POLITECNICO
MILANO 1863

Design Document

Lo Bianco Riccardo - Manzoni Mirco - Mascellaro Giuseppe

January 23, 2017
v1.1

Contents

1 Introduction **1**
- 1.1 Purpose . . . 1
- 1.2 Scope . . . 1
- 1.3 Definitions, acronyms, abbreviations . . . 1
  - 1.3.1 Definitions . . . 1
  - 1.3.2 Acronyms . . . 1
- 1.4 Reference documents . . . 2
- 1.5 Document structure . . . 2

2 Architectural design **3**
- 2.1 Overview . . . 3
- 2.2 Component view . . . 5
  - 2.2.1 High level view . . . 5
  - 2.2.2 Client view . . . 6
- 2.3 Server view . . . 7
- 2.4 Deployment view . . . 8
  - 2.4.1 Physical view . . . 8
  - 2.4.2 Application end to end view . . . 10
- 2.5 Runtime view . . . 13
  - 2.5.1 Sign up . . . 14
  - 2.5.2 Start ride . . . 15
  - 2.5.3 Make reservation . . . 17
  - 2.5.4 End ride . . . 19
  - 2.5.5 Look for a car . . . 20
  - 2.5.6 Navigate to a car . . . 21
  - 2.5.7 Use the money saving option . . . 22
  - 2.5.8 Operator log in . . . 23
  - 2.5.9 Solve technical issue . . . 24
  - 2.5.10 Consult a ride's information . . . 29
- 2.6 Component interfaces . . . 30
  - 2.6.1 Server . . . 30
  - 2.6.2 Car . . . 32
  - 2.6.3 User Mobile Application . . . 33
  - 2.6.4 Operator Application . . . 33
- 2.7 Architerctural styles and patterns . . . 34
- 2.8 Other design decisions . . . 36

3 Algorithm design **38**
- 3.1 Check car distribution algorithm: checkCarDistribution() . . . 38
- 3.2 Money saving option algorithm: moneySavingOption(float K) . . . 39

4 Interface design **41**
- 4.1 User interface design . . . 42
- 4.2 Operator interface design . . . 43
- 4.3 Car screen interface design . . . 44

5 Requirements traceability **45**

6 Appendix **48**
6.1 Used tools 48
6.2 Hours of work 48
6.3 Changelog 48

1 Introduction

1.1 Purpose

This document represents the continuation of the RASD document for the PowerEnJoy application previosly presented. Through the discussion we will explain in depth the architecture of the the system to be deployed, then we will focus on components, algorithms developed to implement the described architecture and user interface.

1.2 Scope

The system must provide the following macro functionalities, which can be mapped from the goals presented in the RASD document

- **User functionalities:** PowerEnjoy will make available for the users (both logged and unlogged) the full set of functionalities described in the RASD.
- **Operator functionalities:** PowerEnjoy will make available for the operators the full set of functionalities described in the RASD.
- **Car functionalities:** PowerEnjoy will manage the full set of sensors installed on the cars, processing the signals and storing the data in the database, as well as processing the input received through the car screen and those dictated by the server.

1.3 Definitions, acronyms, abbreviations

1.3.1 Definitions

- **Unreliable user:** a user that inserted a wrong password through the car screen five times in a row.
- **Unreliable ride:** a ride that is not finished after eight hours from its beginning.
- **Emergency module:** software element that encourages the driver to end the current ride as soon as possible if a technical issue occurs.

1.3.2 Acronyms

- **R:** Requirement
- **G:** Goal
- **JEE:** Java Enterprise Edition
- **API:** Application Programming Interface
- **DBMS:** Database Management System
- **DMZ:** Demilitarized Zone
- **VPN:** Virtual Private Network
- **NAT:** Network Address Translation

- **PPP:** Point to Point Protocol
- **DHCP:** Dynamic Host Confiiguration Protocol
- **DLA:** Driving License Authorithy
- **JSP:** JavaServer Pages
- **SOAP:** Simple Object Access Protocol
- **JDBC:** Java Database Connectivity
- **AJP:** Apache JServ Protocol
- **MVC:** Model View Control
- **IaaS:** Infrastructure as a Service
- **DLA:** Driving License Authority

1.4 Reference documents

1. **Analysis document:** the RASD we previously produced, which is a description of the problem and of the solution abstracted from the physical implementation.
2. **IEEE Standard for Information Technology - Systems Design - Software Design Descriptions:** the standard for production of documentation regarding IT systems deployment.

1.5 Document structure

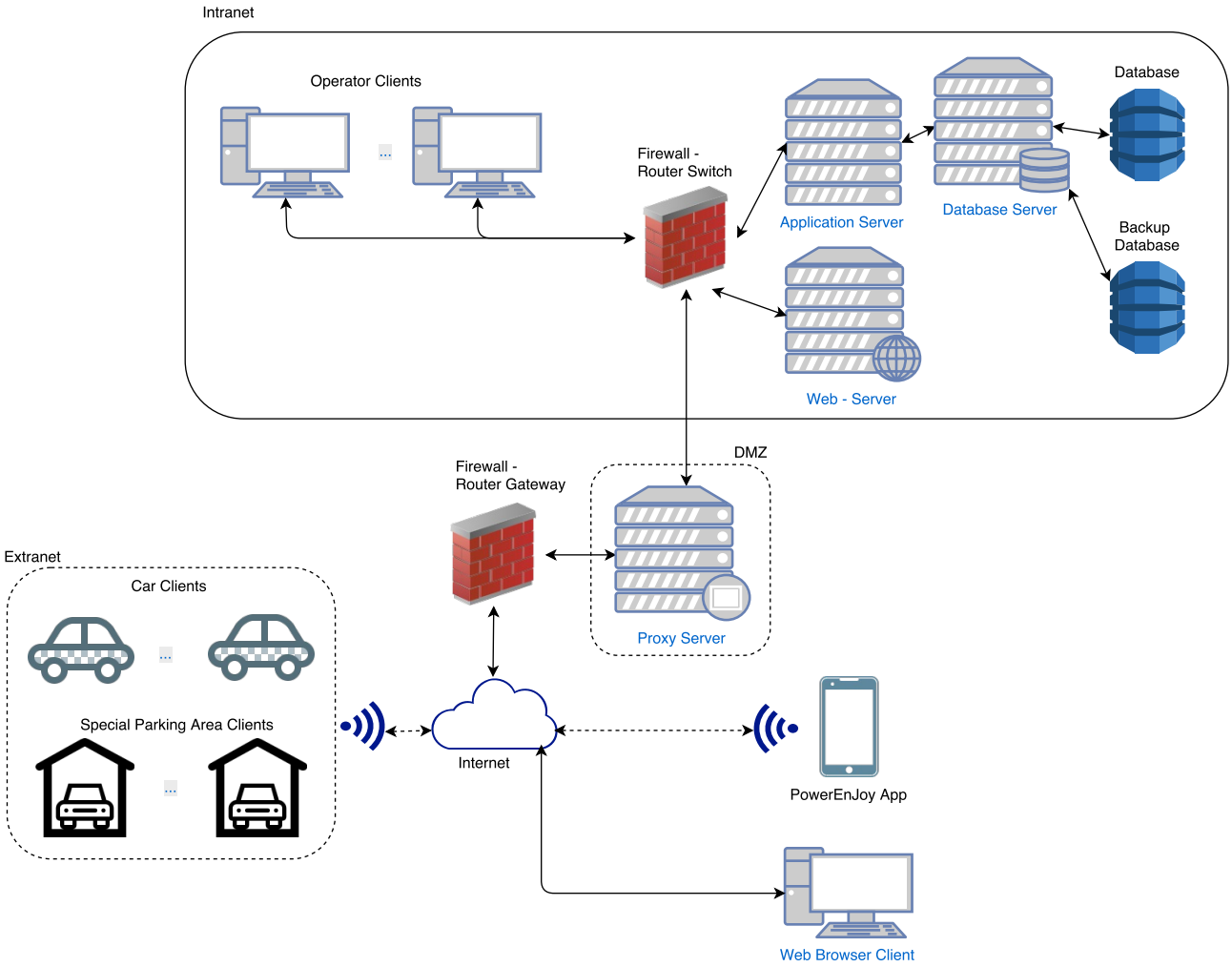
Along the document we will discuss the following arguments:

- **Architectural design:** the complete structure of the system to be deployed, divided into component view, deployment view and runtime view. As a compendium to these three major paragraph, a detailed description of the component interfaces is included in this chapter as well as a discussion regarding the styles and patterns used to obtain a coherent architecture and the design decisions that the team had to face.
- **Algorithm design:** the implementation (in object-oriented pseudo code) of two of the most significant algorithms to be included in the system's code.
- **Interface design:** a refinement of what was introduced in the RASD for what concerns the user interface we want to obtain. In the previous document, the argument was developed through the use of simple mockups of the most important pages of the system, while in this document we exploited class diagrams to obtain a more complete and formal description.
- **Requirements traceability:** the complete mapping of the goals/requirements pointed out in the RASD with the components designed to fullfill them.

2 Architectural design

2.1 Overview

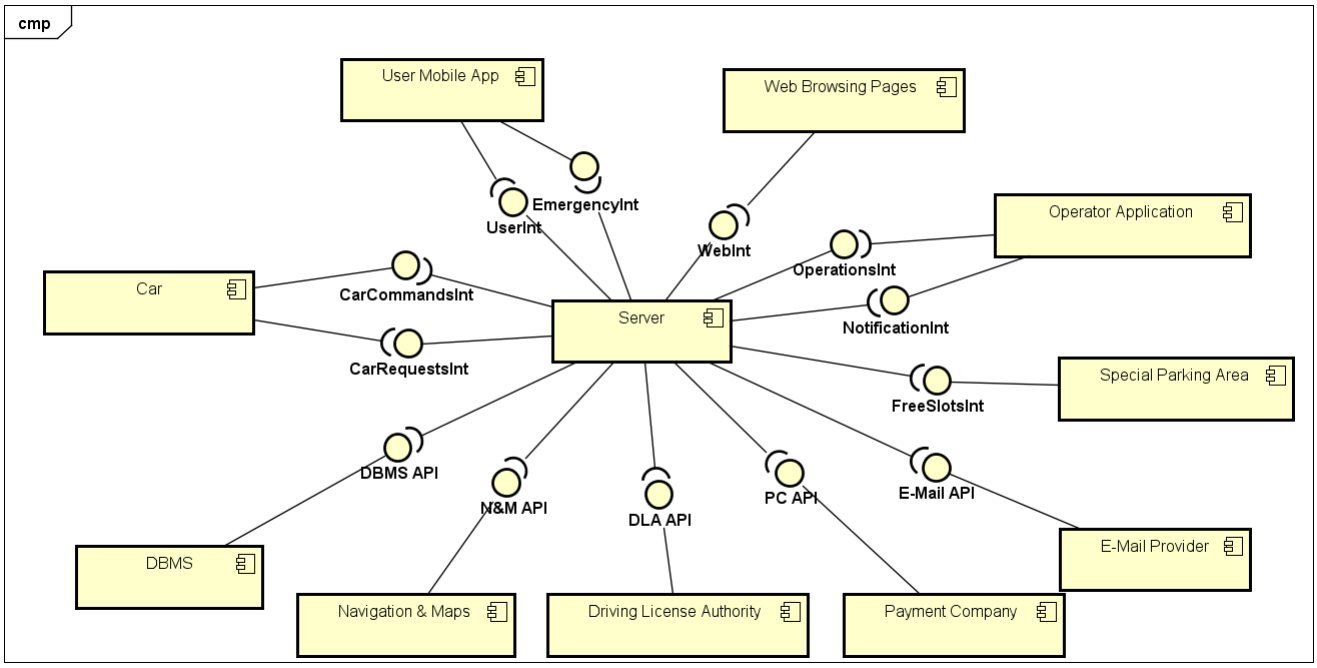
After a deep analysis of requirements, we decided to use a Client/Server web infrastructure with only one part which interacts following the Publisher/Subscriber pattern. These two architectural styles cooperate in a hybrid solution of the system.
As it can be seen in the graphical overview below we have the usual client and server distinction. In particular, the identified clients are: PowerEnJoy App, Web Browser Client, Car Clients, Special Parking Area Clients, and Operator Clients.
Car Clients and Special Parking Area Clients are anyway part of system from a logical perspective. Indeed, they are physically distant from the central servers. In order to keep them logically inside the network, they are connected through a VPN.
The clients will be developed following two different strategies: car clients will be thick clients, since they need to manage a consistent part of the elaboration of the data collected by the sensors, while both user clients and operator clients will be thin client, relying on the computational power of the servers.
The server side is composed of one or many Proxy Servers which act as front end servers. Their main purpose is to accept or discard client requests and to dispatch them through the right server belonging to the intranet. To reach the Proxy Server, a client must know the static well known IP address or the symbolic name. In order to make this possible, this server has to be exposed to the internet in a DMZ, considering that exposing a Server to the Internet is an evident security leak. To avoid any risk, a firewall is interposed between the Internet and the Proxy Server. In addition, the Proxy Server is used as a gateway to manage gateway requests to either external information systems or API managers. Furthermore, the Proxy Server is used for sending either responses to clients' requests or direct server commands to clients.
Another firewall acts as network flow filter to access the intranet and this one is more restrictive than the previous one. The intranet is composed by the Operator Clients on one side and central servers on the other.
Main servers are: Web Server, Application Server and Database Server.
The Web Server deals with the dynamic web pages' generation. As every Web Server it may need the contribution of a business logic layer, furthermore the Web Server may exploit the Application Server to retrieve dynamic content. Web Browser Clients communicate to the Web Server.
The Application Server is the business logic core of the entire system. It is responsible for all clients' requests handling and for commands producer. The Database Server contains the DBMS which deals with the data persistence. It communicates to actual physical data storages. Moreover, it manages a parallel physical data storage which contains backed up data to ensure the absence of data loss. This copy data storage is located in a different area with high security level.
In the image below we represented only one server per type, but in the system to be deployed these single servers will be replaced by server farms that exploits the Proxy Server as load balancer.


