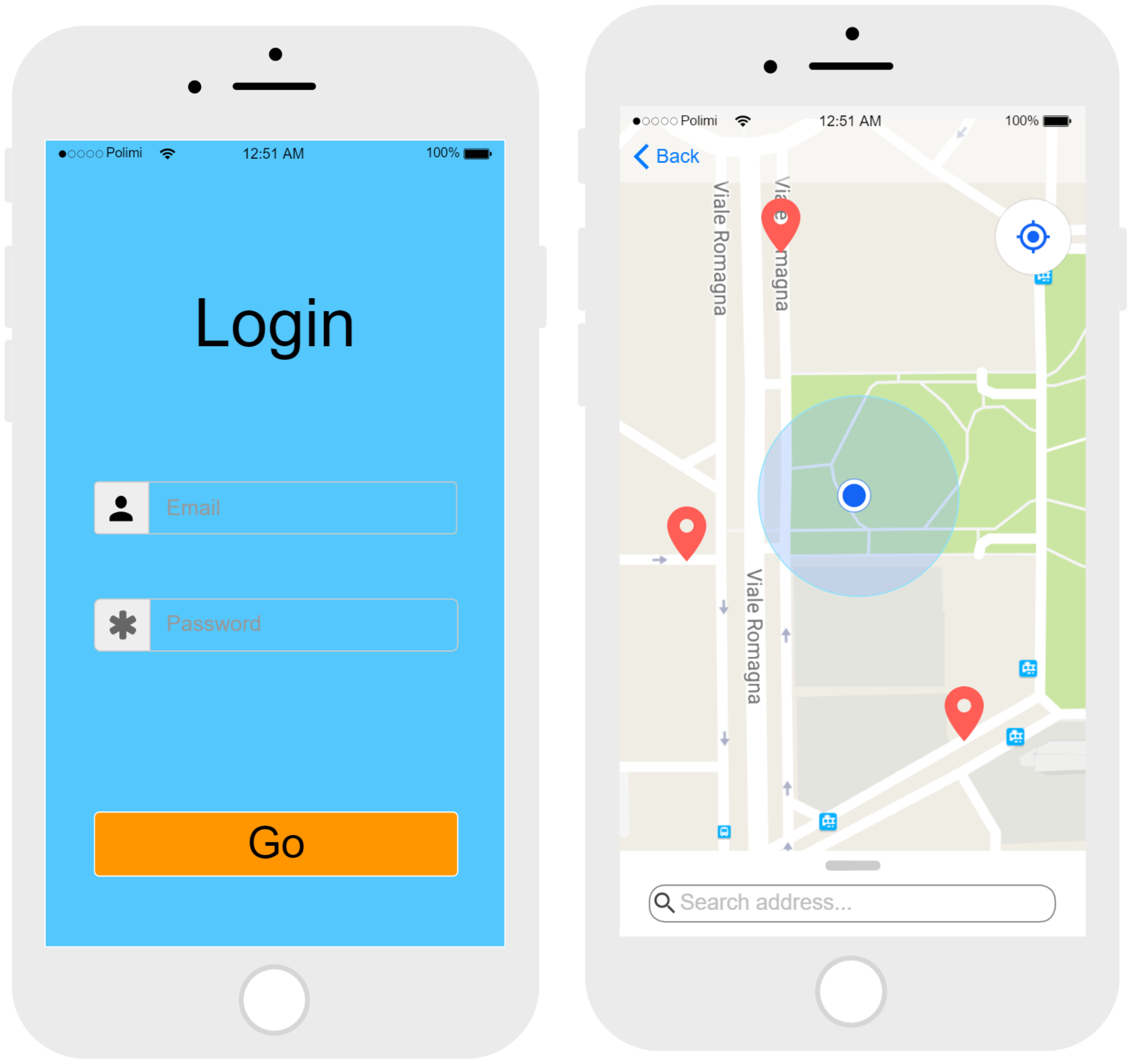


Figure 15: Login and map views

