# DD

**Design Document**
**a.a. 2022/2023**

Version: 2.0
Date: 08/01/23

**Eutizi Claudio**
Person ID: 10812073
Student ID: 995635
email: claudio.eutizi@mail.polimi.it

**Perego Gabriele**
Person ID: 10488414
Student ID: 987104
email: gabriele2.perego@mail.polimi.it

# Contents

# 1 Introduction

## 1.1 Purpose

The purpose of the following **Design Document (DD)** is to deal with the architectural description of the *eMall* system and its services in order to provide more technical details about the project. While the previous RASD provided an abstract overview of the goals of eMall application, this document will focus on an in-depth description of the system's components, the interactions between them and their deployment. Then, it will be shown how the presented components will satisfy the Requirements already expressed in the RASD. This document will be also a guide for the implementation and testing phases.

## 1.2 Scope

As previously explained in the already published RASD, the *eMall* system wants to be a service that supports the charging process of electric vehicles, both for Drivers and for the Charging Point Operators. For this reason, the eMall system has to be able to satisfy needs of different kinds of Users and that is why its functionalities have been divided into two subsystems: *eMSP* and *CPMS*.

The first one is dedicated to provide the end-user services e.g. know the position of nearby charging stations, book a charge in a charging station, charge an electric vehicle and pay for the service etc.

The CPMS system administers the IT infrastructure of a CPO and provides a simple access point to the CPOs Operators that want to monitor the charging stations status, make decisions and apply changes e.g. handle the distribution of energy to the vehicles connected to a charging station, monitor the status of the charging station, decide from which DSO to acquire energy, set prices and offers etc.

In order to avoid the Operators to do too much repetitive operations, some of the listed operation can be also automatically performed by an autonomous system installed in the CPMS infrastructure. As already specified in the RASD, This autonomous system is sometimes mentioned, but its implementation and functionalities are not considered as part of the scope of these R&DD documents; it is, in fact, considered as an already existing and working subsystem that operates autonomously, but whose role is subordinate to the Operator's intervention, that is what this project is focusing on and has always an higher priority.

## 1.3 Definitions and abbreviations

### 1.3.1 Acronyms

- **RASD**: Requirement Analysis and Specification Document.
- **DD**: Design Document.
- **CPO**: Charging Point Operator.
- **eMSP**: e-Mobility Service Provider.

- **CPMS**: Charging Point Management System.
- **DSO**: Distribution System Operator.
- **QR Code**: Quick Response Code.
- **UML**: Unified Model Language.
- **API**: Application Programming Interface.
- **OS** : Operating System.
- **DBMS** : Data Base Management System.
- **HTTP** : HyperText Transfer Protocol.
- **HTTPS** : HyperText Transfer Protocol over Secure Socket Layer.
- **REST** : Representational State Transfer.
- **CRUD** : Create, Read, Update, Delete.
- **SQL** : Structured Query Language.
- **PaaS** : Platform as a Service.
- **MVCS** : Model-View-Controller-Store.
- **GUI** : Graphical User Interface.

### 1.3.2 Definitions

- **Client** : Software System on the user's device that requests services to the Server.
- **Server** : Software System that handles requests from different clients.
- **n-tier** : Distributed architecture composed of n hardware components, each containing one or many layers.
- **n-layer** : Distributed architecture composed of n software levels, each distributed on one or many tiers.
- **Design Pattern** : Reusable software solution to a commonly occurring problem within a given context of software design.
- **DBMS** : System Software for creating and managing databases.
- **Maps API** : Software platform that provides accurate real-time data in order to geolocate Users.
- **Business Logic Layer** : Layer which manages the services that eMSP and CPMS offer to Users (Drivers and Operators).

- **Presentation Layer** : Layer which manages the interaction between application and User.It contains graph interfacing modality and information rendering and it is accessible by the User's own graphical interface.
- **Data Access Layer** : Layer which manages the application's data and database access.

### 1.3.3 Abbreviations

- $[R.n]$ = n-th functional requirements.

## 1.4 Revision history

| Version | Date | Details |
|---|---|---|
| 1.0 | 04/01/23 | DD first draft |
| 2.0 | 08/01/23 | DD final version |

Table 1: revision history

## 1.5 Reference Documents

| Title | Authors |
|---|---|
| R&DD Assignment AY 2022-2023 | M. Camilli, E. Di Nitto, M. Rossi |
| RASD | C. Eutizi, G. Perego |

Table 2: table of documents and references

- ISO/IEC/IEEE 29148:2018(E)
- StarUML for diagrams: StarUML webpage
- Draw.io for schemes and diagrams : Draw.io webpage
- For Deployment part: Deployment documentation

## 1.6 Document Structure

- **Introduction** (section 1): this section presents a general overview of the DD. It provides an overall description of the purpose and scope of the project and summarizes what will be presented in the following chapters.
- **Architectural Design** (section 2): this section details all the system level components. It presents a description of some of the architecture views by the mean of several UML diagrams. In particular the focus will be on the component, deployment and runtime view. The rationales behind the architectural styles and the design patterns adopted close this chapter.
- **User Interface Design** (section 3): this section is dedicated to the User Interfaces, but we have already included this part in the RASD, Section 3.1.
- **Requirements Traceability** (section 4): In this section each requirement of the RASD will be mapped with the design component that satisfies it.
- **Implementation, Integration and Test Plan** (section 5): this section aims to provide a plan that must be followed for the implementation of the components, their integration and the related testing.
- **Effort Spent** (section 6): this section shows a table that records the time spent by each group member for each section working on the DD part.

# 2 Architectural Design

## 2.1 Overview: High-level components and their interaction

The functionalities of eMall, as already mentioned in the RASD, will have different target devices with respect to its subsystems: a mobile application (Android, iOS) will provide Drivers the access to the eMSP functionalities, while both mobile and web applications will be developed for what concerns the CPMS system services. The eMall System will be designed as a distributed application, following the classic **multi-tier architecture**. In this way, each of the layers can be independently modified, giving scalability and maintainability to the application. Also, the decision to use a multi-tier architecture was made in order to separate the business logic of the system from the data, which could be used for other applications in the future, and so decoupling them facilitates that.

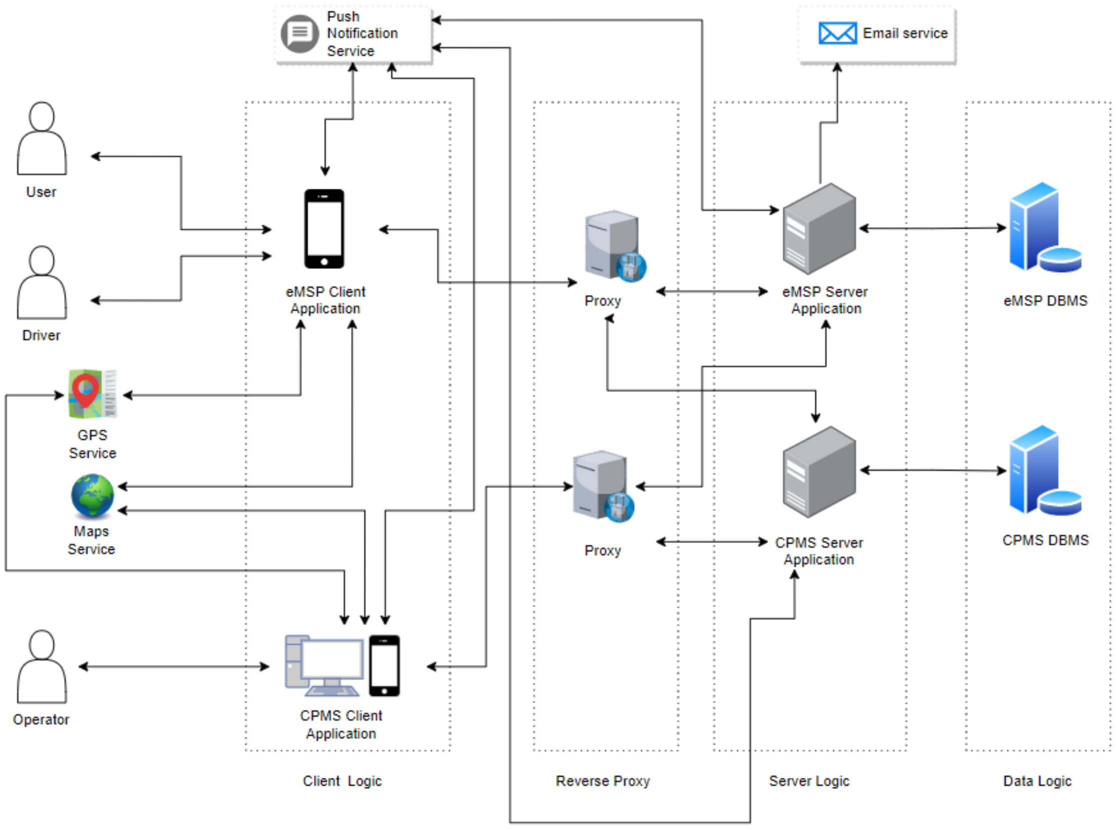


Figure 1: Overview of the System

The above image (fig. 1) provides a general overview of the System's architecture.
We can immediately identify two horizontal layers and four vertical ones. The horizontal layers correspond to eMSP and CPMS architectures which have their Client Application, Proxies, Application Servers and DBMS Systems.
They also communicate to each other: eMSP asks information to CPMS's functionalities by performing requests to its proxy and vice versa. With the vertical layers, we want to separate the Client Logic side, that includes the Client Applications, from the Reverse Proxy side, from the Server Logic side, including the Application Server and the Reverse Proxy, and from the Data Logic side. Finally, also all external services are included in the diagram. The Client is connected to GPS, Maps and Push Notification Services. Both the eMSP and CPMS Servers are connected to the Push Notification Service and only the eMSP is also connected to Email Service.

## 2.2 Component view

Figure (fig. 2) shows the Main Component Diagram of eMall with its interactions.
The figure (fig. 4) shows the CPMS Component Diagram and the figure (fig. 3) shows the eMSP Component Diagram.
The purpose of these UML diagrams is to show the internal architecture of the System's software. It is globally divided into five macro components:

- eMall Application Client
- eMSP Business Logic
- eMSP external Database
- CPMS Business Logic
- CPMS external Database

In addition, some of these macro components are connected to external services through dedicated APIs, and each business logic component communicates directly with its database. However, eMSP's business layer also communicates with CPMS's services through its APIs, and also CPMS's business layer communicates with the Charging Stations and with the DSO through dedicated APIs. These main layers are internally organized in modular components, each providing specific services.
Components communicate between each other by providing interfaces that expose all the methods that other components may require. In our diagrams, this is represented internally to a component by assembly connectors ball-and-socket and externally by delegation connectors, required interfaces and provided interfaces.
Below the main component diagram, including both the eMSP and the CPMS subsystem, is shown. In the next figures, each of the subsystems will be more in detail described, focusing on one component at a time and punctualizing its functionalities and relationships with the other components.

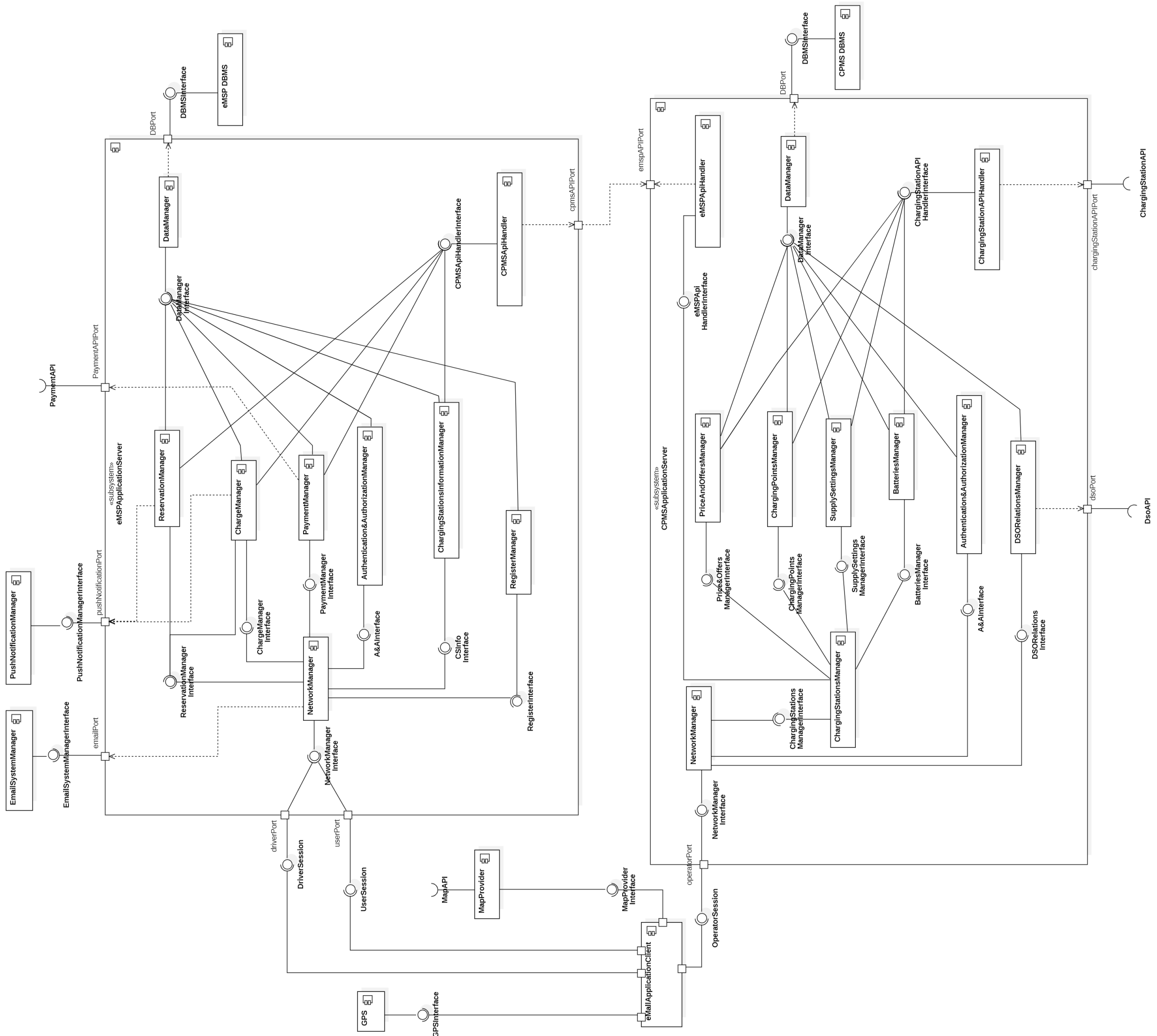


Figure 2: Main Component Diagram

### 2.2.1 eMall Application Client

The main component is represented by the **eMallApplicationClient**, which is located on the User's device.

### 2.2.2 eMSP Business Logic

This component is devoted to the implementation of eMSP's core logic. It is a stateless module that lies between the Client application and its Database. It gathers all components that interact with each other to provide the System's functionalities. We will now detail how each component is structure and what is its purpose.

