AY 2022/2023

Politecnico di Milano

DD: Design Document

Valeria Amato Francesco Dettori Matteo Pancini

Prof.
Elisabetta Di Nitto

Version 1.4
January 3, 2023

Contents

1 Introduction

1.a Purpose

The purpose of this document is to provide the design of the eMall application. It will include the architecture of the system and its functional description with a description of the modules and their interfaces. Furthermore we will show the traceability maps between the requirements and the components. Finally we will discuss our plans for implementation, integration and test.

1.b Scope

The scope of eMall is to develop a solution so that an electric car owner won't even notice the disadvantages related to charging time of this sort of vehicle, creating a platform which is perfectly integrated in the owner's daily schedule. Moreover, eMall wants to help CPOs manage their stations, let them know information about the internal and external situation, deal with DSOs to acquire energy and force the start and stop of charge in case of need. To accomplish its goals, already discussed in the RASD document, the system must be used by as many users as possible. This is why the intuitiveness and ease-of-use will be fundamental during the designing phase, considering also that the users' technological knowledge is not assured. In order to guarantee the correctness of its functionalities and the overall quality of the system will be tested before every update release.

1.c Definitions, Acronyms, Abbreviations

1.c.1 Definitions

- API: Application Programming Interface
- CAP: Consistency, Availability, and Partition tolerance
- DBMS: DataBase Management System
- GPS: Global Positioning System
- HTTP: HyperText Transfer Protocol

- ORM: Object-Relational Mapping
- OS: Operating System
- RASD: Requirements Analysis and Specification Document
- REST: REpresentational State Transfer
- RIA: Rich Internet Application
- TCP/IP: Transmission Control Protocol/Internet Protocol
- MVC: Model-view-controller

1.c.2 Acronyms

- CPO: Charging Point Operator. It refers to the charging station owner and manager.
- CPMS: Charge Point Management System. System through which each CPO can administer his own IT infrastructure.
- DSO: Distribution System Operator. It refers to 3rd party companies that sell energy to CPOs.

1.c.3 Abbreviations

1.d Revision history

- December 9, 2022 (version 1.0)
- December 19, 2022 (version 1.1):
  - Update of the Reference Documents
  - Update of some graphics
  - General revision
- December 21, 2022 (version 1.2):
  - Effort spent
  - General revision

- January 3, 2023 (version 1.3):
  - Typo corrections

1.e Reference documents

- Specification document: R&DD Assignment - A.Y. 2022/23
- eMall - RASD (Requirements Analysis and Specifications Document)
- Course slides

1.f Document structure

- **Section 1: Introduction**
  This section will provide the purpose and scope of the document, as well as the revision history and the documents used overview of the project.
- **Section 2:Architectural design**
  This section will describe the system's architecture including the components, the interactions and the deployment, as well as a runtime view and further details about design decisions.
- **Section 3: User Interface Design**
  This section will provide mockups and their interactions of the application both from the user's and the CPO's perspective that will complete the ones presented in the RASD document.
- **Section 4: Requirements Traceability**
  This section will map the requirements, defined in the RASD, with the components described in this document.
- **Section 5: Implementation, Integration and Test Plan**
  This section will show the implementation, integration and test plan of the system.

2 Architectural Design

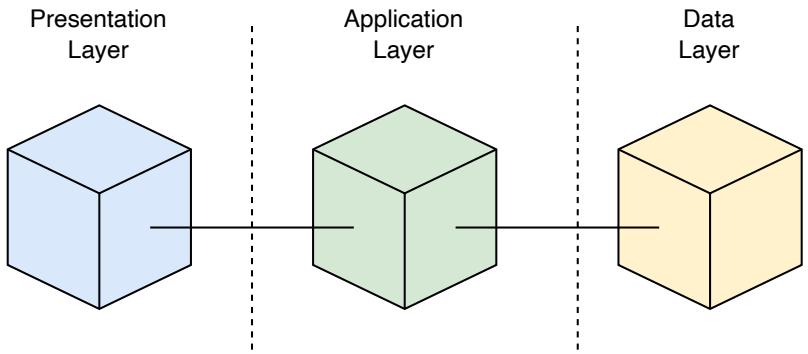
2.a Overview

The system uses the well-known client-server paradigm and is a distributed application.

There are two distinct types of clients in particular, which causes it to either be thin or fat at the same time:

- The first one is an RIA Web Application, used by CPOs, which is a thin client by definition due to its complete reliance on the server. This type of client does not contain the application business logic, but only the presentation layer.
- The second, however, is a mobile application, used by users, that has its own internal database to reduce its need on the server. It becomes a thicker client due to this feature.

In both situations, the server is large and houses all the business logic and data management. This section will provide a short explanation of the architecture while also supporting all of the chosen patterns.



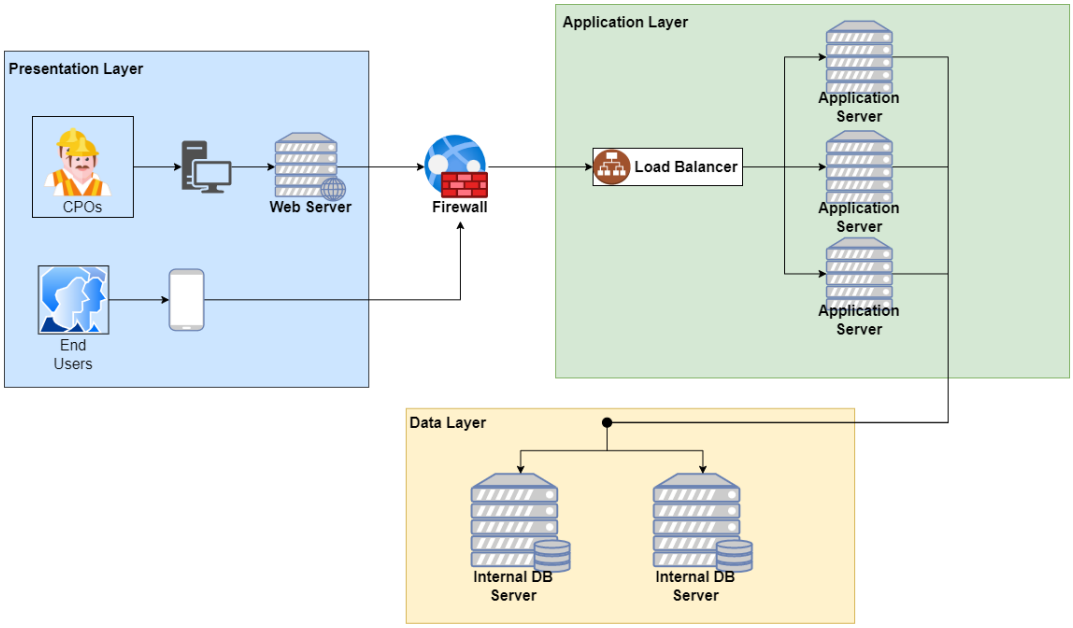
The image shows the layers of the system, which respectively are:

- Presentation Layer: it controls the presentation logic and, as a result, all user interactions.
- Application Layer: it is where the logic of the system is implemented, it manages the business functions.

- Data Layer: it manages the safe storage and the relative access to data.

As shown in the high level representation of the figure below, the system is divided into three layers that are physically separated by installing them on different tiers.

A tier is a physical (or a set of) machine, each of them with its own computational power. The application described in this document is composed of four tiers.



The service is supposed to be accessed through the web application by the CPOs and through a mobile application by the end users.

A client-side scripting paradigm will be used to enable the creation of the web application. In the architectural figure, the application is divided into the aforementioned layers. These servers serve as a middleman between the CPOs' browser and the application servers.