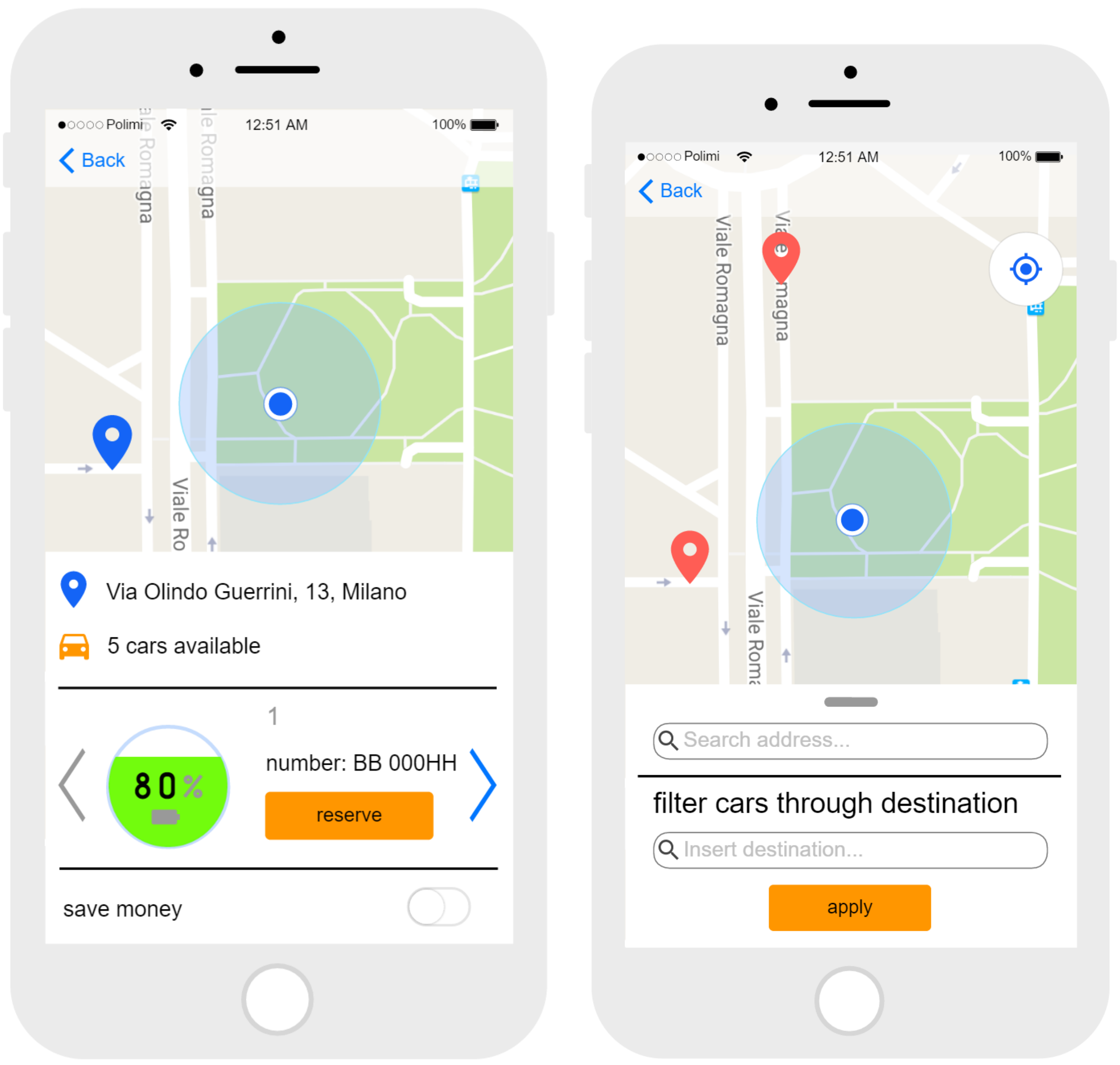


Figure 16: Pick-a-Car and Filter-by-destination views