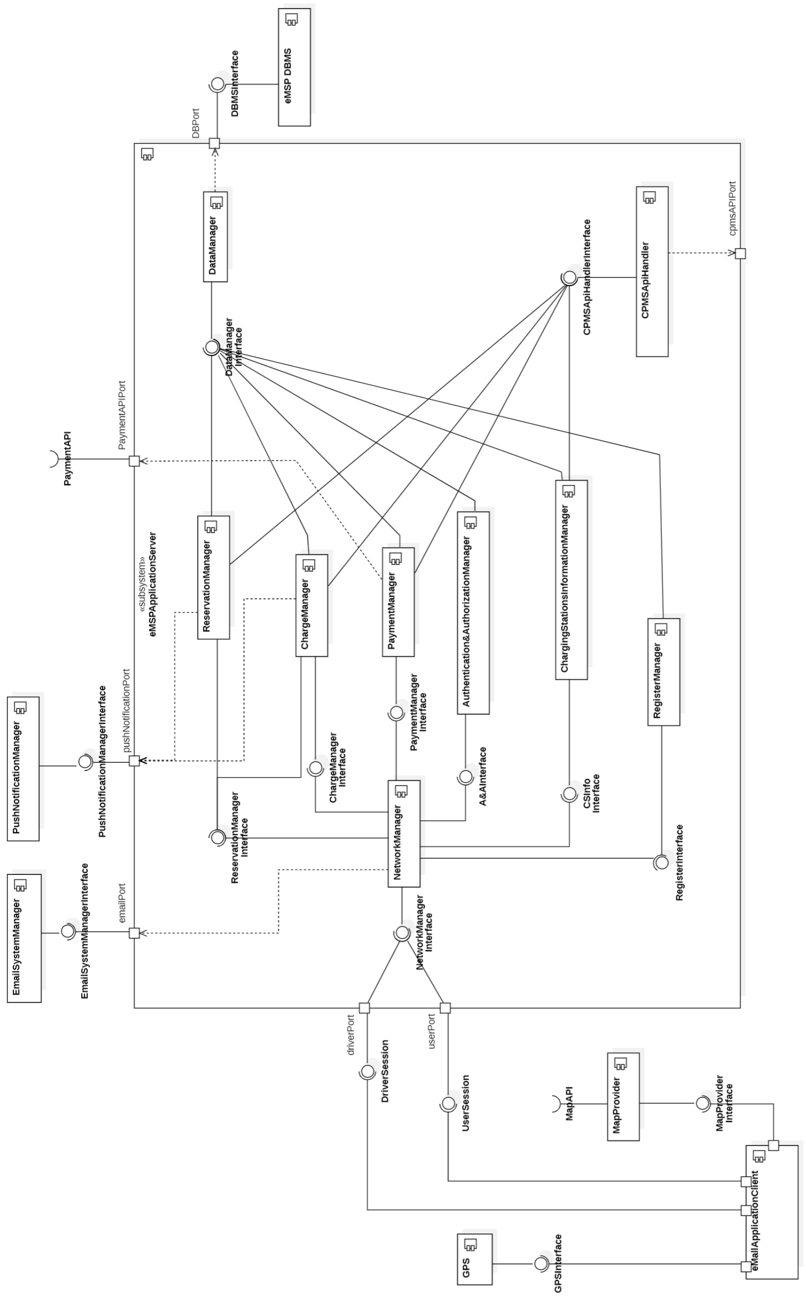


Figure 3: eMSP Component Diagram

- **NetworkManager**: it is the access point to the Business Logic and handles all the requests coming and going from/to the ApplicationClient. It dispatches all the incoming messages coming from the client application to the specific component that is in charge of handling the request. Regarding the outgoing messages, it gathers all the messages and send them using HTTP protocol to the client through the port.

- **ReservationManager**: component responsible for creating and managing a socket reservation. It gathers all the data needed to create a reservation and communicates with the *CPMSApiHandler* in order to transmit the reservation data to the related CPMS system. It also deals with the generation of QR-Codes, that need to be transmitted to the Driver, and reservation tokens that the CPMS needs in order to handle the reservations coming from the eMSP subsystem.

- **ChargeManager**: component that provides all the functionalities in order to handle the charging process. When a Driver scans a reservation QR-Code, this component receives a message from the *CPMSApiHandler* in order to create a charge instance and to communicate with the other internal components in order to update the reservation status and to keep the driver up to date about the information of his/her active charging.

- **PaymentManager**: component that exposes all the methods related to the payment phase. It provides the functionalities in order to add a payment method in the System and to pay for the charging service. It communicates using uniform APIs with the bank or the credit card companies in order to check the data inserted by the Driver and to request money reservations and payments.

- **Authentication&AuthorizationManager**: component responsible for the authentication and authorization of a Driver.

- **ChargingStationInformationManager**: this component deals with obtaining information from the CPMS subsystems about the charging stations for visualization purposes.

- **RegisterManager**: component responsible for the registration of a new Driver in the eMSP system.

- **DataManager**: this component provides the methods for the interaction between the eMSP Business Logic and the DBMS. All the internal functional components are connected to it in order to forward requests that need the interaction with the Data Logic tier.

- **CPMSApiHandler**: component that dispatches all the requests and messages between the eMSP subsystem with the CPMS subsystem. Any information that have to go from eMSP to CPMSs, or vice versa, passes through this component.

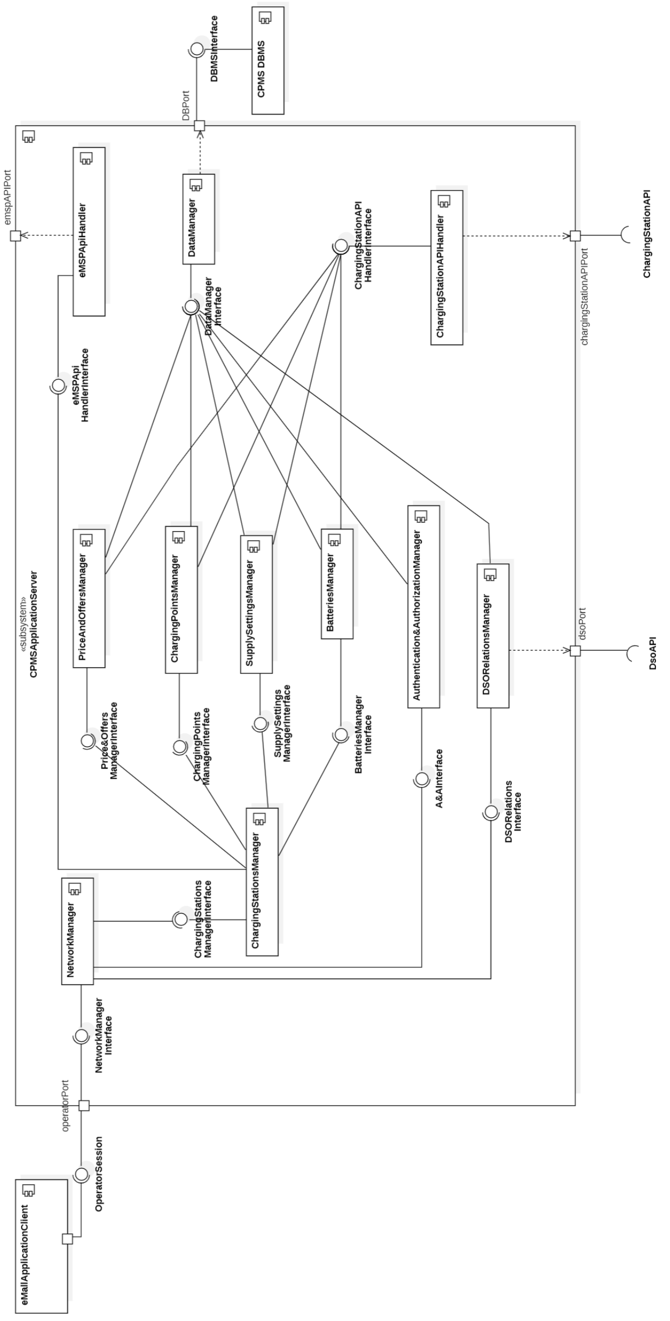


Figure 4: CPMS Component Diagram

### 2.2.3 CPMS Business Logic

This component is devoted to the implementation of a CPMS core logic. It is a stateless module that lies between the Client application and its Database. It gathers all components that interact with each other to provide the CPMS subystem's functionalities. It will now be described each component's structure and purpose.

- **NetworkManager**: as for the eMSP NetworkManager component, it is the access point to the Business Logic and handles all the requests coming and going from/to the ApplicationClient. It dispatches all the incoming messages coming from the client application to the specific component that is in charge of handling the request. Regarding the outgoing messages, it gathers all the messages and send them using HTTP protocol to the client through the port.
- **ChargingStationManager**: it handles requests from the NetworkManager component that involve the participation of components responsible for the charging stations management. It forward the Client requests to the specific charging station component that is required to satisfy the request.
- **PriceAndOffersManager**: component responsible for the management of price and offers of the charging stations.
- **ChargingPointsManager**: component responsible for the visualization and modification of the data about a certain charging station's charging points.
- **Authentication&AuthorizationManager**: component responsible for the authentication and authorization of an Operator to access the system and use its functionalities.
- **SupplySettingsManager**: component responsible for the supply settings management.
- **BatteriesManager**: component responsible for visualizing the batteries state and charge them, if needed.
- **DataManager**: as for the eMSP DataManager component, this component provides the methods for the interaction between the eMSP Business Logic and the DBMS. All the internal functional components are connected to it in order to forward requests that need the interaction with the Data Logic tier.
- **eMSPApiHandler**: this component connects the CPMS subsystem with the eMSP subsystem. It dispatches all the requests and messages from and to the eMSP subsystem. Any information that have to go from eMSP to CPMS, or vice versa, passes through this component.
- **DSORelationManager**: component that provides the Operator functionalities about the management of the relation with the DSO that supplies electric energy to the CPO.
- **ChargingStationAPIHandler**: component that connect the CPMS subsystem with the charging stations in order to retrieve real-time data from them and to send

to data about any possible modification of their settings. Any information that have to go from the charging stations to CPMS, or vice versa, passes through this component.

#### 2.2.4 External Interfaces

Some of the described components in our System are also dedicated to communicating with external services through specific interfaces. These communications are bilateral and essential to guarantee the application's functionalities. These components are both on the Client side and on the server side. We will now describe how each external service used by the System is accessed:

- **eMSP/CPMS DBMS**: this component wants to represent access point to the cloud central database and responds to the Business Logic DataManager components CRUD (Create, Read, Update, Delete) basic requests. It is interposed between all other business logic modules and the external database.
- **GPS**: On the Client side of the application, the eMSP may require the position of the User so GPS is needed. GPS information is essential to identify the location of the User and, consequently, geolocalize the nearest charging stations.
- **MapProvider**: Access to a mapping service is essential for eMSP to provide its core functionalities. In particular, it allows the User to provide the User a real-time and interactive map where the charging stations are placed in their location.
- **PushNotificationManager**: It is a cross-platform cloud solution for messages and notifications. It has the capability of allowing the user to receive push notification messages or data messages which can be deciphered by the client application.
- **EmailSystemManager** : As the name suggests, it is a service that allows sending any type of SMS and email to a target Driver.

## 2.3 Deployment view

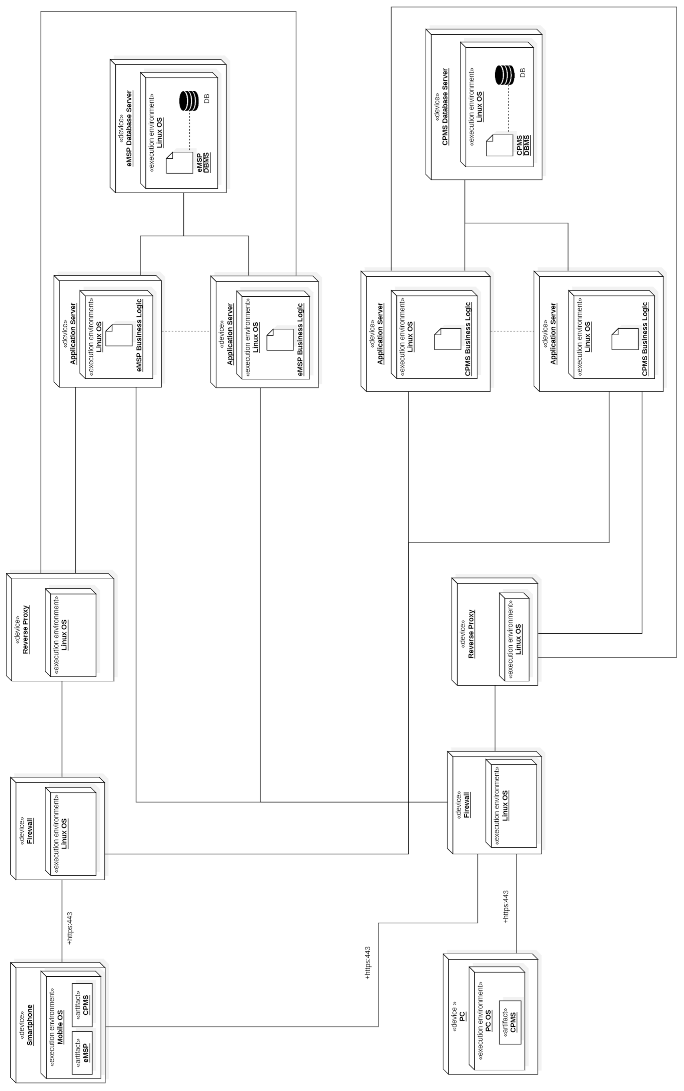

«device» eMSP Database Server
«execution environment» Linux OS
eMSP DBMS
DB
«device» Application Server
«execution environment» Linux OS
eMSP Business Logic
«device» Application Server
«execution environment» Linux OS
eMSP Business Logic
«device» CPMS Database Server
«execution environment» Linux OS
CPMS DBMS
DB
«device» Application Server
«execution environment» Linux OS
CPMS Business Logic
«device» Application Server
«execution environment» Linux OS
CPMS Business Logic
«device» Reverse Proxy
«execution environment» Linux OS
«device» Reverse Proxy
«execution environment» Linux OS
«device» Firewall
«execution environment» Linux OS
«device» Firewall
«execution environment» Linux OS
«device» Smartphone
«execution environment» Mobile OS
«artifact» eMSP
«artifact» CPMS
+https:443
+https:443
+https:443
«device» PC
«execution environment» PC OS
«artifact» CPMS

Figure 5: Deployment Diagram

The System (fig. 5) presents a multi-tier architecture in which the role of each node is specified below.

- **Mobile**: this node acts as a client machine and could hosts eMSP and CPMS applications.
  CPMS and eMSP applications run over a Mobile OS, in particular we consider the two most common ones that are AndroidOS and iOS.

- **PC**: this node works as well as a client and allows Operators to access to CPMS functionalities, through the use of a web browser.

- **Firewall**: this component filters the access to the Reverse Proxy and is used to protect a trusted network from an untrusted network. A firewall provides protection from unauthorized requests or from various types of malicious attacks.

- **Application Servers**: this level of the architecture encloses all the business logic of the Systems. CPMS and eMSP Application Servers are fully replicated to balance the workload. CPMS Application Servers communicate with eMSP ones (and viceversa) in order to make use of their APIs.

- **Reverse Proxy**: this node helps to achieve increased parallelism and scalability.
  It is a server that sits in front of a web servers, intercepting requests from clients. In the eMall System, the reverse proxy servers will intercept both client (Drivers and Users for eMSP and Operators for CPMSs) and server requests, because the eMSP and the CPMSs subsystems need to communicate with each other (e.g. the eMSP needs to retrieve information from a CPMS in order to keep up to date the presented data to the end-user). It is responsible for the load balancing as it distributes requests between all Application Servers. The Reverse Proxy also increases security and anonymity by protecting the identity of our back-end servers and acting as an additional shield against security attacks.

- **Database Server**: this machine is equipped with a relational DBMS and it is used to store and retrieve all data needed by the Application Servers.

## 2.4 Runtime view

Before illustrating the sequence diagrams, we must consider the following premises.

- In all the following diagrams, the proxy will not be included for readability purposes, but keep in mind that it will analyze every request before forwarding it to the Server.
- In some of the following diagrams, the timer check (15 minutes for the reservation and 5 minutes for the payment and start of the charging process) will not be included for readability purposes, but keep in mind that the timer check must be considered.

### 2.4.1 Sequence Diagrams

- **Driver Login**: The driver login sequence diagram (fig. 6) shows the way components interact when a Driver wants to logs in the application. The Driver sends the eMallApplicationClient component a login request with the credentials. The eMallApplicationClient component checks in advance if the input data is valid and forwards the request to the NetworkComponent. The NetworkComponent will dispatch the request to the Authentication&AuthorizationManager that requests a check of the inserted credentials to the eMSP DBMS component. If the credentials are valid, the login is successful and is shown to the Driver the main menu, otherwise, the eMallApplicationClient is notified with an HTTP Error.