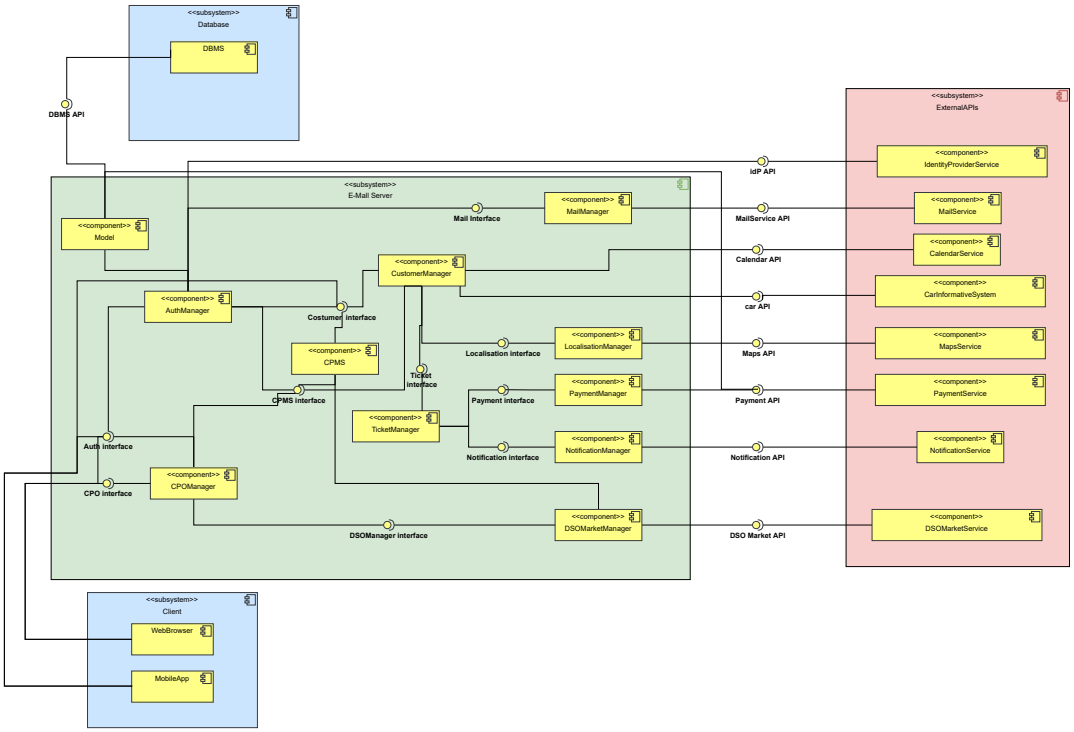
Instead, when using a mobile application, the core of the program installed on the user's device will communicate with the APIs of the business layer, which send and receive all the data necessary for the application to function effectively to the application servers.

Finally, the application server interfaces with the DBMS APIs, in order to retrieve and store the data required for the considered computation.

The applications servers are expected to be stateless, according to the REST standard definitions (more details in section 2.7). For accessing the data, they will use an ORM programming technique in order to interface with the DBMS exploiting the advantages of the object-oriented paradigm. Firewalls are used to offer a higher level of system security by separating the nodes.

The next sections will include a detailed description of each component.

2.b Component view



Model Entry-point to the database, this component represents the data on the server and acquires it through queries. It is connected to most components of the system due to the fact that it's the only component that

interacts with the DBMS. The database contains all information related to the user's account.

AuthManager Handles the authentication of the users and CPOs, when logging in as well as the authorization to access different options of the system. It interacts with Mail Interface in the registration phase. It communicates with the idP API if the "sign up" or "log in" action has been done with the Google or the Facebook account.

CPMS This document considers it to be a legacy. It controls all correspondence between the main server and the charging points. It contains all station-related information.

CustomerManager It serves as the user's representative and provides access to all user-taken actions. To recommend personalized charging slots, it then communicates with the CPMS, the LocalisationManager, the Calendar and Car API; the creation of the ticket is connected to the TicketManager component.

MailManager Interacts with the mail server to send the users a confirmation email when registering in the system.

TicketManager Its duties include managing and creating tickets, as well as tasks relating to payments and notifications. In reality, it communicates with the NotificationManager and PaymentManager to appropriately pay the fee and notify of a new slot or a change in the charging status.

LocalisationManager Communicates with the Maps API to get the user location, useful to load the map in the application and to suggest charging slots based on the calendar. The LocalisationManager is called only by the CustomerManager, the management component of the user's actions.

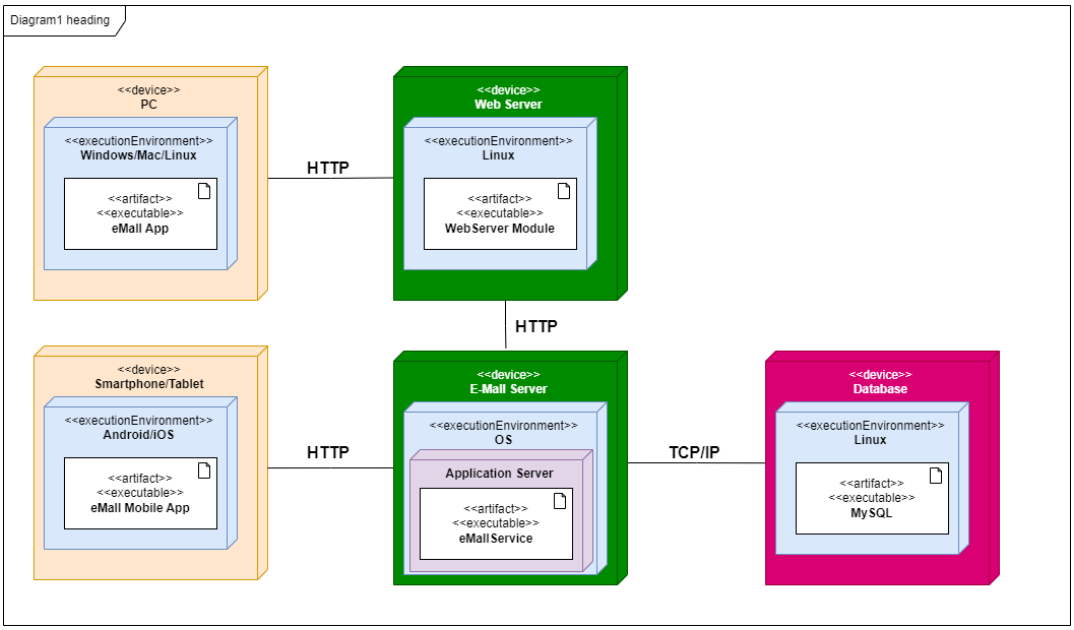
PaymentManager It engages with the PaymentService with the intent of requesting funds from the user's bank account, after a ticket is closed.

NotificationManager Manages the push notifications of the system, these are issued when a new calendar suggestion is elaborated.

DSOMarketManager It works in conjunction with the DSOMarketAPI and enables the CPMS and CPO to modify DSO energy. We agreed that each DSO can only have one energy distribution as a matter of convention.

CPOManager It is the commander of the CPMS component: allows human decisions. All actions made by the CPOManager win on the CPMS automatic resolutions.

2.c Deployment view



The deployment diagram in figure above shows the needed components for a correct system behavior, except the external APIs for CreditProvider, MailService, CalendarService, MapsService, PaymentService, NotificationService, DSOMarketService, IdentityProviderService and CarInformativeSystem. Each device has its own Operating System where the software runs. The tiers in the image are the following:

- Tier1: The client machine, which can be either a downloadable mobile application (available on both the Apple Store and Google Play Store)

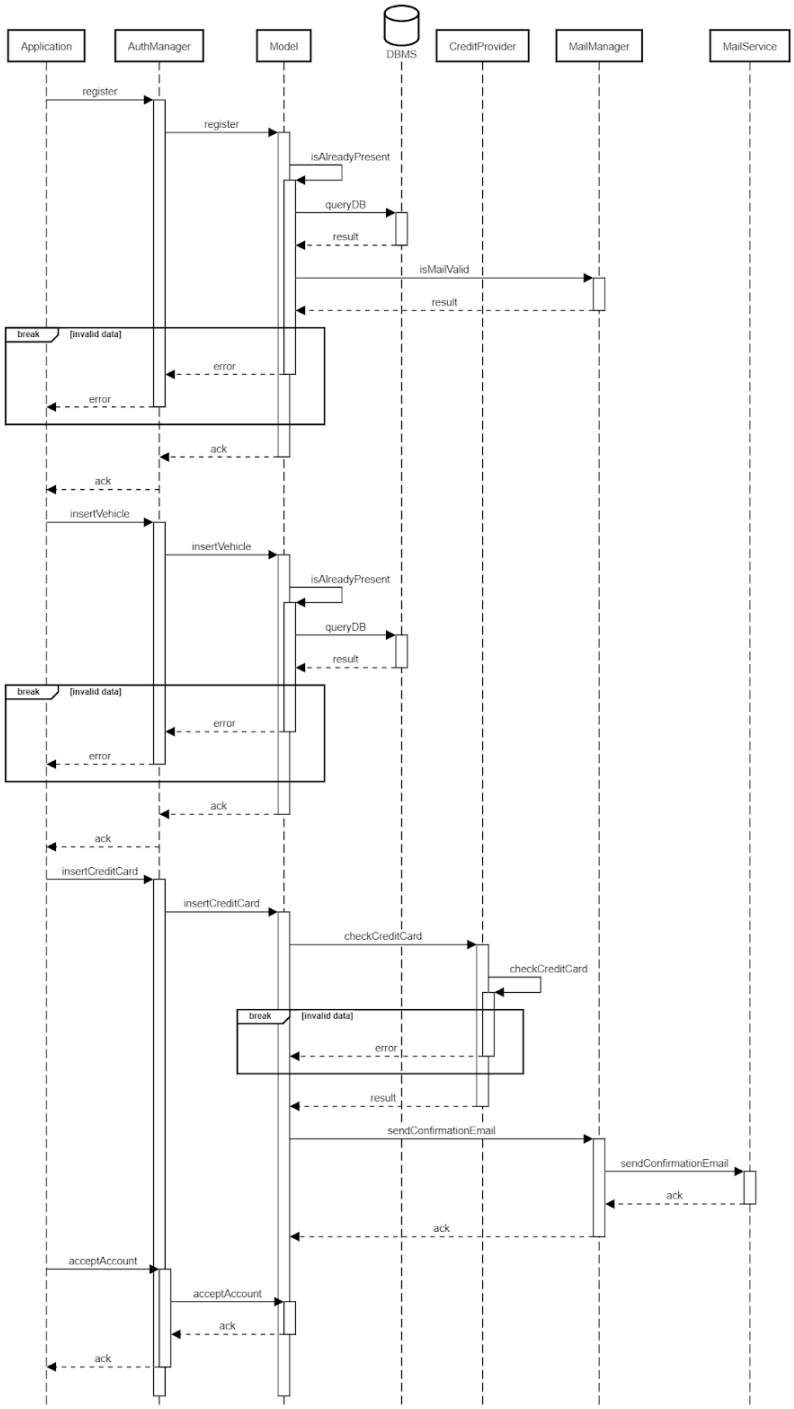
in case of end users or a PC with a web browser (for example, operating on Windows 10 OS) in case of CPOs.

