Version 1.7

Software Engineering Project

Design Document

Vianney Payelle - Rémi Rigal - Noëlie Ramuzat

Revision Notice

Versions	Date	Description	Modifications
V1.1	26/11/2016	Creation of the document	_
V1.2	28/11/2016	Creation of the introduction part	Addition of : - Purpose - Scope - Document Structure
V1.3	04/12/2016	Creation of : - UML diagrams - Architectural part	Addition of : - Component View - Sequence diagrams - Architectural Overview - High level Components
V1.4	08/12/2016	Creation of the Selected architectural styles & patterns part	Addition of : - Overall architecture - Protocols - Design Patterns - Deployment Diagram
V1.5	10/12/2016	Creation of : - Algorithm design - User interface design - Requirement Traceability	Addition of : - Algorithm of reservation - Algorithm of unlock car - UX diagram - BCE diagram - Requirements Traceability
V1.6	11/12/2016	Validation of the document	Modification of : - Protocols - Design Patterns
V1.7	13/12/2016	Validation of the document	Addition of: - Requirements Traceability

Table 1 : Design Document versions

Table of contents

List of tables

List of figures

1. Introduction

1.1. Purpose

The Design Document provides a functional description of the PowerEnjoy's system. Its aim is to present the architectural design of the application and to explain the taken decisions for this choice. This document is a technical description of the interaction between the software and the users and is intended to be a support for the developers.

1.2. Scope

The scope of the project PowerEnjoy, which is a service based on mobile application, is to manage, design, build, and implement a service aimed at facilitating public transportation. The application provides to its target, the client, a way to research an electric car near a position, reserve it and pick it for a ride. At the end the application sends the ride's bill to the client. The PowerEnjoy application needs the client to be registered in its database before he can reserve a car, for security and payment reasons (credential information, driver license, and identity card). When the client logs in, the mobile application allows him to reserve a car around an address or his GPS position. Then it provides him details about his reservation on the main page. The application also allows the client to cancel his reservation, unlock the reserved car when he is near it, and access his account details to modify it.

After the ride the application locks the car and sends an email to the client with the bill of the ride. The mobile application can moreover give discounts and charges in function of the client's ride such as sharing the car or plug it in power grid station. The PowerEnjoy application is built in order to ensure an easy and clear reservation service as well as an effective data collecting and saving. This refers the PowerEnjoy application simplifies the customer's uses, optimises the time to reserve a car and adjusts the price of the ride according to the driver.

1.3. Definitions, Acronyms, Abbreviations

Name	Definition
RASD	Requirements Analysis and Specifications Document
DD	Design Document
SMS	Short message service: notification sent to a mobile phone SMS gateway is needed to use it
SMS Gateway	Service which allows to send SMS via standard API
API	Application Programming Interface
MVC	Model View Controller
Push notification	Notification sent to a smartphone using mobile Application
REST	REpresantional State Transfer
RESTful	REST with no session
DBMS	Database Management System
BCE	Business Controller Entity

Table 2 : Glossary of the Design Document

1.4. Reference documents

Name	Publication Date	Authors	Contents
Assignment AA 2016-2017 Software Engineering 2	14/10/2016	Elisabetta di Nitto	Project goal, schedule and rules
RASD	16/11/2016	Rémi Rigal Vianney Payelle Noëlie Ramuzat	Requirements Analysis and Specifications Document of the project

Table 3 : Reference documents used in the Design Document

1.5. Used Tools

Name	Use
Github & SourceTree	Control the document versions
Modelio	Create the Architecture models
Dia	Create the Component view
Edraw Max 8.4	Create the Architecture models
Adobe Acrobat Reader DC	Create the Design document PDF
Android SDK	Create the smartphone app mock-up

Table 4 : Description of the tools used to create the Design document

1.6. Document structure

In the following parts of the document are described the different part of the application's design and the needed RASD requirements.

Architectural design
This section introduced the high level architecture of the PowerEnjoy system, its main components and interfaces, its runtime behaviour and its design patterns.

Algorithm design
In this second part, the most critical parts are presented. To this end, some explicit algorithms in pseudocode are provided for the implementation.

User interface design
This section provides an overview of the user interfaces of the application by giving mockup.

Requirements traceability
In this part are describing the RASD requirements that cover every design elements defined in the Design Document.

Finally the hours of work repartition follows this section.

2. Architectural design

2.1. Overview

The PowerEnjoy system has a three-tier architecture: a database server, an application server and a user interface. The user interface being composed of the mobile application only, a RESTful API will ensure all the communications between the application server and the client.

The main benefit of this kind of structural implementation is its flexibility as it becomes simple to have several database servers holding the data or it can easily integrates a cloud system.

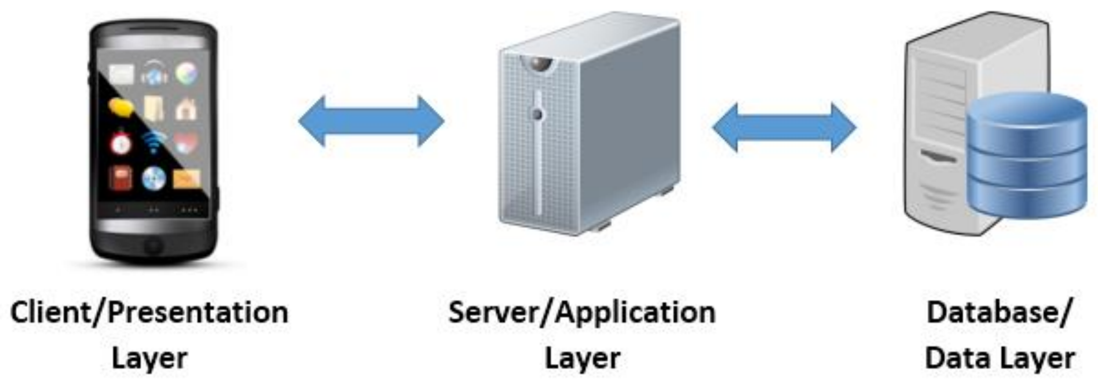

Figure 1: Three-tier architecture schema

2.2. High level components and their interactions

The high level components are composed of different layers, the first one is the database component that holds all the data and provides an exhaustive way to trace all sensitive actions such as payment transactions. Then, the application server is responsible for all the logic of the system and is the only component to have access to the database. Thus it provides all the needed information to the mobile app and the administration computer and the next layer. The latter allows the PowerEnjoy Company to monitor the whole system at any time providing the location of all the cars and eventually their current state so that an employee can be informed if maintenance is needed for example.

The mobile app communication with the server initiates when the driver opens the application and logs in, the RESTful API will ensures the exchange of data between those two components.