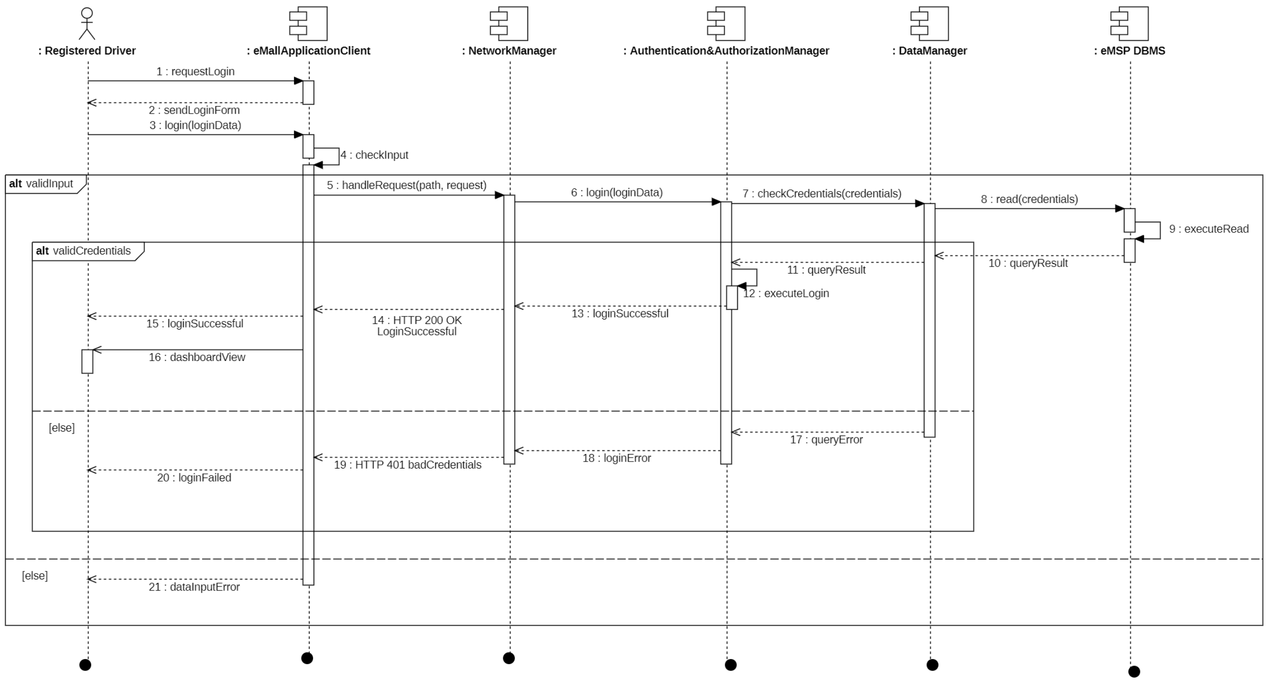


Figure 6: Driver Login Sequence Diagram

- **Driver registration**: the driver registration sequence diagram (fig. 7) shows how the eMSP subsystem architecture handles a request from a User that wants to register in the application. The Driver sends to the eMallApplicationClient component a registration request filling the required data. The eMallApplicationClient, after checking if all the required data has been inserted, forwards the request to the NetworkManager that dispatches the request to the RegisterManager component. A check if the User has already an account is then executed, with the NetworkManager notifying the eMallApplicationClient with an HTTP 403 error, in case. If the check gives no error, the RegisterManager component creates the new account, which data is then saved into the eMSP DBMS, and a view of registration confirmation will be sent to the new Driver. In order to activate the account, the Driver receive an email through the component EmailSystemManager, with a link to click to activate the account. When the Driver clicks the link, a new request will be dispatched from the NetworkManager to the RegisterManager; it will activate the account and request the update of the persistent account data to the DataManager, setting the account as "active".

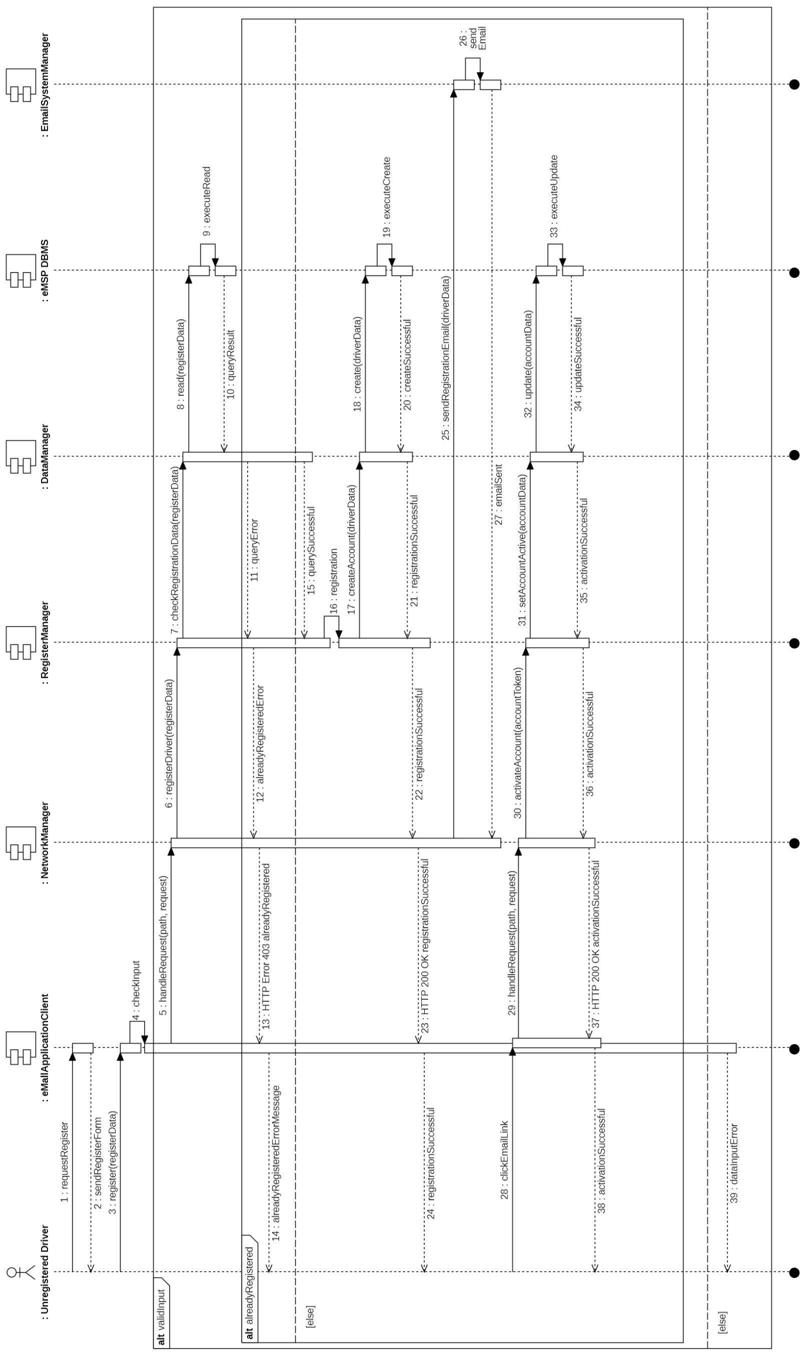

: Unregistered Driver
: eMallApplicationClient
: NetworkManager
: RegisterManager
: DataManager
: eMSP DBMS
: EmailSystemManager
alt validInput
alt alreadyRegistered
[else]
[else]
1 : requestRegister
2 : sendRegisterForm
3 : register(registerData)
4 : checkInput
5 : handleRequest(path, request)
6 : registerDriver(registerData)
7 : checkRegistrationData(registerData)
8 : read(registerData)
9 : executeRead
10 : queryResult
11 : queryError
12 : alreadyRegisteredError
13 : HTTP Error 403 alreadyRegistered
14 : alreadyRegisteredErrorMessage
15 : querySuccessful
16 : registration
17 : createAccount(driverData)
18 : create(driverData)
19 : executeCreate
20 : createSuccessful
21 : registrationSuccessful
22 : registrationSuccessful
23 : HTTP 200 OK registrationSuccessful
24 : registrationSuccessful
25 : sendRegistrationEmail(driverData)
26 : send Email
27 : emailSent
28 : clickEmailLink
29 : handleRequest(path, request)
30 : activateAccount(accountToken)
31 : setAccountActive(accountData)
32 : update(accountData)
33 : executeUpdate
34 : updateSuccessful
35 : activationSuccessful
36 : activationSuccessful
37 : HTTP 200 OK activationSuccessful
38 : activationSuccessful
39 : dataInputError

Figure 7: Driver Registration Sequence Diagram

- **Visualization of Charging Station External Status**: the visualization of charging station sequence diagram (fig. 8) shows the way components interact when a User wants to visualize the external status of a selected charging station. This data can be visualized both by a not registered/not logged in User and by a logged in Driver. Once the User opens the application, it presents immediately a map of the Charging Stations in a certain random area.

  If the User allows the eMallClientApplication to communicate with his/her device's GPS, the map will center on his/her current position. The shown map is provided by the MapProvider component that interacts with a map system i.e. Google Maps In order to place the Charging Stations in their exact position on the map provided to the User.

  While the User interacts with the map, it will automatically update showing the Charging Station in the map. Once the User chooses the Charging Station he/she wants to see data of, he/she sends the request to the eMallApplicationClient and the request will be dispatched to the ChargingStationsInformationManager. This last component will forward the request to the CPMSApiHandler that will subsequently establish a communication with the CPMS that manages the charging station. The request will follow the path into the CPMS Business Logic internal components: from the eMSPApiHandler that receives the message to the ChargingPointsManager that have access to the ChargingStationAPIHandler. With the methods provided by this last component, the request is then forwarded to the selected Charging Station software application that gathers the required data and sends it back. This data will then follow the inverse path its request have crossed, up to the eMallApplicationClient that will graphically show the data to the User.

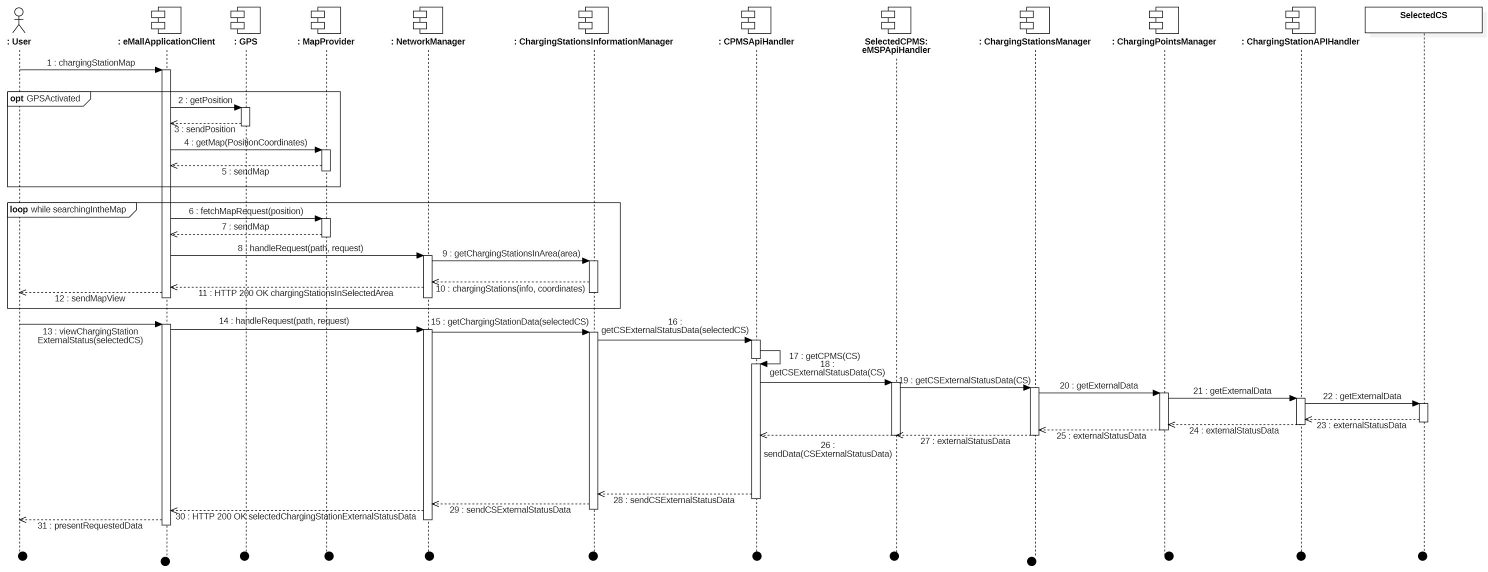

: User
: eMallApplicationClient
: GPS
: MapProvider
: NetworkManager
: ChargingStationsInformationManager
: CPMSApiHandler
SelectedCPMS: eMSPApiHandler
: ChargingStationsManager
: ChargingPointsManager
: ChargingStationAPIHandler
SelectedCS
1 : chargingStationMap
opt GPSActivated
2 : getPosition
3 : sendPosition
4 : getMap(PositionCoordinates)
5 : sendMap
loop while searchingIntheMap
6 : fetchMapRequest(position)
7 : sendMap
8 : handleRequest(path, request)
9 : getChargingStationsInArea(area)
10 : chargingStations(info, coordinates)
11 : HTTP 200 OK chargingStationsInSelectedArea
12 : sendMapView
13 : viewChargingStation ExternalStatus(selectedCS)
14 : handleRequest(path, request)
15 : getChargingStationData(selectedCS)
16 : getCSExternalStatusData(selectedCS)
17 : getCPMS(CS)
18 : getCSExternalStatusData(CS)
19 : getCSExternalStatusData(CS)
20 : getExternalData
21 : getExternalData
22 : getExternalData
23 : externalStatusData
24 : externalStatusData
25 : externalStatusData
26 : sendData(CSExternalStatusData)
27 : externalStatusData
28 : sendCSExternalStatusData
29 : sendCSExternalStatusData
30 : HTTP 200 OK selectedChargingStationExternalStatusData
31 : presentRequestedData

Figure 8: Visualization of Charging Station External Status Sequence Diagram

- **Reservation of a socket**: this sequence diagram (fig. 9) shows how components interact when a Driver wants to book a socket of a selected charging station. The NetworkManager dispatches the Driver request to the ReservationManager that requests a check if the Driver who asked for a new reservation has already a pending one. If so, the NetworkManager will send a HTTP 400 Bad Request error to the ApplicationClient who will show the error to the Driver. In the other case, the ReservationManager then communicates with the CPMSApiHandler with a view of checking the availability of the socket selected by the Driver. The CPMSApiHandler will forward the request to the CPMS that manages the selected Charging Station. Asking for the socket state.

  For the sake of readability, being this diagram already very complex and difficult to follow, it was not included the connection between the CPMS Business Logic and the selected Charging Station through the ChargingStationAPIHandler in order to retrieve real-time data about the selected socket's state. It is then assumed that the ChargingPointsManager component already possesses that data. However, the sequence diagram in fig. 9 shows in detail this passage.

  If the socket is available, the ReservationManager sends a request to the CPMS about the socket to be reserved and then creates the reservation. This reservation is then also saved persistently in the eMSP database.

  After this, the ReservationManager generates a QR-Code and an authorization token to be sent respectively to the Driver and to the selected Charging Station. This is made in order to: firstly, permit the Driver to start the charging scanning the QR-Code once he arrives at the Charging Station; secondly, to give to the CPMS who manages the selected Charging Station the data about the created reservation i.e. reserved socket, driver data. This information will be persistently memorized in the CPMS DB and exploited by the CPMS subsystem once the QR-Code will be scanned in order to start the charging process.

  If the socket is not available, a HTTP 400 Bad Request error message will be forwarded to the eMallApplicationClient component and then presented to the Driver.