- Tier2: It consists of a web server, which only accepts client requests, routes them to the application servers, and serves an HTML page to the client, who will then use client-side scripting to develop the page. It additionally adds the page's styling logic (CSS sheets, JS sheets, etc.).
- Tier3: It houses the application servers that power the system's primary features. This tier corresponds to the entire application layer and communicates with the client tier via APIs that are used by both native applications and web servers (in the case of webapps) (in case of mobile app download). Additionally, it uses the DBMS gateway to interface with the data tier.
- Tier4: It is composed by DMBS servers. In accordance with the instructions provided by the application servers, they store the data and perform operations on it.

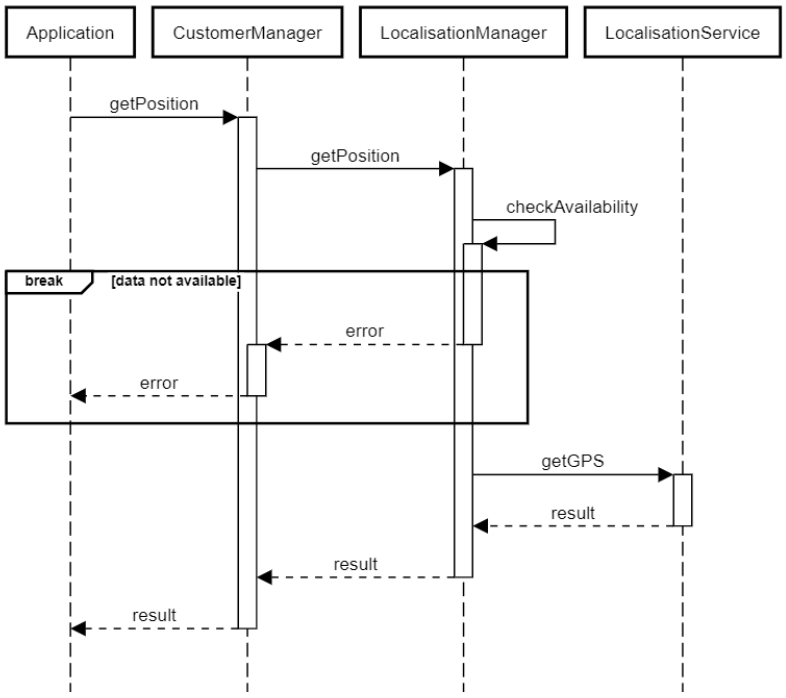
Note that the use of HTTP is required by the REST architecture.

2.d Runtime view

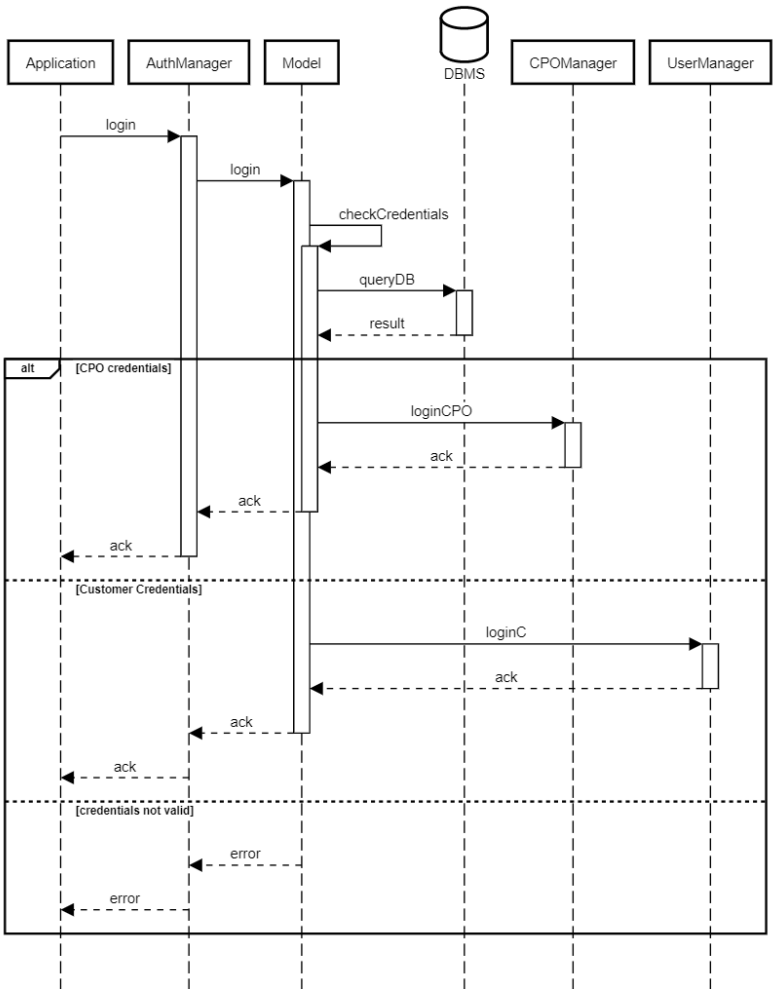
User Registration A user can sign up into the app by filling the requested information. The system than checks if the user is already present in the database and, if not, sends an a verification email to the user. After the email check the user is asked to insert his vehicle and payment information and then he can finally start using eMall.


Application
AuthManager
Model
DBMS
CreditProvider
MailManager
MailService
register
register
isAlreadyPresent
queryDB
result
isMailValid
result
break
[invalid data]
error
error
ack
ack
insertVehicle
insertVehicle
isAlreadyPresent
queryDB
result
break
[invalid data]
error
error
ack
ack
insertCreditCard
insertCreditCard
checkCreditCard
checkCreditCard
break
[invalid data]
error
result
sendConfirmationEmail
sendConfirmationEmail
ack
ack
acceptAccount
acceptAccount
ack
ack

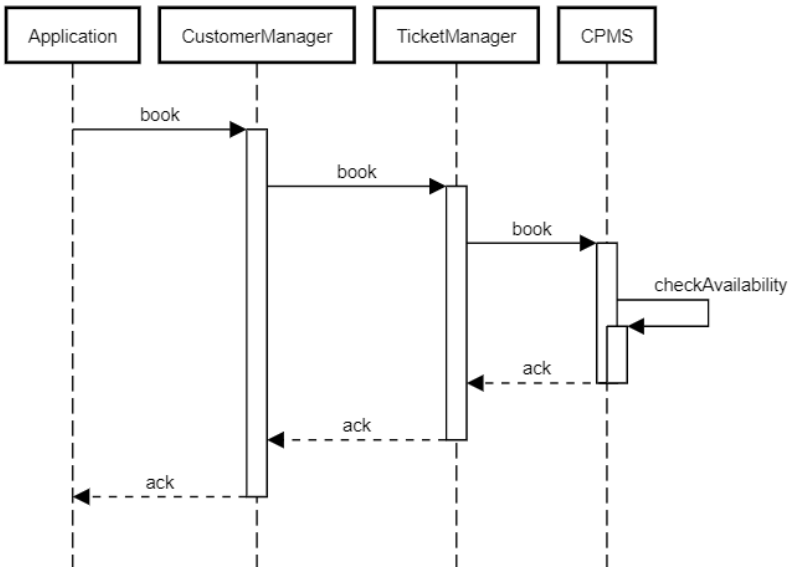
GetPosition The getPosition process is used at runtime, by the system, to check, through an API the GPS position of the user. In case of eMall this process is fundamental in order to show and suggest the user the best charging solutions in the nearby.