2.2.1. Subsystems Structure

For better understanding, the whole system can be divided in several sub-systems holding a unique functionality:

- User Management
  - History
  - Account
  - Credentials
- Sign Up Management
- Log In Management
- Reservation Management
  - Car
  - Ride
- Notification Management

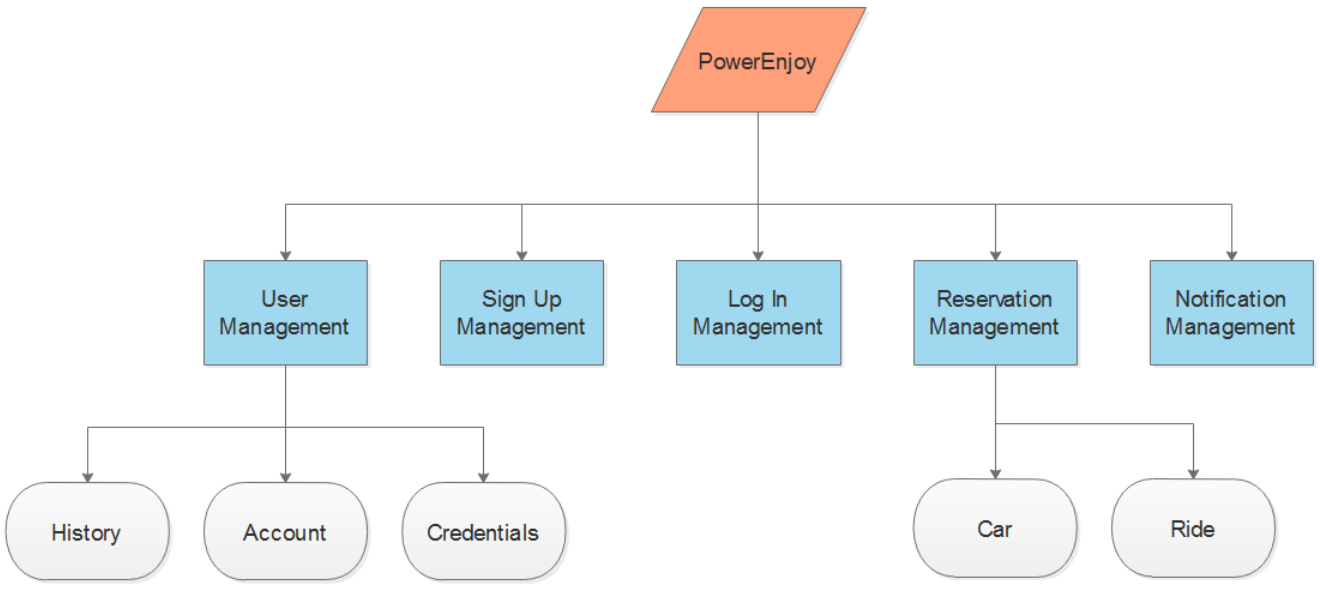

Figure 2: Subsystems diagram

2.3. Component view

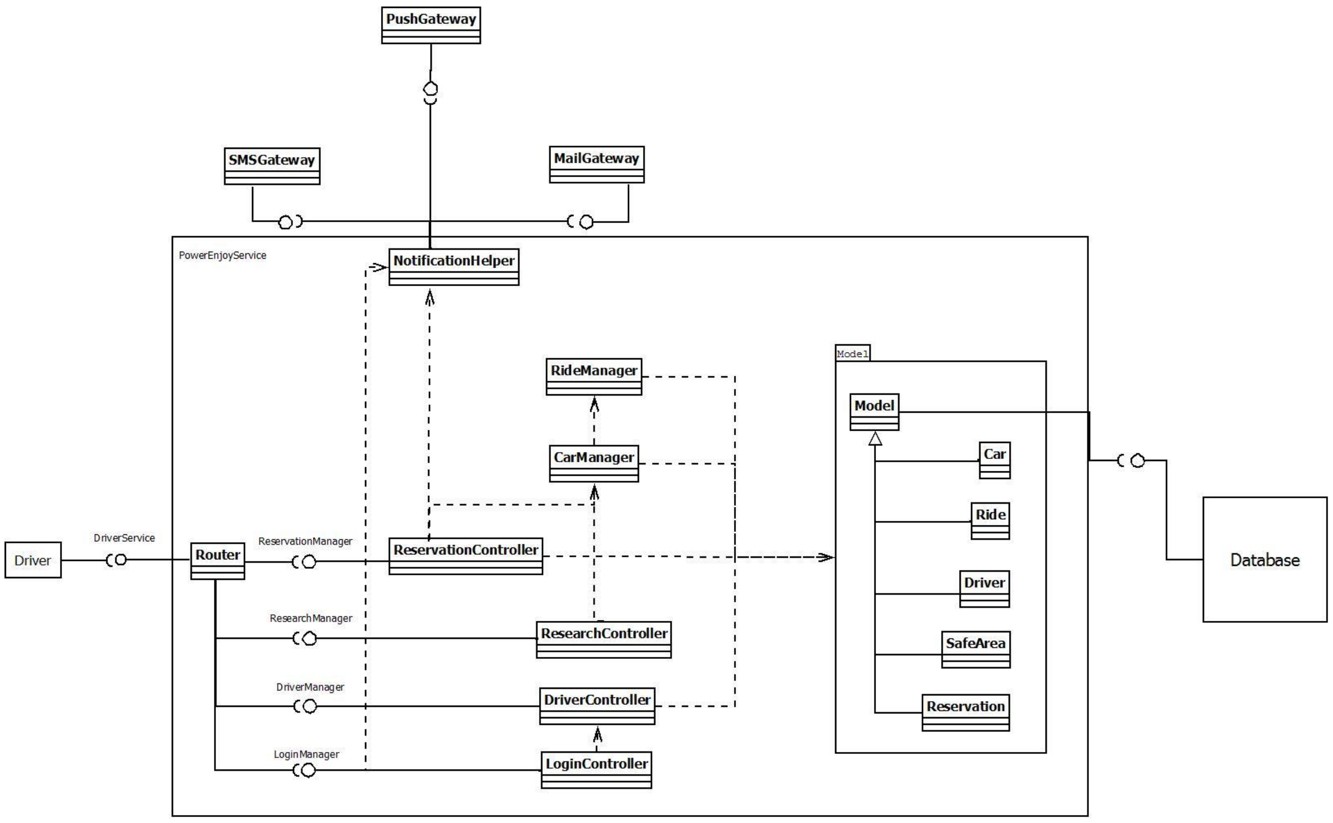

Figure 3: Component View Diagram

- Driver : Represents the driver's mobile application
- Router : Associates the driver's requests to the appropriate Controller
- LoginController : Registers the driver in the database or verifies the email and password of this one. Uses the NotificationHelper to send driver's password by email
- DriverController : Manages the drivers, changes its data in the database
- ResearchController : Manages the research, using the CarManager to access the location of the car
- CarManager : Manages the cars' data in the database
- RideManager : Manages the ride's data in the database
- ReservationController : Manages the reservation, using the CarManager and RideManager to access data
- NotificationHelper : Manages the notifications, using the PushGateway, SMSGateway and MailGateway to interact with the driver by different way.
- PushGateway : Manages the push notifications
- SMSGateway : Manages the SMS notifications
- MailGateway : Manages the email notifications
- Model : Represents the world and the data with which the system interacts
- Database : Represents the database used to store persistent data