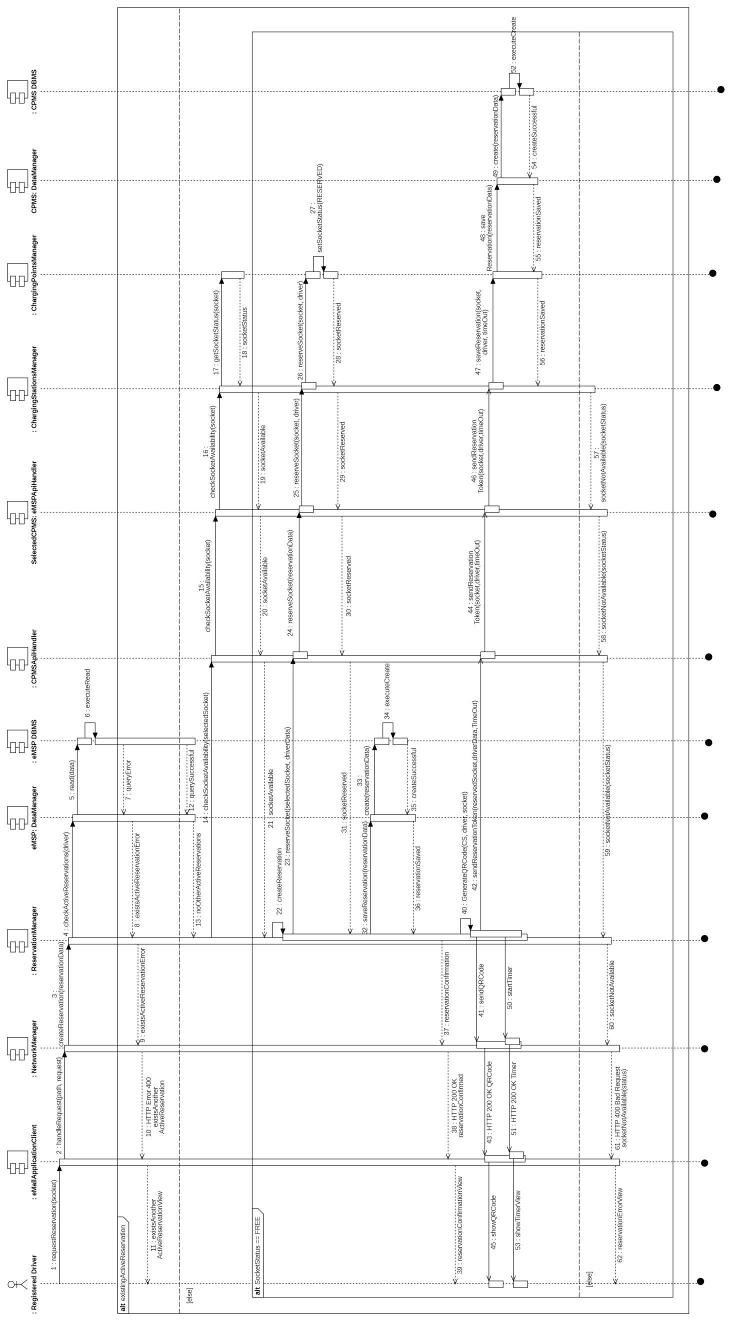


Figure 9: Reserve a socket Sequence Diagram

- **Add Payment Method**: the following sequence diagram (fig. 11) shows the interaction between components in order to let the Driver add a new payment method in the system. The Driver requests the addition of a payment method and the eMallApplicationClient provides him/her the form to be filled with the required data. Once the data has been inserted, the Driver sends it. The Client, then, does a preliminary check if all the data have been inserted and then forwards the request to the NetworkManager. This latter dispatches the request to the PaymentManager component.

This component processes the data and tries to establish a communication with the the bank or the credit company that corresponds to the inserted payment method in order to verify the inserted data truthfulness. If the inserted payment method turns out to be valid, the payment data will be forwarded to the DataManager component in order to save it into the eMSP Database. A confirmation of the successful operation is then sent to the eMallApplicationClient and then presented to the Driver. If, instead, the payment data results as not valid, an error will be notified to the Driver via a HTTP 400 Bad Request message.

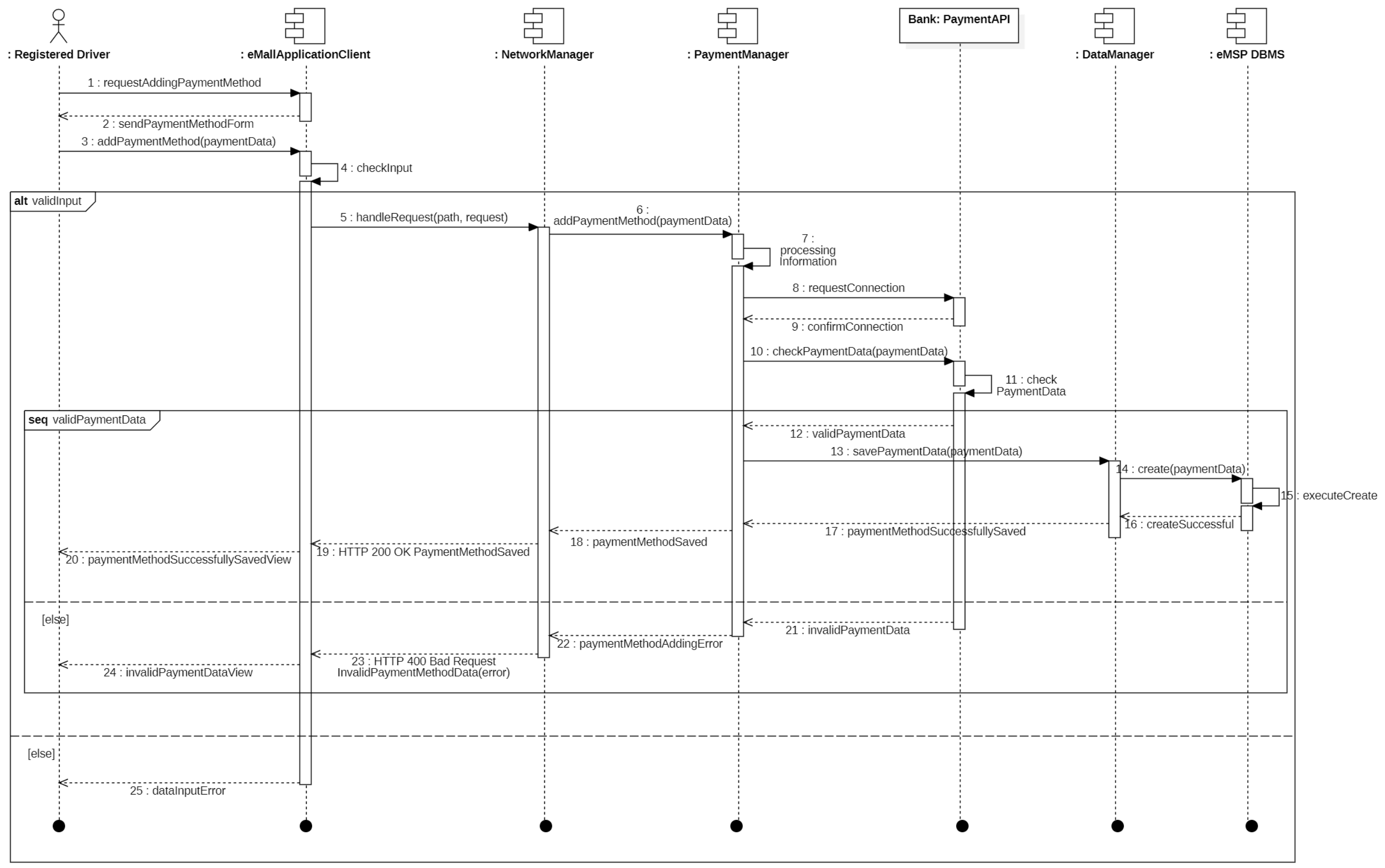

: Registered Driver
: eMallApplicationClient
: NetworkManager
: PaymentManager
Bank: PaymentAPI
: DataManager
: eMSP DBMS
1 : requestAddingPaymentMethod
2 : sendPaymentMethodForm
3 : addPaymentMethod(paymentData)
4 : checkInput
alt validInput
5 : handleRequest(path, request)
6 : addPaymentMethod(paymentData)
7 : processing Information
8 : requestConnection
9 : confirmConnection
10 : checkPaymentData(paymentData)
11 : check PaymentData
seq validPaymentData
12 : validPaymentData
13 : savePaymentData(paymentData)
14 : create(paymentData)
15 : executeCreate
16 : createSuccessful
17 : paymentMethodSuccessfullySaved
18 : paymentMethodSaved
19 : HTTP 200 OK PaymentMethodSaved
20 : paymentMethodSuccessfullySavedView
[else]
21 : invalidPaymentData
22 : paymentMethodAddingError
23 : HTTP 400 Bad Request InvalidPaymentMethodData(error)
24 : invalidPaymentDataView
[else]
25 : dataInputError

Figure 10: Add Payment Method Sequence Diagram

- **Start charging process**: the start charging process sequence diagram (fig. 11) shows the interaction between components in order to starts a charging process. Firstly, a Driver arrives at the reserved socket, requests the QR-Code presentation to the eMallApplicationClient and then scans it at the appropriate QR-Code reader available at the Charging Point. The software application of the Charging Station decodes the QR-Code and this information is then forwarded to the dataManager in order to check if the reservation data is valid and present in the DB. If so, the ChargingPointsManager component sends an async message that will be forwarded to the eMSP subsystem via the eMSPApiHandler with the aim of reporting that the Driver checked in the Charging Station. The ChargeManager then creates a charge instance and sends a request to the NetworkManager in order to present a "start charging view" to the Driver Application. The Driver chooses the payment method and clicks "start charging".

  The NetworkManager dispatches the request to the PaymentManager, that establishes a communication with the bank or the credit company that corresponds to the chosen payment method in order to request a reservation of the maximum amount of spendable money for the charge, as previously explained in the RASD. If money is not available, it will be reported to the Driver.

  In case the transaction is successfully executed, a start charging request will be forwarded to the ChargingPointsManager that updates data about the socket and then sends the request to the Charging Station that will fuel the socket and activate the charging.

  Once the charge has been successfully started, a confirmation message is sent to the Driver.

  If, instead, no reservation corresponds to the QR-Code presented by the Driver, it could be expired or not corresponding to the charging point that scanned it. In this case, an error message is forwarded to the Driver.

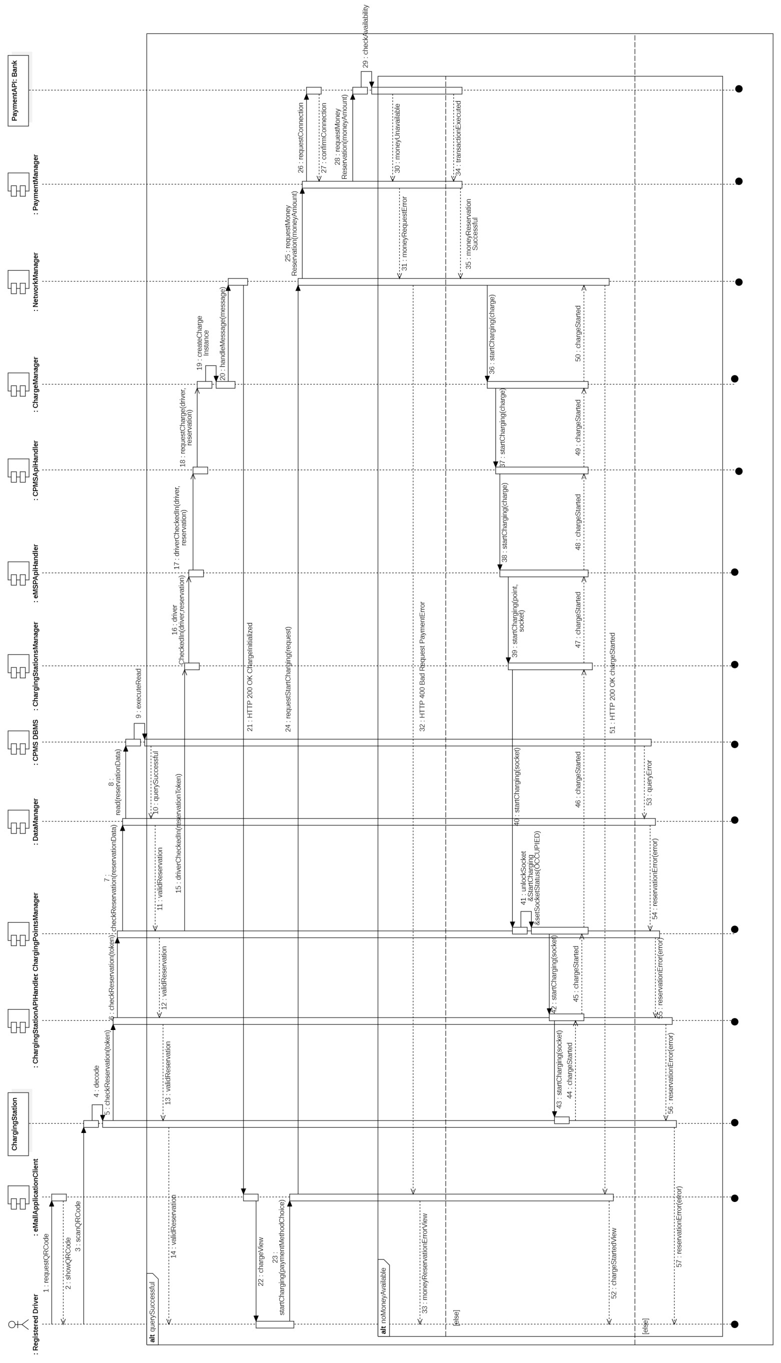
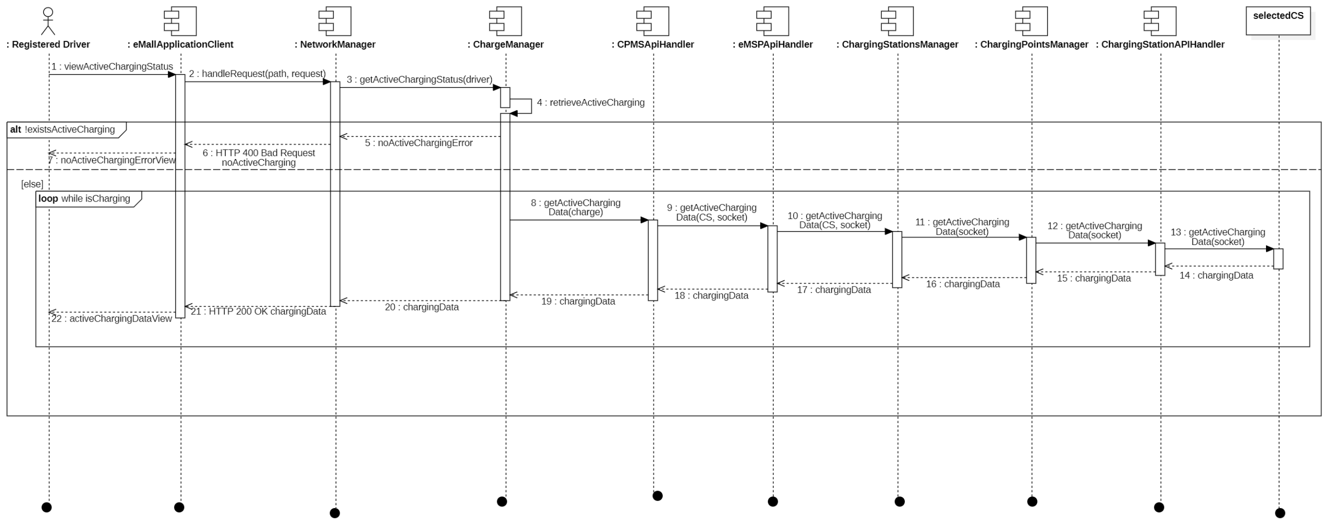

: Registered Driver
: eMallApplicationClient
ChargingStation
: ChargingStationAPIHandler
: ChargingPointsManager
: DataManager
: CPMS DBMS
: ChargingStationsManager
: eMSPApiHandler
: CPMSApiHandler
: ChargeManager
: NetworkManager
: PaymentManager
PaymentAPI: Bank
1 : requestQRCode
2 : showQRCode
3 : scanQRCode
4 : decode
5 : checkReservation(token)
6 : checkReservation(token)
7 : checkReservation(reservationData)
8 : read(reservationData)
9 : executeRead
10 : querySuccessful
11 : validReservation
12 : validReservation
13 : validReservation
14 : validReservation
15 : driverCheckedIn(reservationToken)
16 : driver CheckedIn(driver,reservation)
17 : driverCheckedIn(driver, reservation)
18 : requestCharge(driver, reservation)
19 : createCharge Instance
20 : handleMessage(message)
21 : HTTP 200 OK ChargeInitialized
22 : chargeView
23 : startCharging(paymentMethodChoice)
24 : requestStartCharging(request)
25 : requestMoney Reservation(moneyAmount)
26 : requestConnection
27 : confirmConnection
28 : requestMoney Reservation(moneyAmount)
29 : checkAvailability
30 : moneyUnavailable
31 : moneyRequestError
32 : HTTP 400 Bad Request PaymentError
33 : moneyReservationErrorView
34 : transactionExecuted
35 : moneyReservation Successful
36 : startCharging(charge)
37 : startCharging(charge)
38 : startCharging(charge)
39 : startCharging(point, socket)
40 : startCharging(socket)
41 : unlockSocket &StartCharging &setSocketStatus(OCCUPIED)
42 : startCharging(socket)
43 : startCharging(socket)
44 : chargeStarted
45 : chargeStarted
46 : chargeStarted
47 : chargeStarted
48 : chargeStarted
49 : chargeStarted
50 : chargeStarted
51 : HTTP 200 OK chargeStarted
52 : chargeStartedView
53 : queryError
54 : reservationError(error)
55 : reservationError(error)
56 : reservationError(error)
57 : reservationError(error)
alt querySuccessful
alt noMoneyAvailable
[else]
[else]

Figure 11: Start charging process Sequence Diagram

- **View Active Charging Status**: the view active charging status sequence diagram (fig. 11) shows the way components interact when a Driver wants to visualize the status of the active charging of his electric vehicle. This sequence diagram is very similar to the one in which the Driver requires to see data about the external status of a Charging Station in terms of acquiring information from the Charging Station and presenting it to the Driver. The differences are in the check if there are active charging to monitor, operation carried out by the ChargeManager component, and in the loop of information retrieval that lasts until the electric vehicle is still charging and its charge has not been stopped or is not ended.