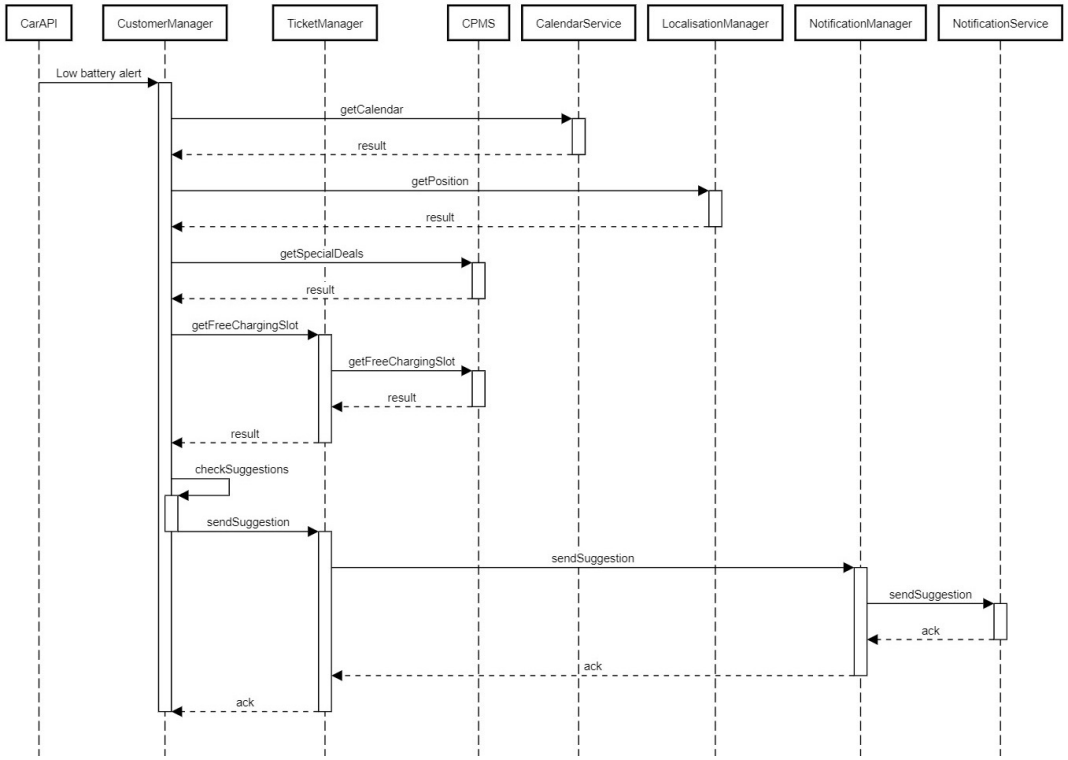
Login A user can login by inserting his credentials in the opening page of the app. The system then check the validity of the credentials in the database in order to let the user to log in. Otherwise an error is showed.



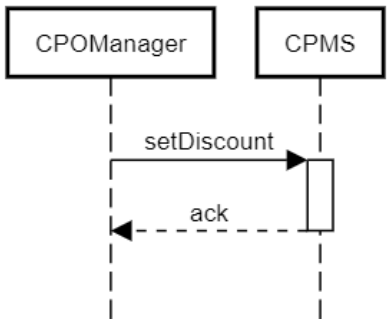
Book a ticket A user that wants to charge is car can start a booking process. When a user selects a charging point only the available slots of that station are shown, retrieved from the CPMS.



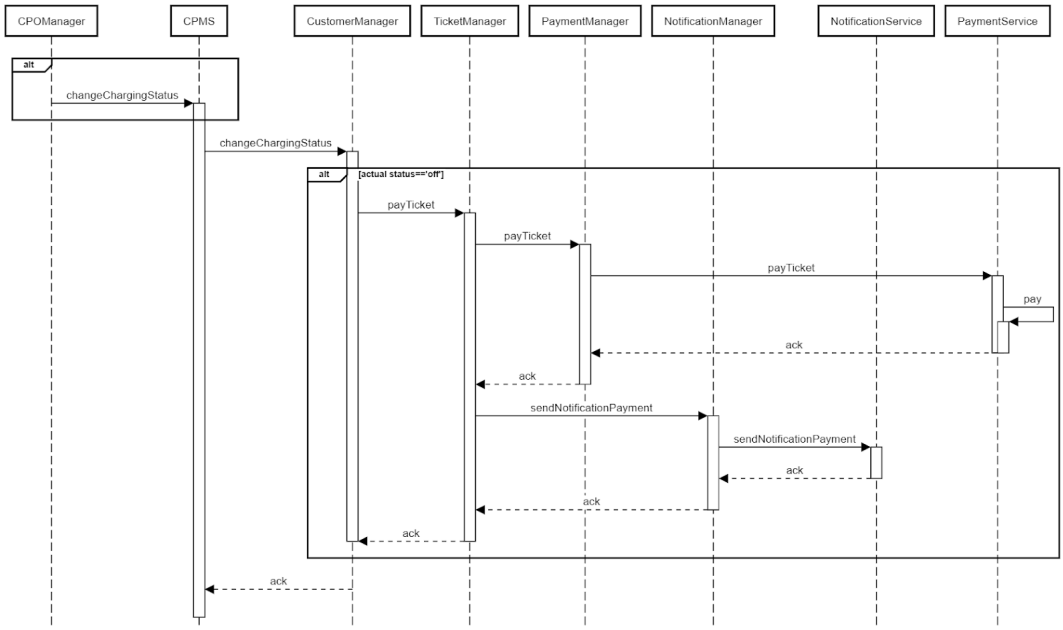
Charging Suggestion The smart eMall system is able to suggest the user charge booking for his car. The system controls the battery status of the car, the location of the activities of the user from the calendar and the availability of slots in the station nearby. Then suggests the user a possible booking that would perfectly fit in his day.



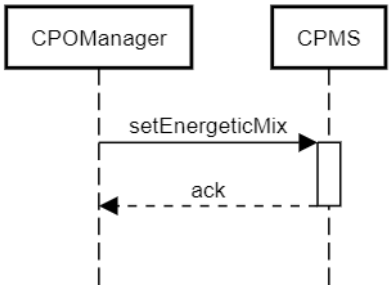
Set discount A CPO can create special offers for his customers by decreasing the price of energy for a limited amount of time.



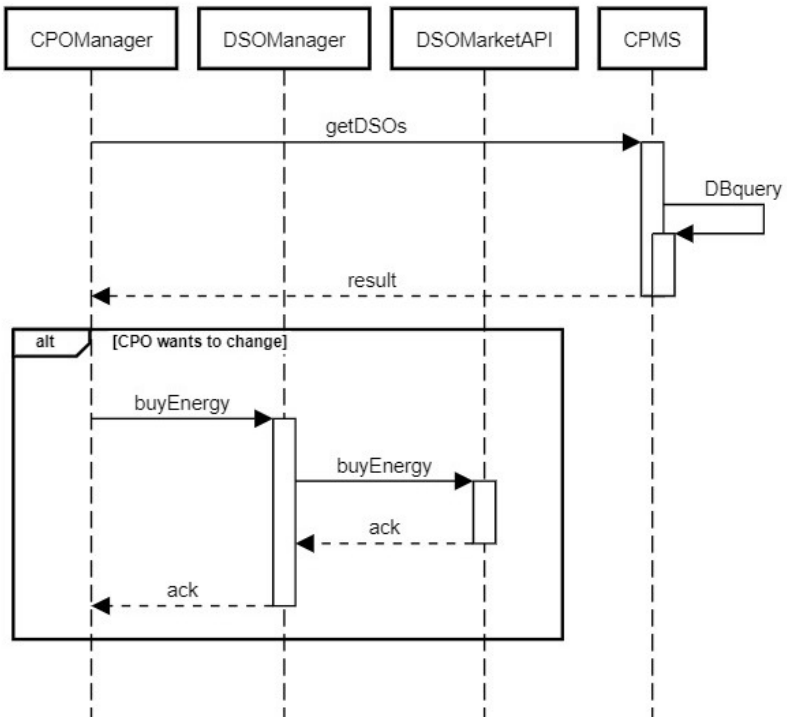
Change charging status A CPO can force the current charging status of a pump. This functionality can be used in case the charging system doesn't work as expected. The stop immediately ends the charging process and pay the ticket for the user, sending a notification for the completed process.



Set energetic mix A CPO can manually change the energetic mix of his station in order to set more or less charging power for charging to his batteries or to the energy coming from DSOs.



Buy energy from DSO A CPO can buy energy from DSO through the DSO market shop. If a DSO, with his related energy, is available, the DSO can buy energy from him.