2.4. Deployment view

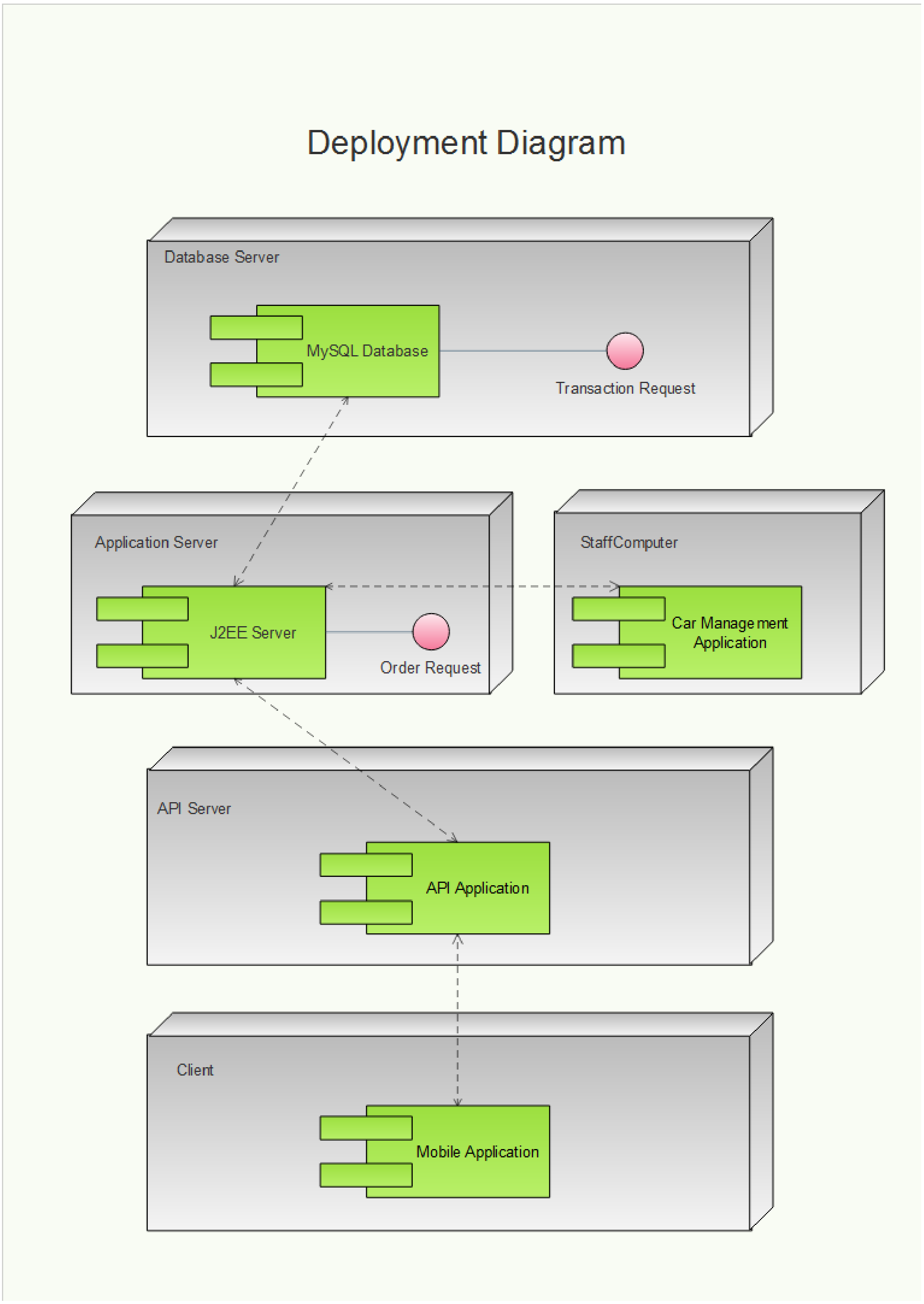

Figure 4: Deployment Diagram

2.5. Runtime View

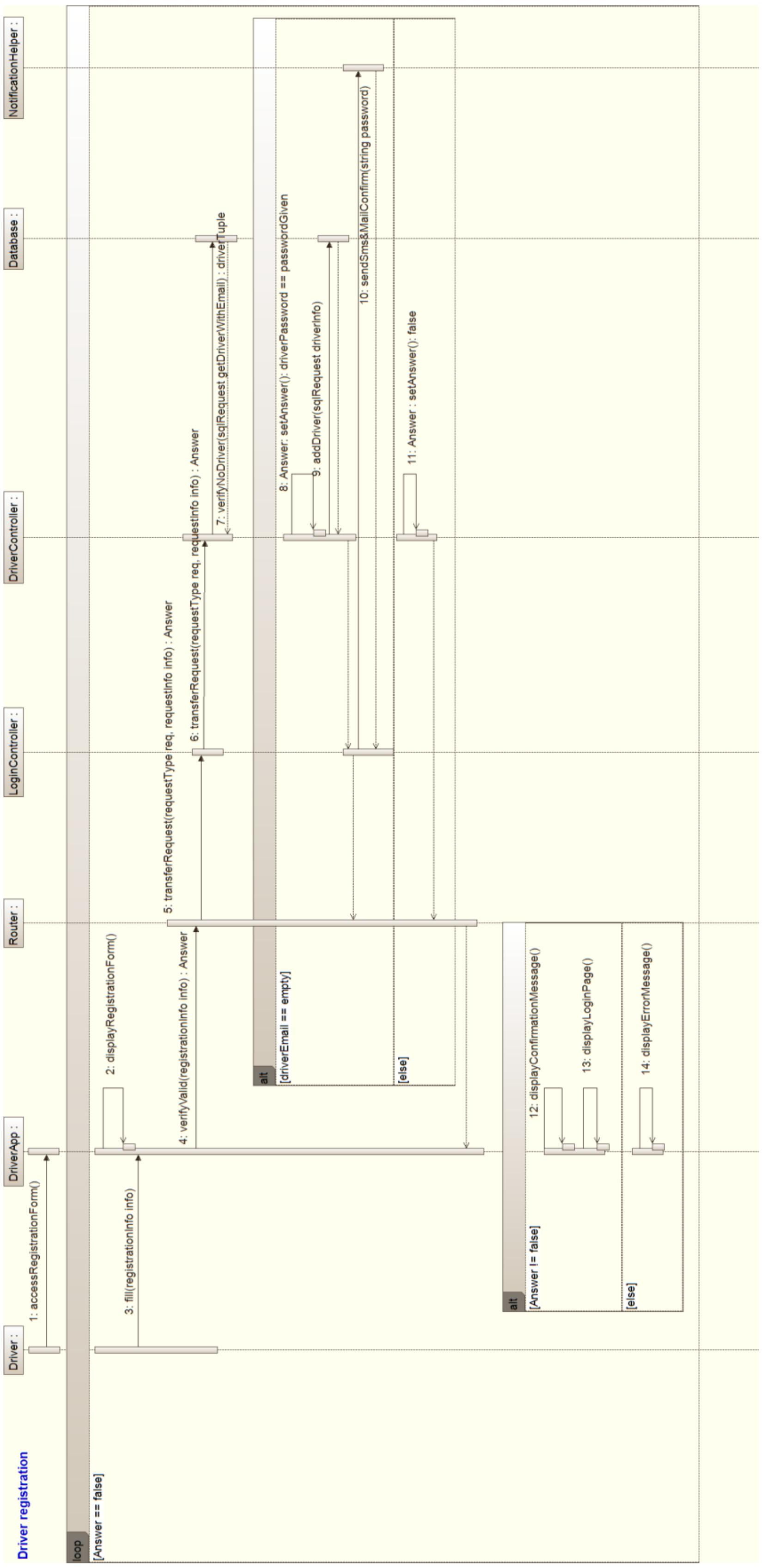
Driver registration
Driver :
DriverApp :
Router :
LoginController :
DriverController :
Database :
NotificationHelper :
loop
[Answer == false]
1: accessRegistrationForm()
2: displayRegistrationForm()
3: fill(registrationInfo info)
4: verifyValid(registrationInfo info) : Answer
5: transferRequest(requestType req, requestInfo info) : Answer
6: transferRequest(requestType req, requestInfo info) : Answer
7: verifyNoDriver(sqlRequest getDriverWithEmail) : driverTuple
alt
[driverEmail == empty]
8: Answer: setAnswer(): driverPassword == passwordGiven
9: addDriver(sqlRequest driverInfo)
10: sendSms&MailConfirm(string password)
[else]
11: Answer : setAnswer(): false
alt
[Answer != false]
12: displayConfirmationMessage()
13: displayLoginPage()
[else]
14: displayErrorMessage()