: Registered Driver
: eMallApplicationClient
: NetworkManager
: ChargeManager
: CPMSApiHandler
: eMSPApiHandler
: ChargingStationsManager
: ChargingPointsManager
: ChargingStationAPIHandler
selectedCS
1 : viewActiveChargingStatus
2 : handleRequest(path, request)
3 : getActiveChargingStatus(driver)
4 : retrieveActiveCharging
alt !existsActiveCharging
5 : noActiveChargingError
6 : HTTP 400 Bad Request noActiveCharging
7 : noActiveChargingErrorView
[else]
loop while isCharging
8 : getActiveCharging Data(charge)
9 : getActiveCharging Data(CS, socket)
10 : getActiveCharging Data(CS, socket)
11 : getActiveCharging Data(socket)
12 : getActiveCharging Data(socket)
13 : getActiveCharging Data(socket)
14 : chargingData
15 : chargingData
16 : chargingData
17 : chargingData
18 : chargingData
19 : chargingData
20 : chargingData
21 : HTTP 200 OK chargingData
22 : activeChargingDataView

Figure 12: View active charging status Sequence Diagram

- **Stop charging process**: the stop charging process sequence diagram (fig. 11) shows the way components interact when a Diver wants to stop the charging process. The Driver forwards the stop charging request through the eMallApplicationClient. The NetworkManager then receives the request and dispatches it to the ChargeManager. It then sends a stop charge message through the CPMSApiHandler that forwards the message to the CPMS that manages the Charging Station related to the charge. This message is then forwarded to the ChargingPointsManager that updates the state of the socket to "FREE" and unlocks the socket. Once the charge process has been stopped, the NetworkManager will dispatch to the PaymentManager component a payment request of the amount of money related to the just ended charge. The PaymentManager component establishes a connection with the bank or the credit company that corresponds to the chosen payment method in order to request the money transfer. When the transaction gives positive result, the Driver will be informed of the just correctly executed operation. The ChargeManager then requests the DataManager component to update in the database the state of the reservation (setting it as "COMPLETED") related to the just stopped charge.

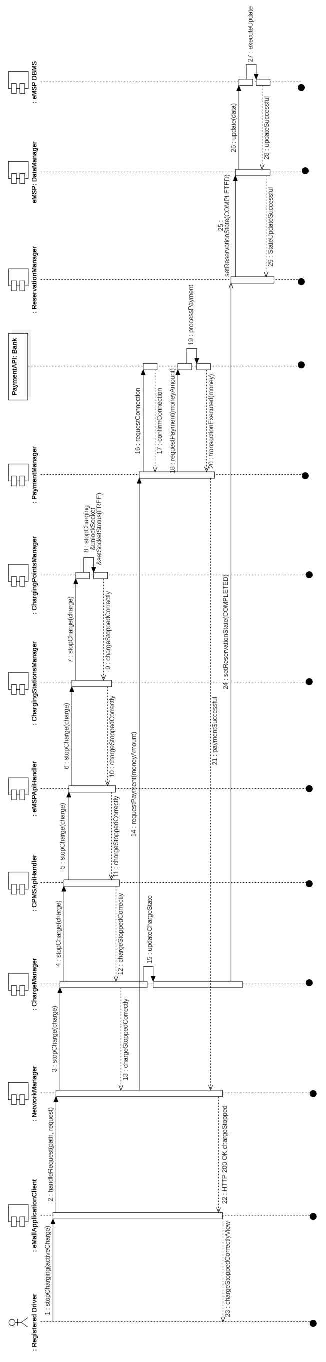

: Registered Driver
: eMallApplicationClient
: NetworkManager
: ChargeManager
: CPMSApiHandler
: eMSPApiHandler
: ChargingStationsManager
: ChargingPointsManager
: PaymentManager
PaymentAPI: Bank
: ReservationManager
eMSP: DataManager
: eMSP DBMS
1 : stopCharging(activeCharge)
2 : handleRequest(path, request)
3 : stopCharge(charge)
4 : stopCharge(charge)
5 : stopCharge(charge)
6 : stopCharge(charge)
7 : stopCharge(charge)
8 : stopCharging &unlockSocket &setSocketStatus(FREE)
9 : chargeStoppedCorrectly
10 : chargeStoppedCorrectly
11 : chargeStoppedCorrectly
12 : chargeStoppedCorrectly
13 : chargeStoppedCorrectly
14 : requestPayment(moneyAmount)
15 : updateChargeState
16 : requestConnection
17 : confirmConnection
18 : requestPayment(moneyAmount)
19 : processPayment
20 : transactionExecuted(money)
21 : paymentSuccessful
22 : HTTP 200 OK chargeStopped
23 : chargeStoppedCorrectlyView
24 : setReservationState(COMPLETED)
25 : setReservationState(COMPLETED)
26 : update(data)
27 : executeUpdate
28 : updateSuccessful
29 : StateUpdateSuccessful

Figure 13: Stop charging process Sequence Diagram

- **Operator Login**: The operator login sequence diagram (fig. 6) shows the way components interact when an Operator wants to logs in the application. The Driver sends the eMallApplicationClient component a login request with the credentials. The eMallApplicationClient component checks in advance if the input data is valid and forwards the request to the NetworkComponent. The NetworkComponent will dispatch the request to the Authentication&AuthorizationManager that requests a check of the inserted credentials to the eMSP DBMS component. If the credentials are valid, the login is successful and the main application dashboard is shown to the Operator, otherwise, the eMallApplicationClient is notified with an HTTP Error that will be presented to the Operator.

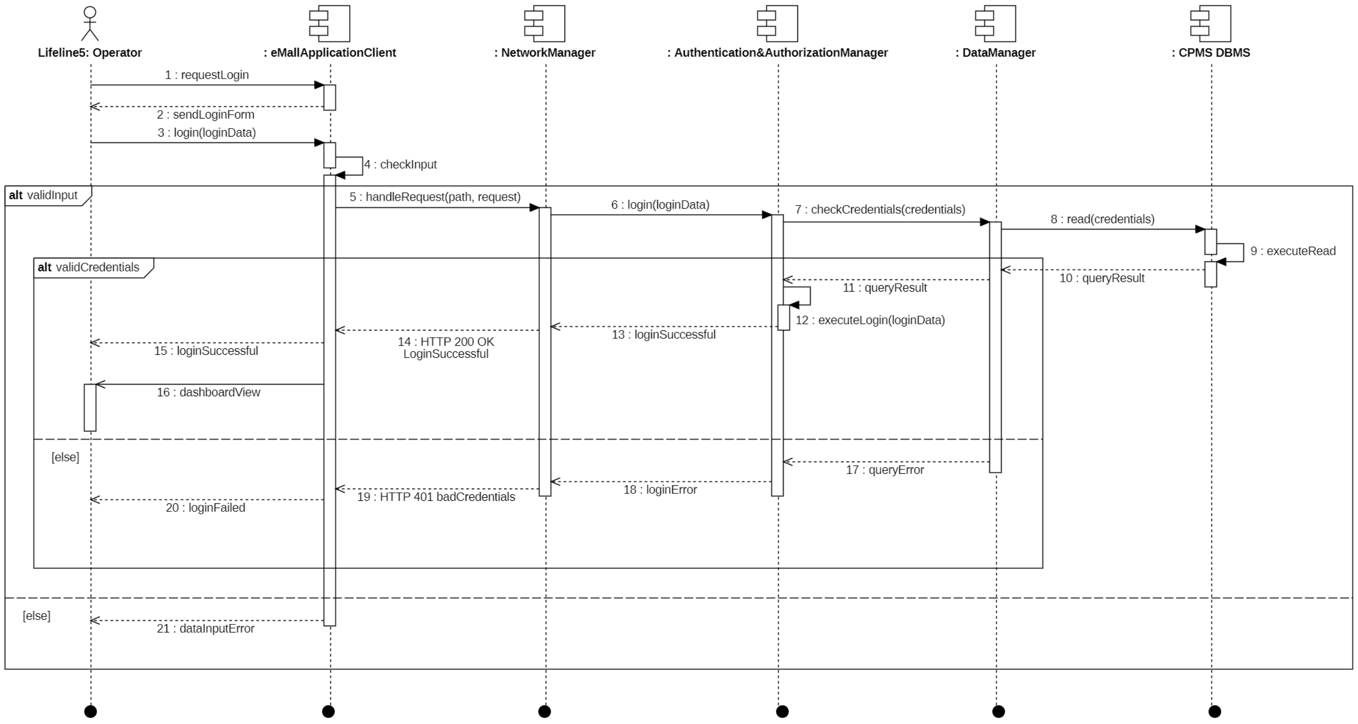


Figure 14: Operator Login Sequence Diagram

In the next fig. 15 and fig. 16, the "X" in "SelectedCS:X" is referred to one of the following components: PriceAndOffersManager, ChargingPointManager, SupplySettingsManager, BatteriesManager.
Due to the fact that the sequence diagrams of visualization and modification of price&offers, charging point, supply settings and batteries data are the same, we decided to generalize the operations of data Visualization and Modification to avoid having to show many times the same sequence diagram changing only the X component for each diagram.

- **Operator General Data Visualization**: the operator general data visualization sequence diagram (fig. 15) shows the way the CPMS Business Logic components interact between each other when an Operator carries out a data visualization request. The Operator chooses between the Charging Station he/she manages and send the data visualization request specifying the Charging Station. The NetworkManager then handles the request and dispatches it to the ChargingStationsManager which forwards the request to the component responsible for the data the Operator wants to visualize.

  The request is then forwarded to the selected Charging Station software application through the ChargingStationApiHandler component. The Charging Station SA then gathers the requested data and send it back. then follow the inverse path its request have crossed up to the eMallApplicationClient that will graphically present the data the Operator requested.

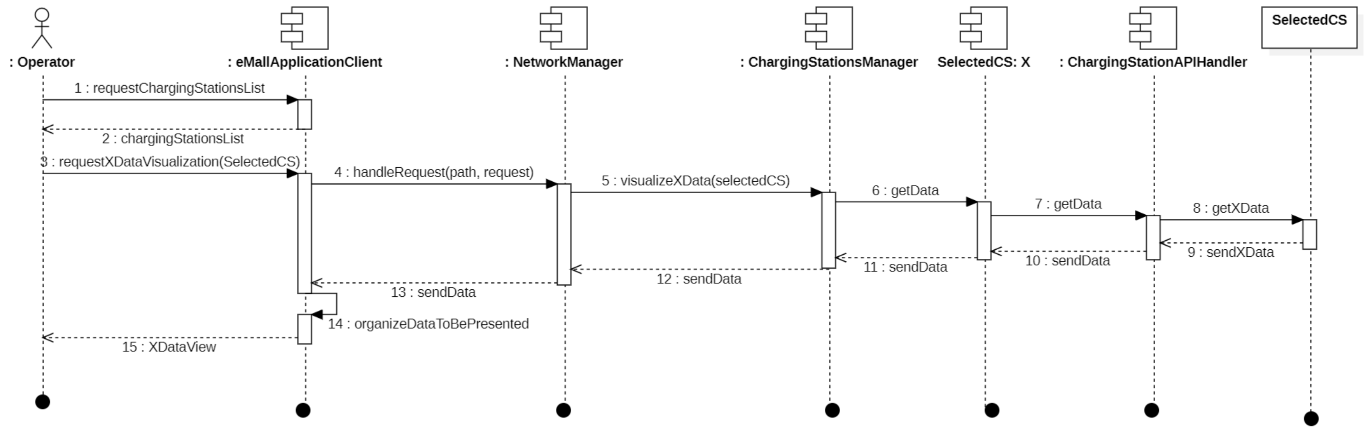

: Operator
: eMallApplicationClient
: NetworkManager
: ChargingStationsManager
SelectedCS: X
: ChargingStationAPIHandler
SelectedCS
1 : requestChargingStationsList
2 : chargingStationsList
3 : requestXDataVisualization(SelectedCS)
4 : handleRequest(path, request)
5 : visualizeXData(selectedCS)
6 : getData
7 : getData
8 : getXData
9 : sendXData
10 : sendData
11 : sendData
12 : sendData
13 : sendData
14 : organizeDataToBePresented
15 : XDataView

Figure 15: Operator general data visualization Sequence Diagram

- **Operator General Data Modification**: the operator general data modification sequence diagram (fig. 16) shows the way components interact when an Operator carries out a data modification request regarding a particular Charging Station. The Operator chooses between the Charging Station he/she manages and send the data modification request specifying the Charging Station. The eMallApplication-Client then provides to the Operator the form of the data to be filled and sent. The Client, then, does a preliminary check if all the data have been inserted and then, if so, forwards the request to the NetworkManager, otherwise it will present to the Operator an error. The NetworkManager will forward the request to the ChargingStationsManager that, in turn, will dispatch the request to the component responsible for the data the Operator wants to modify. The component in question will check the validity of the data and in negative case it will send back a data error that will be forwarded to the Operator.

  If the data is valid, the component will forward the data to the selected Charging Station SA through the ChargingStationApiHandler component. The Charging Station then set the new data and confirms the operation.

  After this, the data inserted by the Operator will be forwarded to the DataManager component in order to save it into the CPMS Database.

  When the operation is successfully carried out, an HTTP 200 message will be sent back to the eMallApplicationClient that will show to the Operator a view of operation successful.

