

1 Architectural design

1.1 Overview

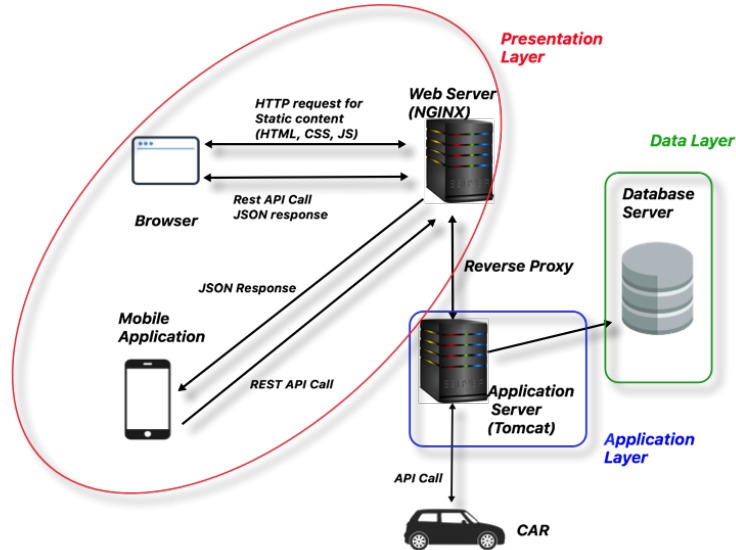


Figure 1: High level architecture

In this section is presented an high level architecture. It shows how is structured the system and the main interaction between subsystems.

There is a WebServer with Nginx technology that serves static content (like .html, .js, .css) to browser. When WebServer receives certain API calls it acts as reverse proxy. Proxying is used to distribute load among several servers. When web server proxies a request, it send the request to the Application Server, fetches the response and sends back to the client.

Application server contains the business logic of the software to be and interacts with the Database Server and with the CarOS.

Mobile Application only makes API calls to WebServer and receives back the JSON response of Application Server.

This project starts as a monolithic application because of simplicity of development, but with particular attention to the modularization. So, in case will be the need of scaling, the refactoring to micro-services won't be difficult.

1.2 High level components and their interaction

In this section are presented the high level component and it's described how they interact with each other. The client component is made of the Web Browser and the Mobile Application. Both communicates with the server through its interface.

The application server communicates with DBMS through DBMS API and with CarOS through it's API.

In this way the central system always knows the status (position, battery, etc.) of all cars and also cars can initiate a communication with server when they need to communicate important event. This is done with observer pattern as specified later in this document.

The communication between client and server can be synchronous or asynchronous depending on the kind of interaction. The server can communicate asynchronously with client with notification or messages (email). This is the class diagram of the data core (likely during the coding phase a more detailed data structure will come up). There are two important consideration to do about this model: the first one is that the two relations user-reservation and user-ride will guarantee the reconstruction of users "story" in the use of the Power Enjoy service, and the second one related to a bit more technical solution about car system. In the car class there are some informations that in general shouldn't be stored, like battery life and position, because they change rapidly. The explanation of this decision is in the component diagram description, in particular in the explanation of how the car proxy component works.

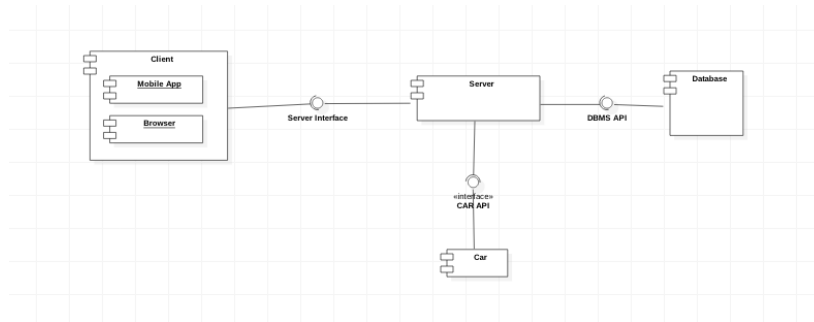


Figure 2: High level component diagram

1.3 Data model

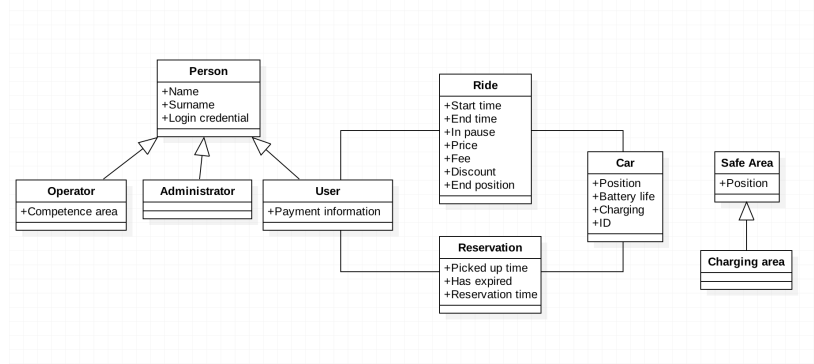


Figure 3: Data class diagram

This is the core of the data stored in the db. It is not complete, it contains only the more important information and it is possible that during the coding phase more structures will come up. There are two important considerations to do: the first one, is that the two relations User-Ride and User-Reservation lead to the construction of the whole story of a user, and the second one, more technical, is related to the Car class. This class contains some information that in general aren't stored in a database, because they change rapidly, like Position and battery life. This decision is motivated and explained in the description of the car proxy component, in the following section.