Intranet
Operator Clients
Firewall -
Router Switch
Application Server
Database Server
Database
Backup
Database
Web - Server
DMZ
Firewall -
Router Gateway
Proxy Server
Extranet
Car Clients
Special Parking Area Clients
Internet
PowerEnJoy App
Web Browser Client

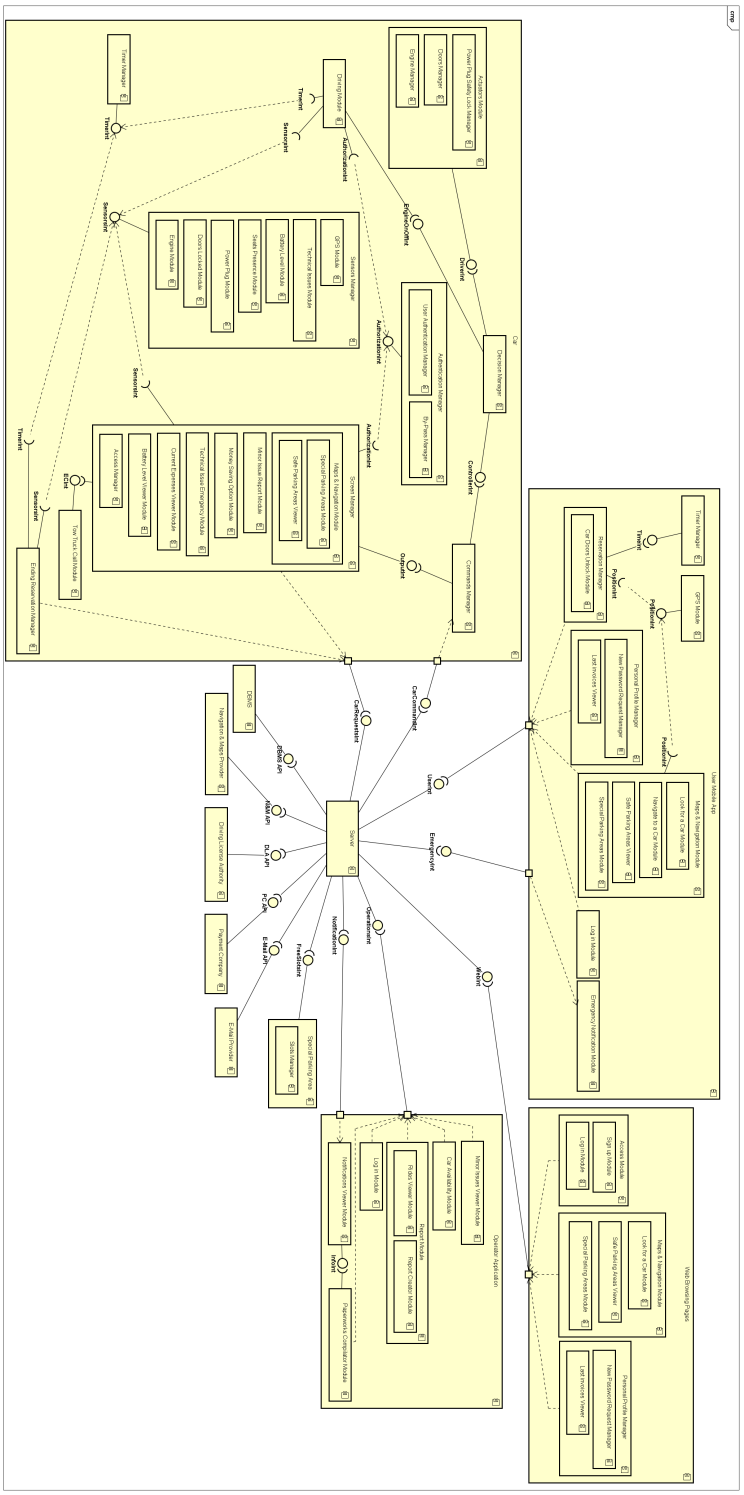
2.2 Component view

The components, which are represented below, were designed to interact following the desing principles of low coupling and high cohesion. While in the first and in the second diagram the inspection strategy adopted was of type *top down*, in the last one we used a hybrid technique which better fitted the high complexity. We decided to show the software components that constitute our system in two different diagrams. In the first one we decided to create a "client view" to show which are the components present in all the clients (website, users app, operators app, car app). In the second one, the "server view", we described which are the components present in the server to implement the business logic. The interactions between server and client are represented by interfaces that we named "external interfaces". We decided to show some sub-components as a grouped "big" component to better contextualize the interested sub-components.

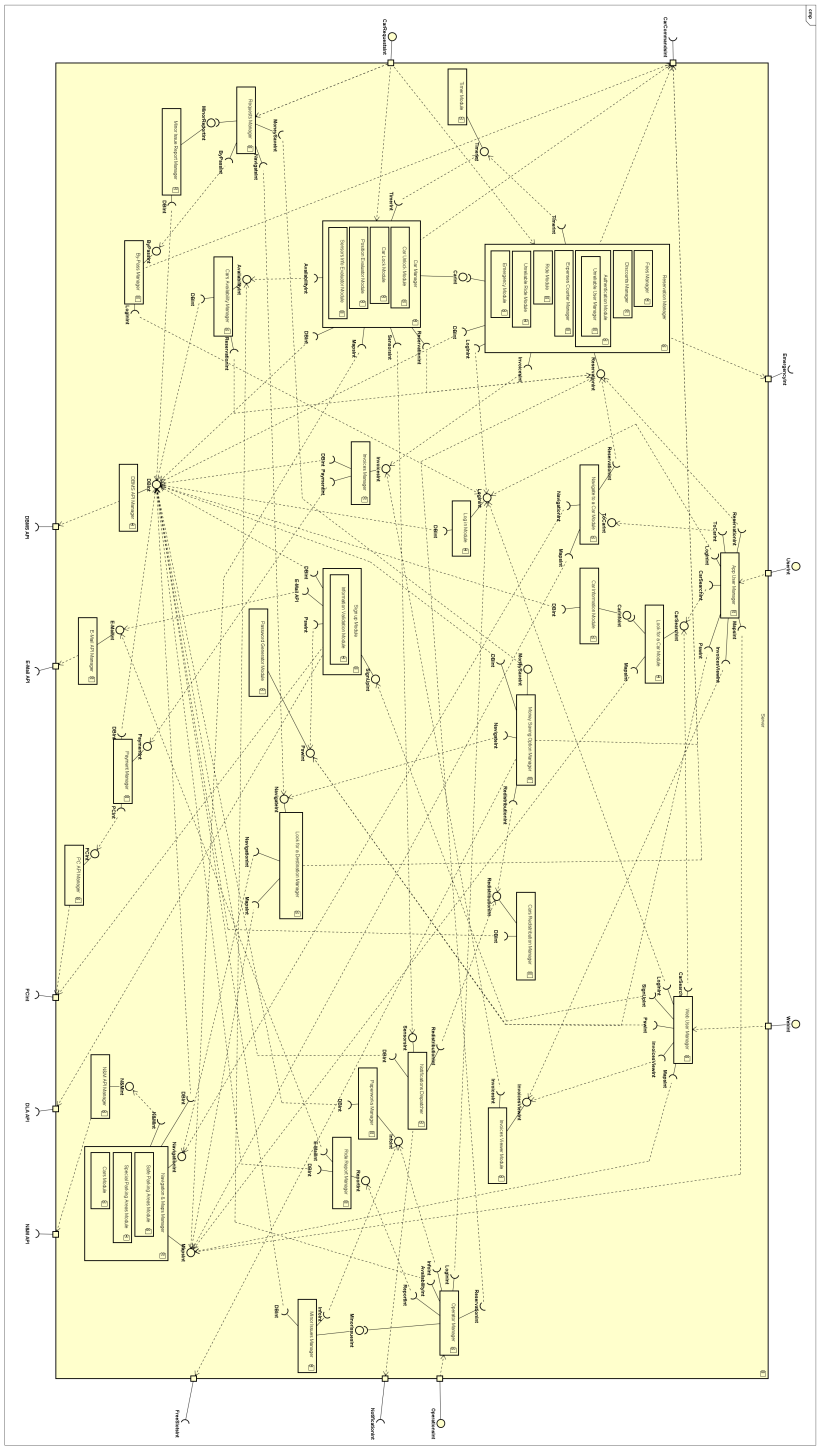
2.2.1 High level view



2.2.2 Client view



2.3 Server view

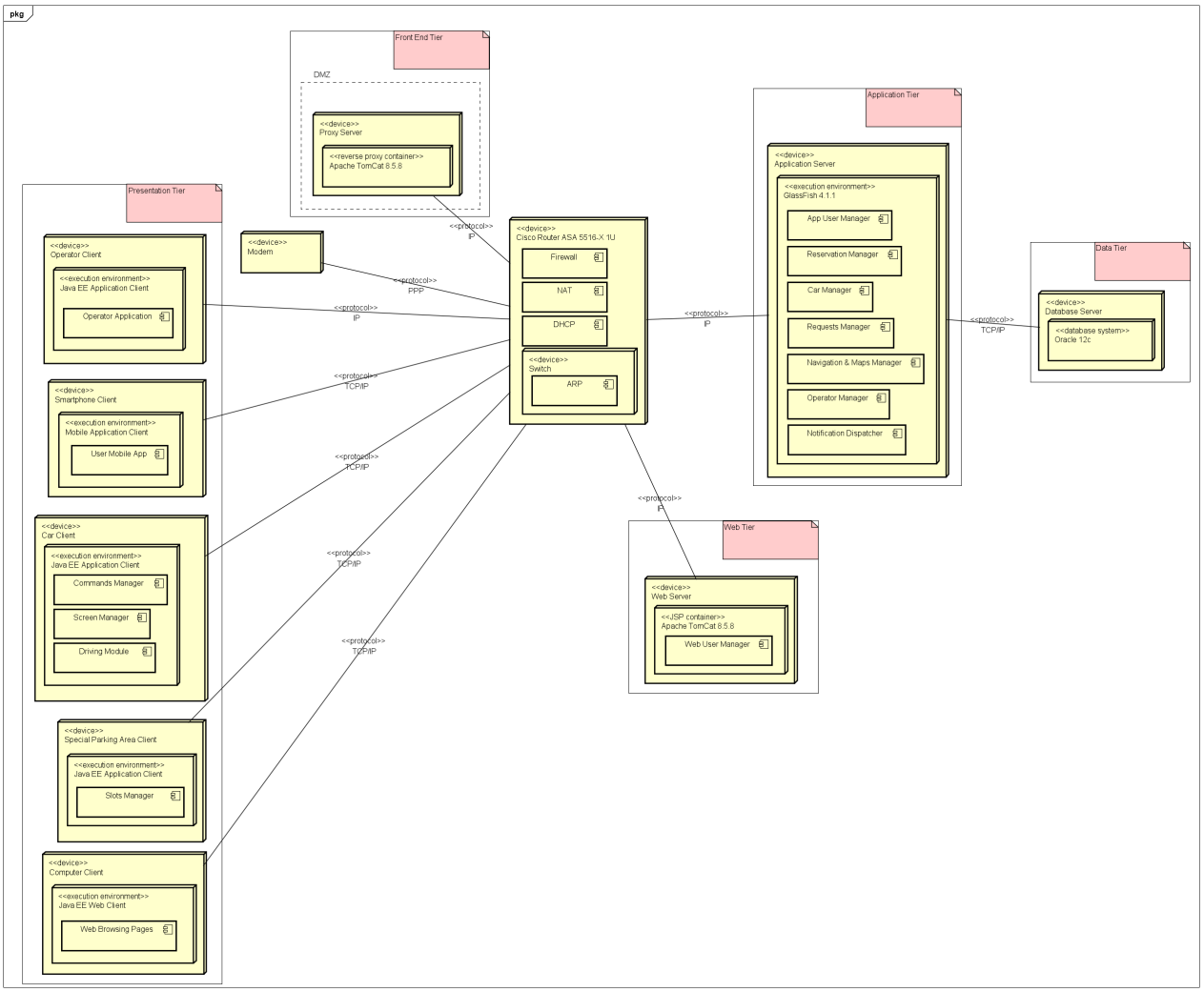


2.4 Deployment view

The deployment view highlights the physical deployment of artifacts on nodes. In our case, we preferred to split the deployment diagram in two main views, one concerning the actual physical connections amongst nodes and the other concerning the logical end to end connection amongst artifacts inside nodes. Only the most relevant software artifacts have been inserted in the deployment diagrams.
Like in every Client/Server architectural style, a tier partitioning through clients and servers and one for each tier there is a correspondent logical layer are present. The tier partitioning is visible in the physical view while the layer one in the logical view.

2.4.1 Physical view

The fundamental element of the physical view is the Cisco Router. This device is connected to a modem for signals modulation and demodulation that uses PPP. Indeed, the router is connected to a network backbone. The router acts as firewall for filtering both front-end network flows and intranet accesses. In addition, it has NAT in order to address requests to the right server inside the intranet. Furthermore, the router exploits DHCP for dynamic assignment of IP addresses to servers inside the intranet.
As stated before, the Proxy Server is directly reachable from web distributed clients due to his exposition on the Internet.
The presentation tier is composed by all clients, but only Operator Clients belong to the intranet. Car Clients and Special Parking Area Clients belong to the extranet (VPN). The router takes care of network partitioning exploiting subnet separation.
Clients can directly communicate to either the Web Server or the Application Server. The Web Server may exploit some of the Application Server functionalities communicating to it internally as well. All these connections are handled by the router.
The Application Tier is connected directly to the Data Tier because it is the only user of the data services. They are connected to plain Ethernet cable. All communications listed below rely on the TCP/IP protocols set.


pkg
Front End Tier
DMZ
<<device>>
Proxy Server
<<reverse proxy container>>
Apache TomCat 8.5.8
Application Tier
<<device>>
Application Server
<<execution environment>>
GlassFish 4.1.1
App User Manager
Reservation Manager
Car Manager
Requests Manager
Navigation & Maps Manager
Operator Manager
Notification Dispatcher
Data Tier
<<device>>
Database Server
<<database system>>
Oracle 12c
Presentation Tier
<<device>>
Operator Client
<<execution environment>>
Java EE Application Client
Operator Application
<<device>>
Modem
<<device>>
Cisco Router ASA 5516-X 1U
Firewall
NAT
DHCP
<<device>>
Switch
ARP
<<device>>
Smartphone Client
<<execution environment>>
Mobile Application Client
User Mobile App
<<device>>
Car Client
<<execution environment>>
Java EE Application Client
Commands Manager
Screen Manager
Driving Module
<<device>>
Special Parking Area Client
<<execution environment>>
Java EE Application Client
Slots Manager
<<device>>
Computer Client
<<execution environment>>
Java EE Web Client
Web Browsing Pages
Web Tier
<<device>>
Web Server
<<JSP container>>
Apache TomCat 8.5.8
Web User Manager
<<protocol>>
IP
<<protocol>>
PPP
<<protocol>>
TCP/IP

2.4.2 Application end to end view

Each node is characterized by its execution environment.
The Proxy Server runs a reverse proxy container. There are several reasons for installing reverse proxy servers:

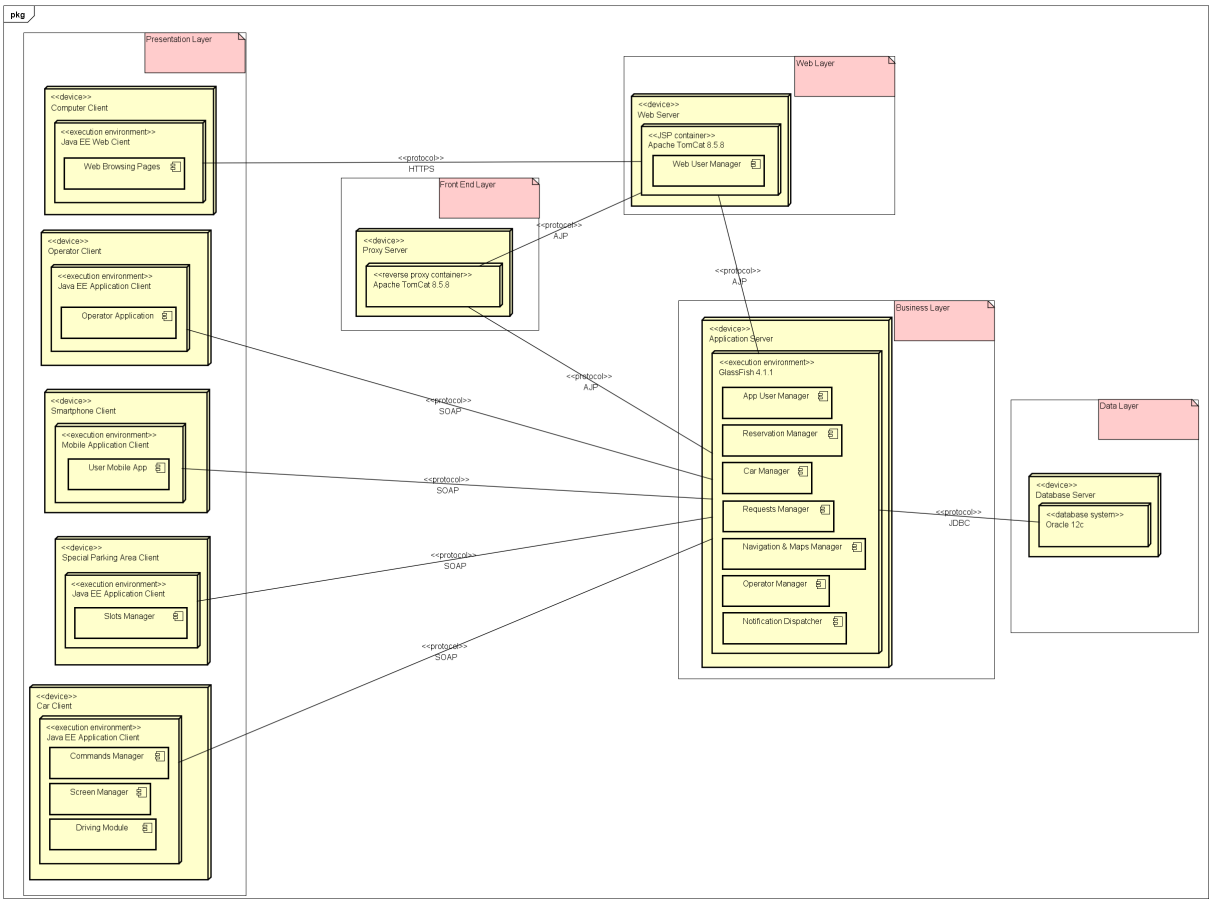
- **Encryption / SSL acceleration:** a reverse proxy is equipped with SSL acceleration hardware.
- **MLoad balancing:** the reverse proxy distributes the load to several Web Servers and Application Servers.
- **Server/cache static content:** a reverse proxy can offload the web servers and application servers by caching static content like pictures and other static graphical content.
- **Compression:** the proxy server can optimize and compress the content to reduce the load time.
- **Security:** the proxy server is an additional layer of defense and can protect against some Web Server or Application Server specific attacks.
- **Extranet Publishing:** a reverse proxy server facing the Internet can be used to communicate to a firewall server internal to an organization, providing extranet access to some functionalities while keeping the servers behind the firewalls.

The Application Server adopts GlassFish execution environment: this is the JEE engine that is going to run the application logic, which is going to be implemented in terms of Enterprise Java Beans. GlassFish is stable, well-known and open source. These are the main reasons for us to adopt it.
The Proxy Server and the Web Server adopt Apache TomCat: contrary to GlassFish, which provides a full implementation of the JEE framework, TomCat is specifically designed to run JSP. For this reason, it can be used to run the server-side of applications, and it's a good practice to use it along with GlassFish.
The Database Server runs the well-known Oracle DBMS, which is robust and reliable. In addition, Oracle DBMS supports Entity/Relationship model schemas and therefore SQL, which fits perfectly our system's needs.
A Web client runs on Computer Clients and consists of two parts:

1. dynamic web pages containing various types of markup language (HTML, XML, and so on), which are generated by web components running in the web tier
2. a web browser, which renders the pages received from the server.

Web clients usually do not query databases, execute complex business rules. Such heavyweight operations are off-loaded to enterprise beans executing on the Java EE server, where they can leverage the security, speed, services, and reliability of Java EE server-side technologies.
An Application Client runs on Operator Clients, Smartphone Clients, Special Parking Area Clients and Car Clients. It provides a way for users to handle tasks that require a richer user interface than can be provided by a markup language.

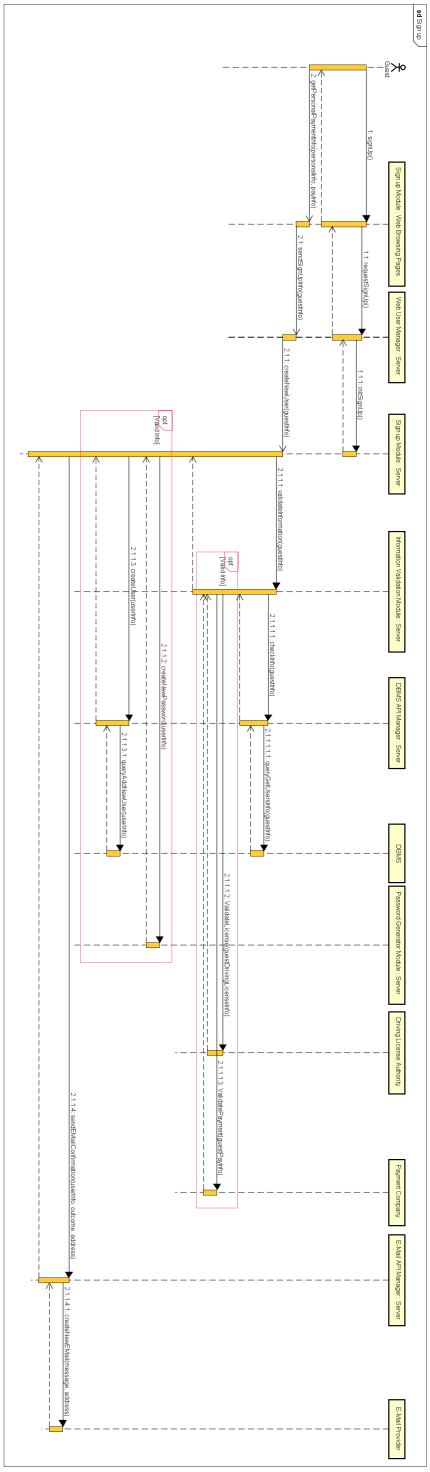
It typically has a graphical user interface (GUI). Application clients directly access enterprise beans running in the business tier. However, if application requirements warrant it, an application client can open an HTTP connection to establish communication with a JSP running in the web tier.
All these software containers are physically distributed and for this reason it is necessary to specify application communication protocols.
Communications between Application Clients and Application Servers occur via SOAP. SOAP can work with different operating systems and, because of this, we can fulfill nonfunctional requirement of portability. Applications can run on Android, iOS, Windows Phone, Linux, Mac OS X and Windows. Basically, SOAP relies on HTTP and XML to exchange data through the web.
Communication between Proxy Server, Web Server and Application Server can be made using AJP which comes as a software module inside Apache TomCat execution environment.
JDBC is an API which provides methods to query and update data in a database, and is oriented towards relational databases.


pkg
Presentation Layer
<<device>>
Computer Client
<<execution environment>>
Java EE Web Client
Web Browsing Pages
<<device>>
Operator Client
<<execution environment>>
Java EE Application Client
Operator Application
<<device>>
Smartphone Client
<<execution environment>>
Mobile Application Client
User Mobile App
<<device>>
Special Parking Area Client
<<execution environment>>
Java EE Application Client
Slots Manager
<<device>>
Car Client
<<execution environment>>
Java EE Application Client
Commands Manager
Screen Manager
Driving Module
<<protocol>>
HTTPS
Front End Layer
<<device>>
Proxy Server
<<reverse proxy container>>
Apache TomCat 8.5.8
<<protocol>>
AJP
Web Layer
<<device>>
Web Server
<<JSP container>>
Apache TomCat 8.5.8
Web User Manager
<<protocol>>
AJP
<<protocol>>
AJP
Business Layer
<<device>>
Application Server
<<execution environment>>
GlassFish 4.1.1
App User Manager
Reservation Manager
Car Manager
Requests Manager
Navigation & Maps Manager
Operator Manager
Notification Dispatcher
<<protocol>>
SOAP
<<protocol>>
SOAP
<<protocol>>
SOAP
<<protocol>>
SOAP
<<protocol>>
JDBC
Data Layer
<<device>>
Database Server
<<database system>>
Oracle 12c

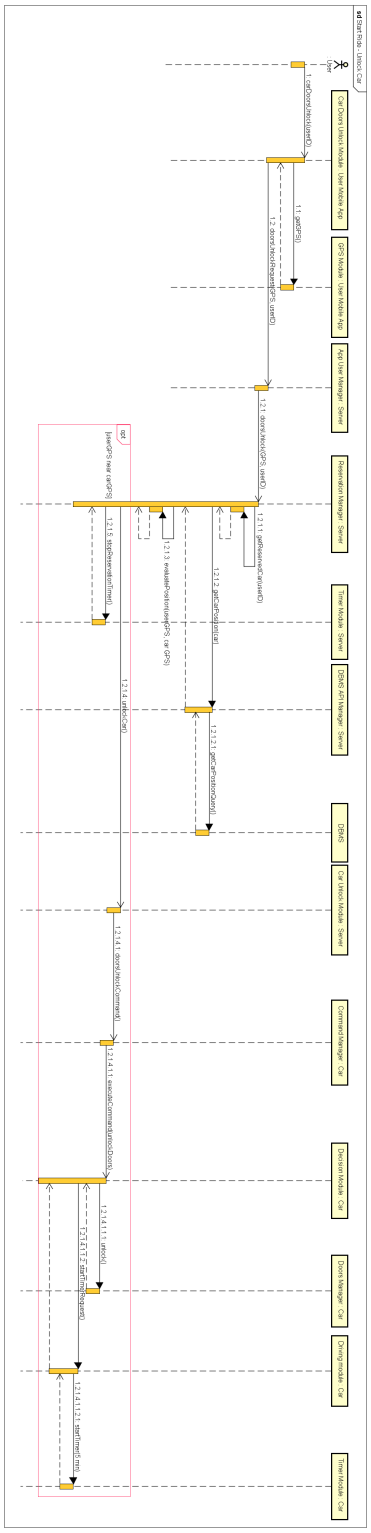
2.5 Runtime view

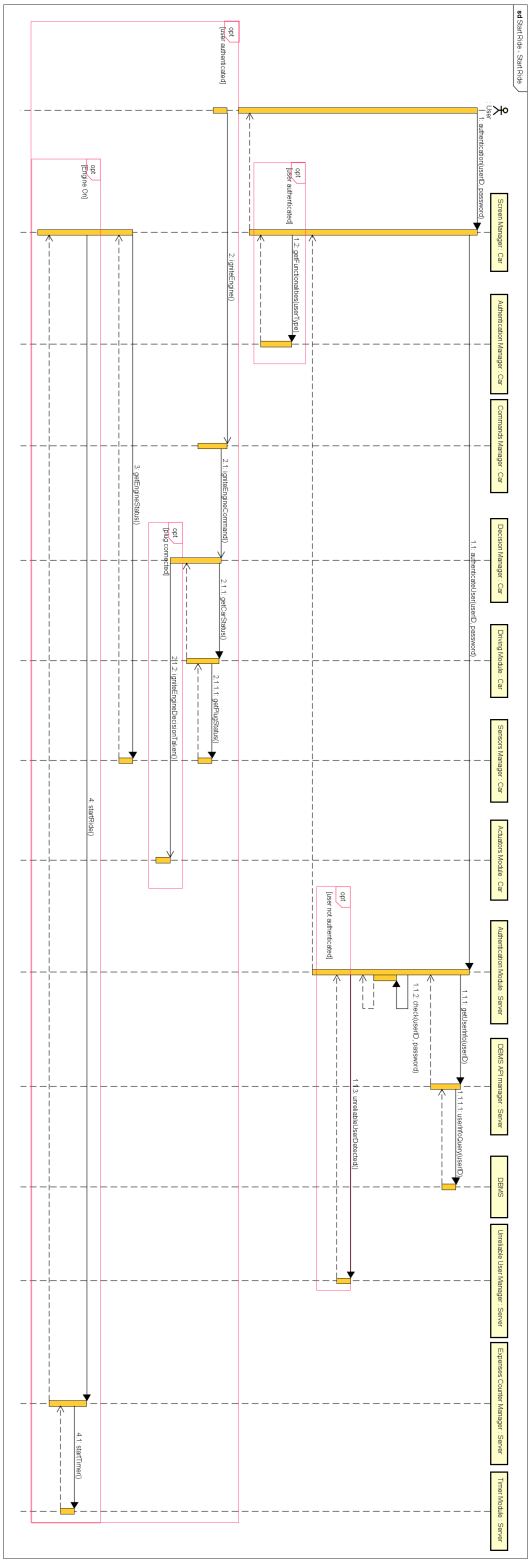
We decided to describe the most significant interactions between the system components to show how some of our functionalities are actualized. In the process, we kept into account only flows and alternative flows that we considered relevant. In some cases we divided the same requirement sequence diagram in two or more different sequence diagrams because the complexity of the original diagram was too high or because the alternative flows interacted with components that were not considered in the main flow of the original sequence diagram. The flows we chose not to consider are only those with few, simple iterations with the system.