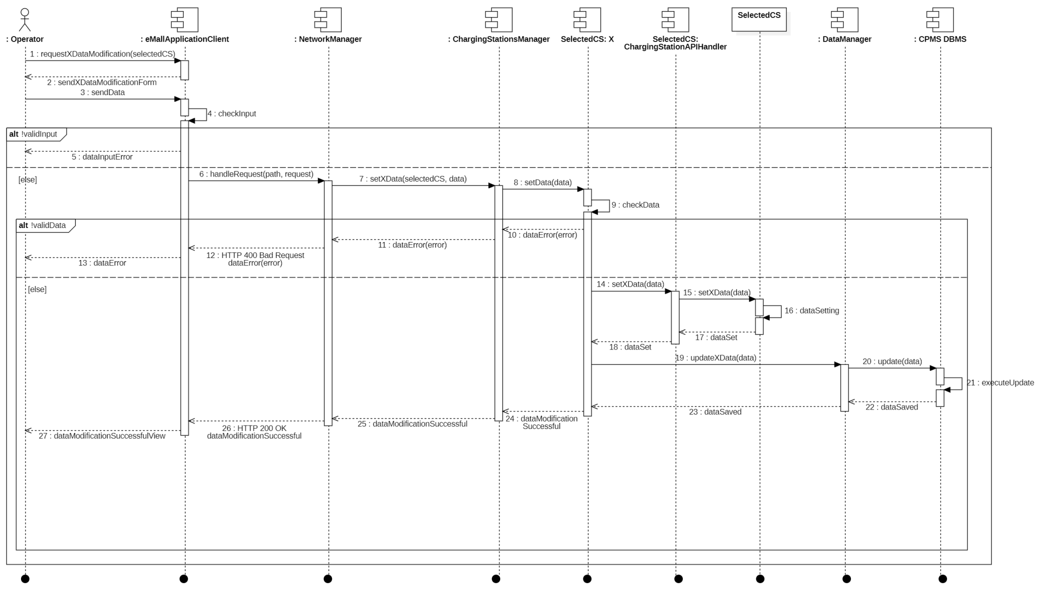


Figure 16: Operator general data modification Sequence Diagram

- **Operator Choose DSO**: the choose DSO sequence diagram (fig. 17) shows how the system manages the request of an Operator that wants to visualize the list of available DSO and to choose one between them. Through the eMallApplication-Client, the Operator asks to view the list of available DSOs. The list is passed by the DSORelationManager component through the NetworkManager of the CPMS using a HTTP message. After that, the operator selects the new DSO from which to buy electric energy and sends the request. The request is then forwarded to the DSORelationManager which establishes connection with the selected DSO through the DSOApi and requests to check the availability of the DSO choosen. If the DSO is not available, the DSORelationManager will notify the unavailability to the NetworkManager that, using a HTTP error, will notify the Operator. Otherwise, the DSORelationManager will send the data of the new DSO to the DataManager in order to save it into the CPMS DBMS; finally, the DSORelationManager will notify the successful operation to the NetworkManager that, using a HTTP message, will notify the eMallApplicationClient that will present a confirmation view to the Operator.

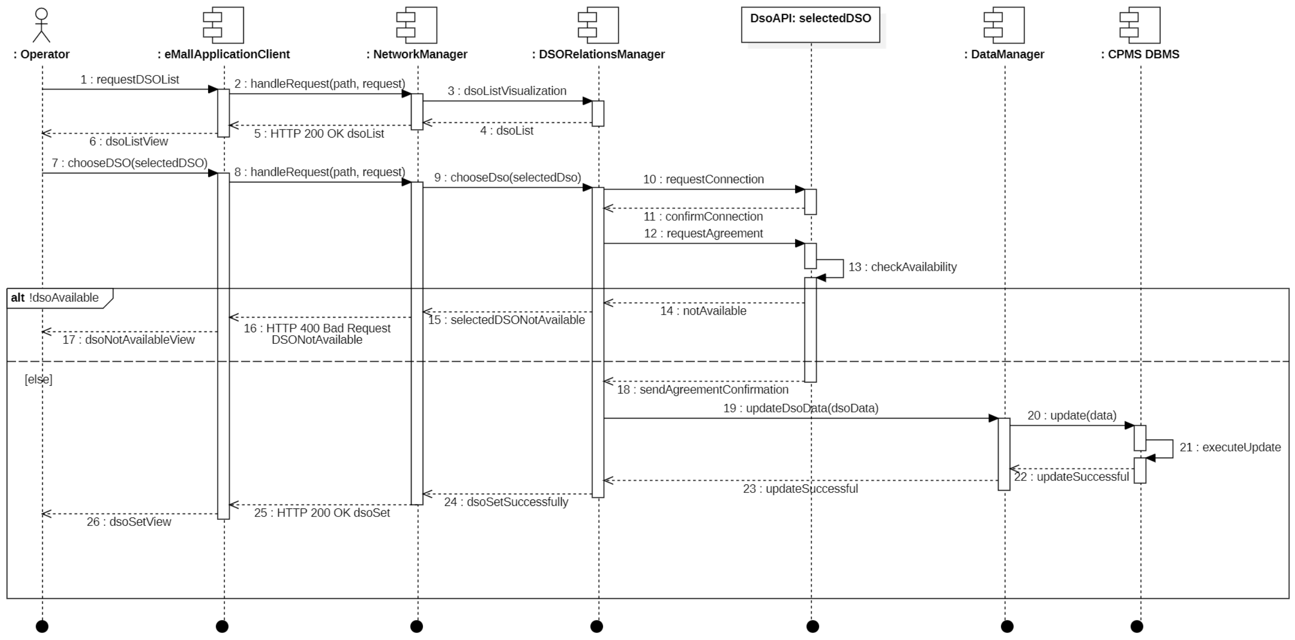


Figure 17: Operator choose DSO Sequence Diagram

## 2.5 Component interfaces

In the following Component Interfaces Diagrams, we will show interactions between the component interfaces of our System.
The Component Interfaces Diagram (fig. 18) shows the interaction between all the component interfaces of our System. The Component Interfaces Diagram (fig. 19) shows the interaction between the component interfaces of our eMSP subsystem. The Component Interfaces Diagram (fig. 20) shows the interaction between the component interfaces of our CPMS subsystem.

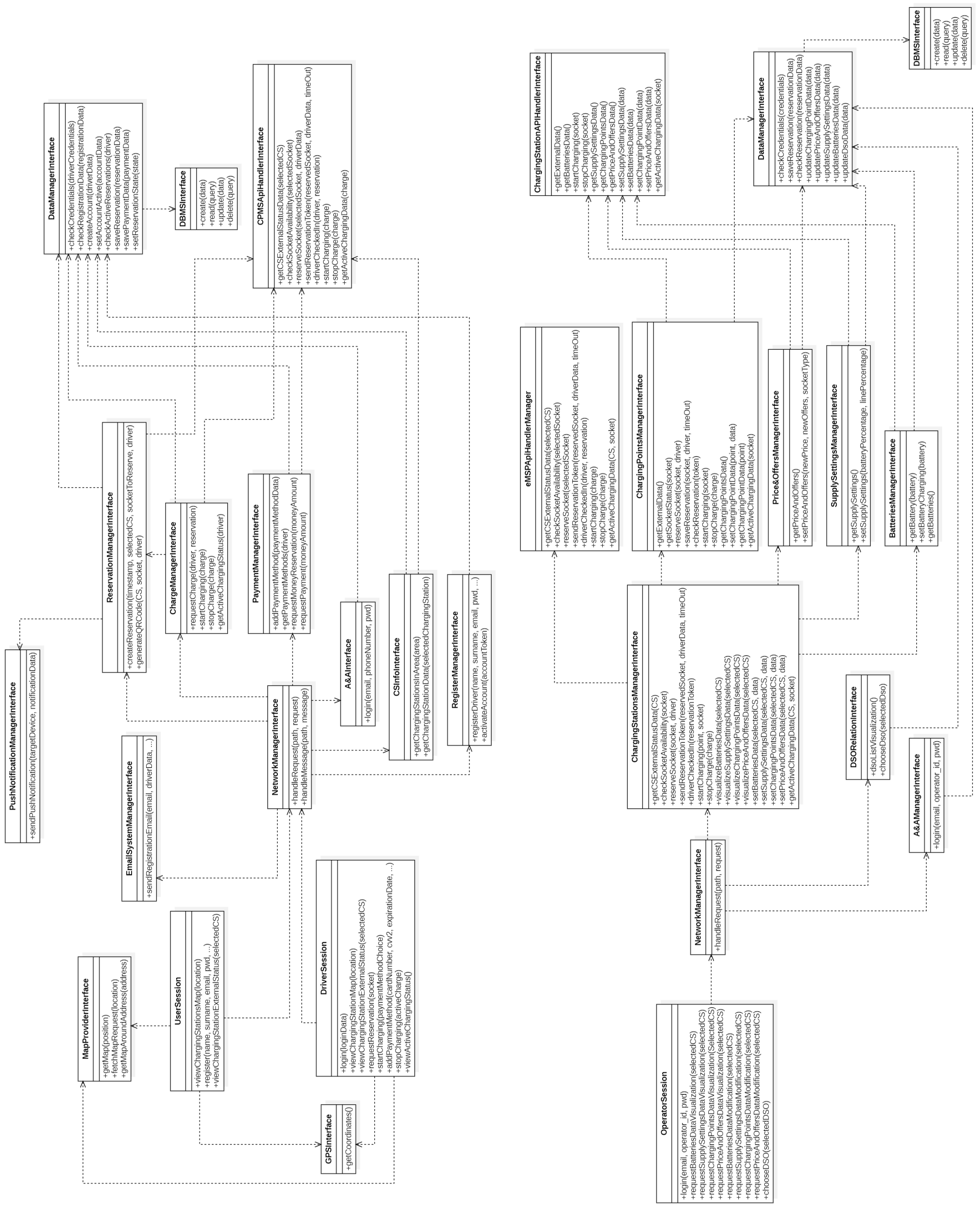

DataManagerInterface
+checkCredentials(driverCredentials)
+checkRegistrationData(registrationData)
+createAccount(driverData)
+setAccountActive(accountData)
+checkActiveReservations(driver)
+saveReservation(reservationData)
+savePaymentData(paymentData)
+setReservationState(state)
DBMSInterface
+create(data)
+read(query)
+update(data)
+delete(query)
CPMSApiHandlerInterface
+getCSExternalStatusData(selectedCS)
+checkSocketAvailability(selectedSocket)
+reserveSocket(selectedSocket, driverData)
+sendReservationToken(reservedSocket, driverData, timeOut)
+driverCheckedIn(driver, reservation)
+startCharging(charge)
+stopCharge(charge)
+getActiveChargingData(charge)
ReservationManagerInterface
+createReservation(timestamp, selectedCS, socketToReserve, driver)
+generateQRCode(CS, socket, driver)
ChargeManagerInterface
+requestCharge(driver, reservation)
+startCharging(charge)
+stopCharge(charge)
+getActiveChargingStatus(driver)
PaymentManagerInterface
+addPaymentMethod(paymentMethodData)
+getPaymentMethods(driver)
+requestMoneyReservation(moneyAmount)
+requestPayment(moneyAmount)
PushNotificationManagerInterface
+sendPushNotification(targetDevice, notificationData)
EmailSystemManagerInterface
+sendRegistrationEmail(email, driverData, ...)
NetworkManagerInterface
+handleRequest(path, request)
+handleMessage(path, message)
A&AInterface
+login(email, phoneNumber, pwd)
CSInfoInterface
+getChargingStationsInArea(area)
+getChargingStationData(selectedChargingStation)
RegisterManagerInterface
+registerDriver(name, surname, email, pwd, ...)
+activateAccount(accountToken)
MapProviderInterface
+getMap(position)
+fetchMapRequest(location)
+getMapAroundAddress(address)
UserSession
+viewChargingStationsMap(location)
+register(name, surname, email, pwd, ...)
+viewChargingStationExternalStatus(selectedCS)
DriverSession
+login(loginData)
+viewChargingStationMap(location)
+viewChargingStationExternalStatus(selectedCS)
+requestReservation(socket)
+startCharging(paymentMethodChoice)
+addPaymentMethod(cardNumber, cvv2, expirationDate, ...)
+stopCharging(activeCharge)
+viewActiveChargingStatus()
GPSInterface
+getCoordinates()
ChargingStationAPIHandlerInterface
+getExternalData()
+getBatteriesData()
+startCharging(socket)
+stopCharging(socket)
+getSupplySettingsData()
+getChargingPointsData()
+getPriceAndOffersData()
+setSupplySettingsData(data)
+setBatteriesData(data)
+setChargingPointData(data)
+setPriceAndOffersData(data)
+getActiveChargingData(socket)
DataManagerInterface
+checkCredentials(credentials)
+saveReservation(reservationData)
+checkReservation(reservationData)
+updateChargingPointData(data)
+updatePriceAndOffersData(data)
+updateSupplySettingsData(data)
+updateBatteriesData(data)
+updateDsoData(data)
DBMSInterface
+create(data)
+read(query)
+update(data)
+delete(query)
eMSPApiHandlerManager
+getCSExternalStatusData(selectedCS)
+checkSocketAvailability(selectedSocket)
+reserveSocket(selectedSocket)
+sendReservationToken(reservedSocket, driverData, timeOut)
+driverCheckedIn(driver, reservation)
+startCharging(charge)
+stopCharging(charge)
+getActiveChargingData(CS, socket)
ChargingPointsManagerInterface
+getExternalData()
+getSocketStatus(socket)
+reserveSocket(socket, driver)
+saveReservation(socket, driver, timeOut)
+checkReservation(token)
+startCharging(socket)
+stopCharge(charge)
+getChargingPointsData()
+setChargingPointData(point, data)
+getChargingPointData(point)
+getActiveChargingData(socket)
Price&OffersManagerInterface
+getPriceAndOffers()
+setPriceAndOffers(newPrice, newOffers, socketType)
SupplySettingsManagerInterface
+getSupplySettings()
+setSupplySettings(batteryPercentage, linePercentage)
BatteriesManagerInterface
+getBattery(battery)
+setBatteryCharging(battery)
+getBatteries()
ChargingStationsManagerInterface
+getCSExternalStatusData(CS)
+checkSocketAvailability(socket)
+reserveSocket(socket, driver)
+sendReservationToken(reservedSocket, driverData, timeOut)
+driverCheckedIn(reservationToken)
+startCharging(point, socket)
+stopCharge(charge)
+visualizeBatteriesData(selectedCS)
+visualizeSupplySettingsData(selectedCS)
+visualizeChargingPointsData(selectedCS)
+visualizePriceAndOffersData(selectedCS)
+setBatteriesData(selectedCS, data)
+setSupplySettingsData(selectedCS, data)
+setChargingPointsData(selectedCS, data)
+setPriceAndOffersData(selectedCS, data)
+getActiveChargingData(CS, socket)
DSORelationInterface
+dsoListVisualization()
+chooseDso(selectedDso)
A&AManagerInterface
+login(email, operator_id, pwd)
NetworkManagerInterface
+handleRequest(path, request)
OperatorSession
+login(email, operator_id, pwd)
+requestBatteriesDataVisualization(selectedCS)
+requestSupplySettingsDataVisualization(selectedCS)
+requestChargingPointsDataVisualization(selectedCS)
+requestPriceAndOffersDataVisualization(selectedCS)
+requestBatteriesDataModification(selectedCS)
+requestSupplySettingsDataModification(selectedCS)
+requestChargingPointsDataModification(selectedCS)
+requestPriceAndOffersDataModification(selectedCS)
+chooseDSO(selectedDSO)