1.4 Component view

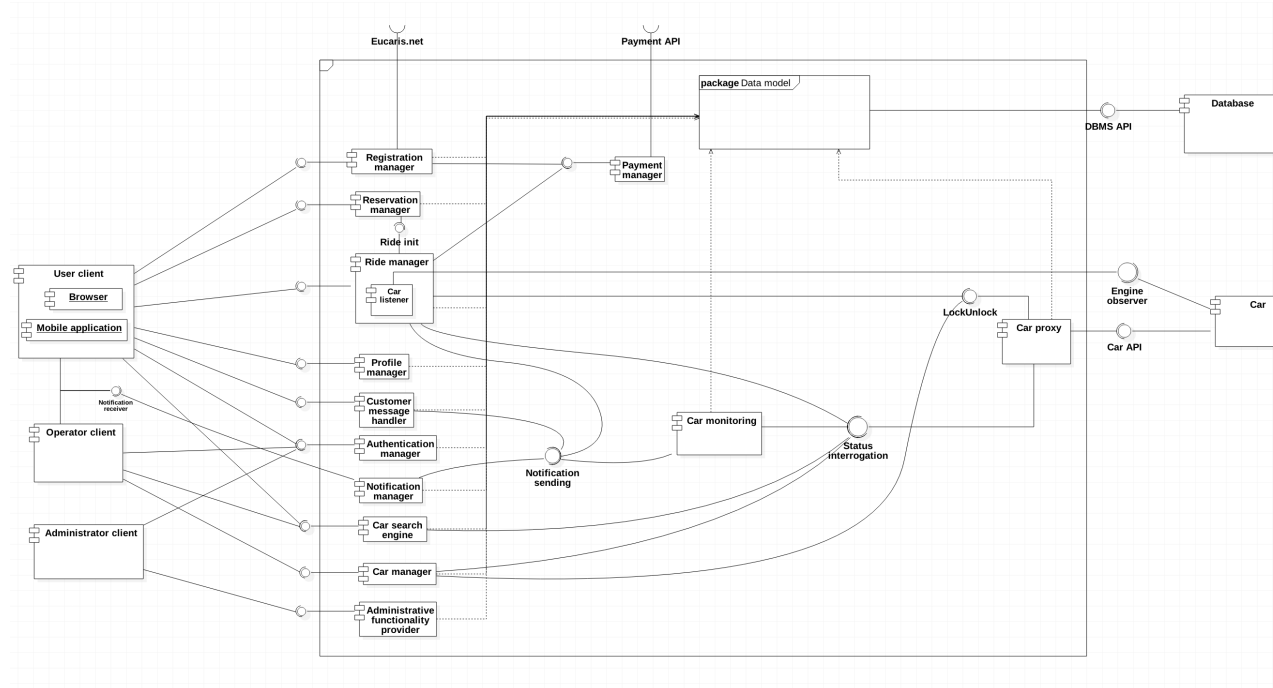


Figure 4: More detailed component diagram

1.4.1 Car proxy component

Car proxy is the abstraction of cars in the server. It must be invoked by other internal subsystems that need information about cars physical status (real time informations), like position, battery life etc. This component absolutely doesn't care about information related to reservations or rides.

The main purpose of this component is to request cars using their API, but its implementations should also guarantee that the systems working on cars don't have to support high amount of parallel request. In order to do that, car proxy will use the database, storing a physical characteristic related to the timestamp of the API call that has provided that information. When a request arrives, car proxy decides which data should be provided: the one got through the API call to the car, or the one stored in the database. If the stored data are sufficiently recent then they can be provided, otherwise it is needed to make the request. As it can be seen in the diagram, this component has a considerable fan-in, it could be necessary in future to make it scalable.

1.4.2 Car monitoring component

This component is substantially a daemon. It periodically asks to the car proxy component the informations that operators need to do car maintenance. If a damage has been detected, the notification component should be called, in order to communicate it to an operator.

1.4.3 Ride manager

Ride manager is the component that takes care about the data in the server corresponding to rides in the real world. Furthermore, when a ride is set to pause, it use the car proxy component to lock or unlock the car.

1.4.4 Reservation manager

This component manages users reservation request, setting the status of a car from available to reserved and vice versa. It should also take care about reservation expiration, resetting the car state from reserved to available, and taxing the user through the component payment manager.