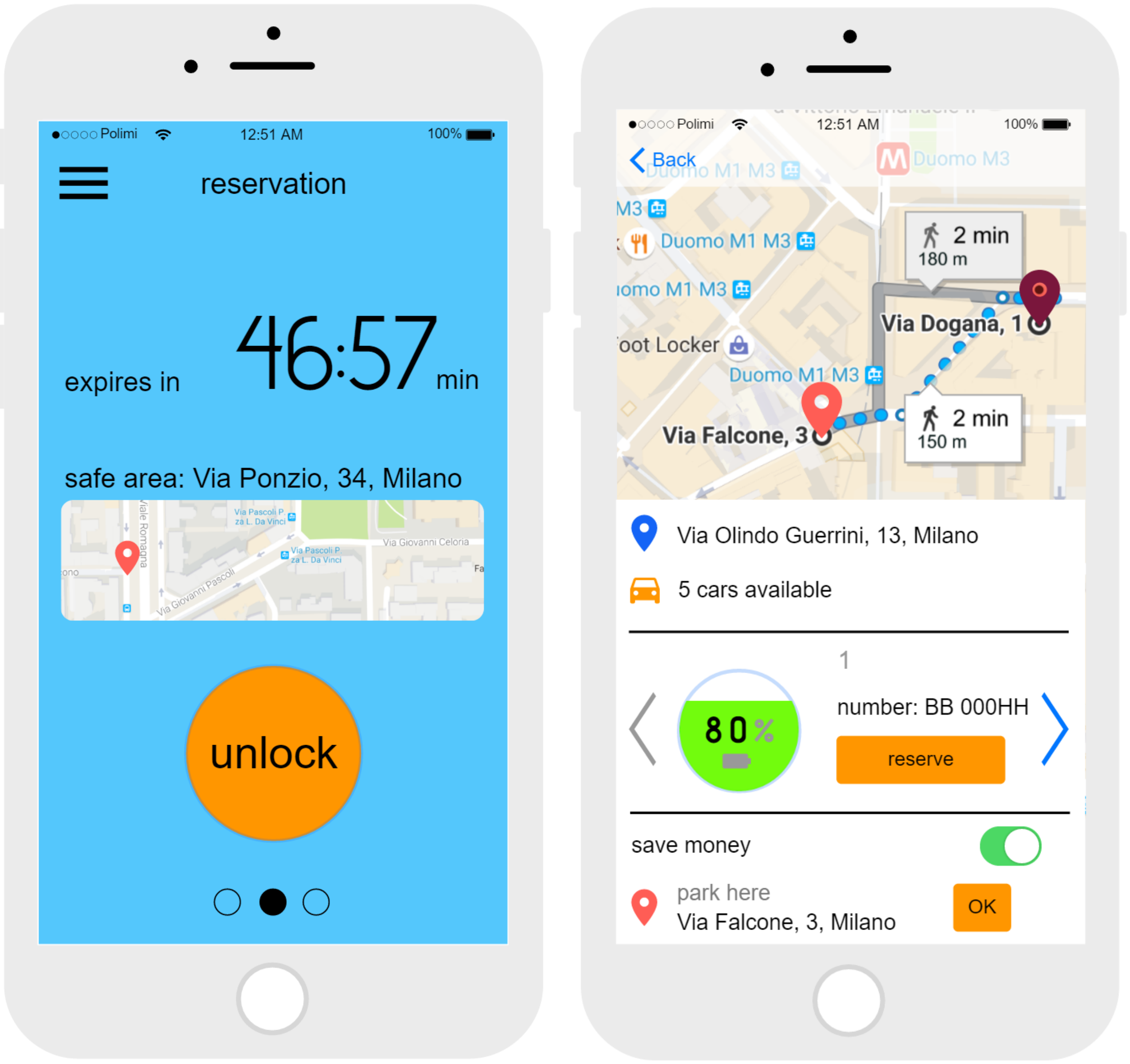


Figure 17: Reservation and Money-saving-option views