Figure 5: Sequence Diagram - Driver Registration

This sequence diagram explains how the components interact during a driver's registration.

The driver has to fill the registration form on his mobile application. The registration request is sent with this data as parameters to the system. The router transfers the request to the LoginController which transfers it to the DriverController which verifies on the database if the driver's email doesn't exist yet. If yes, the driver is not registered yet, so the DriverController add him in the database and the LoginController sends a request to the NotificationHelper. It sends a sms and an email with a password to the driver. Finally the LoginController returns the result to the DriverApplication which displays the appropriate message.

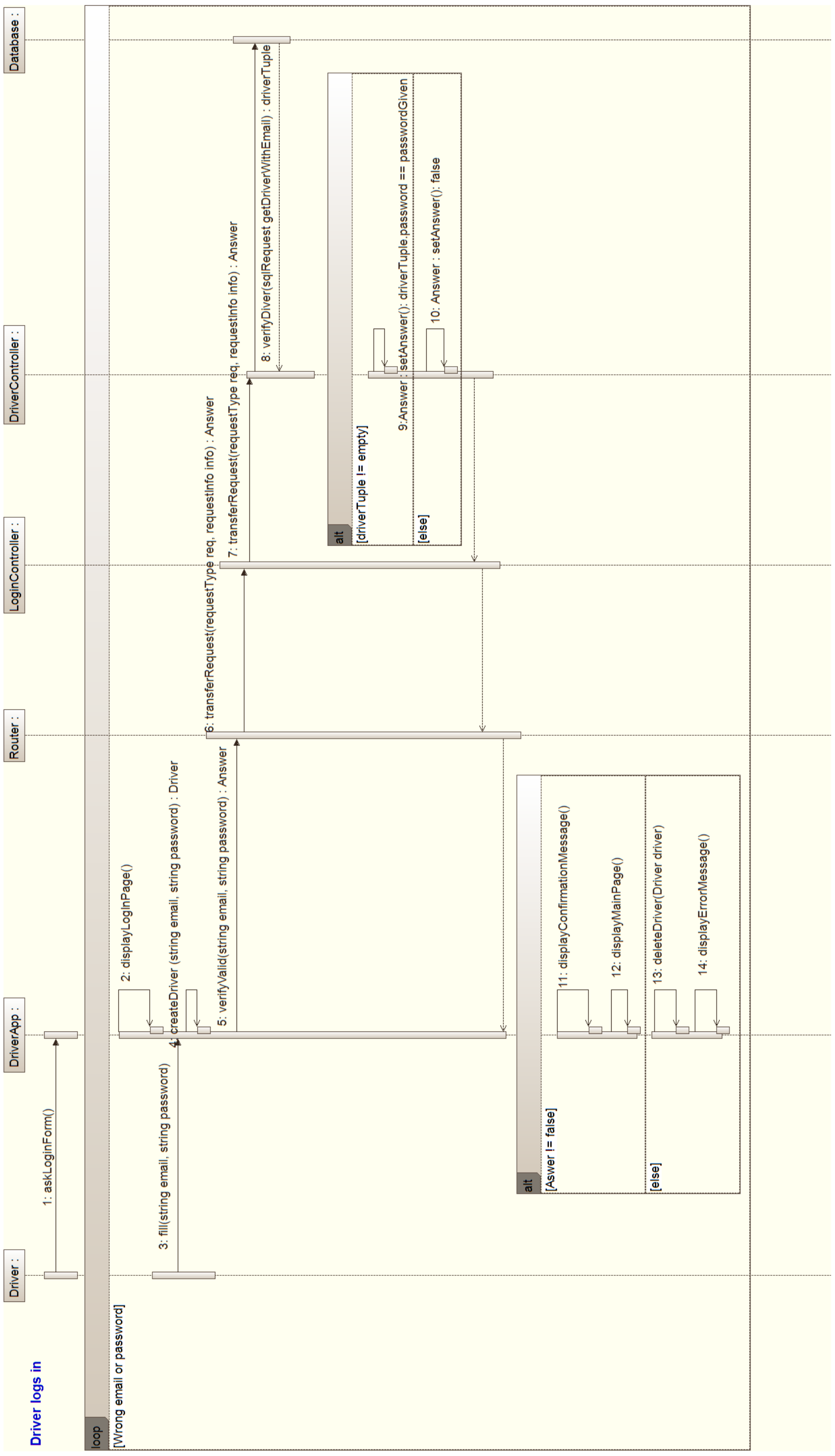
Driver logs in
Driver :
DriverApp :
Router :
LoginController :
DriverController :
Database :
loop
[Wrong email or password]
1: askLoginForm()
2: displayLogInPage()
3: fill(string email, string password)
4: createDriver (string email, string password) : Driver
5: verifyValid(string email, string password) : Answer
6: transferRequest(requestType req, requestInfo info) : Answer
7: transferRequest(requestType req, requestInfo info) : Answer
8: verifyDiver(sqlRequest getDriverWithEmail) : driverTuple
alt
[driverTuple != empty]
9:Answer : setAnswer(): driverTuple.password == passwordGiven
[else]
10: Answer : setAnswer(): false
alt
[Aswer != false]
11: displayConfirmationMessage()
12: displayMainPage()
[else]
13: deleteDriver(Driver driver)
14: displayErrorMessage()

Figure 6: Sequence Diagram - Driver Logs In

This sequence diagram presents the login operation.

The driver has to fill his email and password on his mobile application. The login request is then sent to the system with this two data in parameters. The router transfers the request to the LoginController which transfers it to the DriverController which verifies on the database if the driver exists (with his email). And then if it's true, it checks if the password is correct. Then the LoginController returns the result to the DriverApplication which displays the appropriate screen.