1.4.5 Registration manager

This component accepts request from guest of joining the Power Enjoy service. It checks if the guest has a valid drive license and adds it to the users. It also verifies the validity of the payment informations received by the guest using the services provided by the payment manager component.

1.4.6 Profile manager

This component reply to users that want to see their past utilization of the power enjoy services.

1.4.7 Customer messages handler

This component receives messages about users regarding malfunctions of cars, and use the notification manager to notify the operator that should take care about that.

1.4.8 Authentication manager

The authentication manager the component that manages the login of user, operator, and administrator, and administrates session.

1.4.9 Car search engine

This component interrogates the car proxy component and get cars position, performing the two type of car research.

1.4.10 Car manager

The car manager component can be used by operators to change car status in the server. For instance, if an operator is going to work on a car, it should use this component to switch the car status to “under maintenance”, and then to unlock it.

1.4.11 Administrator functionality provider

This component provides the functionalities accessible only by the administrators that are listed in the RASD document.

1.4.12 Payment manager

This component uses external API of the accepted payment service. It should be used not only for carry out the payments, but also to verify during the registration of a user that the provided payment informations are correct.

1.4.13 Notification manager

Notification manager is used to send notifications to users (when their reserved car changes status), and to operator (when the system detects that an intervention on a car is needed, or a user make a communication).

1.4.14 Car listener

This is a subcomponent of the ride component, that is instantiated for a ride when the car is successfully unlocked. After the unlock of a car, probably the engine will be turned on. This component fulfil the problem of getting the engine ignition time, in order to calculate the correct ride cost. The runtime flow is explained with a sequence diagram in the section Runtime view.

1.5 Deployment view

The deployment diagram shows the hardware of the system and the software that it's installed on it.

In this diagram there is the Mobile Application installed on Mobile Phone of the user.

The browser runs on user/operator/admin PC.

The Application Server and the Web Server are on different Nodes and on different environment. In this way they are totally decoupled and it's ensured the scalability of the system. The DBMS runs on a different node in order to not overload a node with a huge load.

If the need of scalability becomes more important for the system there could be more application server on different nodes with a load balancer before them. This is compatible with a cloud approach.