2.5.1 Sign up

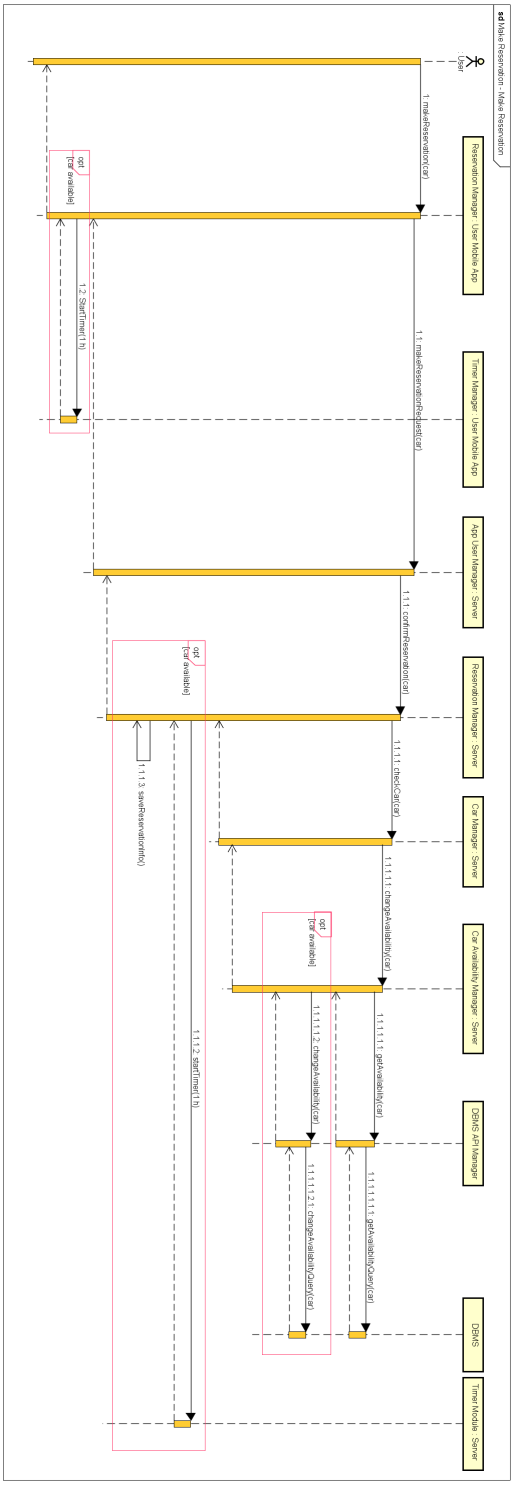

sd Sign up
Guest
Sign up Mobile
Web Browsing Pages
Web User Manager
Server
Sign up Mobile
Server
Information Validation Mobile
Server
DBMS API Manager
Server
DBMS
Password Generator Mobile
Server
Driving Licence Authority
Payment Company
E-Mail API Manager
Server
E-Mail Provider
opt
[valid info]
opt
[valid info]

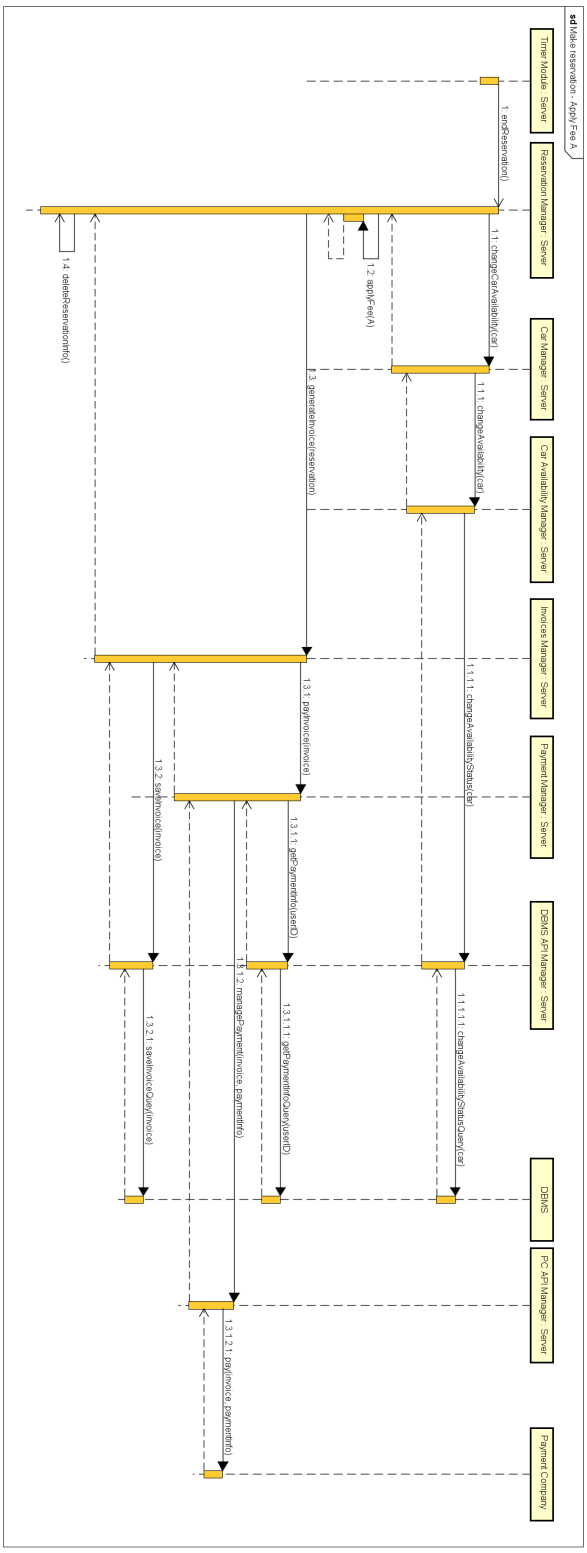
2.5.2 Start ride



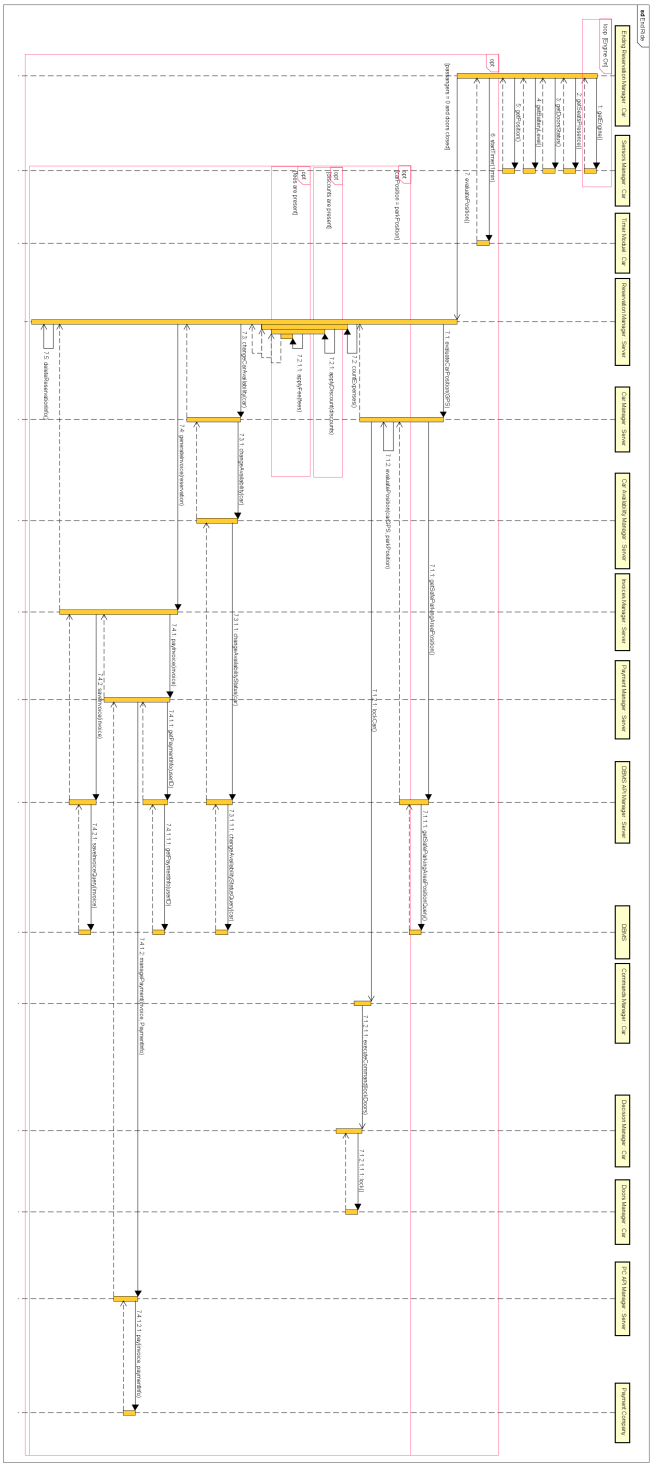

sd StartRide - StartRide
Link
Screen Manager - Car
Authentication Manager - Car
Command Manager - Car
Decision Manager - Car
Driving Module - Car
Sensors Manager - Car
Actuators Module - Car
Authentication Module - Server
DBMS API manager - Server
DBMS
Immutable's User Manager - Server
Expenses Counter Manager - Server
Timer Module - Server
opt
[user authenticated]
opt
[user authenticated]
opt
[engine On]
opt
[plug connected]
opt
[user not authenticated]
1: authenticate(userID, password)
1.1: authenticate(userID, password)
1.2: getExpensesHistory(userID)
2: startEngine()
2.1: startEngine(command)
2.1.1: getPlugStatus()
2.1.1.1: getPlugStatus()
2.2: getRechargeAmount(amount)
3: getEngineStatus()
4: stopRide()
4.1: stopTimer()
1.1.1: getUserInfo(userID)
1.1.1.1: userInfo(userID)
1.1.2: checkUserID, password()
1.1.3: unmarkableUserSelection()

2.5.3 Make reservation

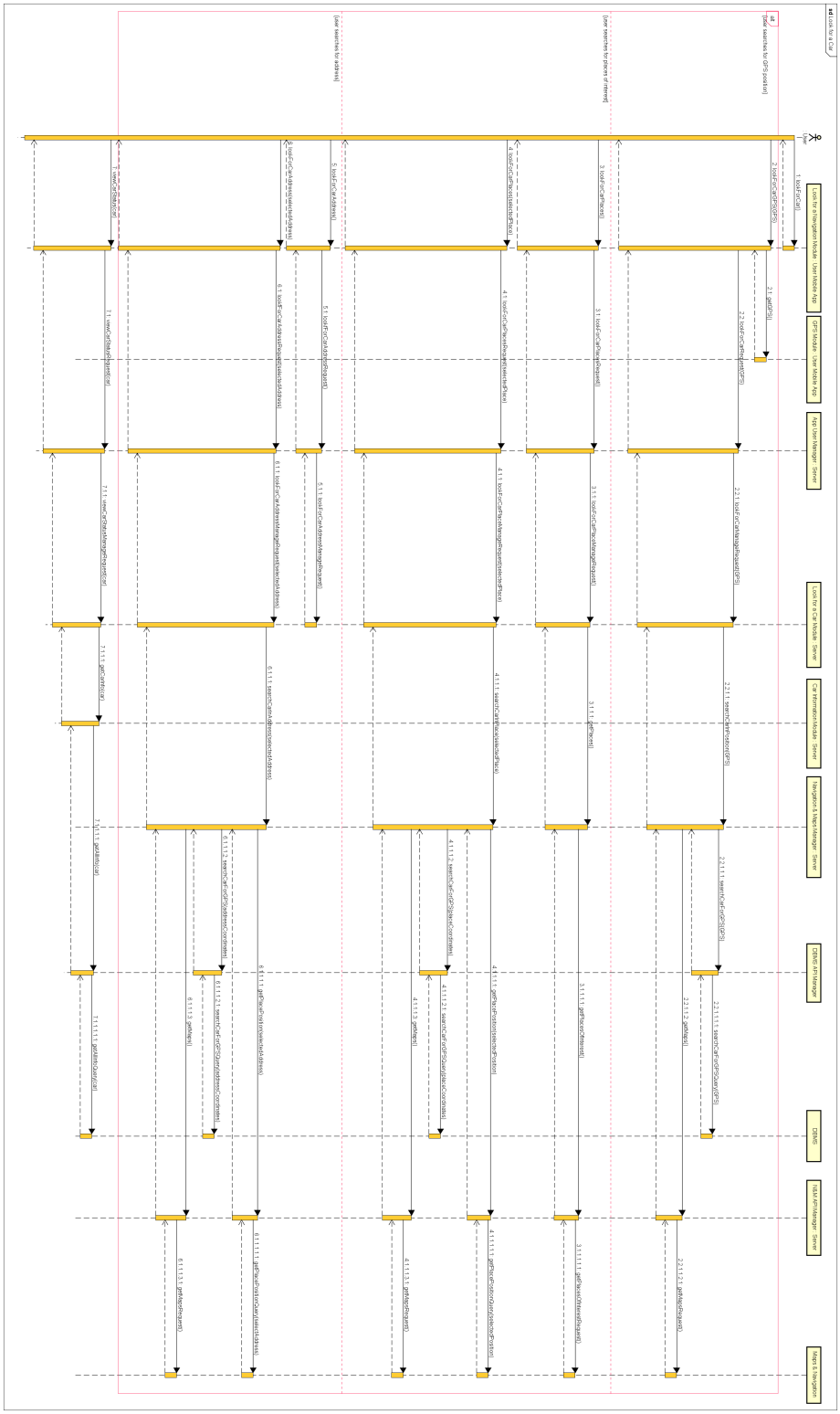



sd Make reservation - Apply Fee A
Timer Module : Server
Reservation Manager : Server
Car Manager : Server
Car Availability Manager : Server
Invoices Manager : Server
Payment Manager : Server
DBMS API Manager : Server
DBMS
PC API Manager : Server
Payment Company
1: endReservation()
1.1: changeCarAvailability(car)
1.2: applyFee(A)
1.1.1: changeAvailability(car)
1.1.1.1: changeAvailabilityStatus(car)
1.1.1.1.1: changeAvailabilityStatusQuery(car)
1.4: deleteReservationInfo()
3: generateInvoice(reservation)
1.3.1: payInvoice(invoice)
1.3.1.1: getPaymentInfo(userID)
1.3.1.1.1: getPaymentInfoQuery(userID)
1.3.1.2: managePayment(invoice, paymentInfo)
1.3.1.2.1: pay(invoice, paymentInfo)
1.3.2: saveInvoice(invoice)
1.3.2.1: saveInvoiceQuery(invoice)