2.e Component interfaces

In this section we provide the main methods of each interface previously presented in the high-level component view.

```
public interface AuthInterface {
        void register(String mail, String password);
        boolean insertVehicle(Vehicle vehicle);
        boolean insertCreditCard(String cardNumber);
        boolean sendConfirmationEmail(String dest);
        User acceptAccount(User user)
        User login(String mail, String password, String
          ↪ role)
```

```
}
public interface ModelInterface {
        void register(String mail, String password)
        Vehicle insertVehicle(String mail, Vehicle
          ↪ vehicle) #the vehicle is associated to
          ↪ the mail
        boolean isAlreadyPresent(String email)
        boolean isAlreadyPresent(Vehicle vehicle)
        boolean insertCreditCard(Card card)
        boolean sendConfirmationEmail(String dest)
        boolean checkCreditCard(Card card)
        User acceptAccount(User user)
        User checkCredentials(String mail, String
          ↪ password, String role)
        CPOManagerInterface loginCPO(String mail,
          ↪ String password)
        CustomerManagerInterface loginC(String mail,
          ↪ String password)
        void setCalendar(User user, Calendar calendar)
        void setNewDSO(CPO cpo, DSO dso)
}
public interface MailManagerInterface {
        boolean isMailValid(String mail)
        boolean sendConfirmationEmail(String dest)
}
public interface CustomerManagerInterface {
        GPS getPosition(User user)
        Calendar getCalendar(User user)
        Calendar getFreeChargingSlot(Calendar calendar)
        boolean sendSuggestion(User user)
        Ticket book(User user, Time time, Charger
          ↪ charger)
        boolean payTicket(Ticket ticket, User Buyer)
        void startCharging(Ticket ticket)
        void stopCharging(Ticket ticket)
        int getBatteryStatus(Vehicle vehicle)
        void getSpecialDeals()
```

```
}
public interface LocalisationManagerInterface {
        boolean checkAvailability(User user)
        GPS getPosition(User user)
}

public interface CPOManagerInterface {
        boolean changeChargingStatus(Charger charger)
        boolean setDiscount(Charger charger, int
          ↪ discount)
        boolean setPrice(Charger charger, int price)
        int getDiscount(Charger charger)
        boolean setEnergeticMix(Charger charger, float
          ↪ percentage, energy type)
        int getEnergeticMix(Charger charger, float
          ↪ percentage, energy type)
        boolean buyEnergy(DSO dso, float percentage)
        ChargerStatus getChargerStatus(Charger charger)
        List<DSO> getDSOs()
}
public interface TicketManagerInterface {
        Calendar getFreeChargingSlot(Calendar calendar)
        boolean sendSuggestion(User user)
        boolean checkTicketAvailability(Date date, Time
          ↪  time, Charger charger)
        Ticket book(User user, Date date, Time time,
          ↪ Charger charger)
        boolean payTicket(Ticket ticket, User Buyer)
        boolean sendNotificationPayment(Ticket ticket,
          ↪ User user)
}
public interface NotificationManagerInterface(){
        boolean sendSuggestion(User user)
        boolean sendNotificationPayment(Ticket ticket,
          ↪ User user)
}
public interface PaymentManagerInterface(){
        boolean payTicket(Ticket ticket, User Buyer)
```

```
}
public interface DSOManagerInterface(){
        boolean buyEletricity(DSO dso, float percentage
            ↪ )
        int getPrice(DSO dso)
}
```

2.f Selected architectural styles and patterns

Four-tired architecture We chose a four-tired architecture for many reasons:

- Load Distribution: having multiple application servers and a load balancer before them ensures a fair distribution of requests. Otherwise, a single node could experience over-requesting, which would bring the entire system to a halt.
- Flexibility: the internal logic of the system is dependent on the external logic once the system's interfaces have been specified. This finding implies that it is possible to improve one module without altering the others.
- Clients: different client types, including thick, thin, Web, and mobile clients, can share the same application logic thanks to the 4-tiered architecture.
- Scalability: a multi-tiered application ensures that only the most essential components will use the scaling strategy for the design. The final product maximizes performance while lowering expenses.

RESTful Architecture The restful application will be adopted both on web and mobile side. REST is an architectural style for software that outlines a set of guidelines for developing web services. RESTful web services are web services that adhere to the REST architectural design. By employing a consistent and predetermined set of rules, it enables requesting systems to access and change web resources. The six guiding principles or constraints of the RESTful architecture are:

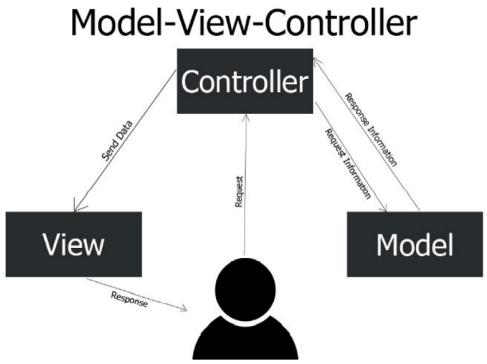
- Uniform Interface

- Client-Server
- Stateless
- Cacheable
- Layered System
- Code on Demand

The only optional constraint of REST architecture is code on demand. If a service violates any other constraint, it cannot strictly be referred to as RESTful. However, the attribute that prompts us to select this style of design is especially the Code on Demand. In fact, it allows the client to get some code snippets from the server and then execute them locally (usually in the web browser). This behavior ensures that the server will be under less computational load and that the service will be dynamic. The program is then designed to be created using client-side scripting, meaning that all page updates and requests are made on the client. By avoiding page refreshes after each activity, this approach further enhances the user experience.

Model View Controller (MVC) Model-View-Controller (commonly abbreviated as MVC) is a software design pattern that splits the relevant program logic into three interrelated pieces and is frequently used for creating user interfaces. By doing this, it is possible to distinguish between internal representations of information and the ways in which information is provided to and received by users. These three components are:

- Model: the central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application.
- View: generates the user interface, any visual depiction of data, like a table, chart, or diagram.
- Controller: responds to the user input and performs interactions on the data model objects. The controller receives the input, optionally validates it and then passes the input to the model.



2.g Other design decisions

Adoption of Identity Providers We decided to adopt some external IdP providers (such as Facebook, Google, etc.) in order to simplify the process of user registration, without asking him any additional information. This service is built on the provider's API, which will interact with our service to exchange the required data (such an email address).

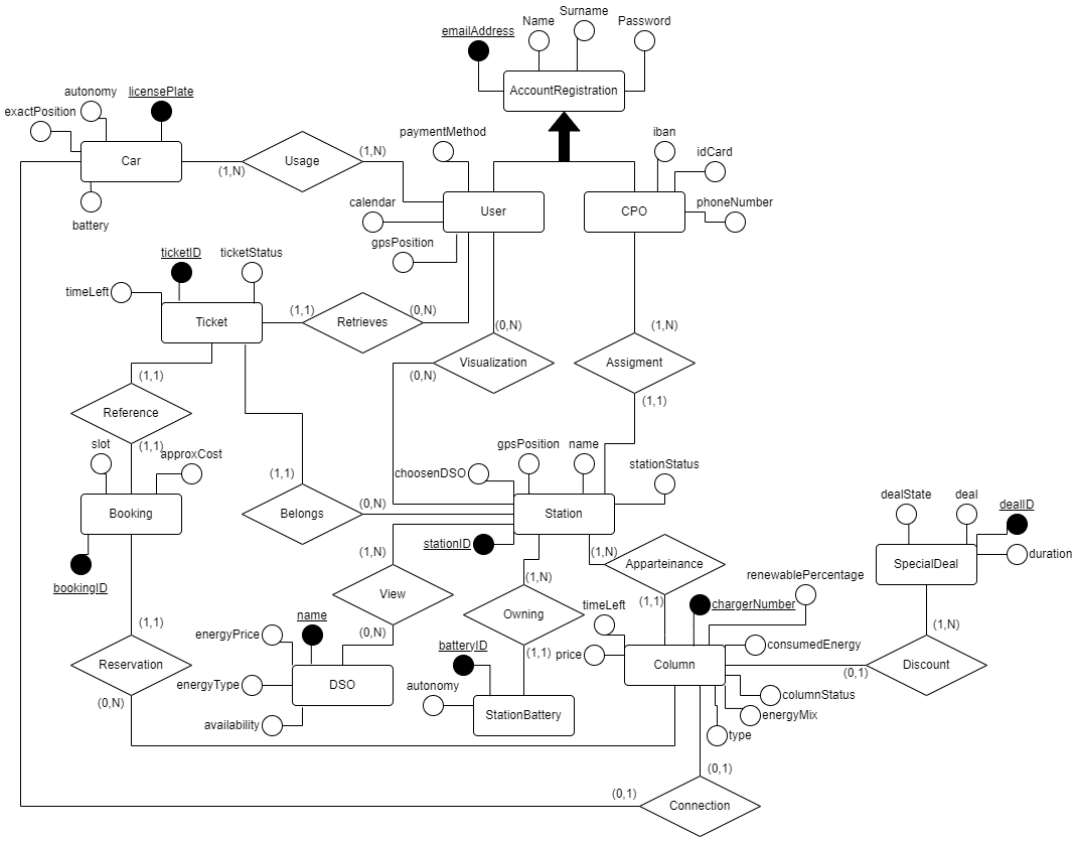
Adoption of notification service The adoption of a notification service enables third party application developers to send notification data to applications installed on mobile devices (e.g. Apple Push Notification). The notification information sent can include badges, sounds, newsstand updates, or custom text alerts.