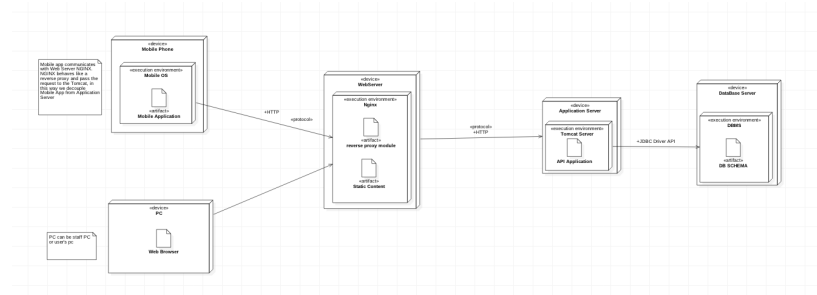


Figure 5: Deployment view

1.6 Runtime view

1.6.1 Registration

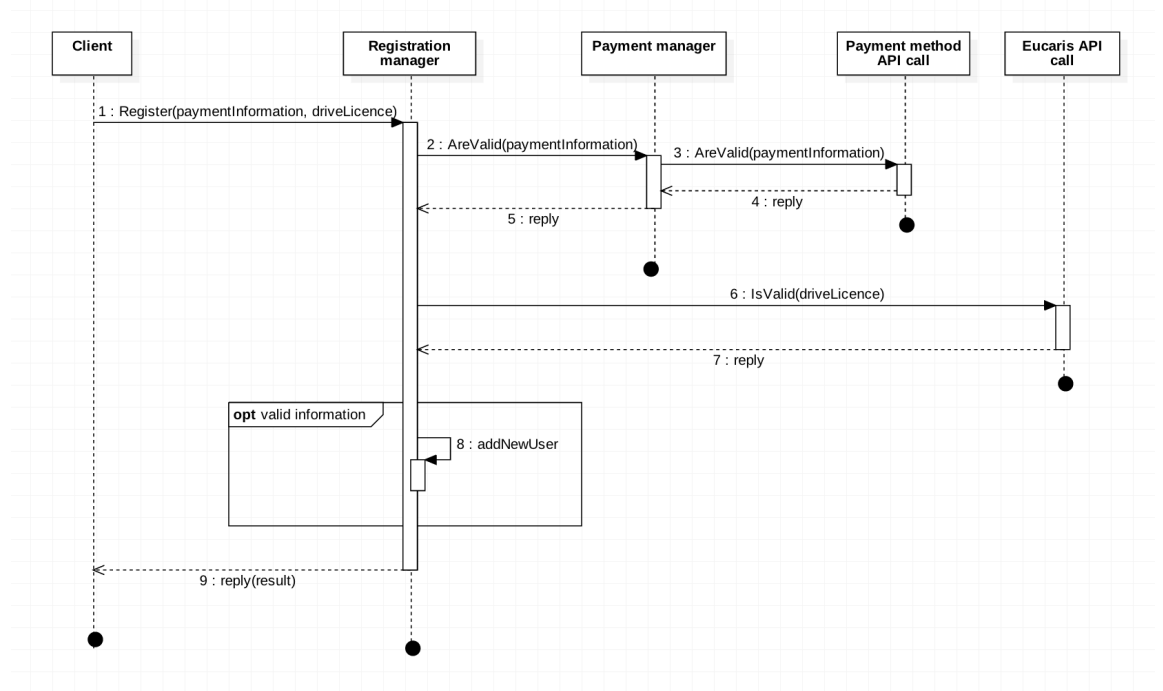


Figure 6: Registration runtime view

The client application request is sent to the registration manager component, that have to verify that the payment information and the driver license are

valid. To do this, as mentioned in the product perspective section of the RASD document, external services are used. Then, if the information are valid, a new user is created and inserted in the database, and the result of the operation is sent to the user.

1.6.2 Car search

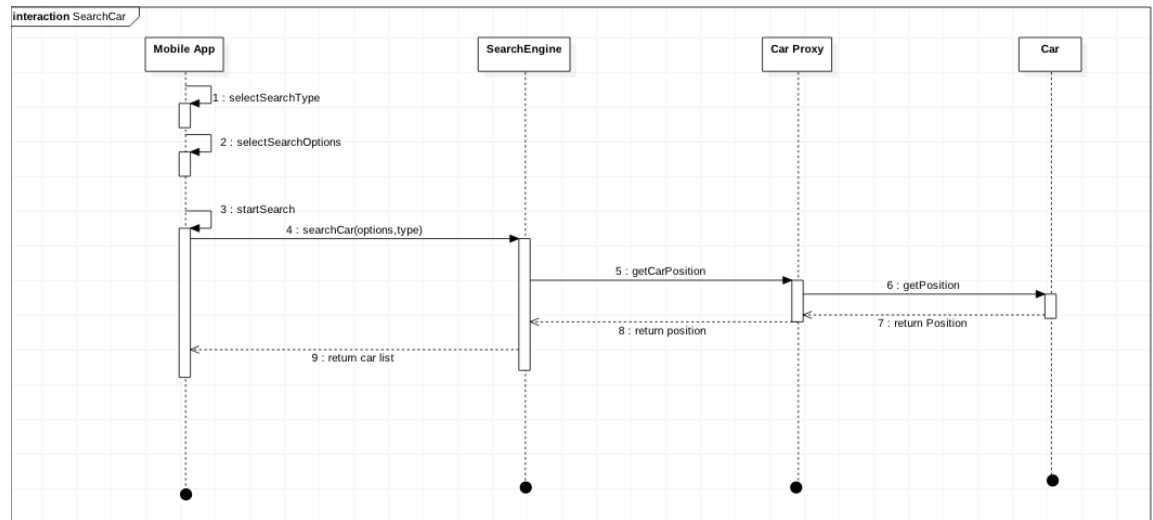


Figure 7: Car search runtime view

The user through the application insert the information needed in order to do the research, then the information are sent to the search engine that performs the research using the algorithm described in the algorithm section of this document. The position of the cars are obtained interrogating the car proxy component.

