Politecnico di Milano
School of Industrial and Information Engineering
Master of Science in Computer Science and Engineering

Software Engineering 2 Project

eMall - eMSP system

Design Document

Authors:
Federico Bono
Daniele Cipollone

Academic Year 2022/2023

Milano, 08/01/2023
Version 1.0

Contents

List of Figures

1 Introduction

1.1 Purpose

The purpose of this document is to describe and explain in details the design choices for the development and deployment of the eMSP system of eMall. The description is structured on multiple levels to give an overview of the system from multiple viewpoints. In the following pages you will find:

1. High level overview of the architecture
2. Overview of the components of the system
3. Deployment overview
4. Overview of the interactions between components
5. Overview of the interfaces offered by the various components
6. The UI of the mobile application used by the final user
7. The patterns and technologies used in the system

1.2 Scope

eMall App is a platform that helps the end users to plan the charging process, by getting information about Charging Points nearby, their costs and any special offer; book a charge in a specific point, control the charging process and get notified when the charge is completed. It also handles payments for the service.

In the e-Charging ecosystem, there are many different actors involved that we need to keep into consideration while collecting requirements and designing the system. The first information to consider is that Charging Points are owned and managed by Charging Point Operators (CPOs) and each CPO has its own IT infrastructure, managed via a Charge Point Management System (CPMS).

In order to comunicate with the various CPMS, the OCPI (Open Charge Point Interface) protocol is used.

To be able to process payments, the system will need to communicate with a Payment Service Provider (PSP) via the HTTP(s) protocol, using the propertary APIs offered by the provider.

Given that our system is not the producer of the data, and that there is the need for implementing different functionalities (e.g. payments) a three-tier architecture has been chosen, to separate the data layer (that mostly acts as a cache layer) and the business logic layer.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

1.3.2 Acronyms

1.4 Revision History

- v1.0 - 05 January 2023

1.5 Related Documents

- eMSP RASD (RASD_eMSP.pdf)
- OCPI specifications document (OCPI-2.2.1.pdf)

1.6 Document structure

The document is structured in six sections:

1. Description and introduction of the various design choices made during the design of the system. Descriptions are written at different levels of abstractions: from the general point of view to the detailed view of the single component.
2. User interfaces and design mockups.
3. The requirement traceability matrix is used to map each component to the requirement(s) that fulfils.
4. Implementation and test plans for the entire system
5. Total effort
6. References used

2 Architectural Design

2.1 Overview: high-level components and interactions

As anticipated in the previous chapter, the architecture selected for the design and development of the system is the three-tier architecture.

This architecture allow us to split the implementation into three layers:

1. Presentation: is the mobile application that will be used by the final users. It allows all the interactions with the system and will also be used as a communication endpoint for the notifications.
2. Application: is the backend of the system, all the business logic and the various connections between the system and the external services are implemented here.
3. Data: is the layer responsible to expose connectivity interfaces from the database. It will be a DBMS.

All the layers communicate in a linear way: the Presentation one interacts only with the Application layer, the same as the Data layer. With this architecture the presentation and the Data layers have no direct communication path, this allow to develop all the business logic only in the Application layer. Other advantages of using this architecture are that the various layers can be developed with different technologies and that they can be duplicated and differentiated (i.e. there can be multiple presentation layers that interact with the same application layer)

The main reasons behind the choice are:

- We are not the producer of the data
- We need to integrate different external services
- Separate the business logic from the data to:
  - Allow a parallel development with multiple teams specialized in the single tiers
  - Allow to use different technologies for the different tiers