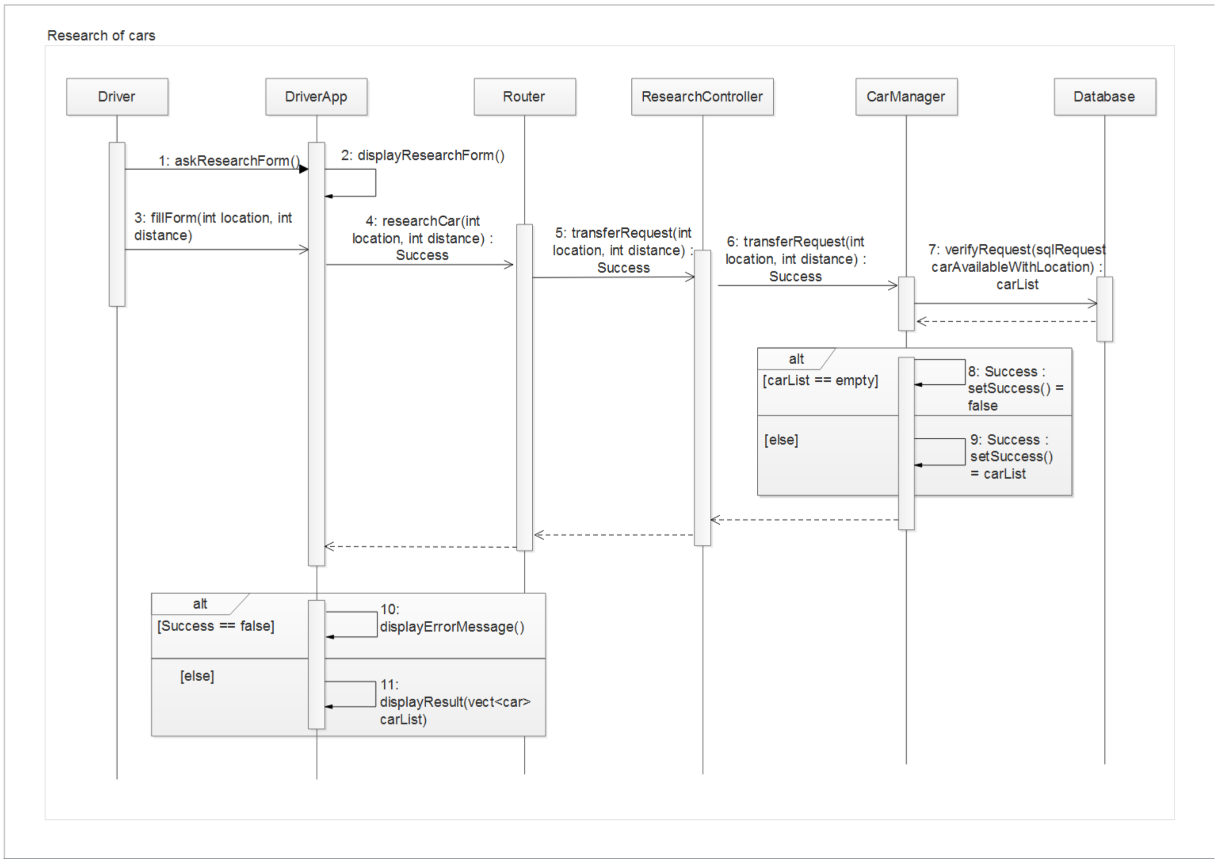
Research of cars
Driver
DriverApp
Router
ResearchController
CarManager
Database
1: askResearchForm()
2: displayResearchForm()
3: fillForm(int location, int distance)
4: researchCar(int location, int distance) : Success
5: transferRequest(int location, int distance) : Success
6: transferRequest(int location, int distance) : Success
7: verifyRequest(sqlRequest carAvailableWithLocation) : carList
alt
[carList == empty]
8: Success : setSuccess() = false
[else]
9: Success : setSuccess() = carList
alt
[Success == false]
10: displayErrorMessage()
[else]
11: displayResult(vect<car> carList)

Figure 7: Sequence Diagram - Car Research

This sequence diagram explains the running of the research.

The driver has to fill the location and distance he wants to find a car, on his mobile application. The research request is then sent to the system with this two data in parameters. The router transfers the request to the ResearchController which transfers the request to the CarManager. This one verifies on the database if there is some car available around the location according to the distance. Then the CarManager returns the result (false if the list of car is empty, the list otherwise) to the DriverApplication which displays the appropriate screen.

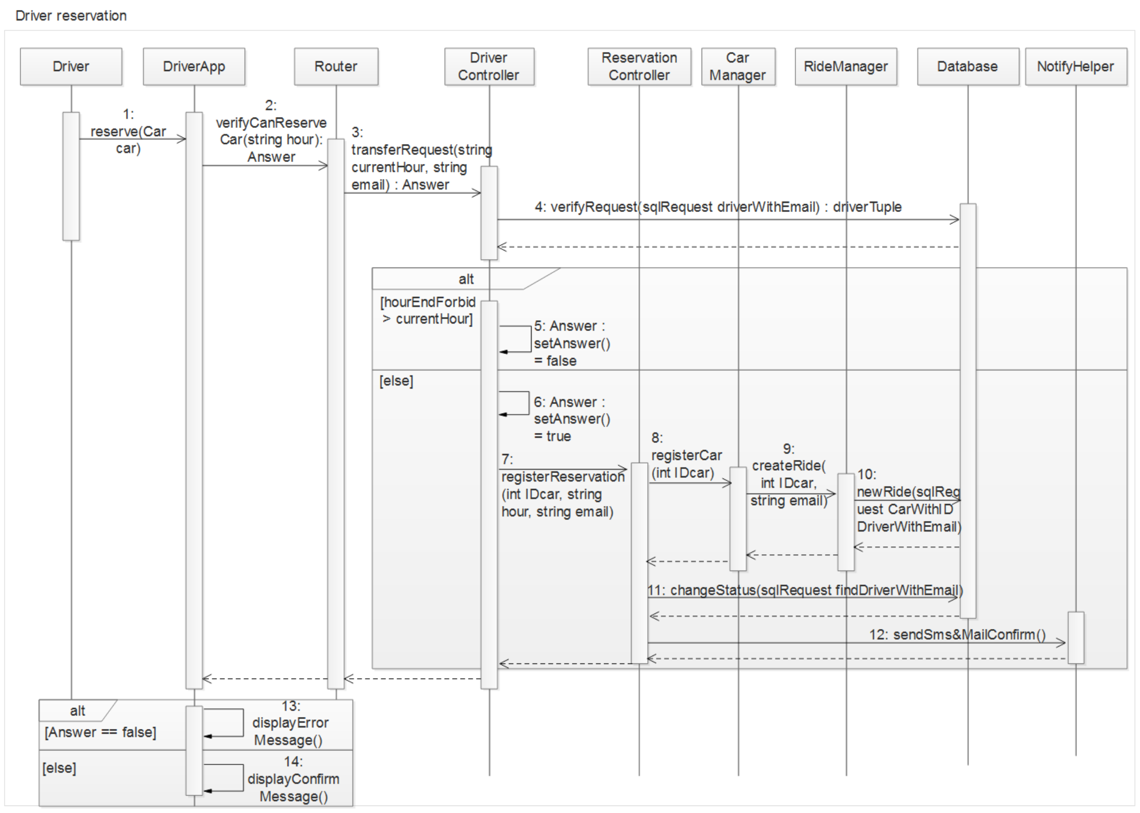
Driver reservation
Driver
DriverApp
Router
Driver Controller
Reservation Controller
Car Manager
RideManager
Database
NotifyHelper
1: reserve(Car car)
2: verifyCanReserveCar(string hour): Answer
3: transferRequest(string currentHour, string email) : Answer
4: verifyRequest(sqlRequest driverWithEmail) : driverTuple
alt
[hourEndForbid > currentHour]
5: Answer : setAnswer() = false
[else]
6: Answer : setAnswer() = true
7: registerReservation(int IDcar, string hour, string email)
8: registerCar(int IDcar)
9: createRide(int IDcar, string email)
10: newRide(sqlRequest CarWithIDDriverWithEmail)
11: changeStatus(sqlRequest findDriverWithEmail)
12: sendSms&MailConfirm()
alt
[Answer == false]
13: displayErrorMessage()
[else]
14: displayConfirmMessage()