1.6.3 Reservation

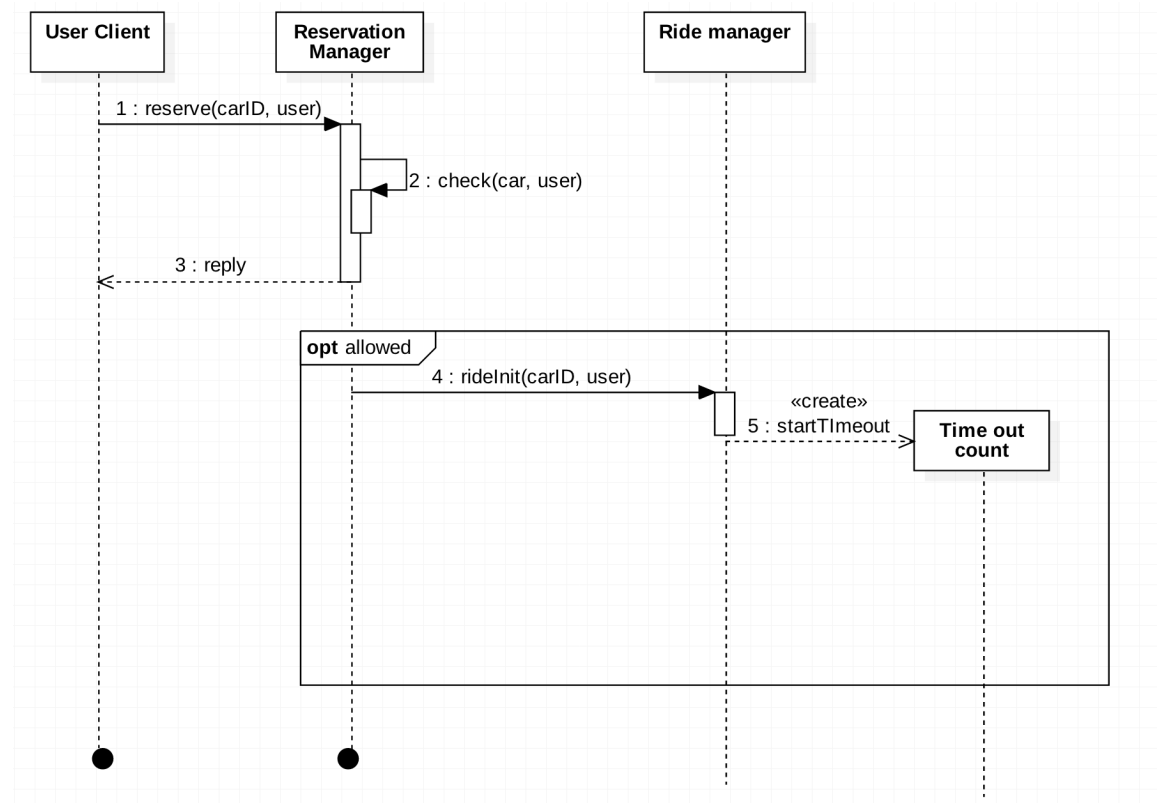


Figure 8: Car reservation runtime view

When the request is received the reservation manager check if it the user can reserve the car identified by carID, and send back the outcome. Then if the reservation is allowed, the reservation manager delegates to the ride manager the managing of the car picking up.

1.6.4 Unlock car

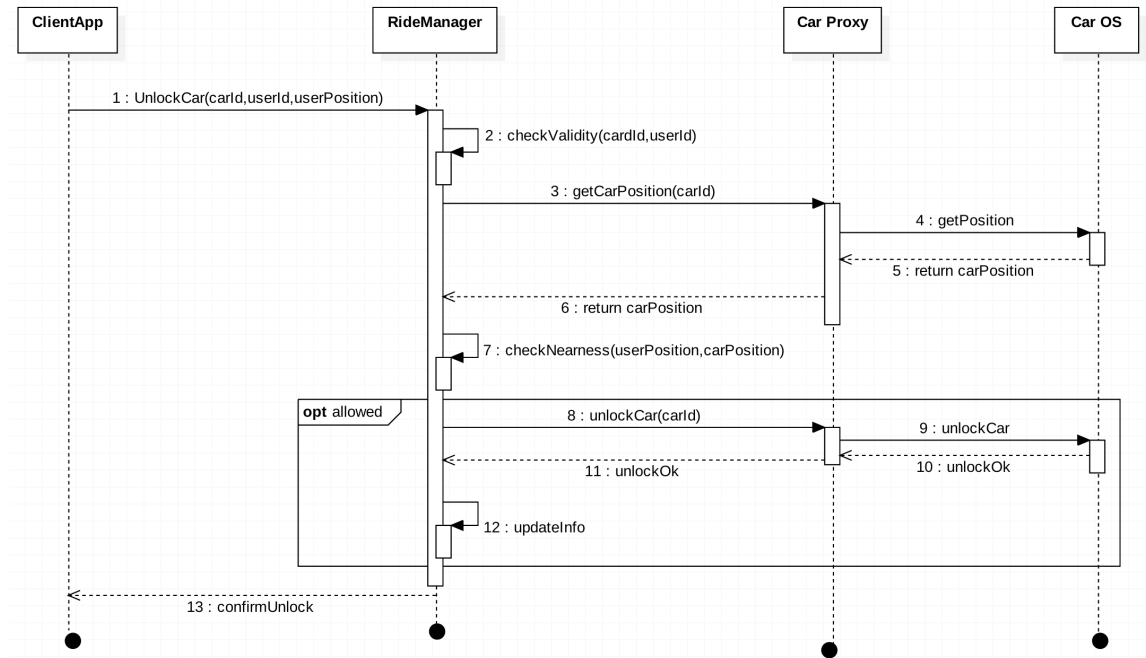


Figure 9: Unlock car runtime view

When a client make the unlock request, the system verifies that the request owner is allowed to unlock the specified car. This operation is carried out checking two things: that the user has actually reserved the car, and that it is sufficiently close to that car. Then through the car proxy the car is unlocked, and the outcome is sent to the user.