Thin and thick client and fat server The web application will be the thin client. The goal of this architecture is to keep client-side information as little as feasible. It denotes that only the server side contains the business logic. A reliable connection between the components is the bare minimum need for this option; otherwise, the application would not function as intended. Of course, the biggest benefit of selecting this implementation approach is that a powerful client machine is not necessary. To avoid constant requests to the server (reduced computational burden), and to preserve information even when an Internet connection is not available, the best option in the case of mobile applications is to maintain essential personal information

on a local database. It is claimed that the client in the second instance is thick.

Scale-Out This technique involves cloning the nodes where we expect a bottleneck to occur in order to improve the overall system scalability. When the limits are reached, this option results in a larger deployment effort but a reduced hardware upgrade cost. In summary, the scale-out is a better route. The system needs a load balancer once it has been divided in order to correctly direct incoming requests to the node with the lightest workload.

Database Structure Given that this system adheres to the CAP (Consistency, Availability, and Partition tolerance) theorem and favors consistency and high availability, as mentioned in the RASD paper, MySQL was selected as the database management system.



The diagram in the figure shows a logic representation of what kind of data is stored in the internal database of the system.

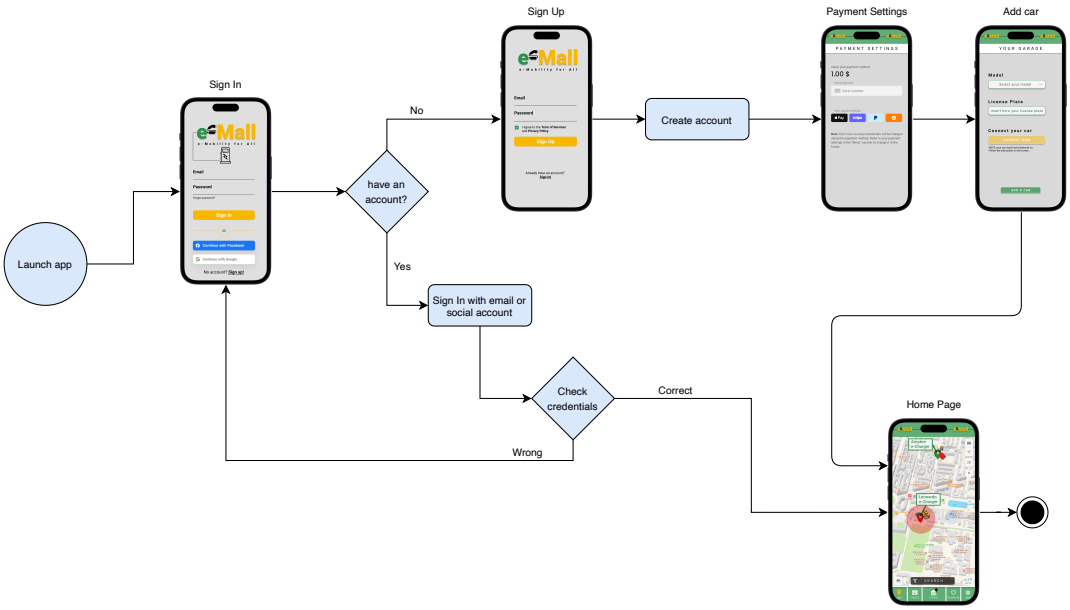
3 User Interface Design

The aim of this section is to show the design of the main screens of the eMall app. Through mockups we want to describe the main functionalities for which the application was intended. In particular each functionality is described through a realistic flow of actions that could be done by the final user. Note that in the following flow diagrams we used mockups presented in the RASD, but we added some other mockups in order to better clarify the user experience. With the main use cases in mind, we thought that the user interface should be based on a mobile application while CPO's one on a web application. We tried to reproduce the most useful flows, already described from the components point of view with the sequence diagrams in section 2.D. Since the interface of the CPOs is full of buttons and little components, we decided to condense the main features into just a few interfaces, since most of the changes wouldn't be appreciated or useful for a descriptive purpose.

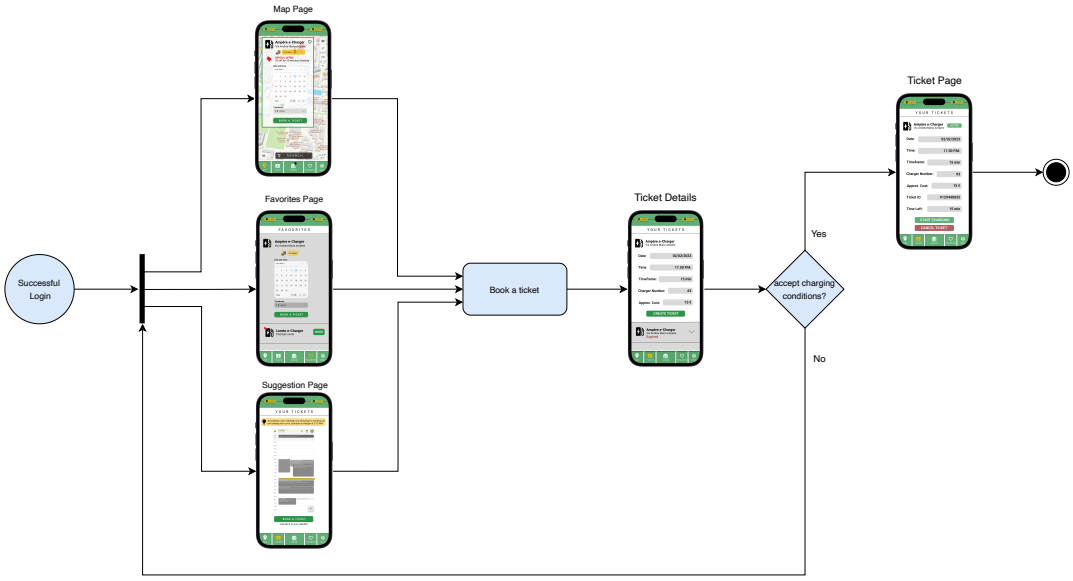
Note that all the functionalities are detailed in the Product Functions section (2.b) of the the RASD document.

3.a User Mobile Interface

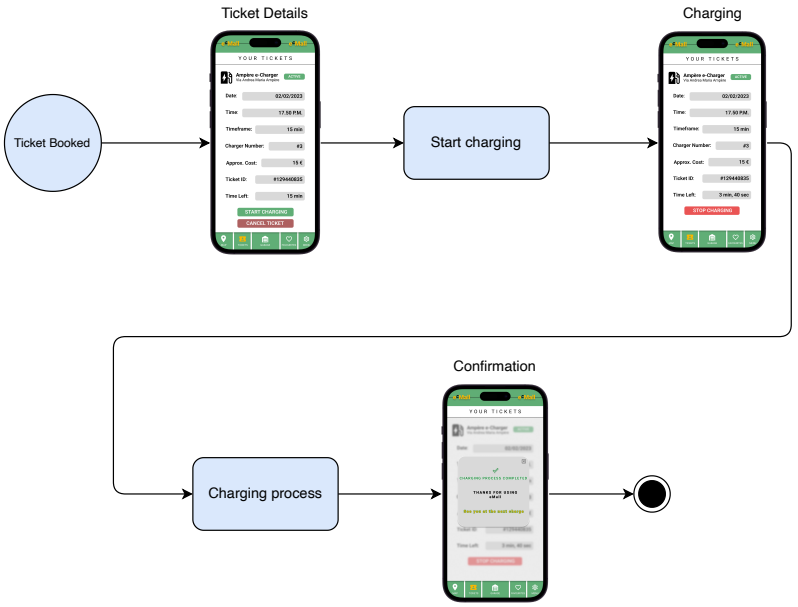
SignUp and SignIn Once in the opening screen a user can SignUp or LogIn to eMall. The SignUp process consists on the creation of the account through an eMail confirmation, the insertion of payment details and the addition of a car.