Figure 8: Sequence Diagram - Car Reservation

This sequence diagram presents the reservation operation.

The driver chooses a car and reserves it on his mobile application. The reservation request is then sent to the system with the current hour in parameter. The router transfers the request to the DriverController which verifies on the database if the driver is not forbid to reserve a car. If true, the request is transferred to the ReservationController which transfers the request to the CarManager. This one sends a create ride's request to the RideManager with the ID of the car and the email of the driver in parameters. Then the RideManager creates a new ride in the database. These actions done, the ReservationController updates the status of the driver in the database and asks the NotificationHelper to send a sms and an email of confirmation. Finally the ReservationController returns the result to the DriverApplication which displays the appropriate screen.

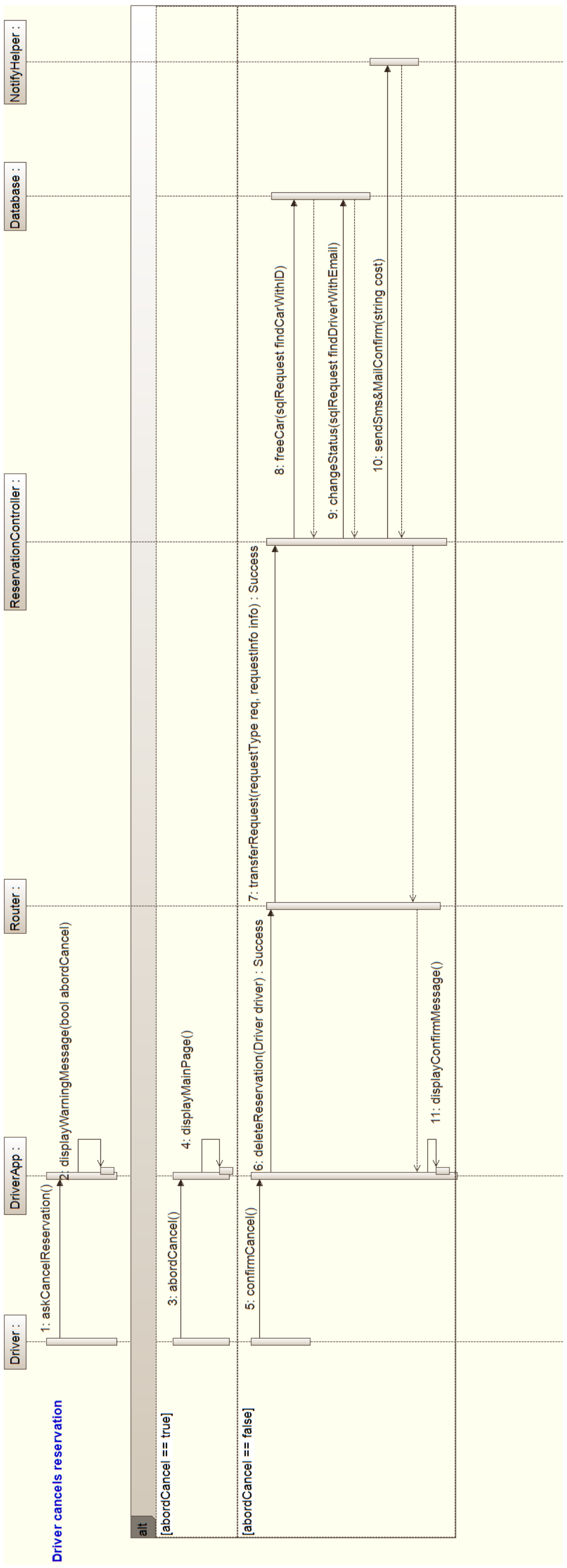
Driver cancels reservation
Driver :
DriverApp :
Router :
ReservationController :
Database :
NotifyHelper :
alt
[abordCancel == true]
[abordCancel == false]
1: askCancelReservation()
2: displayWarningMessage(bool abordCancel)
3: abordCancel()
4: displayMainPage()
5: confirmCancel()
6: deleteReservation(Driver driver) : Success
7: transferRequest(requestType req, requestInfo info) : Success
8: freeCar(sqlRequest findCarWithID)
9: changeStatus(sqlRequest findDriverWithEmail)
10: sendSms&MailConfirm(string cost)
11: displayConfirmMessage()

Figure 9: Sequence Diagram - Reservation Cancellation

This sequence diagram explains the running of the reservation's cancel.

The driver has to confirm his cancel on his mobile application. If he abords the cancel the DriverApplication displays the main screen, if not the cancel request is sent to the system. The router transfers the request to the ReservationController which transfers the request to the CarManager. This one updates the car status in the database. Then the ReservationController transfers the request to the DriverController which updates the driver status in the database and asks the NotificationHelper to send a sms and an email of confirmation. Finally the ReservationController returns the result to the DriverApplication which displays the confirmation screen.