1.6.5 End ride

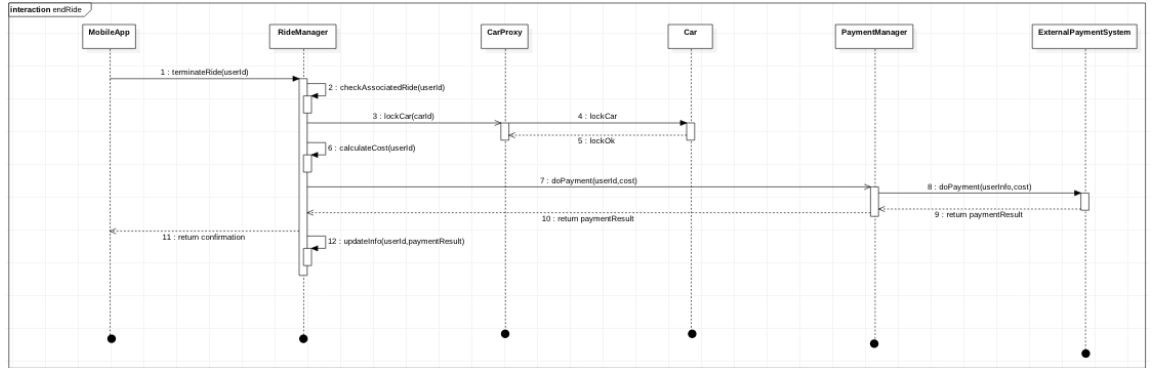


Figure 10: End ride runtime view

The `stopEngineCallback` remote method is called by the car system on the subcomponent of the ride manager (car listener) when the stop of the engine is detected. The call contains as parameter the `cardID`, so the ride manager component can find in the persistent data the user that is riding the car. In this way the system can ask to a user that has stop the engine if it wants to end the ride, or put it in pause (it is necessary to manage the situations in which there is no reply). Then if the ride is finished, the payment can be carried out through the payment method API call.

1.6.6 Car monitoring

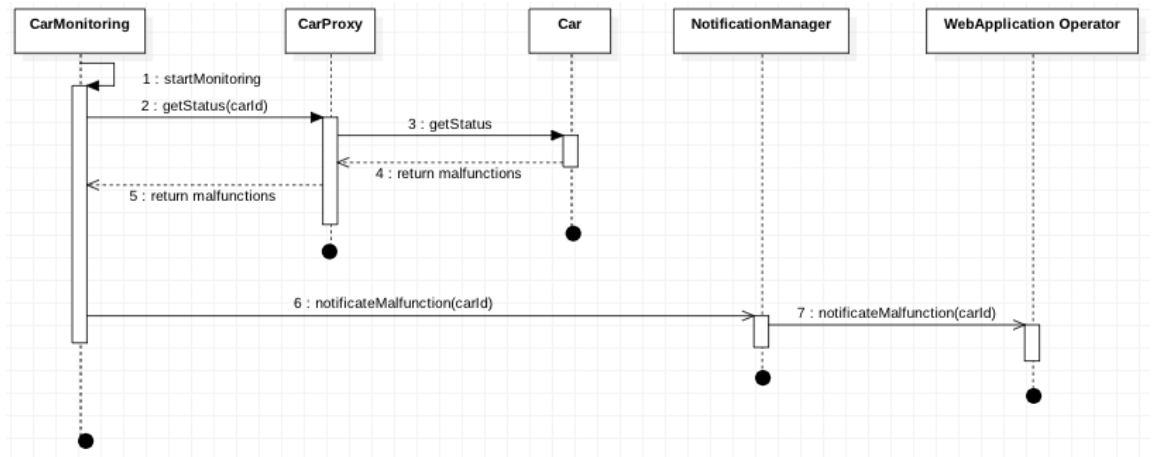
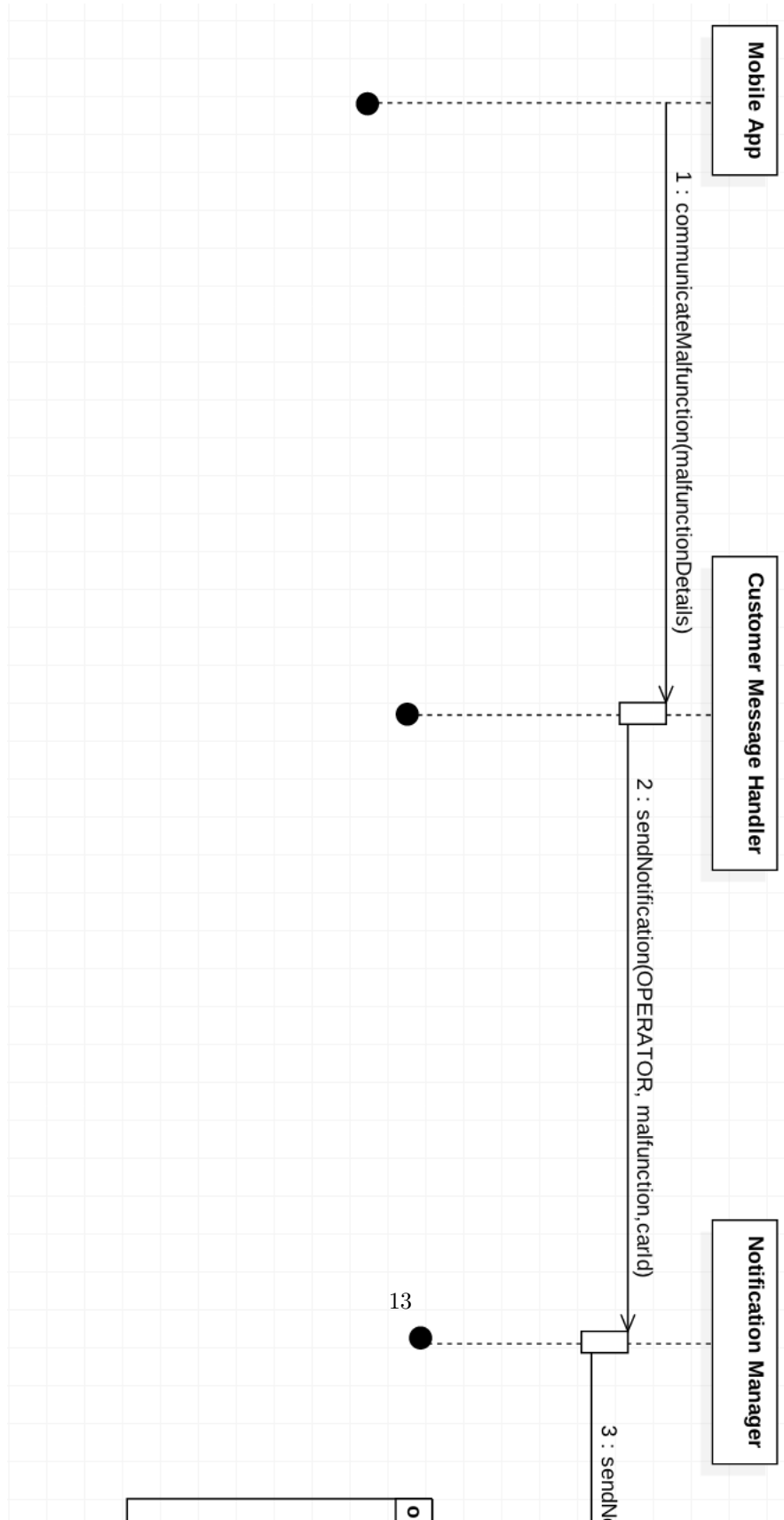