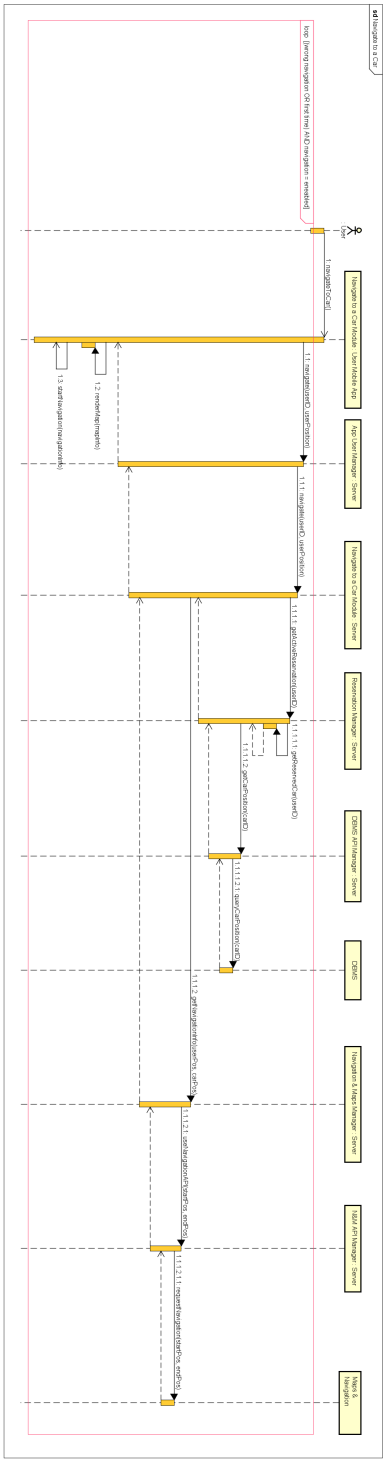
2.5.4 End ride



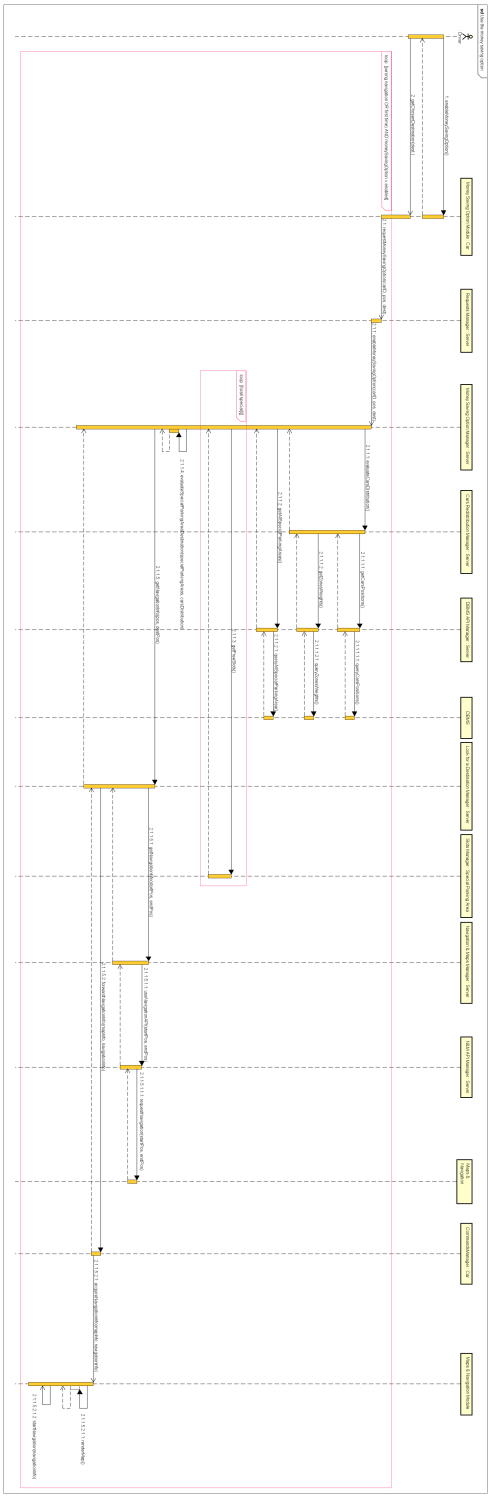
2.5.5 Look for a car



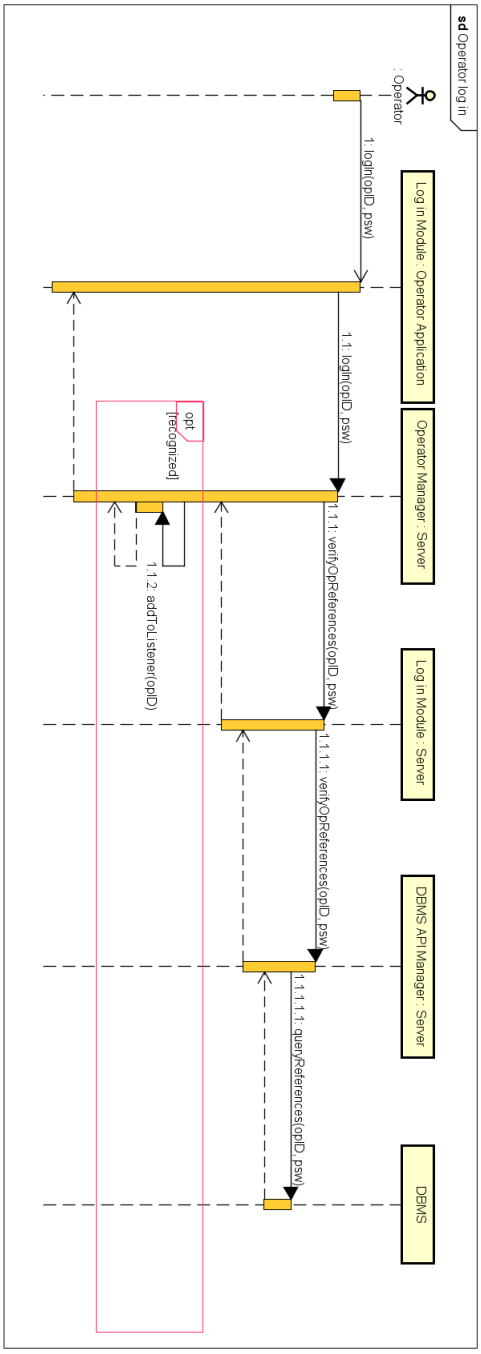
2.5.6 Navigate to a car



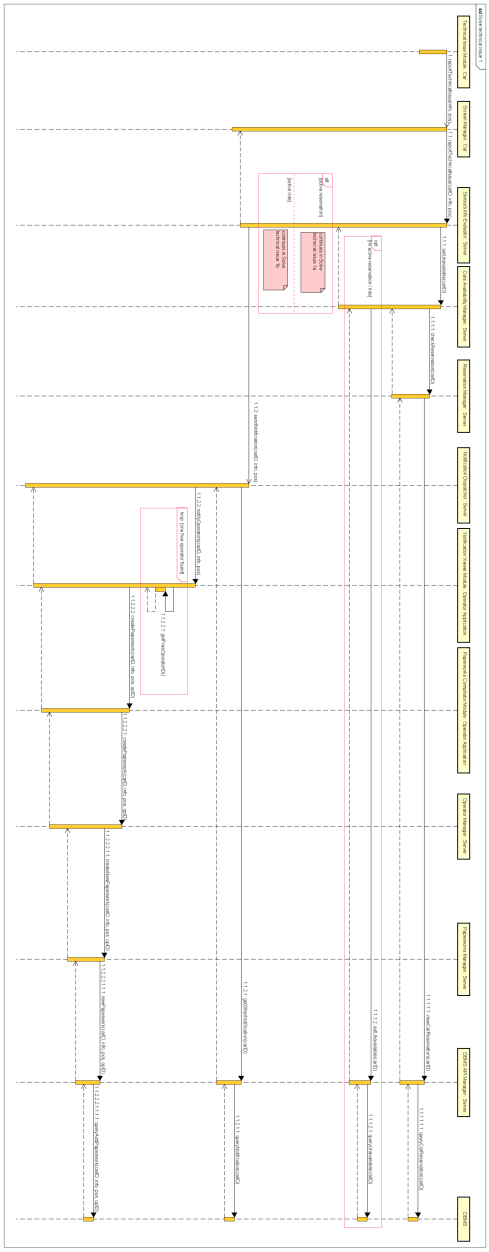
2.5.7 Use the money saving option

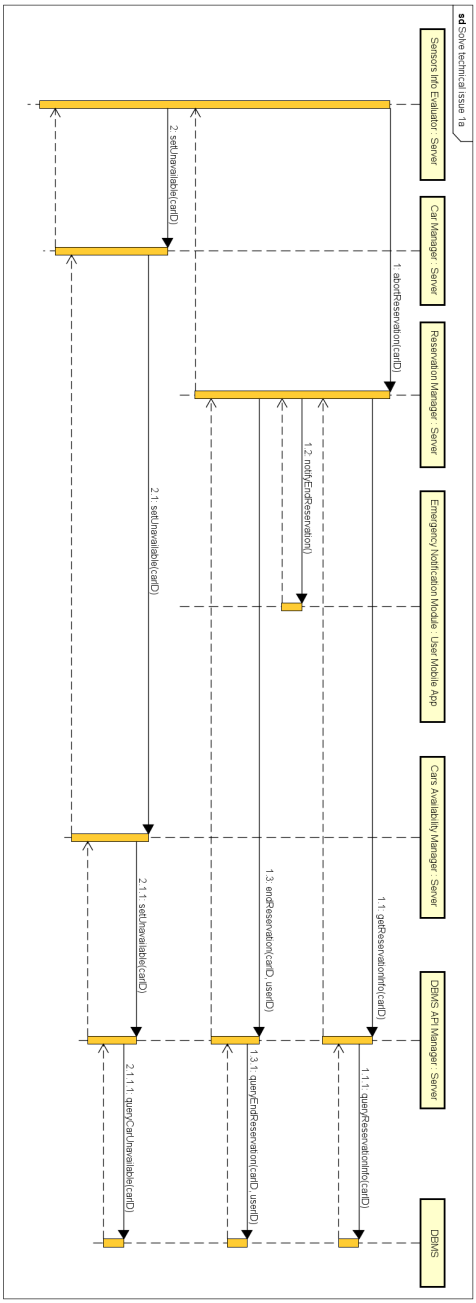


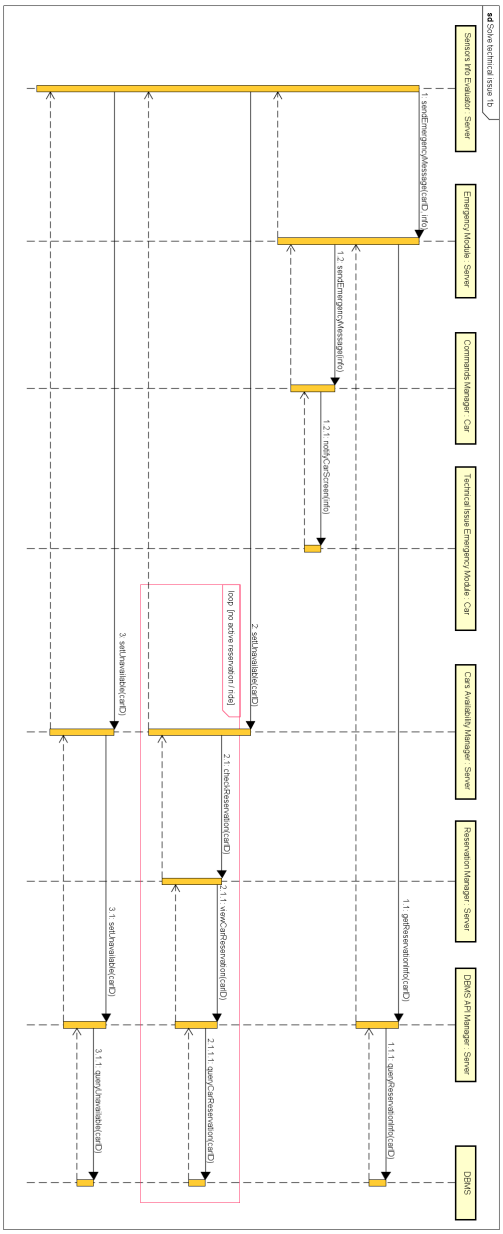
2.5.8 Operator log in

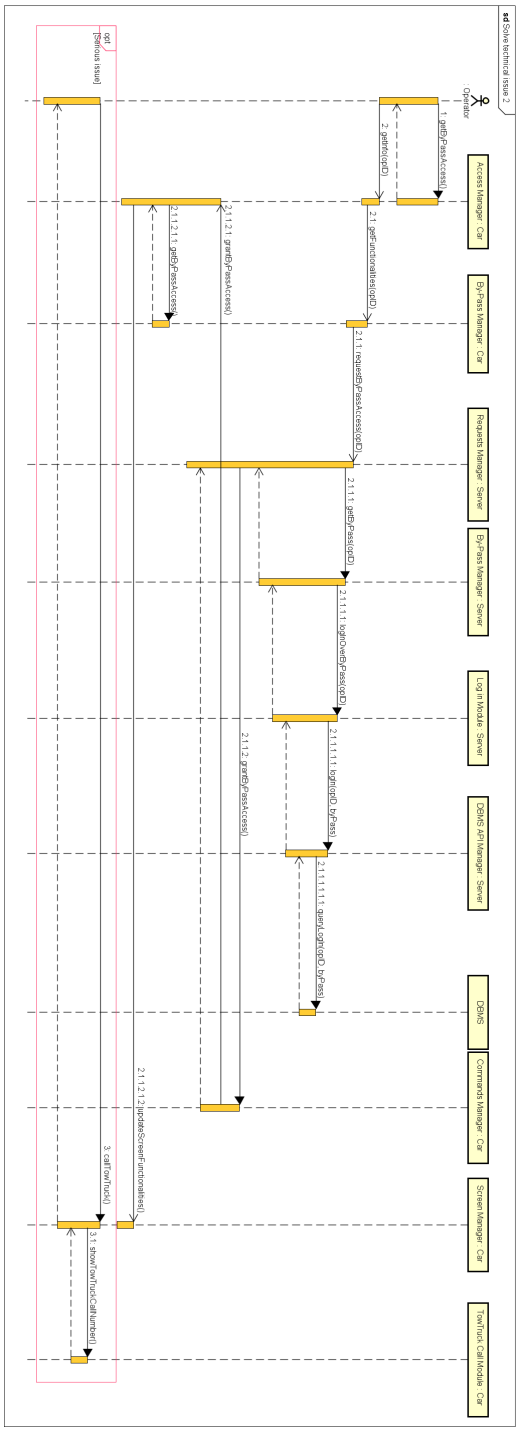


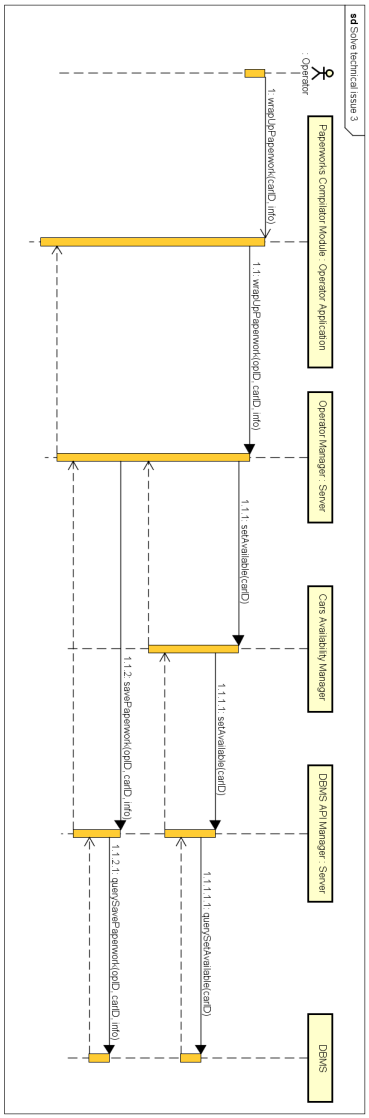
2.5.9 Solve technical issue



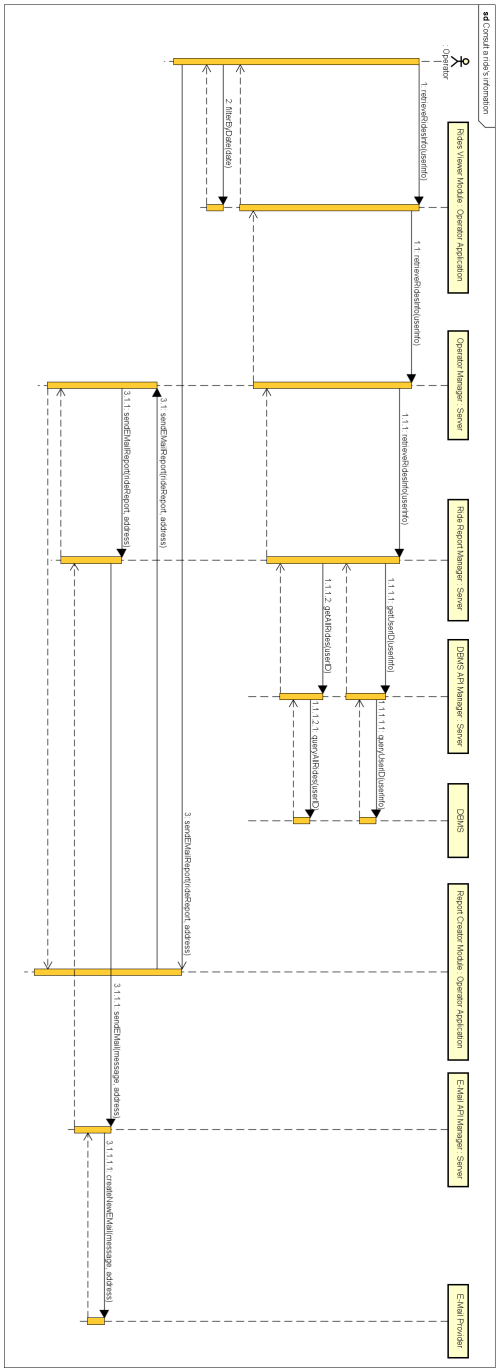

sd Solve technical issue 1a
Sensors Info Evaluator : Server
Car Manager : Server
Reservation Manager : Server
Emergency Notification Module : User Mobile App
Cars Availability Manager : Server
DBMS API Manager : Server
DBMS
2: setUnavailable(carID)
1: abortReservation(carID)
1.2: notifyEndReservation()
2.1: setUnavailable(carID)
1.1: getReservationInfo(carID)
1.3: endReservation(carID, userID)
2.1.1: setUnavailable(carID)
1.1.1: queryReservationInfo(carID)
1.3.1: queryEndReservation(carID, userID)
2.1.1.1: queryCarUnavailable(carID)


sd Solve technical issue 1b
Sensors Info Evaluator : Server
Emergency Module : Server
Commands Manager : Car
Technical Issue Emergency Module : Car
Cars Availability Manager : Server
Reservation Manager : Server
DBMS API Manager : Server
DBMS
1 s: sendEmergencyMessage(carID, info)
1.1: getReservationInfo(carID)
1.1.1: queryReservationInfo(carID)
1.2: sendEmergencyMessage(info)
1.2.1: notifyCarScreen(info)
2: setUnavailable(carID)
loop [no active reservation ride]
2.1: checkReservation(carID)
2.1.1: viewCarReservation(carID)
2.1.1.1: queryCarReservation(carID)
3: setUnavailable(carID)
3.1: setUnavailable(carID)
3.1.1: queryUnavailable(carID)


sd Store terminal issue 2
Operator
Access Manager : Car
ByPass Manager : Car
Requests Manager : Server
ByPass Manager : Server
Log In Module : Server
DBMS API Manager : Server
DBMS
Commands Manager : Car
Screen Manager : Car
TownTork Call Module : Car
opt
[isRoute issue]
1: getByPassAccess()
2: getInfo(opID)
2.1: getFunctionalities(opID)
2.1.1: requestByPassAccess(opID)
2.1.1.1: getByPass(opID)
2.1.1.1.1: loginOverByPass(opID)
2.1.1.1.1.1: login(opID, byPass)
2.1.1.1.1.1.1: queryLogin(opID, byPass)
2.1.1.2: grantByPassAccess()
2.1.1.2.1: grantByPassAccess()
2.1.1.2.1.1: getByPassAccess()
2.1.1.2.1.2: updateScreenFunctionalities()
3: callTownTork()
3.1: showTownTorkCallNumber()


sd Solve technical issue 3
: Operator
Paperwork& Compilation Module: Operator Application
Operator Manager: Server
Cars Availability Manager
DBMS API Manager: Server
DBMS
1: wrapUpPaperwork(carID, info)
1.1: wrapUpPaperwork(opID, carID, info)
1.1.1: setAvailable(carID)
1.1.1.1: setAvailable(carID)
1.1.1.1.1: querySetAvailable(carID)
1.1.2: savePaperwork(opID, carID, info)
1.1.2.1: querySavePaperwork(opID, carID, info)

2.5.10 Consult a ride's information



2.6 Component interfaces

2.6.1 Server

External interfaces

- **CarCommandInt:** allows the server to communicate with the car in order to send messages to the driver or to execute commands in the car like lock and unlock the car doors.
- **CarRequestInt:** allows the car to communicate to the server all information about car status, driver requests or to evaluate the conditions for ending a reservation.
- **DBMS API:** allows the server to communicate with the database to retrieve all information the server needs.
- **N&M API:** allows the server to communicate with the Maps and Navigation Provider to obtain all information about maps and tips for navigation.
- **DLA API:** allows the server to communicate with the Driving License Authority in order to validate the driving licenses of new users.
- **PC API:** allows the server to communicate with the Payment Company in order to validate all the payment information of a new user and to perform payments after a reservation ends.
- **E-Mail API:** allows the server to communicate with the E-Mail Provider in order to send e-mail to users and operators.
- **FreeSlotsInt:** allows the server to communicate with the special parking areas in order to know how many slots are free in a special parking area if a user has enable the money saving option functionality.
- **NotificationInt:** allows the server to dispatch notifications regarding the status to all the operators.
- **OperatorsInt** allows the operator application to interact with the server in order to log in, work on paperworks, report or minor issues solutions and changing cars availability.
- **WebInt:** allows the web browser to communicate to the application in the server in order to let a guest or a user use all the functionalities our web site provides.
- **EmergencyInt:** allows the server to dispatch notifications regarding the status of the car reserved by the user.
- **UserInt:** allows the mobile app to use all the functionalities our mobile app provides forwarding all required information to the application server.

Internal interfaces

- **TimerInt:** allows the managing of the reservation ending timer and car lock doors command when the car sends a message to the server stating that there are the conditions to end a reservation.