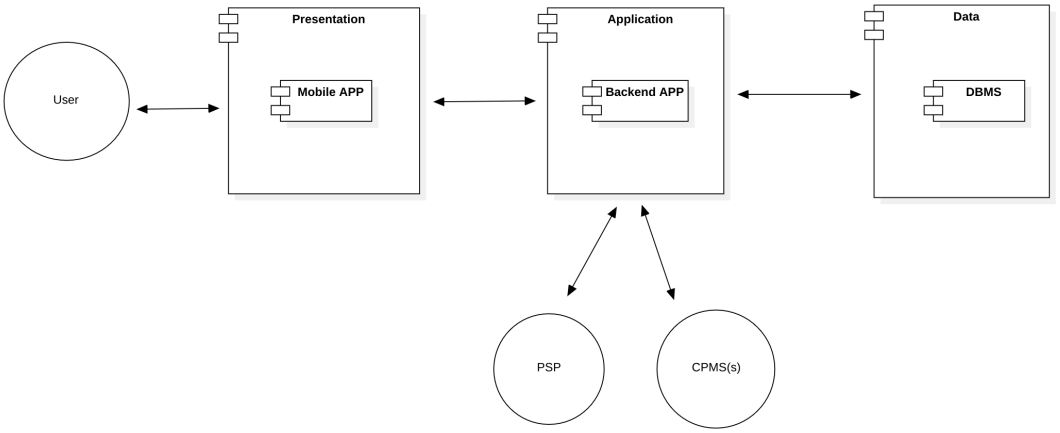

Figure 2.1: Overview of the chosen three-tier architecture with actors

2.2 Component view

The following schema shows all the main components and interfaces of the system. Later on you will find the details of each component.