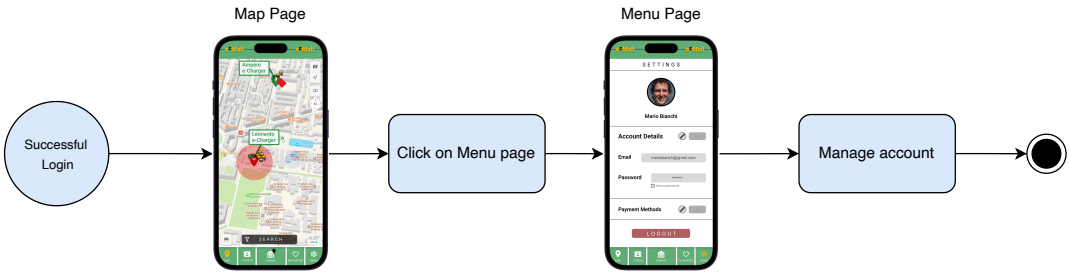
Book a ticket The booking process can be triggered from the map page, the favorite page and in case of a suggestion from the system. Then the ticket is created and the user can use it during his slot timeframe.



Charging Process Once arrived at the station and plugged-in the smart pump, the user can start the chargin process through the button. If the process ends, or the user decides to stop it before, a confirmation notification is sent ending the entire charging process.



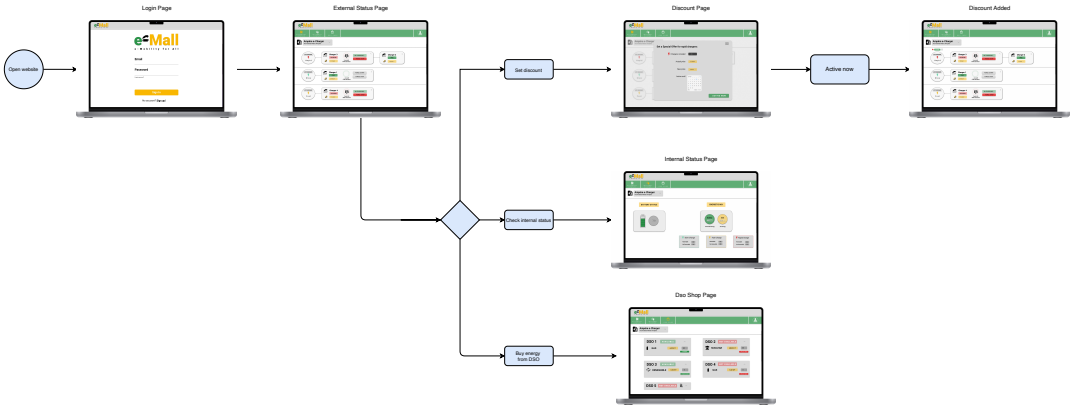
Account Management From the menu page the user can logout, change his personal information and his payment method.



3.b CPO Web Interface

Once logged-in the CPO can control the external status of the station, adding eventually special deals to specific types of chargers. Moreover the CPO can change his energetic mix (batteries and DSO energy) from the internal status

page. Finally a CPO can change provider and buy energy from different DSO from the DSO shop page.



4 Requirements Traceability

In this section we will map the requirements, already shown in the RASD, and the components described previously in the component diagram. In other words, we will show the components necessary to implement the functionalities of each requirement.

- R1: The system allows the user to sign up and provide their personal information, payment method and vehicle information
  - AuthManager: in order to sign up
  - EmailManager: in order to send a confirmation email to the user
  - PaymentService: in order to check the payment method
  - Model: in order to save data about the user
  - IdentityProviderService: in order to let the user sign up through external services (e.g Google, Facebook)
- R2: The system allows the user to log in by entering the credentials used at the time of registration.
  - AuthManager: in order to log in
  - Model: in order to check the credentials of the user
- R3: The system must regularly update charging station data regarding cost and special offers
  - Model: in order to manage the update of the charging station data
  - CPOManager: in order to check and eventually update data about charging stations
- R4: The system allows the user to give permission about GPS location
  - LocalisationManager: in order to check the user availability and get his position
  - CustomerManager: in order to update the current position of the user
- R5: The system shall allow users to select a day, time and timeframe slot from the available ones

  - TicketManager: in order to check the ticket availability
  - CPOManager: in order to get the status of the chargers of the selected charging station
  - Model: to retrieve information about tickets available

- R6: The system generates a TicketID associated to a charging reservation
  - TicketManager: in order to create the unique TicketID during the booking process
  - Model: in order to update the ticket the ticket information

- R7: The system computes a prediction of expected cost of the charge, based on the timeframe and charging info and lets the user pay automatically at the end of the charge.
  - TicketManager: in order to insert the information in the ticket
  - Model: in order to retrive information about the cost of the charger and calculate the estimation

- R8: The system enables users to insert the license plate onto a smart pump.
  - Model: to check information about the license plate of the user

- R9: The system enables users to press the "start charging" button on the application
  - CustomerManager: in order to start the charging process

- R10: The system allows the user to connect its own calendar to the e-mall application
  - Model: in order to save the calendar related to the user

- R11: The system recommends the user a ticket reservation in case it detects car's battery need by elaborating the best charging time to fit its calendar
  - CustomerManager: in order to get the possible charging slots from the user's calendar and send a suggestion

  - TicketManager: in order to check chargers' availability
  - CalendarService: in order to retrieve information about his daily schedule
  - NotificationService: in order to send user suggestions through popups

- R12: The system allows the CPO to know main information (occupied, type of charging, cost, time remaining, energy consumed, energy mix)
  - CPOManager: in order to call CPMS information module
  - CPMS: in order to get the main information about the internal status of the station
  - Model: in order to retrieve data about chargers' information

- R13: The system allows the CPO to know main information related to the whole station (battery, status, number of cars in charge)
  - CPOManager: in order to call CPMS information module
  - CPMS: in order to get the main information about the external status of the station
  - Model: in order to retrieve data about the station

- R14: The system allows the CPO to see the price of energy from the DSO
  - CPOManager: in order to call CPMS module
  - CPMS: in order to get information from DSOs
  - DSOMarketManager: in order to get information about the current price of energy

- R15: The system allows the CPO to change the DSO provider and the energy mix for the whole station (all charging points)
  - CPOManager: in order to call CPMS module
  - CPMS: in order to get information from DSOs
  - DSOMarketManager: in order to change the provider of energy

  - Model: in order to save the data of the new energetic mix and the DSO provider

- R16: The system allows the CPO to insert special offers
  - CPOManager: in order to call the CPMS module
  - CPMS: in order to set special offer
  - Model: in order to save the settings of the new prices

5 Implementation, Integration and Test Plan

5.a Overview

This section concerns the implementation, integration and test plan of the eMall system. The software-to-be will be divided, as already described in the Component View (section 2.B), as follows:

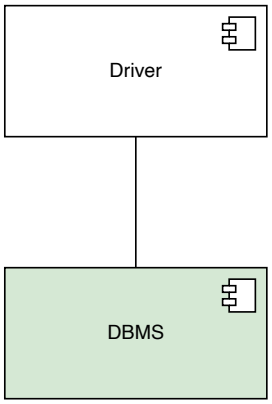
- Client: both web interface for CPOs and mobile application for users
- Web Server
- Application Server
- Internal Database
- External Services: services like GoogleMaps, payment providers, etc.

The implementation, integration of the system will follow a bottom-up approach, in order to avoid stub structures that would be more difficult to implement and test. Also the testing will follow a bottom-up approach so that all the four main phases of unit, integration, system and acceptance testing will be conducted. The integration will be incremental in order to make bug tracking easier. To increase the overall robustness the testing process will start in parallel with the implementation, since components will be implemented, tested and integrated in a hierarchical order, from bottom to top level. The main focus is on the module of the application server, because of its importance in the software development and also due to its testing difficulty. Server side and the client side will be implemented and tested in parallel, in order to immediately check for inconsistencies, since they strongly rely on the other. For external services we assume they have already been tested and they are reliable so we don't introduce further testing.