- **CarInt:** allows the reservation manager to get all the information about the reserved car.
- **AvailabilityInt:** allows the car manager to get information about the availability of a car.
- **MinorReportInt:** allows to generate a report of a minor issue when the appropriate report is sent by the user from the car screen.
- **ByPassInt:** allows the server to recognize the operator who wishes to unlock a car.
- **DBInt:** gives an access point to the data base in order to perform queries.
- **ReservationInt:** gives an access point to all information of a reservation in order to interact with the mobile app or to give information about the car to other components.
- **LogInInt:** provides functionalities to perform the log in of a person in our system.
- **ToCarInt:** allows the app user manager to gather information to provide the Navigate to a Car functionality from the mobile app.
- **CarSearchInt:** allows the app user manager or the web user manager to provide the Look for a Car functionality to a user or a guest.
- **CarInfoInt:** allows the look for a car component to gather information regarding a selected car in order to show it to a user or to a guest.
- **MoneySaveInt:** allows the request manager to get all information to perform the Money Saving Option functionality.
- **InvoiceInt:** allows the reservation manager to generate the invoice related to an ended reservation.
- **SignUpInt:** provides functionalities for guests to sign up in our system saving all the information in the database or to generate another password if a user requests it.
- **PswInt:** allows the sign up functionalities to generate a password to communicate to a user.
- **E-MailInt:** provides the e-mail functionality.
- **PaymentInt:** allows the sign up component to understand if a user has inserted valid payment information or not and save them in the database.
- **PCInt:** provides the payment management functionality.

- **NavigateInt:** provides functionalities to guide a user to a specific location according to the inserted data.
- **RedistributionInt:** provides functionalities to avoid bad car distribution in the same area for the money saving option functionality and communicates to the notification dispatcher whether there are too many cars in the same area in order to notify the operators.
- **InvoiceViewInt:** allows the user to access all his past invoices generating a list of invoice gathering information from the database.
- **SensorInt:** allows the notifications dispatcher to be notified from sensors located in cars if some issues occurred.
- **InfoInt:** allows to the operator manager to access and edit paperworks.
- **ReportInt:** allows the operator manager to create a ride report to send to legal authority using the e-mail system when needed.
- **MinorIssueInt:** allows the operator manager to report the solution of a minor issue.
- **MapsInt:** provides functionalities to show maps, cars' locations, places of interest, safe and special parking areas.
- **NavigationInt:** provides functionalities to use the navigation systems either through the mobile application and through car's screen.
- **N&MInt:** provides the navigation and maps functionalities.

2.6.2 Car

- **DriverInt:** allows the decision module to command the actuator in order to perform actions like enabling the power plug lock when a car is parked in a special parking area or look and unlock the doors during a reservation.
- **ControllerInt:** allows to execute commands received by the server.
- **EngineOffInt:** allows to turn off the engine after the driver has left the car.
- **SensorsInt:** allows to retrieve information about all sensors present in the car in order to display the car status to the user, to know the position and the status of the car at the end of the reservation or to turn off the engine if the driver leaves the car as a safety measure.
- **TimerInt:** allows to manage the timer to turn off the engine if the driver leaves the car as a safety measure or to wait before sending all information to the server to end the reservation.
- **ECInt:** allows drivers to call a tow truck in case of an emergency.
- **AuthorizationInt:** allows to access a different set of functionalities in the car depending on which person wants to use the car: a user or an operator.
- **OutputInt:** allows to show information retrieved by the server on the car screen in the form of navigation tips.

2.6.3 User Mobile Application

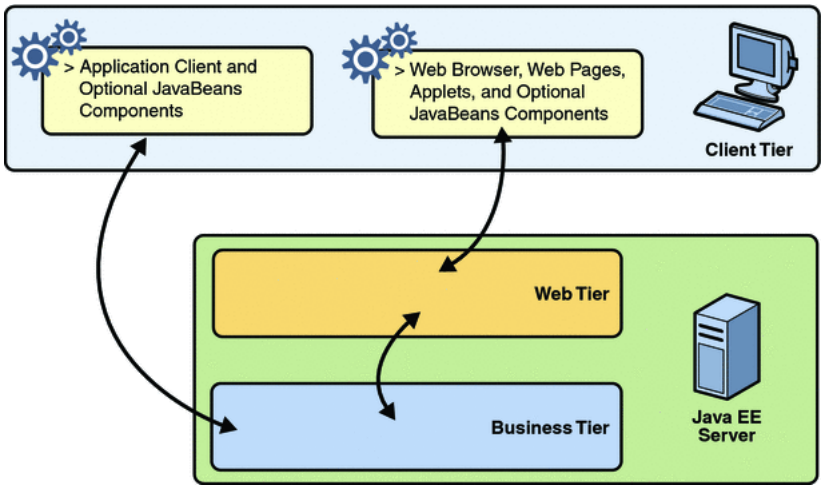
- **TimerInt:** allows the mobile app to set the timer when a user wants to show how much time remains before the end of his reservation. This timer allows to reduce the data request on the server. When a user starts the app a synchronization between server and mobile app is performed and this is the only data that the server communicates to the user.
- **PositionInt:** allows to communicate the GPS position in order to unlock the car for a reservation or, using maps and navigation module, to search a car and navigate to it.

2.6.4 Operator Application

- **InfoInt:** allows to get all the information about the notifications in order to generate the paperwork related to a notification.