Figure 11: Car monitoring runtime view

Here is explained how the system try to keep cars in a good state. The car monitoring component makes periodic requests to the cars system, asking if the system needs some kind of operators intervention. If for instance a malfunction is detected, or the battery level is too low, the central system can notify the operator that is working in that area.

1.6.7 Communication of a malfunction



In this diagram is explained how a user can communicate to the system eventual cars problem. Further, when an operator decides to work on a car, it can set its status to “under maintenance”, so no user can reserve it.

1.6.8 Car proxy

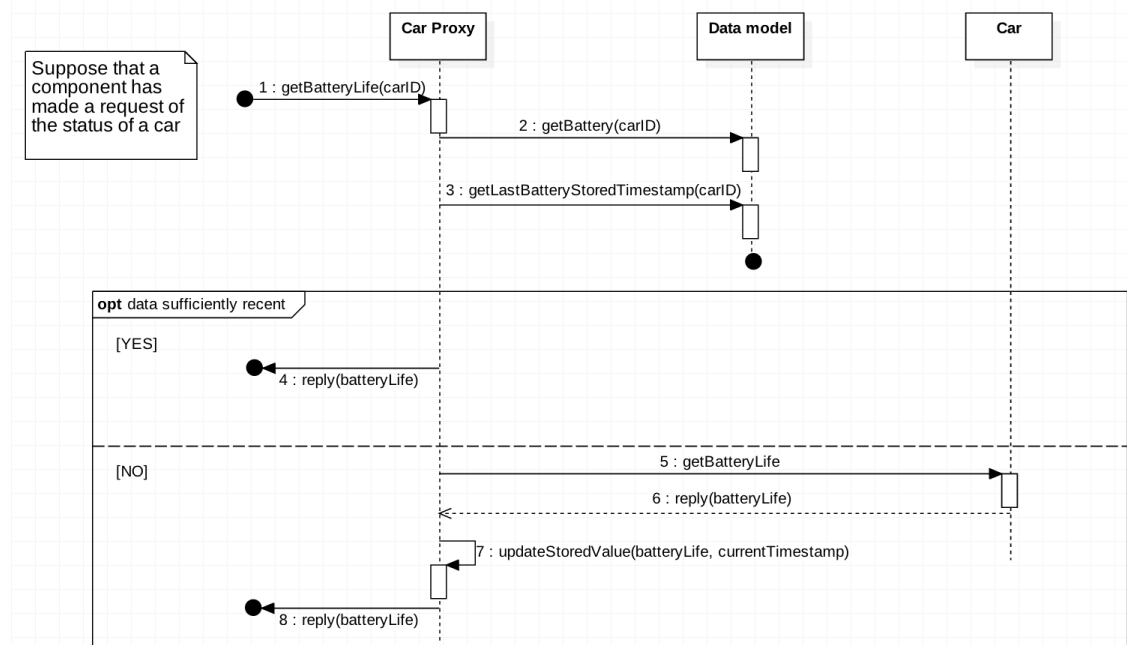


Figure 13: Example of car proxy runtime view

In this diagram is shown an example of the functioning of the car proxy. Attention was paid in particular on how it checks, when a request to a car should be done, if the stored value is sufficiently recent or not. In this way it can decide if it is necessary to make an actual remote request, or the if cached value can be used.