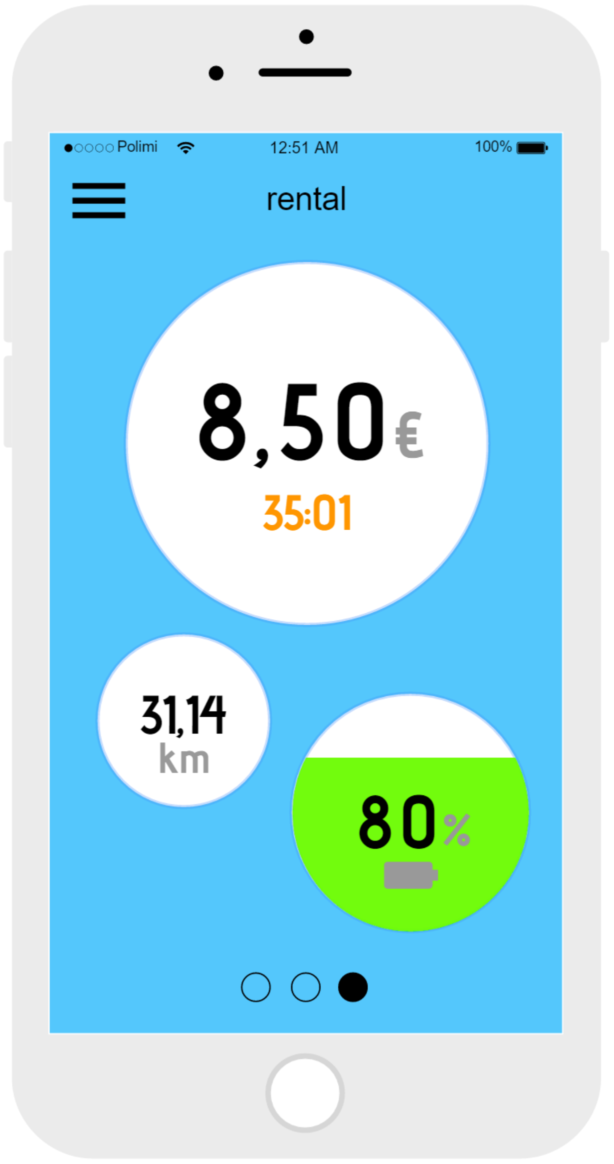
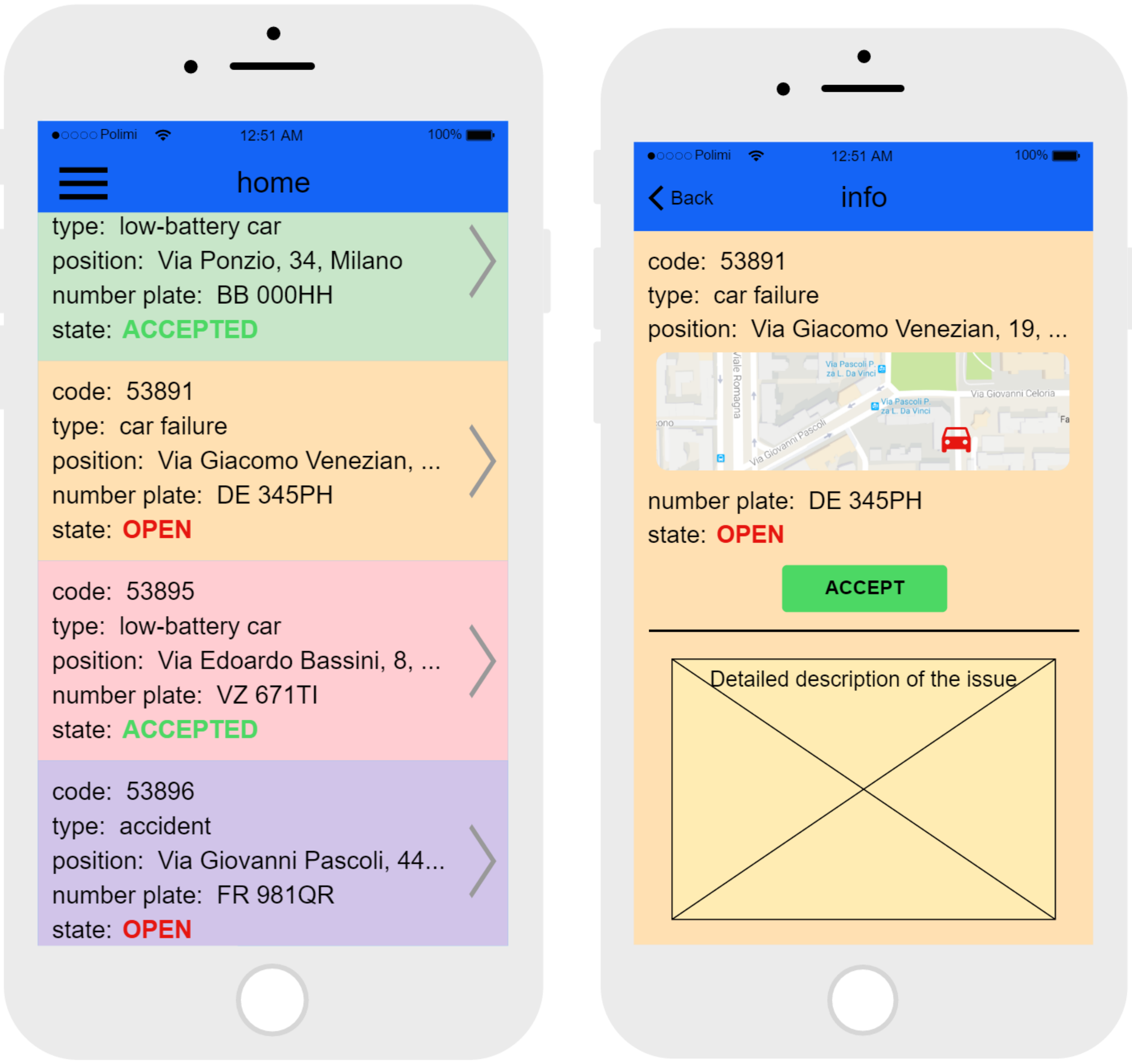