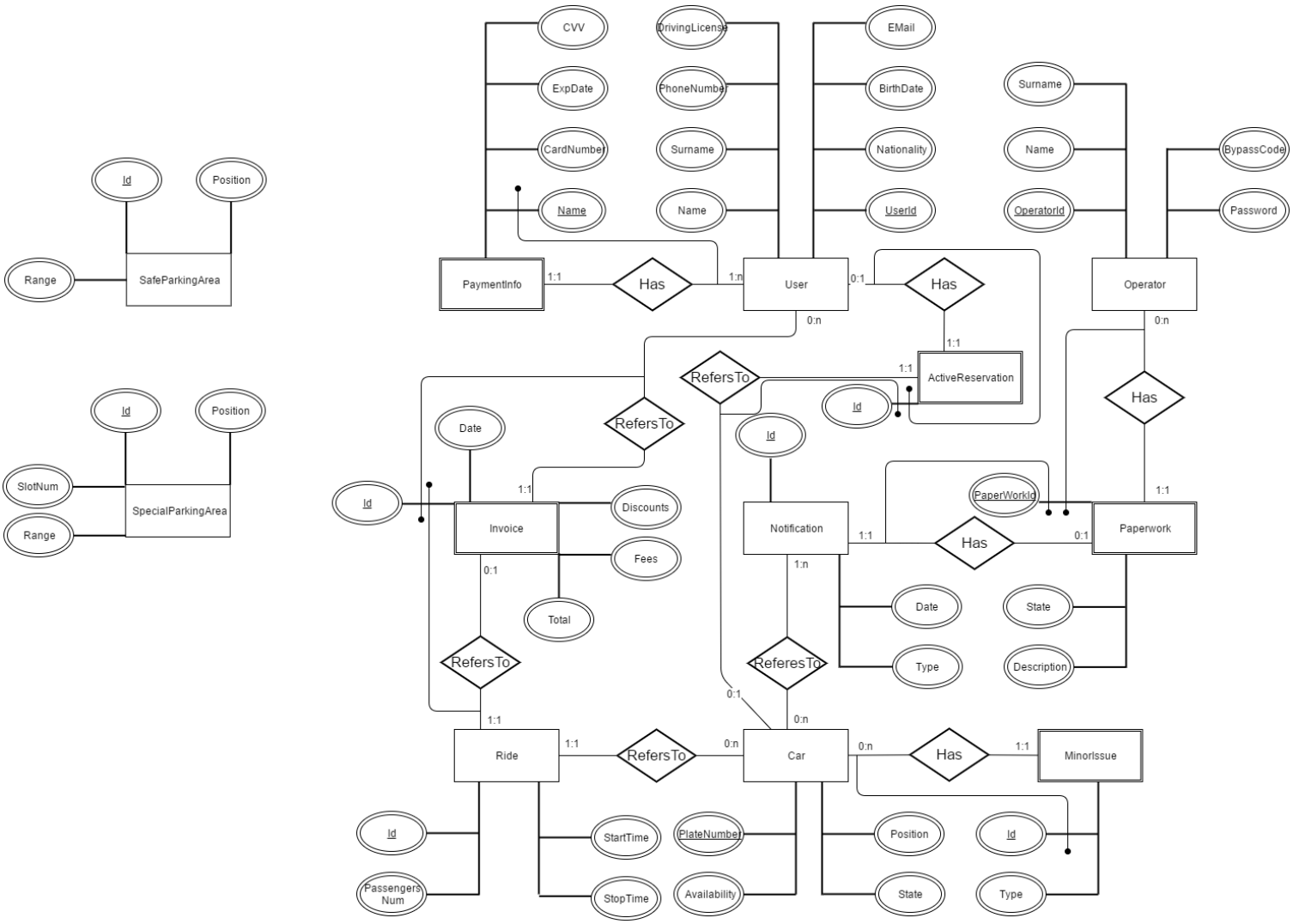
2.7 Architerctural styles and patterns

Use cases analysis in the RASD suggests that main actors of PowerEnJoy system will be Guests, Users, Drivers and Operators. These actors must not share anything with each other but they only need to send requests to the system and to retrieve responses. In addition, each of them can access only a limited set of the system's functionalities according to their role. Furthermore, actors are geographically distributed and they can even move during their communication with the system.
It will be the necessary to control some distributed system devices such as cars and special parking areas.
All these systems' properties lead us to use a Client/Server architectural style which basically relies on the MVC and Observer design patterns (even though it has not been explicitly stated before but they are inherently used in the Java EE environment). The server side will manage all what concerns part of the presentation layer, the business logic and the persistent data management. All these components are centralized in order to be available for every client that requests them. So, most of the functionalities have not to be replicated on the clients but just collected from the central nodes.
The client side is divided into two categories: PowerEnJoy clients and customer clients. PowerEnJoy clients are Operator Clients, Car Clients and Special Parking Area Clients. Customer Clients are Computer Clients and Smartphone Clients.
The operator management is integrated into the Client/Server infrastructure but it does not follow this architectural style. Indeed, a Publisher/Subscriber pattern fits better in this case: once an Operator Client is available to serve the system, it requests to the Server Dispatcher to be added to the listeners list. In this manner, every time the Server will send a notification to operators, it will have the list of available operators ready to serve (listeners, exactly). The two used patterns cooperate to come up with a hybrid infrastructure solution for our system.
Nowadays, the Client/Server infrastructure is one of the most suitable for most of web oriented software applications. The fact that the core business is centralized and under strict control makes the infrastructure reliable and (at least in theory) more secure. Many of the execution parameters such as robustness and reliability can be easily monitored and further software improvements can be taken basing on feedbacks. Moreover, performance evaluations can be performed as the system starts to be used by users and, in the case it is necessary, it can be horizontally scaled. The horizontal scaling is possible thanks to the load balancer inside the Front End Server. A backup database is employed in order to recovery from disasters. The centrality of the core system makes it easily maintainable by system and software engineers.
The main reason to choose Java is portability. As the Java team (Sun Microsystems) states: "write once, run everywhere". Portability is fundamental to make it possible to reach as most as possible catchment area. A typical Java EE Client/Server infrastructure is presented below.


> Application Client and Optional JavaBeans Components
> Web Browser, Web Pages, Applets, and Optional JavaBeans Components
Client Tier
Web Tier
Business Tier
Java EE Server

2.8 Other design decisions

In order to abstract from actual server machines, it is assumed that the execution environments will be installed in virtual machines provided by a cloud-service supplier following the IaaS model.
This choice is relevant because, by doing so, we foresee no physical hardware investment and, in case it will be needed, the number of server machines working on a certain layer may be easily scaled just adding new replica virtual machines of the same execution environment.
Another interesting point concerns costs: one of the main advantages of the Iaas model is that the service must be paid only for the amount of time it is actually used.
A possible future extension may deal with building a proprietary server farm in case of a favorable earning/cost prevision analysis.
We also chose to describe the structure of the database used in the application, since we believe it is a major characteristic of the designed application. In the following page the reader can find the complete ER modeling of database to be deployed with the most significant tables and their relationships.


Id
Position
Range
SafeParkingArea
Id
Position
SlotNum
Range
SpecialParkingArea
CVV
DrivingLicense
EMail
ExpDate
PhoneNumber
BirthDate
CardNumber
Surname
Nationality
Name
Name
UserId
Surname
Name
OperatorId
BypassCode
Password
PaymentInfo
1:1
Has
1:n
User
0:1
Has
Operator
0:n
0:n
1:1
1:1
ActiveReservation
RefersTo
Id
Has
Date
RefersTo
Id
Id
1:1
Discounts
Invoice
Fees
Notification
1:1
Has
0:1
Paperwork
PaperWorkId
1:1
0:1
Total
1:n
Date
State
RefersTo
ReferesTo
Type
Description
0:1
1:1
0:n
Ride
1:1
RefersTo
0:n
Car
0:n
Has
1:1
MinorIssue
Id
StartTime
PlateNumber
Position
Id
Passengers Num
StopTime
Availability
State
Type

3 Algorithm design

In the following paragraph we represent the implementation of the most significant algorithms to be used in PowerEnjoy. In particular, we chose to represent the algorithm for implementing the activation of the money saving option and the one used for the detection of bad distribution of the cars amongst the areas covered by the service. The two algorithms are expressed in an object-oriented pseudo-programming language, enriched and sometimes integrated with comments in natural language, which are useful to comprehend the logical process of the functionality to be developed without going too deep into technical details, expecially when they are related to the interaction between code and navigation and maps API.

3.1 Check car distribution algorithm: checkCarDistribution()

The car distribution amongsts different areas needs to be checked periodically in order to avoid having some overcrowded areas and others that are almost empty at the same time. The main idea of the following algorithm is to give every area a weight that is variable in time to represent the attractivity of the area for possible customers. The perfect distribution of cars is achieved when the summation of the number of cars present in each area per area multiplied per the weight of the area is equal to a constant value represented with the name of OPTWEIGHT. If the value of the summation is higher than (OPTWEIGHT+UPPERTHRESH) or lower than (OPTWEIGHT-LOWERTHRESH), where UPPERTHRESH and LOWERTHRESH are constants, the distribution is considered not acceptable and a notification containing the desired changes in the cars' distribution is produced.

/*Before the first iteration of the algorithm we load the data in temporary variables, so we do not change their real values
while updating the variables used in the algorithm*/
for *zone* in *zoneList* **do**
  $carsNum = zone.getAvailableCars()$
  /*actWeight(date, time) calculates the actual weight of the zone*/
  $zoneActWeight = zone.actWeight(date, time)$
  $zoneTotWeight = carsNum * zoneActWeight$
  $totWeight = totWeight + zoneTotWeight$
  $zoneWeightsRec.append(zoneTotWeight)$
end for
$maxZone = zoneWeights.getMaxIndex()$
$minZone = zoneWeights.getMinIndex()$
while $totWeight > OPTWEIGHT + UPPERTHRESH$ and
$zoneWeightsRec[maxZone].getAvailableCars() > 2$ and
$zoneWeightsRec[minZone].getAvailableCars() < zoneWeightsRec[minZone].getMaxCarNum()$
do
  /*moveRndCar(zone1, zone2) is a function that selects a random car and moves it from zone1 to zone2,
  returning the id of the moved car*/
  $movedCar = moveRndCar(minZone, maxZone)$

$movedCarList.append(movedCar)$
end while
/*We generate a message containing the list of the cars to be moved from maxZone to minZone*/
while $totWeight < OPTWEIGHT - LOWERTHRESH$ and $zoneWeightsRec[minZone].getAvailableCars() > 2$ and $zoneWeightsRec[maxZone].getAvailableCars() < zoneWeightsRec[maxZone].getMaxCarNum()$ **do**
$movedCar = moveRndCar(minZone, maxZone)$
$movedCarList.append(movedCar)$
end while
/*We generate a message containing the list of the cars to be moved from minZone to maxZone*/
$newTotWeight = sum(zone) for zone in zoneWeightsRec$
if $newTotWeight$ not in $range(OPTWEIGHT+UPPERTHRESH, OPTWEIGHT-LOWERTHRESH)$ **then**
/*Recursive call to the function*/
$checkCarDitribution()$
else
$exit()$
end if

3.2 Money saving option algorithm: moneySavingOption(float K)

The money saving options is meant to make expenses as low as possible for a given ride. To do so, the primary objectives are to minimize the time spent driving and to park the car in a special parking area. To achieve this, the algorithm exploits the maps and navigation APIs for navigation to produce a list of all the special parking areas in a range of K metres from the desired destination and sets the one with less cars parked as the new destination. If no special parking areas are found within K metres from the original destination, the algorithm is relaunched with a new K equal to the double of the original K. The process continues following this policy until K reaches a constant value indicated as MAXDIST. If this happens an exception is raised and the algorithm stops.

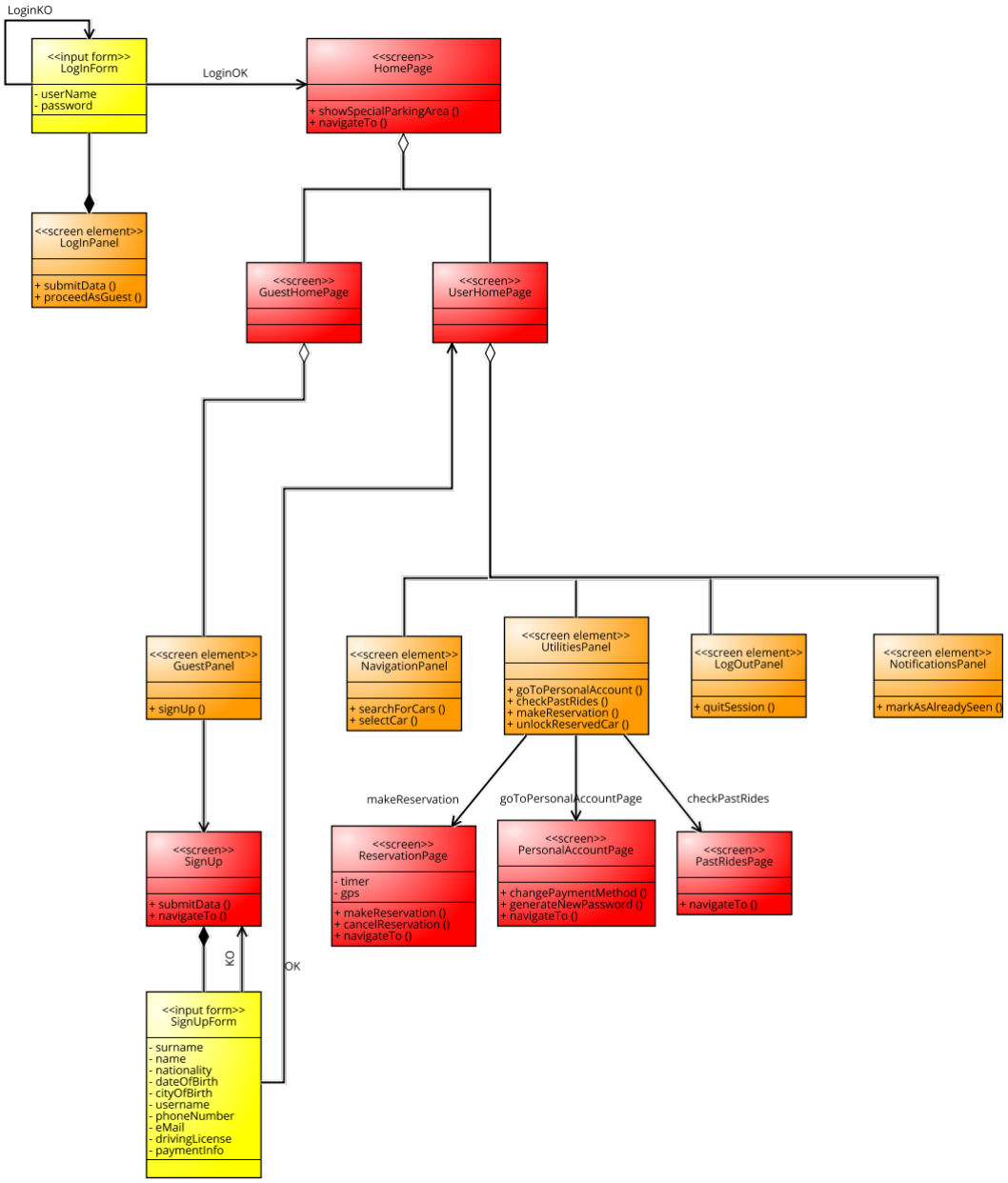
/*The function is called passing an argument K ,
representing the maximum distance of the special parking areas from the desired destination*/
/*We refer to the APIs for navigation and geolocalization provided by*/
$carPos = car.getCarPosition()$
$destinationPos = getDestPosition(dest)$
$currentWeight = MAXCURRENTWEIGHT$
for spa in $specParkAreaList$ **do**
/*The estimated cost of the ride is initially set to the maximum possible amount*/
if $spa.getPosition()$ in $range(destPos, destPos+K)$ and $spa.getCarsNum() < spa.getMaxCarsNum()$ **then**
$possibleSpecParkAreaList.append(spa)$

```
    end if
  end for
  if possibleSpecParkAreaList.lenght() == 0 and k >= MAXDIST then
    /*We launch an exception signaling the impossibility of activate the money
    saving option and abort the operation*/
  else if possibleSpecParkAreaList.lenght() == 0 then
    moneySavingOption(2K)
  else
    for pspa in possibleSpecParkAreaList do
      actWeight = pspa.actWeight(date, time) * pspa.getCarNum()
      if actWeight < currentWeight then
        currentWeight = actWeight
        chosenSpa = pspa
      end if
    end for
  end if
  /*We provide the user with the information regarding the chosen special
  parking area, the estimated cost of the ride and all the obtainable discounts*/
  exit()
```