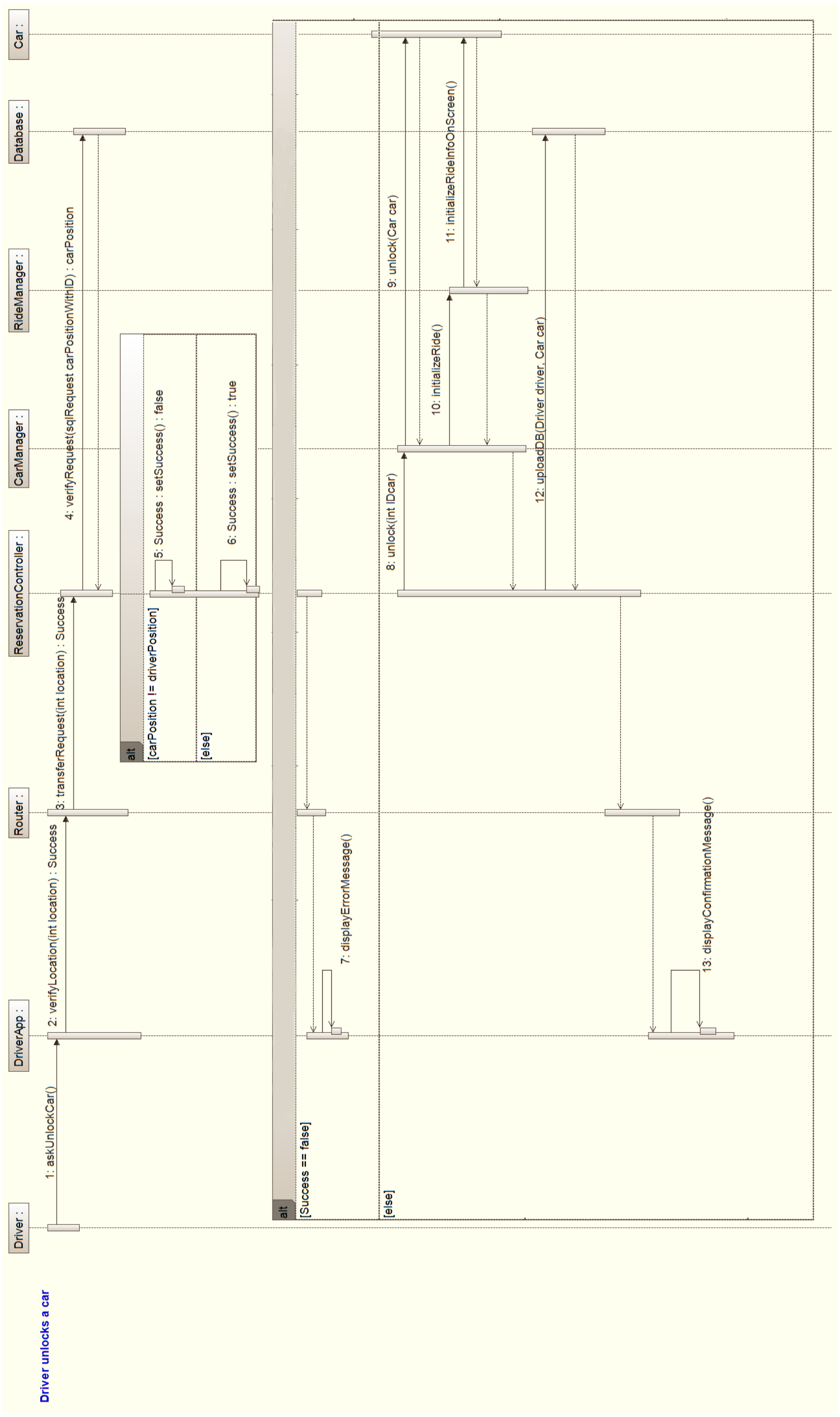
Driver unlocks a car
Driver :
DriverApp :
Router :
ReservationController :
CarManager :
RideManager :
Database :
Car :
1: askUnlockCar()
2: verifyLocation(int location) : Success
3: transferRequest(int location) : Success
4: verifyRequest(sqlRequest carPositionWithID) : carPosition
alt
[carPosition != driverPosition]
5: Success : setSuccess() : false
[else]
6: Success : setSuccess() : true
alt
[Success == false]
7: displayErrorMessage()
[else]
8: unlock(int IDcar)
9: unlock(Car car)
10: initializeRide()
11: initializeRideInfoOnScreen()
12: uploadDB(Driver driver, Car car)
13: displayConfirmationMessage()

Figure 11: Sequence Diagram - Car Unlocking

This sequence diagram presents the unlock operation.

The driver has to choose the unlock option on his mobile application. The unlock request is then sent to the system with the driver's location in parameter. The router transfers the request to the ReservationController which transfers the request to the CarManager which transfers to the RideManager. This one verifies in the database if the location of the reserved car is the same as the driver's location. If it's true, the RideManager returns the result to the CarManager which will sends the unlock request to the car and the start ride request to the RideManager. This one will then sends the initialize ride request to the car (initializes the navigation screen). Then the ReservationController will update the database with the new status of the car and the car. Finally the ReservationController returns the result to the DriverApplication which displays the confirmation message or the error message (if the result is false).

2.6. Component interfaces

All the transactions between the mobile app and the business logic server will start with an identity verification to ensure the origin of every message. Once the user is logged into the system, an authentication key is attributed to him and used by every forthcoming transaction. Then, the server provides a Message Management component in charge of redirecting all the authenticated messages to the corresponding components of the system. If needed, the server can retrieve data from the Database Server. The DBMS contains an active database in which the data are organized with a B+ Tree to ensure short response time.

2.7. Selected architectural styles and patterns

2.7.1. Overall Architecture

The application will be built on a three-tier architecture:

- Database (Db)
- Application Logic Layer (ALL)
- Client (interfacing with the ALL)

2.7.2. Protocols

Our tiers are connected through network and exchange data with the following protocols.

PDO: PHP Data Objects used by the Db to communicate with the ALL.

PDO is a common protocol used to communicate between an ALL and a Db. I also fit well with our project. As a common protocol it will be easier to implement and keep updated (example, community, support etc...). And it also works with most of database system.

An API with JSON used by clients (mobile apps for user and hidden client in the car) to interact with the ALL through http methods. API calls that need authentication are required to authenticate via HTTP basic authentication for each request. Exchanged data will be secured using SSL.

Here are few methods that we may use:

- api/user/info [auth]
    - GET: get user info
    - POST/PUT: update/set user data
- api/user/request
    - POST: create a new request for car
- api/user/reservation

  - POST: create a new reservation
- api/car/ride
  - POST: create a new ride
- api/car/park
  - POST: car properly parked

2.7.3. Design patterns

The MVC pattern will fit well with our three tier architecture. As a common design pattern and known by most programmers, it will provide simplicity and efficiency: the component will be logically grouped and divided; improving task repartition and providing efficient work process.

The API is based on a REST design pattern as you can guess from the http methods.

As the http and JSON protocol are well known, we can use the many, already available, tools to work with it and there is no need to redefine such protocols (http) like in a Service Oriented Architecture. Moreover, a REST architecture will improve the creation of many services like security, monitoring, service quality and many other non-functional part of web services.