Figure 18: Rental view

4.2 Road operator app



type: low-battery car
position: Via Ponzio, 34, Milano
number plate: BB 000HH
state: ACCEPTED

code: 53891
type: car failure
position: Via Giacomo Venezian, ...
number plate: DE 345PH
state: OPEN

code: 53895
type: low-battery car
position: Via Edoardo Bassini, 8, ...
number plate: VZ 671TI
state: ACCEPTED

code: 53896
type: accident
position: Via Giovanni Pascoli, 44...
number plate: FR 981QR
state: OPEN

code: 53891
type: car failure
position: Via Giacomo Venezian, 19, ...
number plate: DE 345PH
state: OPEN

Figure 19: Home and Detailed-description views

5 Requirements traceability

5.1 Customer

[U.1] Register to the service

- *Access Manager*
- Interaction with *Payment Service*

[U.2] Reserve a car

- *Rental Manager*
- *Safe Areas Manager*

[U.3] Car reservation expires

- *Rental Manager*

[U.4] Drive a rented car

- *Rental Manager*
- *Communication Manager* of the car app
- *Car Remote Manager* of the *Car logic*
- *Navigator Controller*
- *Payment Service*

[U.5] Park and plug the car in

- *Rental Manager*
- *Car Status Manager*

[U.6] Temporarily park the car

- *Rental manager*
- *Car Remote Manager*

[U.7] Share a ride

- *Rental Manager*

[U.8] Managing a failure

- *Rental manager*
- *Car Status Manager*

5.2 Road operator

[U.8] Manage a failure, [U.9] Recover a car involved in an accident, [U.10] Recharge a car with low battery

- *Assistance Request Manager*
- *Status Manager* of the Road Operator Logic
- *Car Control Manager*
- *Car Status Manager*

5.3 Assistance service operator

[U.8]Manage a failure, [U.9]Recover a car involved in an accident

- *Assistance Request Manager*
- *Car Status Manager*

6 Effort spent

Arcari Leonardo

- 29/11/2016 - 4h30m
- 30/11/2016 - 5h
- 02/12/2016 - 3h
- 06/12/2016 - 5h30m
- 07/12/2016 - 3h30m
- 11/12/2016 - 2h
- 08/01/2017 - 1h
- 09/01/2017 - 1h

Bertoglio Riccardo

- 29/11/16 - 2h
- 02/12/16 - 2h
- 06/12/16 - 4h
- 08/12/16 - 1h
- 10/12/16 - 4h

Galimberti Andrea

- 29/11/2016 - 2h - group meeting
- 30/11/2016 - 1h - structuring document
- 01/12/2016 - 3h - added safe areas to component diagram + other minor fixes + meeting (removed safe areas)
- 03/12/2016 - 4h - composite structure diagrams for R and AS Operator + split Car APIs
- 04/12/2016 - 2h - added View Nearby Cars and Money Saving Option algorithms
- 05/12/2016 - 3h
- 07/12/2016 - 2h
- 08/12/2016 - 4h - added RO/ASO Composite Structure and RO interaction diagrams to the document

7 References

RESTful API

- https://www.tutorialspoint.com/restful/index.htm

WebSocket

- https://en.wikipedia.org/wiki/WebSocket
- http://stackoverflow.com/a/29149314
- http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/HomeWebsocket/WebsocketHome.html

Java Persistence API

- https://docs.oracle.com/javaee/7/tutorial/persistence-intro.htm#BNBPZ

JSON

- http://www.json.org/
- http://json-schema.org/
- https://spacetelescope.github.io/understanding-json-schema/

8 Changelog

- 15/01/2017 - Version 1.3.1
  - Removed Google Maps API in Customer logic
- 09/01/2017 - Version 1.3
  - Refactored *Car Logic* components diagrams according to the new defined standards
  - Added *Logic Components Composite Structure* section
- 08/01/2017 - Version 1.2
  - Updated RO and ASO Logic Component views
  - Updated Customer Logic Component diagram and description
- 03/01/2017 - Version 1.1
  - Changes to *Customer Logic composite structure diagram*: deleted *Access manager* specialization
- 14/12/2016 - Version 1.0.1
  - *Layout fixes*
- 11/12/2016 - Version 1.0