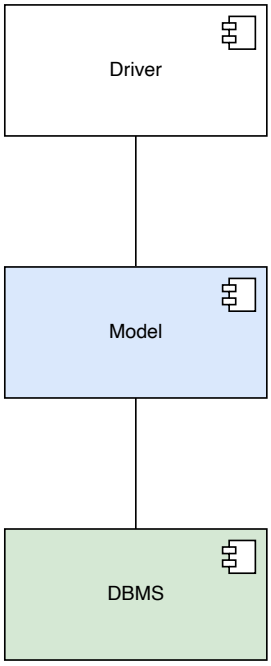
5.b Plan execution

This section describes the implementation, integration and test plan for both server and client, since, as said before, the two are thought in parallel. For readability purposes we will describe the server-side of the implementation and integration and we will add client-side applications in the final diagrams in order to show the whole system. Note that each unit test is incrementally

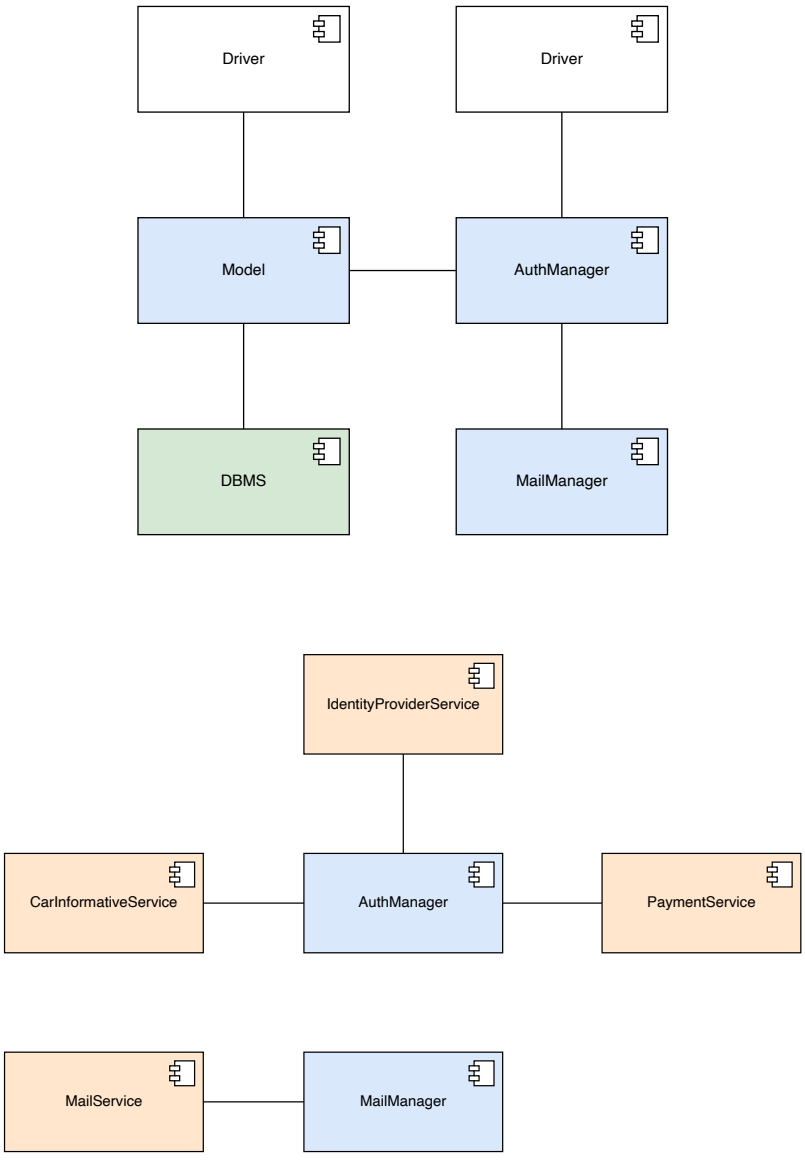
done through drivers, for the components that are still under development. The unit test are performed during the whole process in order to maintain control over the entire system. Note also that, for clarity purposes, we decided to represent the external APIs service managers, connected to our components, in different diagrams. All the components described in this document rely on DBMS of the internal database, which must be implemented as the first component, in order to check possible misconfigurations or incorrectness in the interface.



Immediately after the Model is implemented since it is responsible for all the interactions with the database and is needed by most of the other components to fulfill their functionalities.

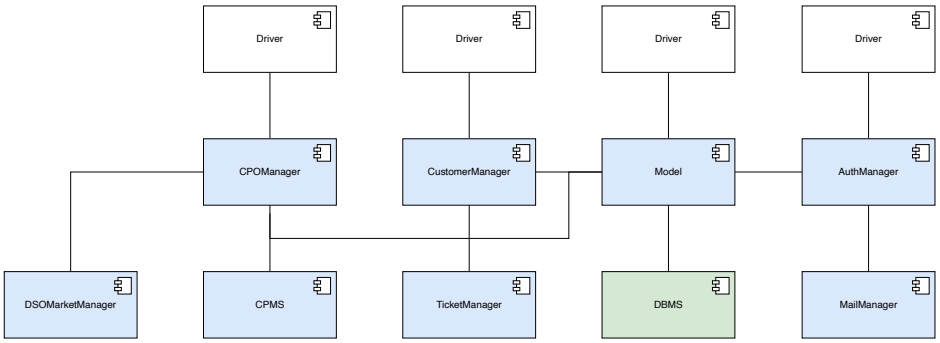


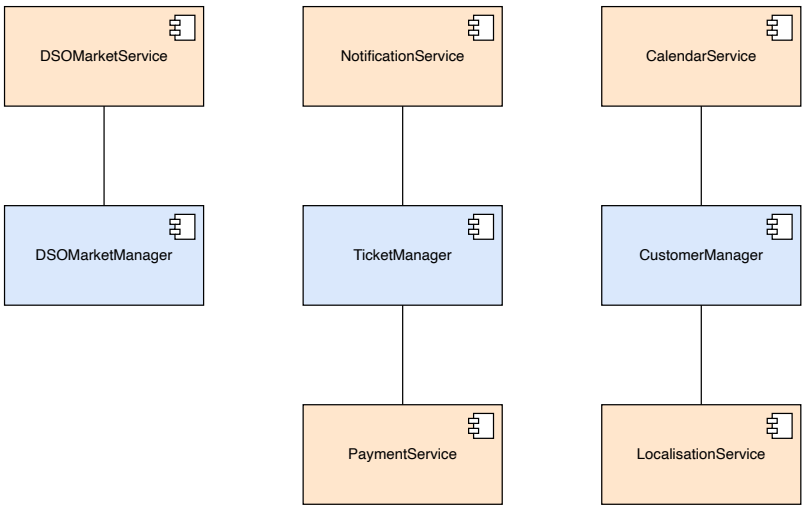
Then we start with the most important functionalities of our software: Login and SignUp. For this reason we implement all the components related to authentication, in order to give priority to the access to the app, since it will be required by all the other components. Since the two modules are strictly related, for testing purposes, it would be a good idea to execute a test on the Sign Up module, immediately followed by one on the Login module. Moreover the test should use a demo of the DBMS, containing some pre-existing accounts, in order to check even more corner cases. In this phase we could also add the implementation and test of the Identity Providers' external service, in order to simulate another use case. Moreover the implementation and test of the PaymentService and CarInformativeService should be considered in order to control all the scenarios related to the SignUp process.



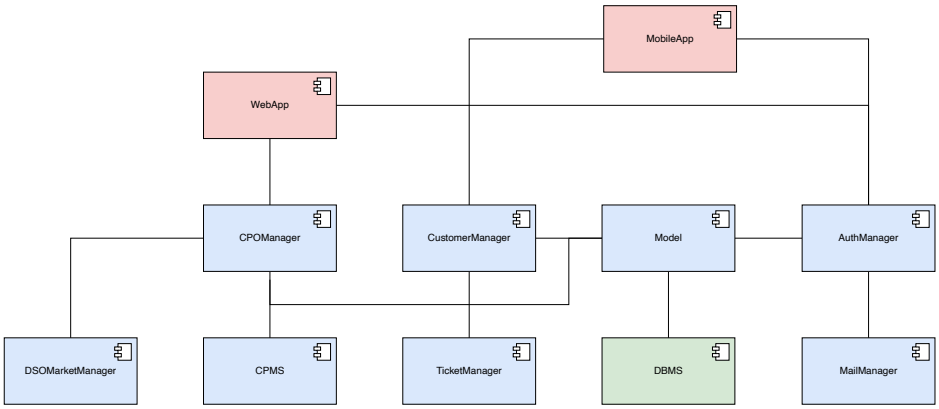
The next step is the implementation of the core modules of the application: the CPOManger and the CustomerManager. These modules have to be integrated and tested since they contain the interface with the users. In this phase we need also to consider the implementation of the TicketManager, the CPMS and the DSOMarketManager since they are strictly related to the

functionalities that could be triggered by CPOs or customers. In this case the best idea is the use of automated test class, eventually with simplified stub extension in order to make the testing process easier. In this way we can simulate realistic flows of actions and check if all the called modules act properly. In this phase we also add the implementation and test of other external services such as: MapsService, DSOMarketService, CalendarService and NotificationService.





The final step is the implementation of the Mobile App and WebApp to test the whole system.



5.c Non-Functional Testing Phase

After unit and integration testing the system should also be tested for non-functional requirements. The most important parameters to be tested in this phase are reliability and availability and this tests could be performed in different ways, such as:

- Volume testing: in order to test the application with a huge amount of data in the database and see the performances on it.
- Load testing: in order to check how the application performs with pre-loaded user data and simulate concurrent users' uses. The main objective is to identify eventual bottlenecks before the release of the application.
- Stress testing: in order to test the application under extreme workloads and see how it handles high data processing. The objective is to identify the breaking point and the limits of the application.

6 Effort Spent

Student	S.1	S.2	S.3	S.4	S.5
Valeria Amato	1h	13h	2h	3h	2h
Francesco Dettori	1h	13h	2h	3h	2h
Matteo Pancini	2h	5h	5h	7h	7h