1.7 Component Interfaces

Components provide some interfaces through which it's possible to other components to use the services provided.

Some interfaces are protected and are accessible only to specific components.

Here is the list of components and their interfaces.

Registration Manager:

- doRegistration(username,password): creates a new user if possible.

Reservation Manager:

- doReservation(userId, reservationInfo): creates a new reservation and starts the timer.

Ride Manager:

- unlockRequest(userId,carId): calls unlock interface provided by Car Proxy
- lockRequest(userId,carId): the same as before.
- startRide(ride): called when the user wants to start a ride.
- pauseRide(ride): called when the user wants to pause a ride.
- terminateRide(ride): called when the user wants to terminate a ride.
- rideInit(ride): called by Reservation Manager when the user creates the ride.

Profile Manager:

- getProfileInfo(userId): returns the profile information associated to userId.

Customer Message Handler:

- sendCommunication(communication): called when the user wants to send a communication regarding malfunctions or damages to the system.

Authentication Manager

- doLogin(username,password): manages login and sessions.

Notification Manager

- sendNotification(notificationInfo, notificationType): sends a notification of type notificationType with the content notificationInfo.

Car Search Engine

- searchCars(position,radius): searches for cars near the position.

Car Manager

- setCarStatus(carId,status): set the status of car
- getCarStatus(carId): get the status of the car (i.e. available, busy)
- unlockCar(carId)
- lockCar(userId)

Administrative Functionality Provider

- `addSafeArea(position,safeAreaType)`: adds a safe area. `SafeAreaType` stands for simple safe area or charging area.
- `deleteSafeArea(safeAreaId)`
- `addOperator(username,password,competenceArea)`
- `deleteOperator(username,password)`
- `addAdmin(username,password)`
- `deleteAdmin(username,password)`

Car Monitoring: it doesn't provide any interface to other components.

Car Proxy:

- `lockCar`
- `unlockCar`
- `getCarStatus`

Payment Manager:

- `doPayment(userId)`: it calls the external payment API in order to complete the payment.

Notification Receiver: this interface is provided by both Web Application and Mobile application in order to receive notifications.

Car Listener:

- `startEngineCallback(carId)`: called by the carOS when the engine starts. It's a callback method in order to be informed by important event of the car.
- `stopEngineCallback(carId)`: the same as above.

1.8 Selected architectural styles and patterns

1.8.1 Tiers

The system will be divided into 5 tiers:

1. Database Server
2. Application Server
3. Web Server
4. Mobile Application

5. Car

Note that here the car is considered as a tier but actually is a blackbox capable of receiving request from our system and to call some interfaces provided by components of the system.

This division is in order to not overload any machine. Application Server and Web Server are deployed on different machine because if due to great load there will be the need of another Application Server will be easier to instantiate another machine.

1.8.2 Layers

As said before the system will be divided into 3 Layers:

- Presentation Layer: it's distributed on the mobile application or on the Web Server.
- Business Layer: it's on the Application Server.
- Data Layer: it's on the Database Server.

Dividing layers is important for decoupling and for dividing responsibilities. In this way different developers can focus on different tasks abstracting from other layers. Diving in layers is also important for mental clearness because in the case of a problem it should be clear where the problem is located.

1.8.3 Protocols

In this section is described how different tier communicate with each other and how they exchange data.

JDBC

Used by the Application Server to communicate with the database server. JDBC API is the standard for database-independent connectivity between the Java programming language and a wide range of databases SQL databases and other tabular data sources. The JDBC API provides a call-level API for SQL-based database access.

RESTful API

Used by both Mobile Application and Web Application to access the services provided by Application Server component.

1.8.4 Design Patterns

Client-Server

Client server is the base of the architecture. The central system is the server and provides services to both web application and mobile application. The

central server communicates with car that acts as a server with respect to central system.

Monolithic

It's decided to use a monolithic approach because of simplicity of development. Particular attention is give to decoupling and modularization. In this way the refactoring to micro-services will be easy if there will be the need of scalability. With the attention to decoupling a cloud approach it's possible and it's possible to deploy different part of the system on different machines.

Proxy

Proxy Pattern is used to communicate with the car. Car Proxy is the abstraction of the car in the system. When a component needs to communicate with the car, it calls the method provided by car proxy. Then car proxy decides wether to call car API or to get information from the DB.

Observer

Observer Pattern is used to get callback from car's important event like engine start or engine stop. CarObserver is instantiated when the user unlocks the car and the car is informed that it needs to call the observer when an important event takes place.