Figure 18: Main Component Interface Diagram

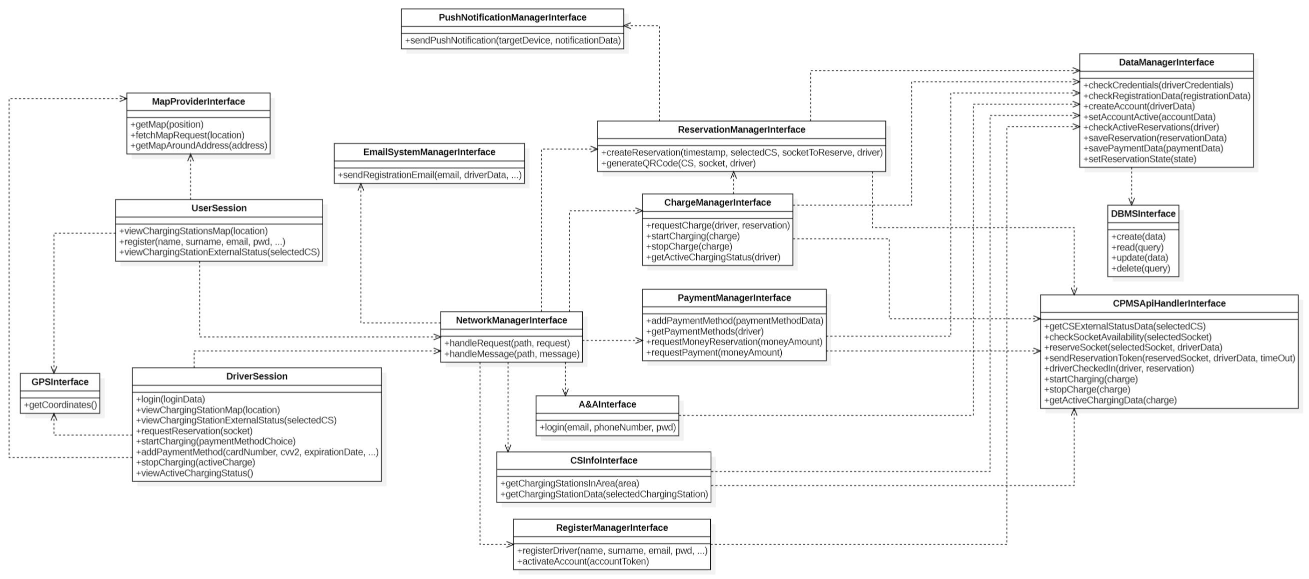

PushNotificationManagerInterface
+sendPushNotification(targetDevice, notificationData)
DataManagerInterface
+checkCredentials(driverCredentials)
+checkRegistrationData(registrationData)
+createAccount(driverData)
+setAccountActive(accountData)
+checkActiveReservations(driver)
+saveReservation(reservationData)
+savePaymentData(paymentData)
+setReservationState(state)
MapProviderInterface
+getMap(position)
+fetchMapRequest(location)
+getMapAroundAddress(address)
EmailSystemManagerInterface
+sendRegistrationEmail(email, driverData, ...)
ReservationManagerInterface
+createReservation(timestamp, selectedCS, socketToReserve, driver)
+generateQRCode(CS, socket, driver)
DBMSInterface
+create(data)
+read(query)
+update(data)
+delete(query)
UserSession
+viewChargingStationsMap(location)
+register(name, surname, email, pwd, ...)
+viewChargingStationExternalStatus(selectedCS)
ChargeManagerInterface
+requestCharge(driver, reservation)
+startCharging(charge)
+stopCharge(charge)
+getActiveChargingStatus(driver)
PaymentManagerInterface
+addPaymentMethod(paymentMethodData)
+getPaymentMethods(driver)
+requestMoneyReservation(moneyAmount)
+requestPayment(moneyAmount)
CPMSApiHandlerInterface
+getCSExternalStatusData(selectedCS)
+checkSocketAvailability(selectedSocket)
+reserveSocket(selectedSocket, driverData)
+sendReservationToken(reservedSocket, driverData, timeOut)
+driverCheckedIn(driver, reservation)
+startCharging(charge)
+stopCharge(charge)
+getActiveChargingData(charge)
NetworkManagerInterface
+handleRequest(path, request)
+handleMessage(path, message)
GPSInterface
+getCoordinates()
DriverSession
+login(loginData)
+viewChargingStationMap(location)
+viewChargingStationExternalStatus(selectedCS)
+requestReservation(socket)
+startCharging(paymentMethodChoice)
+addPaymentMethod(cardNumber, cvv2, expirationDate, ...)
+stopCharging(activeCharge)
+viewActiveChargingStatus()
A&AInterface
+login(email, phoneNumber, pwd)
CSInfoInterface
+getChargingStationsInArea(area)
+getChargingStationData(selectedChargingStation)
RegisterManagerInterface
+registerDriver(name, surname, email, pwd, ...)
+activateAccount(accountToken)

Figure 19: eMSP Component Interface Diagram

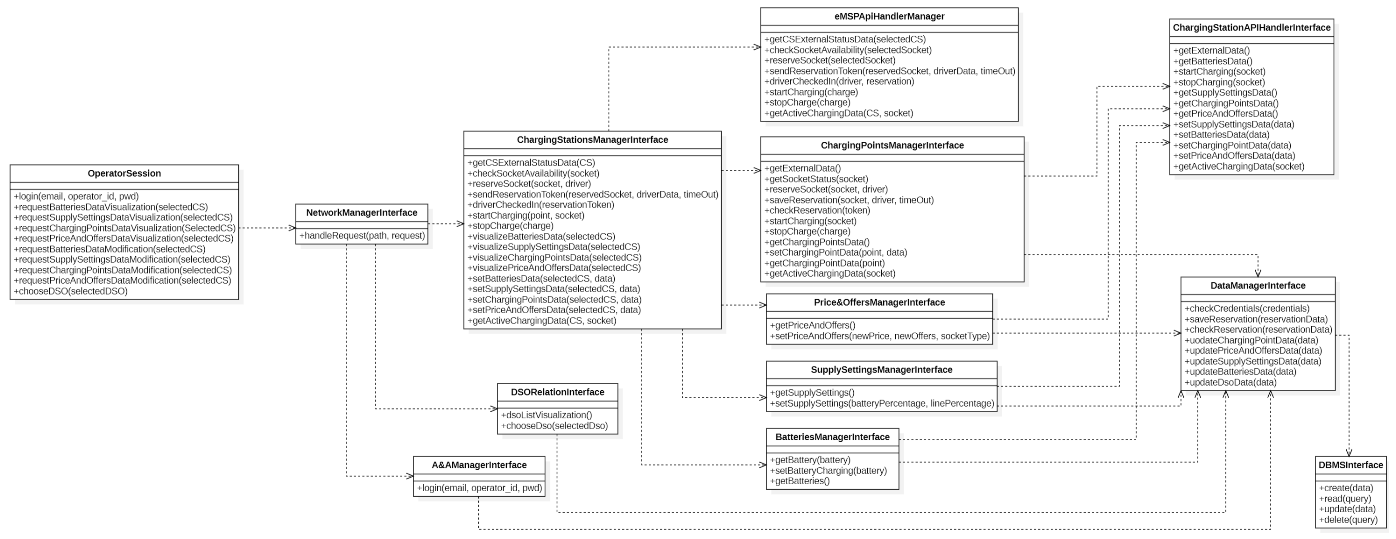

OperatorSession
+login(email, operator_id, pwd)
+requestBatteriesDataVisualization(selectedCS)
+requestSupplySettingsDataVisualization(selectedCS)
+requestChargingPointsDataVisualization(SelectedCS)
+requestPriceAndOffersDataVisualization(selectedCS)
+requestBatteriesDataModification(selectedCS)
+requestSupplySettingsDataModification(selectedCS)
+requestChargingPointsDataModification(selectedCS)
+requestPriceAndOffersDataModification(selectedCS)
+chooseDSO(selectedDSO)
NetworkManagerInterface
+handleRequest(path, request)
ChargingStationsManagerInterface
+getCSExternalStatusData(CS)
+checkSocketAvailability(socket)
+reserveSocket(socket, driver)
+sendReservationToken(reservedSocket, driverData, timeOut)
+driverCheckedIn(reservationToken)
+startCharging(point, socket)
+stopCharge(charge)
+visualizeBatteriesData(selectedCS)
+visualizeSupplySettingsData(selectedCS)
+visualizeChargingPointsData(selectedCS)
+visualizePriceAndOffersData(selectedCS)
+setBatteriesData(selectedCS, data)
+setSupplySettingsData(selectedCS, data)
+setChargingPointsData(selectedCS, data)
+setPriceAndOffersData(selectedCS, data)
+getActiveChargingData(CS, socket)
DSORelationInterface
+dsoListVisualization()
+chooseDso(selectedDso)
A&AManagerInterface
+login(email, operator_id, pwd)
eMSPApiHandlerManager
+getCSExternalStatusData(selectedCS)
+checkSocketAvailability(selectedSocket)
+reserveSocket(selectedSocket)
+sendReservationToken(reservedSocket, driverData, timeOut)
+driverCheckedIn(driver, reservation)
+startCharging(charge)
+stopCharge(charge)
+getActiveChargingData(CS, socket)
ChargingPointsManagerInterface
+getExternalData()
+getSocketStatus(socket)
+reserveSocket(socket, driver)
+saveReservation(socket, driver, timeOut)
+checkReservation(token)
+startCharging(socket)
+stopCharge(charge)
+getChargingPointsData()
+setChargingPointData(point, data)
+getChargingPointData(point)
+getActiveChargingData(socket)
Price&OffersManagerInterface
+getPriceAndOffers()
+setPriceAndOffers(newPrice, newOffers, socketType)
SupplySettingsManagerInterface
+getSupplySettings()
+setSupplySettings(batteryPercentage, linePercentage)
BatteriesManagerInterface
+getBattery(battery)
+setBatteryCharging(battery)
+getBatteries()
ChargingStationAPIHandlerInterface
+getExternalData()
+getBatteriesData()
+startCharging(socket)
+stopCharging(socket)
+getSupplySettingsData()
+getChargingPointsData()
+getPriceAndOffersData()
+setSupplySettingsData(data)
+setBatteriesData(data)
+setChargingPointData(data)
+setPriceAndOffersData(data)
+getActiveChargingData(socket)
DataManagerInterface
+checkCredentials(credentials)
+saveReservation(reservationData)
+checkReservation(reservationData)
+uodateChargingPointData(data)
+updatePriceAndOffersData(data)
+updateSupplySettingsData(data)
+updateBatteriesData(data)
+updateDsoData(data)
DBMSInterface
+create(data)
+read(query)
+update(data)
+delete(query)

Figure 20: CPMS Component Interface Diagram

## 2.6 Selected architectural styles and patterns

### 2.6.1 Multi-tier Architecture

As stated in the Overview, the chosen architecture for the System is a multi-tier architecture, with two partially distributed layers. This division guarantees that each tier only deals with a specific task, so that computation is correctly spread amongst the System and there isn't a central node in charge of everything. Also, modifications to a specific layer won't affect the others, so the overall maintenance of the System is much easier.

One tier is dedicated to the Client's physical device, which can be a mobile device (used by both Drivers and Operators ) or a desktop PC (used only by Operators directly accessing the Application Server of the CPMS). The Presentation Layer is completely managed by this tier and its aim is to display all the information received by the the Application Servers (eMSP or CPMS) to the User. This is done through a dynamic GUI on the mobile device that shows also the external services, such as Map visualization service. User's devices have also a partial part of the Business Logic Layer, in order to guarantee some functional requirements, such as timers that control reservation status or payment status.

One tier is dedicated to the Application Servers of eMall (eMSP and CPMS). In order to improve our System's performances, we have opted for a scale-out approach, so each server is replicated as many times as necessary to guarantee the non-functional requirements expressed in the RASD. In case of a sudden increase in the number of requests, we could seamlessly add more machines to handle the traffic, as each one is stateless. In fact, this tier only implements the Business and Domain Logic layer, but not the Data Storage layer.

One tier is dedicated to the Nginx Web Servers. We decided to use it because it functions both as a reverse proxy and as a load balancer, that is necessary in order to optimize the scale-out approach.
The purposes of these machines, used by eMSP and CPMS, are:

- to retrieve data from the their Application Servers and provide it to the Client , or to the other Application Server, as if the proxy were the server itself.
- to redirect requests to the idle servers, avoiding congestions on each one.

Nginx was chosen for its flexibility in handling requests and its effcient use of main memory, thanks to the event-driven approach used. This tier contributes to the implementation of the Business Logic layer, even though it doesn't do it directly, but it helps the Application Servers in doing it.

The last tier we present is dedicated to the Databases, one for eMSP and one for CPMS. In this tier there are the Databases and the DBMSs and it manages a part of Data Access Layer.

### 2.6.2 REST

The Representational State Transfer style has been used as a guideline to implement the services offered by the Application Servers, and their interactions with the Clients.
All services are accessible through stateless operations, allowing for a scalable System. HTTP is the communication protocol used underneath, so CRUD operations are implemented through GET, POST, DELETE and PUT primitives. Using HTTP and complying with REST guidelines allows to provide clear and complete APIs to the Clients, thus ensuring Portability.

### 2.6.3 MVCS Pattern

The software System's architecture will follow the MVCS paradigm.
The acronym stands for Model View Controller Store.

- The **Model** corresponds to the component containing all the application's data and integrity constraints. In our System the Models are completely described in the external databases.
- The **Controller** is responsible for the logic of the application and in our case is also stateless in order to be replicated and seamlessly support high traffic of requests. It corresponds to the business logic layers (eMSP and CPMS) of our System.
- The **View** corresponds to the presentation layer, the client's only interface with the System.
- Finally the **Store** component handles the fetching and saving of data and is responsible for the interaction between logic and storage. In our case it is represented by the DBMS Interface, the unique point of communication with the external cloud Database server in both eMSP and CPMS.

This best practice design pattern is commonly used in client-server applications due to its power of decoupling all main components thus increasing maintainability: modifications to one component will be transparent to others.

## 2.7 Other design decisions

### 2.7.1 Firewall

The Systems architecture includes different network firewalls that keep a separation between the reverse proxy and the external network and from the application server and cloud storage services.
The external network is regarded as untrusted and may represent a menace to the internal trusted network of the System.
Network firewalls operate by filtering the packets trying to pass through them.
The reverse proxy may have been used to perform firewall functions as well but it has been preferred to decouple packet filtering from packet dispatchment.

### 2.7.2 Database

It was decided to use a relational database model for both eMSP and CPMS, as opposed to a non-relational model, because it allows to represent the intrinsic relationships present in the data needed to be stored. Also, many complex operations will be made on the database, including Joins and Updates, which are not well handled by a non-relational model. For the data storage, it was decided to use a cloud approach. In order to take advantage of REST protocol, we can use *Microsoft azure SQL Database*, for example, for the DBMS of our application. It is well known cloud patterns: PaaS.

## 3 User Interface Design

User interfaces have already been described in section 3.1.1 of the RASD document where all mockups can be found.

# 4 Requirements Traceability

The system's design presented in previous chapters has been conceived with the goal of guaranteeing that all requirements specified in the RASD document can be satisfied. It is now presented the mapping between those requirements and the design components that ensure their fulfillment.

### 4.0.1 Requirements from RASD

Below, we will show the requirements that we have already mentioned in the RASD (section 3.2.4 ) as a reference for the Traceability Matrix.

- [$R$.1] = The system shall allow a unregistered User to register an account as a Driver.
- [$R$.2] = The system must allow registered Drivers to login.
- [$R$.3] = The system must allow registered Operators to login.
- [$R$.4] = The system shall allow a User, Driver and Operator, to see the map of CSs.
- [$R$.5] = The system shall allow a User, Driver and Operator, to select a CS on the map.
- [$R$.6] = The system shall allow a Driver to reserve a socket of a selected CS.
- [$R$.7] = The system shall allow a Driver to receive a unique QR-code ticket for a successful reservation.
- [$R$.8] = The system shall give a timer of 15 minutes to the Driver to scan the QR-code at the Charging Point (otherwise, the booking is rejected).
- [$R$.9] = The system shall be able to correctly associate the socket with the Driver who reserved it.
- [$R$.10] = The system shall allow a Driver to insert payment data.
- [$R$.11] = The system shall give a timer of 5 minutes to the Driver in order to connect the EV to the socket and insert valid payment data (otherwise, the booking is rejected).
- [$R$.12] = The system shall allow a Driver to visualize correct and real-time data about the charging process.
- [$R$.13] = The system shall allow a Driver to stop the charging process.
- [$R$.14] = The system shall allow the Driver to book a socket if and only if the Driver has no active reservations.
- [$R$.15] = The system shall allow the Driver to book a socket if and only if the socket is available.

- $[R.16]$ = The system shall allow a Operator to select one of the CSs managed.
- $[R.17]$ = The system shall allow the Operator to visualize data about the "internal status" of the managed CSs.
- $[R.18]$ = The system shall allow the Operator to visualize the status of the batteries of a charging station, if any.
- $[R.19]$ = The system shall allow the Operator to charge the batteries of a charging station, if any.
- $[R.20]$ = The system shall allow the Operator to select the price and the offers of a certain type of socket.
- $[R.21]$ = The system shall allow the Operator to select the supply settings of the CSs.
- $[R.22]$ = The system shall allow the Operator to visualize the list of available DSOs, with their prices and details.
- $[R.23]$ = The system shall allow the Operator to visualize the current DSO from which the CPO is acquiring electric energy.
- $[R.24]$ = The system shall allow the Operator to change the current DSO with another one from the list of available DSOs.
- $[R.25]$ = The system shall allow a correct and coherent communication between the operations of the CPMS automatic system and the manual interventions handled by the Operators.
- $[R.26]$ = The system shall be able to notify Drivers or Operators of incorrect actions.
- $[R.27]$ = The software of the CPs must be consistent with the system.