3. Algorithm design

3.1. Part 1

```
Reservation(Driver, Car)
Pre-condition: Car->status == AVAILABLE, Driver->status == FREE
{
        As  transaction process
        (do everything single action or none,
        return True if they are done else False ):
        Process =
        {
                //Map of association between User and Car
                //It work in both direction and each association have a unique ID
                driver-Car-BidirectionalMap.add(Driver,Car)
                Driver->status = WAITTOPICKUP
                Car->status = UNAVAILABLE
        }
        if Process
                //Get the id of the association
                reservationId = driver-Car-BidirectionalMap.getId(Driver,Car)
                //Send a message on Driver's phone to confirm the reservation
                Driver.sendConfirmedReservation()
                //Add a one hour timer to the link between the driver and the car
                ReservationId.oneHourClock.start()
        else
                //Send a message on Driver's phone to signal an error
                Driver.sendErrorReservation()
}
```

3.2. Part 2

```
Open(User)
Assumption: driver-Car-BidirectionalMap is global
{
        userPosition = User.getPosition();
        car = driver-Car-BidirectionalMap.getCar(User);
        carPosition = car.getPosition();
        if distance(userPosition,carPosition) < 1
        {
                //Send a signal to the car to open it
                //We suppose it won't fail
                car.sendOpenSignal();
                //wait return True if the Driver open the car before the timeout
                ack = wait(car.satus == OPENED, car.timeoutOpening)
                if ack == False
                {
                        //Send a message to the user
                        //asking him to redo the open procedure
                        //and open the car before the timeout
                        user.sendTooLateTryAgain()
                }
        }
        else
        {
                //Send a message: he is too far form the car
                user.sendToFarMessage();
        }
}
```

4. User interface design

4.1. Mockups

Mockups of the user interface can be found on the RASD, in section 3.1.1.

4.2. BCE diagram

In this Business Controller Entity are presented the interactions due to the user actions on his mobile application. For the sake of clarity the NotificationHelper is not part of the diagram, the ReservationController and the LoginController have the operation "sendNotification()" which call the NotificationHelper.

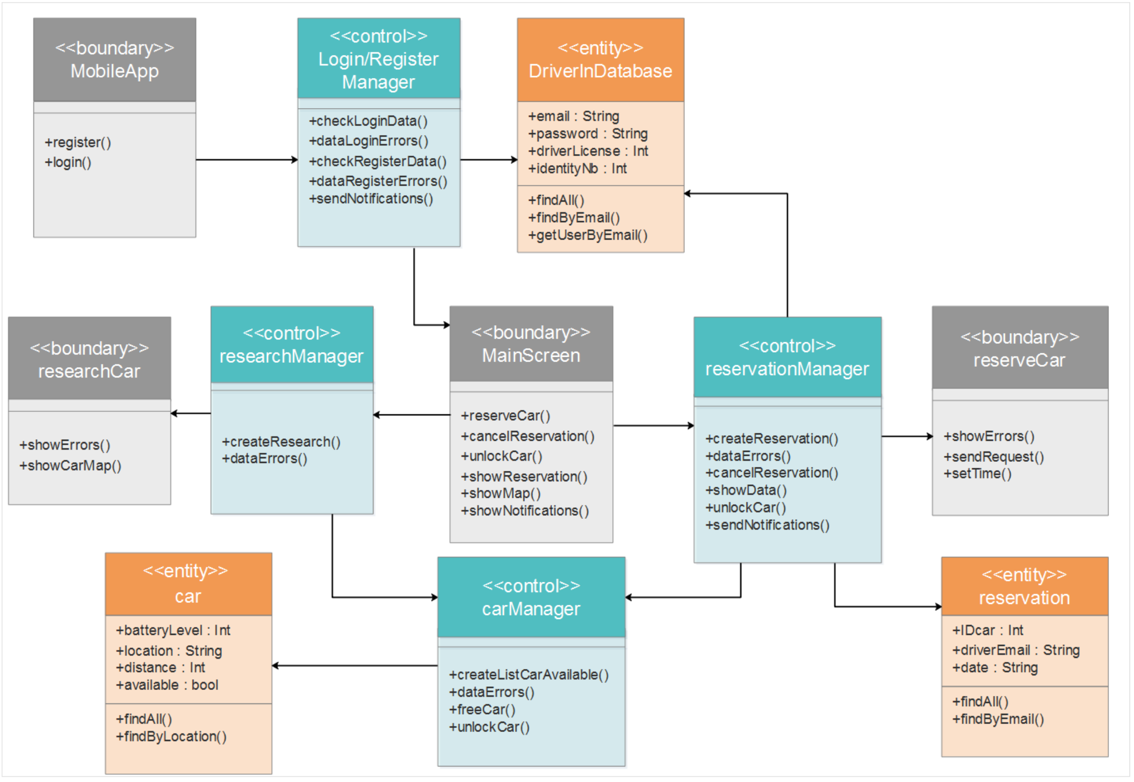

Figure 12: BCE Diagram

5. Requirements traceability

The design of the project was conceived to cover efficiently the requirements and goals defined in the RASD. In this part are presented the design components, defined in this document, associated with the requirements it fulfils.

G1: Allows driver to register the system by providing credential and payment details and provides him a password by email.

- The DriverApplication component
- The Router
- The LoginController
- The DriverController and its interface with the database
- The NotificationHelper
- The MailGateway

G2: Allows driver to log into the system with provided password

- The DriverApplication component
- The Router
- The LoginController
- The DriverController and its interface with the database

G3: Driver can locate available cars within a certain distance around him or an address

- The DriverApplication component
- The Router
- The ResearchController
- The CarManager and its interface with the database

G4: Driver can reserve a car for up to 1 hour before pick it up and should receive notifications of the confirmation.

- The DriverApplication component
- The Router
- The DriverController and its interface with the database
- The ReservationController and its interface with the database
- The CarManager
- The RideManager and its interface with the database
- The NotificationHelper
- The SMSGateway
- The PushGateway

G5: A reserved car but not picked up within one hour generate 1€ fee for the driver and forbidden him to reserve another one for 3 hours. The driver should receive an email of this ban.

- The ReservationController and its interface with the database
- The NotificationHelper
- The MailGateway

G6: Driver can cancel his reservation within the hour after he reserves it and should receive notifications of the confirmation.

- The DriverApplication component
- The Router
- The ReservationController
- The NotificationHelper
- The PushGateway

G7: A driver close enough to a car reserved by him must be able to open it

- The DriverApplication component
- The Router
- The ReservationController and its interface with the database
- The CarManager and its interface with the cars
- The RideManager and its interface with the database

G8: The system starts charging once engine ignite

- The RideManager

G10: The system stop charging once the driver leaves the car parked in a safe area

- The RideManager

G11: 10% discount on last ride if the car detects at least two passengers

- The RideManager

G12: The system apply 20% discount on the last ride if the car is left with more than 50% battery (over full battery)

- The RideManager

G13: If the driver parks the car in a power grid station, where the car can be charged, and takes care to plug the car. The system applies 30% discount on the last ride.

- The RideManager

G14: If the car is left with less than 20% battery or at more than 3km from the nearest power grid station, the system charges 30% more on the last ride.

- The RideManager

G15: The driver can enable the saving money option and so by giving his final destination to the system, this one is able to give him a station as destination and if the driver leaves the car and plugs it at this place he will get a special discount.

- The RideManager
- The CarManager and its interface with the database

6. Hours of work

6.1. Vianney Payelle

- 03/12: 1h
- 06/12: 1h
- 09/12: 1h
- 10/12: 2h
- 11/12: 3h

6.2. Rémi Rigal

- 28/11: 2h
- 04/12: 1h
- 05/12: 1h
- 10/12: 1h
- 11/12: 5h

6.3. Noëlie Ramuzat

- 26/11: 1h
- 28/11: 2h
- 04/12: 6h
- 05/12: 3h
- 10/12: 3h
- 11/12: 5h