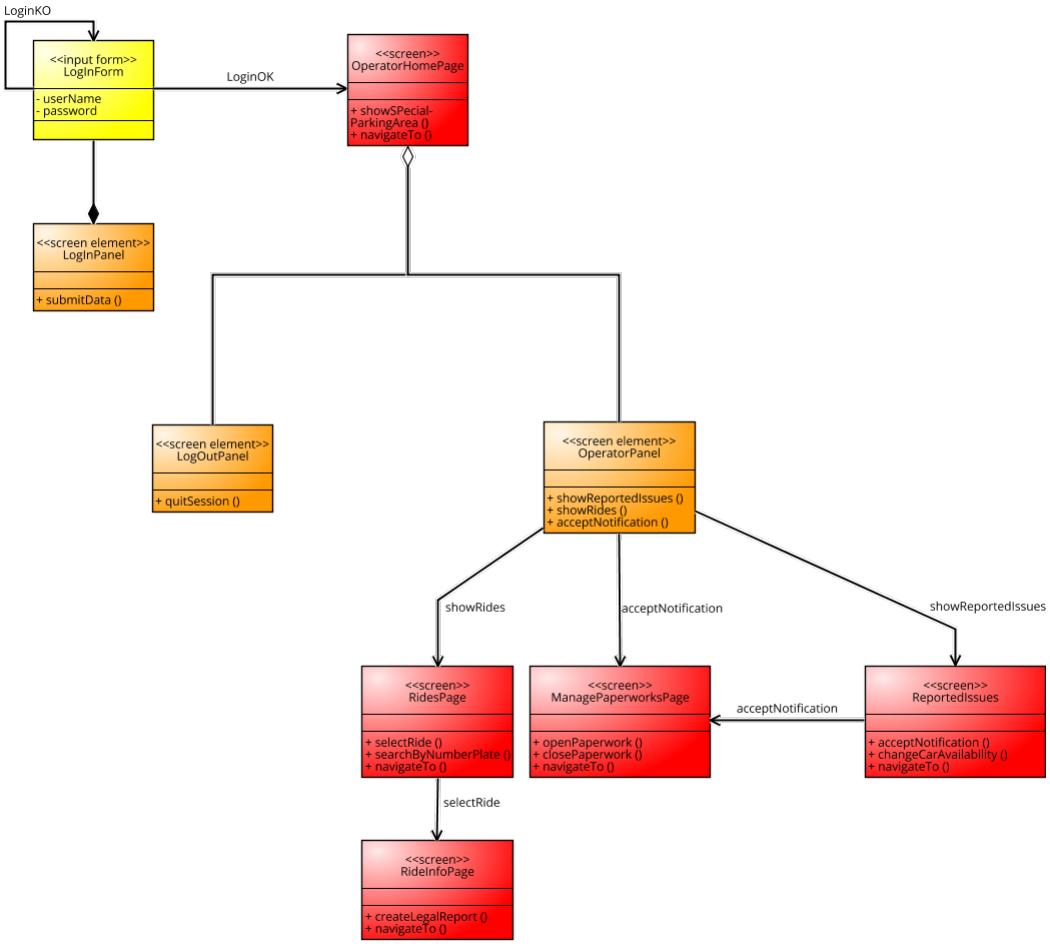
4 Interface design

The following paragraph is strongly connected to the information regarding user experience we already provided in the RASD. In particular, basic mockups useful to understand the interface to be provided to the user were already present in the previous document, so in the following pages we concentrated our efforts in describing the same concepts using UX diagrams. The three diagrams represent the different aspects of the application interface: the view of the user's app, of the operator's app and of the car screen.

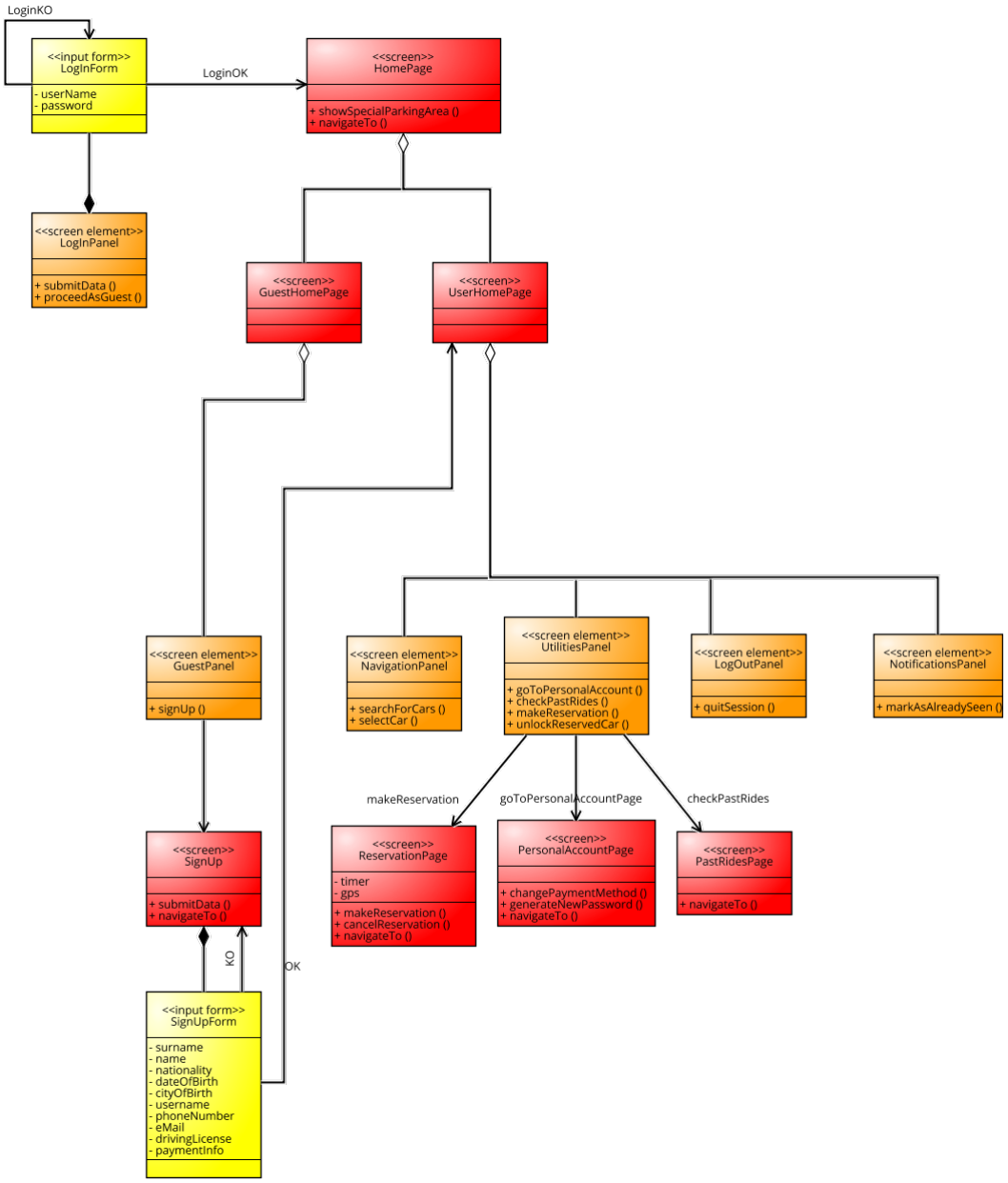
4.1 User interface design



4.2 Operator interface design



4.3 Car screen interface design



5 Requirements traceability

The primary aim of this document was to cover all the goals and requirements specified in the RASD through the use of appropriate software and hardware components. In this paragraph we provide a clear mapping of the relation between the goals described in the RASD and the components designed in the DD.

- **G1:**
  - **R1:** Sign Up Module, Password Generator, Web User Manager, E-Mail API Manager, Access Module in Web Browsing Pages, Sign Up Module in Web Browsing Pages
  - **R2:** Information Validation Module, Payment Manager, PC API Manager
  - **R3:** Log In Module, App User Manager, Web User Manager, Access Module in Web Browsing Pages, Log In Module in App User Manager and Browsing Pages
  - **R4:** Car Manager, Car Lock Module, Car Unlock Module, GPS Module in User Mobile App, Reservation Manager in User Mobile App, Car Doors Unlock Module in User Mobile App, GPS Module in Car, Command Manager in Car, Decision Manager in Car, Actuator Manager in Car, Doors Manager in Car
  - **R5:** Expenses Counter Manager, Engine Module in Car, Screen Manager in Car, Current Expenses Viewer Manager
  - **R6:** Expenses Counter Manager, Payment Manager, PC API Manager, Ride Manager, Reservation Manager, Invoices Manager, Ending Reservation Manager in Car
  - **R7:** Invoices Viewer Module, Invoices Manager, Personal Profile Manager in User Mobile App and in Web Browsing Pages, Last Invoices Manager in User Mobile App and in Web Browsing Pages
- **G2:**
  - **R1:** Reservation Manager, Car Manager, Cars Availability Manager, Reservation Manager in User Mobile App
  - **R2:** Reservation Manager, Fees Manager, Timer Manager, Car Manager, Cars Availability Manager, Reservation Manager in User Mobile App, Timer Manager in User Mobile App
  - **R3:** Cars Availability Manager
  - **R4:** Car Manager, Car Lock Module, Command Manager in Car, Decision Manager in Car, Actuator Module in Car, Doors Manager in Car
  - **R5:** Reservation Manager, Ride Module, Expenses Module, Invoices Manager
  - **R6:** Reservation Manager, Car Module, Cars Availability Manager
  - **R7:** Reservation Manager, Reservation Manager in User Mobile App

- **G3:**
  - **R1:** Look for a Car Module, Car Information Module, App User Manager, Web User Manager, Navigation and Maps Manager, N&M API Manager, Navigation and Maps Module in User Mobile App and Web Browsing Pages, Look for a Car Module in User Mobile App and Web Browsing Pages
  - **R2:** Navigate to a Car Module, App User Manager, Navigation and Maps Manager, N&M API Manager, Navigation and Maps Module in User Mobile App, GPS Module in User Mobile App, Navigate to a Car in User Mobile App
- **G4:**
  - **R1:** Request Manager, Look for a Destination Manager, Navigation and Maps Manager, N&M API Manager, Screen Manager in Car, Navigation and Maps Module in Car
  - **R2:** Money Saving Option Manager, Cars Redistribution Manager, Request Manager, Navigation and Maps Manager, N&M API Manager, Screen Manager in Car, Navigation and Maps Module in Car, Money Saving Option Module in Car
- **G5:**
  - **R1:** Reservation Manager, Discounts, Manager, Sensors Manager in Car, Seats Presence Manager in Car, Ending Reservation Manager in Car
- **G6:**
  - **R1:** Navigation and Maps Manager, Special Parking Areas Module, App User Manager, Web User Manger, Screen Manager in Car, Navigation and Maps Module in Car, Special Parkinng Areas Module in User Mobile App and in Web Browsing Pages
  - **R2:** Reservation Manager, Discounts Manager, Sensors Manager in Car, Battery Level Module, Ending Reservation Manager in Car
  - **R3:** Reservation Manager, Discounts Manager, Sensors Manager in Car, Power Plug Manager, Ending Reservation Manager in Car
  - **R4:** Car Manager, Cars Avaiilability Module, Sensors Info Evaluator Module, Notifications Dispatcher, Sensors Manager in Car, Battery Level Module in Car, Notification Viewer inModule in Operator Application
  - **R5:** Car Manager, Cars Availability Module, Battery Level Module in Car, Screen Manager in Car
  - **R6:** Reservation Manager, Fees Manager, Sensors Manager in Car, Battery Level Module, GPS Module, Ending Reservation Manager in Car

- **G7:**
  - **R2:** Cars Redistribution Manager, Notification Dispatcher, Notification Viewer Module in Operator Application
- **G8:**
  - **R1:** Car Manager, Notification Dispatcher, Cars Availability Manager, Notification Viewer Module in Operator Application, Sensors Manager in Car, Technical Issue Emergency Module in Car, Tow Truck Call Module in Car
  - **R2:** Request Manager, Minor Issue Report Manager, Screen Module in Car, Money Saving Option Module in Car
  - **R3:** Operator Manage, Minor Issue Manager, Paperwork Manager, Minor Issue Viewer Module in Operator Application
- **G9:**
  - **R1:** Reservation Manager, Authentication Module, Unreliable User Manager, Log In Module, Screen Manager in Car, Authentication Manager in Car, User Authentication Manager in Car
  - **R2:** Reservation Manager, Authentication Module, Unreliable User Manager, Log In Module, Screen Manager in Car, Authentication Manager in Car, User Authentication Manager in Car
  - **R3:** Reservation Manager, Authentication Module, Unreliable User Manager, Log In Module, Screen Manager in Car, Authentication Manager in Car, User Authentication Manager in Car, Driving Module in Car, Timer Module in Car
  - **R4:** Operator Manager, Ride Report Manager, E-Mail API Manager, Report Manager in Operator Application, Rides Viewer Module in Operator Application, Report Creator Module in Operator Application
  - **R5:** Sign In Module, Password Generator Module, App User Manager, Web User Manager, E-Mail API Manager, Personal Profile Manager in App User Manager and Web Browsing Pages, New Password Request Manager in App User Manager and Web Browsing Pages
  - **R6:** Reservation Manager, Timer Module, Unreliable Ride Module, Car Manager, Notification Dispatcher, Operator Manager, Ride Report Manager, E-Mail API Manager, Report Manager in Operator Application, Rides Viewer Module in Operator Application, Report Creator Module in Operator Application

6 Appendix

6.1 Used tools

The following software tools were used to produce the DD document:

1. TexWorks (www.tug.org/texworks/): framework for LaTex text format
2. Signavio (www.signavio.com): online framework for modeling class diagrams
3. Astah Professional 7.1.0 (www.astah.net): tool for modeling component, deployment sequence diagrams
4. Draw.io (www.draw.io): online tool for modeling ER diagrams

6.2 Hours of work

Lo Bianco Riccardo	36h
Manzoni Mirco	45h
Mascellaro Giuseppe	43h

6.3 Changelog

Version 1.1:

- Integration of section *Architecture overview*
- Integration of subsection *subsection Component view*
- Several typos.