Since, we have some components with the same name in both eMSP and CPMS (e.g. NetworkManager, DataManager etc...), to distinguish the components of the two subsystems we will put at the end of the name of the component (eMSP) if it is part of the eMSP subsystem, otherwise (CPMS) if is part of the CPMS subsystem.

### 4.0.2 Traceability Matrix

| **Requirement (RASD)** | **Component (DD)** |
|---|---|
| R.1 | eMallApplicationClient<br>NetworkManager(eMSP)<br>RegisterManager<br>DataManager(eMSP)<br>eMSP DBMS<br>EmailSystemManager |
| R.2 | eMallApplicationClient<br>NetworkManager(eMSP)<br>Authentication&AuthorizationManager(eMSP)<br>DataManager(eMSP)<br>eMSP DBMS |
| R.3 | eMallApplicationClient<br>NetworkManager(CPMS)<br>ChargingStationsManager<br>Authorization&AuthenticationManager(CPMS)<br>DataManager(CPMS)<br>CPMS DBMS |
| R.4, R.5 | eMallApplicationClient<br>GPS<br>MapProvider<br>NetworkManager(eMSP)<br>ChargingStationInformationManager<br>CPMSApiHandler<br>eMSPApiHandler<br>ChargingStationManager<br>ChargingPointManager<br>ChargingStationApiHandler |
| R.6, R.14, R.15 | eMallApplicationClient<br>NetworkManager(eMSP)<br>ReservationManager<br>DataManager(eMSP)<br>eMSP DBMS<br>CPMSApiHandler<br>eMSPApiHandler<br>ChargingStationManager<br>ChargingPointsManager<br>DataManager(CPMS)<br>CPMS DBMS<br>ChargingStationApiHandler |
| R.7, R.8, R11 | eMallApplicationClient<br>NetworkManager(eMSP)<br>PushNotificationManager<br>ReservationManager<br>DataManager(eMSP)<br>eMSP DBMS |

| | |
|---|---|
| R.9 | eMallApplicationClient<br>ChargingStationAPIHandler<br>ChargingPointsManager<br>DataManager<br>CPMS DBMS |
| R.10 | eMallApplicationClient<br>NetworkManager(eMSP)<br>PaymentManager |
| R.12, R.13 | eMallApplicationClient<br>NetworkManager(eMSP)<br>ChargeManager<br>CPMSApiHandler<br>eMSPAPiHandler<br>ChargingStationManager<br>ChargingPointManager<br>ChargingStationAPIHandler |
| R.16, R.17 | eMallApplicationClient<br>NetworkManager(CPMS)<br>ChargingStationsManager<br>ChargingPointsManager<br>ChargingStationAPIHandler |
| R.18, R.19 | eMallApplicationClient<br>NetworkManager(CPMS)<br>ChargingStationsManager<br>BatteriesManager<br>ChargingStationAPIHandler |
| R.20 | eMallApplicationClient<br>NetworkManager(CPMS)<br>ChargingStationsManager<br>PriceAndOffersManager<br>ChargingStationAPIHandler<br>DataManager(CPMS)<br>CPMS DBMS |

| | |
|---|---|
| R.21 | eMallApplicationClient<br>NetworkManager(CPMS)<br>ChargingStationsManager<br>SupplySettingsManager<br>ChargingStationAPIHandler<br>DataManager(CPMS)<br>CPMS DBMS |
| R.22, R.23, R.24 | eMallApplicationClient<br>NetworkManager(CPMS)<br>DSORelationsManager<br>DataManager(CPMS)<br>CPMS DBMS |
| R.25, R.27 | NetworkManager(CPMS)<br>ChargingStationManager<br>DSORelationmanager<br>ChargingStationAPIHandler |
| R.26 | eMallClientApplication<br>NetworkManager(eMSP)<br>NetworkManager(CPMS)<br>PushNotificationManager |

Table 3: Traceability Matrix

# 5 Implementation, Integration and Test Plan

### 5.0.1 Overview

In the design of the System we followed a top-down approach: first of all we presented an high abstraction level of its components and macro functionalities and then we analyzed it in detail under different point of view.
We can say that our System is composed by three different subsystems that are:

- The Client subsystem, containing presentation, some business logic and temporary data storing.
- The Server subsystem, containing most of the components related to Business Logic and interaction with the two storage systems.
- External services, such as : storage systems, Maps service etc.

Our implementation and testing approach will be incremental and we will follow different strategies that are discussed in detail with their advantages in the following paragraphs.

### 5.0.2 Implementation

The order of implementation takes into account the necessity of parallelising the workload between developers, and can be identified as:

- Setup of the Database and creation of the Application Client.
- Implementation of the components that connect eMSP and CPMS servers.
- Implementation of the server and client subsystems at once.
- Integration with external services.

As much as regards this order of implementation, considering that it has been thought that implementation and testing of the components must go on almost in parallel (as a component is implemented, it also has to be tested), the database's set-up was a priority to allow for more efficient testing throughout the implementation's progression. Having the database up and running would allow real data usage and would allow to quickly catch, solve and exclude database connection-related problems, shrinking the possible error sources in the code. Then the server and client implementation would follow, with their components' implementation order that will be soon presented, and lastly the external devices would be added and set-up to better ensure the functionalities of components implemented.
Now we present and explain the server's and client's components implementation order, using the following diagram as an aid for a better understanding.

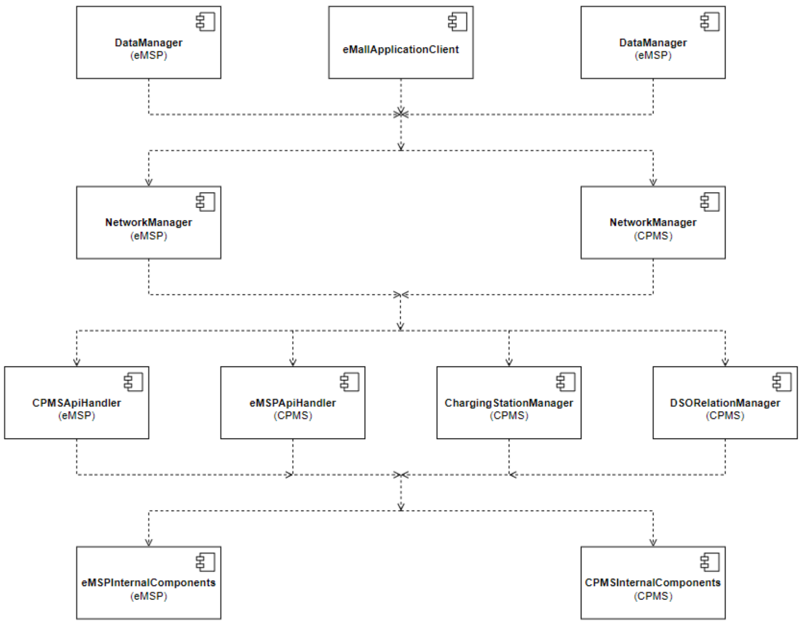


Figure 21: eMall implementation components

The scheme above (fig. 21) shows the implementation of the components of the system. Having already completed the set-up of the database, it's pivotal to implement first the components which control access to the databases from the servers, which are the DataManagers, while implementing the GUI for the client (eMallApplicationClient)in parallel: this will allow to improve the testing quality by excluding errors related to the connection to the databases, while also allowing the client-side app to have a functioning interface. Then there will be the implementation of the Network Manager for both the client and the server, to allow the connection management and testing, followed by the implementation of the ChargingStationManager and the APIHandlers: CPMSApiHandler, eMSPApiHandler, ChargingStationManager, DSORelationManager. These last components are core to eMalls' functioning, so their implementation and testing are to be done very minutely. Then, after and only after having implemented (and tested) both the connection between server and client, the Data Managers and the ApiHandlers, the other components of the two subsystem (RegisterManager, PaymentManager, PriceAndOffersManager, batteriesManager etc.) will be implemented and tested.

### 5.0.3 Unit Testing

In this part we want to explain how to test each component independently. In this case we prefer to use an incremental approach: unit tests will be added as new code is written. During the whole duration of the implementation phase, unit tests will be performed in order to spot as soon as possible bugs and flaws inside the System components. This is essential to be able to fix them with the lowest cost of repair in terms of effort and time and it also helps guiding the design of the System.
A support tool can be used to perform tests execution automatically once a new version of the software is available.
An automated test tool will be used for two different reasons:

- It saves time for developers that do not have to manually perform all tests every time they add or change something.
- It allows to make sure that a test that was valid for a previous version of the code still holds after some changes or additions in order to preserve consistency and compatibility.

Unit tests are focused not only on internal behavior but also on single modules' interaction with the System (fig. 22).

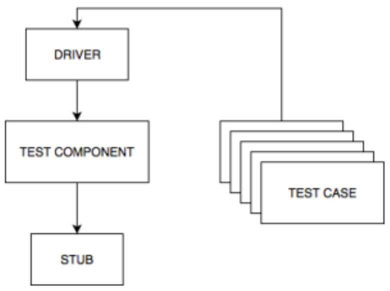


Figure 22: Test scaffolding

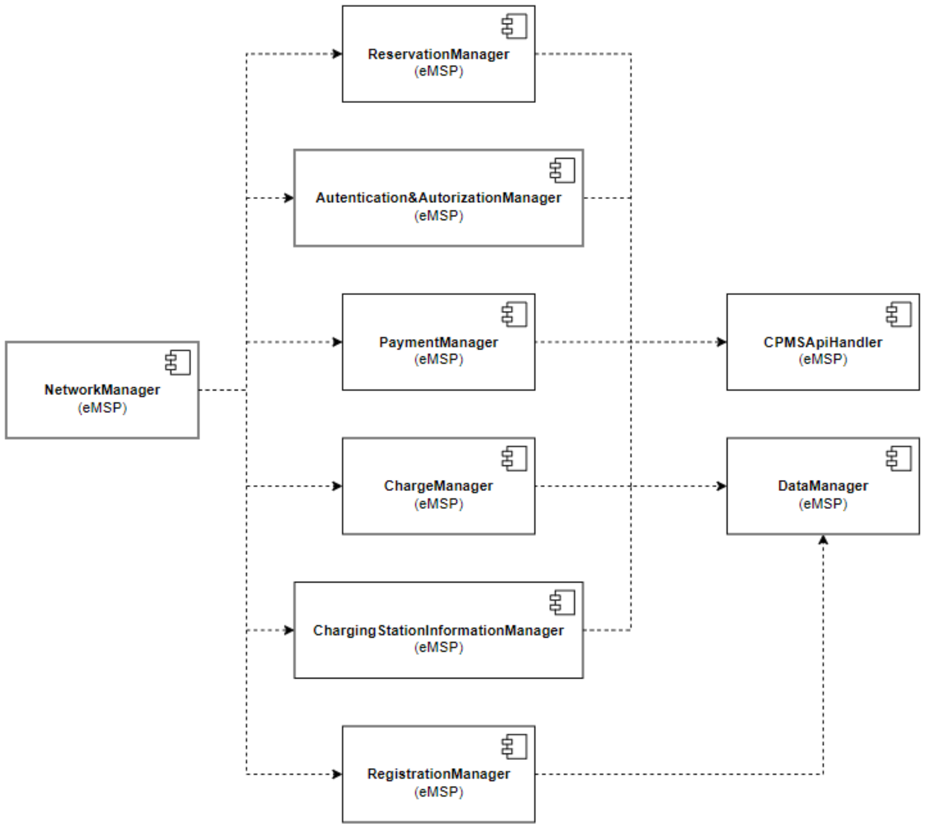
In order to make the single module work even in isolation,*stubs* and *drivers* are used to replace the missing software and simulate the interface between the software components.
*Drivers* will be used to simulate calls to a certain component.
*Stubs* will be used to simulate the calls made by a component under testing.

### 5.0.4 Integration Testing

Once we have defined how we will approach the implementation of each component of our System and how to test it independently, we shall focus on how to test their interaction and validate their joint behavior. Integration between components means testing groups of components that depend on one another in order to assess their interaction and possibly expose any defects.

We will perform integration testing at two different levels: at subsystem level, meaning between components of the same macro System such as modules of the Controller or of the View, and at a System level, linking all systems together.

As previously mentioned, subsystem integration will be performed within a macro System and will follow an incremental approach as done during implementation and testing. This approach will follow a top down procedure according to the use hierarchy of modules. Previously created stubs in fact will be now substituted by real components. The only stubs and drivers we will use in this type of integration will be those of the Client (*driver*), the DBMS(*stub*).
We will adopt this approach both for eMSP and CPMS's architectures. In order, we will first integrate the first two levels of the use hierarchy and proceed until the very last that is composed by the DataManager, the component with functions that are called by all other components and that interacts with external services.



ReservationManager (eMSP)
Autentication&AutorizationManager (eMSP)
PaymentManager (eMSP)
CPMSApiHandler (eMSP)
NetworkManager (eMSP)
ChargeManager (eMSP)
DataManager (eMSP)
ChargingStationInformationManager (eMSP)
RegistrationManager (eMSP)

Figure 23: eMSP Server SubSystem

The scheme (fig. 23) represents the use hierarchy of modules within the eMSP's server, the order of integration we have opted for follows modules from the root to the leaves.

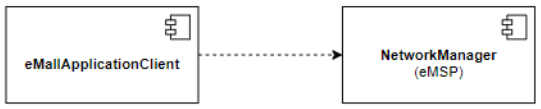


Figure 24: eMSP Client SubSystem

The simplified version of scheme (fig. 24) instead contains the use hierarchy within the eMSP client and shows our integration flow (from the root to the leaves).

As mentioned above, the same approach will be adopted with CPMS's subSystems.

After having integrated each component within a System, we will focus on integrating all systems together. Our strategy in this case will be a bottom up approach based on the use hierarchy. At the very bottom level there is the Model of our System, the first element that we have created and that is interrogated by all other modules. At first we shall integrate Controller and Model in order to test calls from the former to the latter. When integrating these two components we will also use some kind of services (for example Postman) to simulate client requests.
Secondly, we will implement the integration and testing the communication between eMSP and CPMS, it is essential to test integration between the two systems in order to guarantee a seamless communication in front of the Users and also a good work of the whole system.
Also in this case, we can use some kind of services to simulate client requests, for testing if all the complete system works well.
Afterwards, we will add the Client to our previously integrated systems and verify the complete functioning of our System.
The Client corresponds to the root of our hierarchy as it interacts only with the controller that handles its requests and communicated with the database.
Finally, when performing System integration, we will also include external services, such as: Push notification Service and Email service. Those used by the Server will be integrated with it after integration with the Model. Those instead used by the Client will be included at the very end.

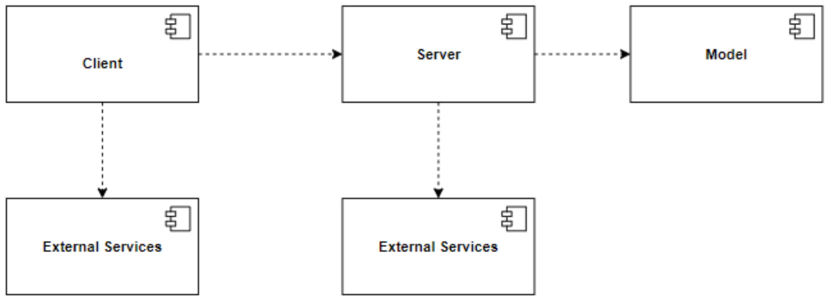


Figure 25: System Testing

The scheme above (fig. 25) represents how our System test will be performed.
It is true both for eMSP and CPMS subsystems. Arrows represent the use hierarchy, meaning that the arrows start from modules that use those corresponding to the end points. As mentioned before, our integration procedure will begin from the leaves to the root.

## 6 Efforts

| **Individual Work** | | |
|---|---|---|
| | ***Eutizi Claudio*** | ***Perego Gabriele*** |
| **Tasks** | **Hours** | **Hours** |
| Introduction (section 1) | 3 | 3 |
| Architectural Design (section 2) | 18 | 12 |
| User Interface Design (section 3) | 1 | 1 |
| Requirements Traceability (section 4) | 4 | 6 |
| Implementation,Integration and Test Plan (section 5) | 3 | 7 |
| Final Revision | 6 | 6 |
| **Total** | **35** | **35** |

Table 4: Time spent by each team member