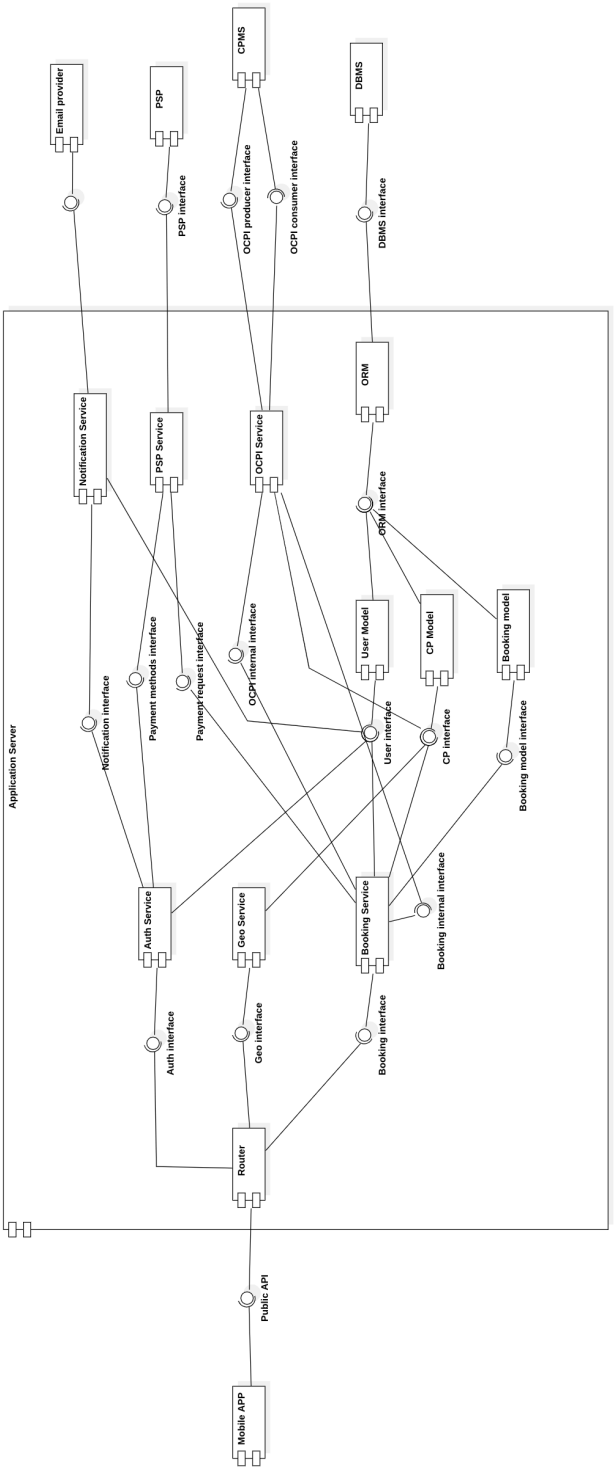

Figure 2.2: Component diagram of the system

- **Application server**: not a real "component" it shows the division between the backend, the frontend, the data layer and the external service. In the three-tier architecture it represents the Application layer.
- **Mobile APP**: mobile application for the system, used by the users.
- **Router**: handles all the requests directed to the Application APIs (i.e. OCPI PUSH endpoints are not handled here) that comes into the Application Server, uses the Auth Service to Authenticate and Authorize them. After that, if all the checks are passed, it will forward the request to the correct service that exposes the required route. It basically acts as a middleware, its presence is useful as it acts as a single point to develop all the authentication/authorization logic so that if it needs changes we can just update the router (and eventually the auth service) instead of updating all the services.
- **Auth Service**: is an internal service that handles authentication (Signup and login) and authorization (even if at the moment our system does not need that). It is mainly used internally but it exposes a couple of endpoints to the router for both signup and login.
- **Geo Service**: an highly reusable service, it uses a set of geographical points and exposes functionalities to list, filter and search those points based on their position. For our system is needed for the map functionality of the Mobile APP.
- **Booking Service**: it manages the booking for the application, it exposes some endpoints to the router for the booking functionalities of the Mobile APP and it is also used from other internal services.
- **Notification Service**: it manages all the notifications that the system needs to send to the user, both via email as via push services (e.g. Firebase Cloud Messaging)
- **PSP Service**: it manages the communication with the Payment Service Providers, it is useful as we can easily swap it with proprietary SDKs from the various providers.
- **OCPI Service**: it manages the communication with the CPMS, it exposes the endpoints for the PUSH part of the protocol (is needed to receive real-time updated from CPMS)
- **ORM**: a library that allows an easy to use mapping between Models and DB table, it handles queries and relationships. It is not worth it to develop an in-house solution, so we will use a library for that.
- **User Model, CP Model, Booking Model**: Models components that sit between services and ORM, useful to attach event listeners and other effects that need to run when updating the data.
- **Email Provider**: external service (or services) that exposes API to send emails
- **PSP**: external service that exposes API to process payment
- **CPMS**: Charging Point Management System that exposes OCPI compliant API
- **DBMS**: Database Management System

2.2.1 Class diagram

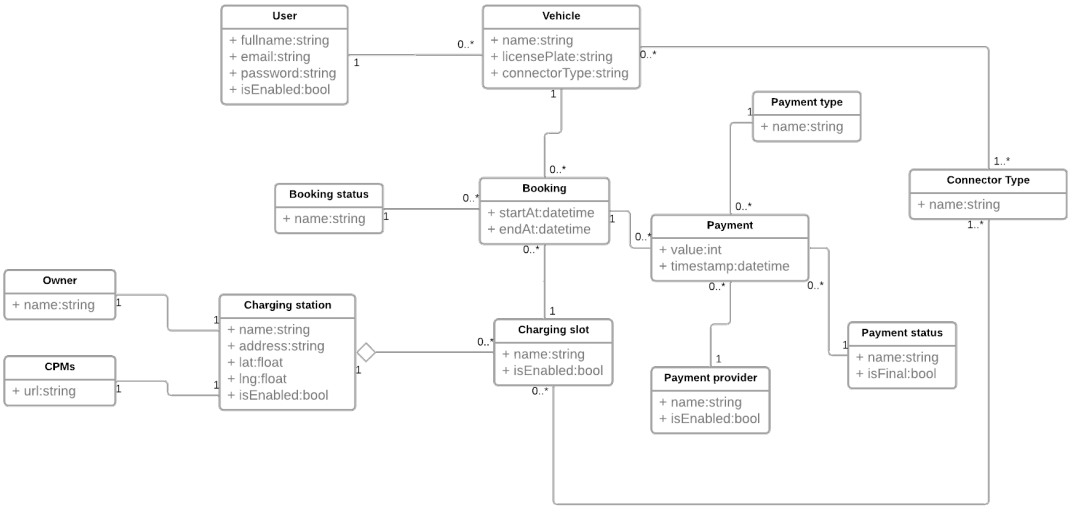

Figure 2.3: Class diagram for the main models of the System

2.3 Deployment view

In this section we introduce a detailed view of the system and various component from a deployment cycle perspective. The following schemes highlight the environments and the tools to be used for the system.

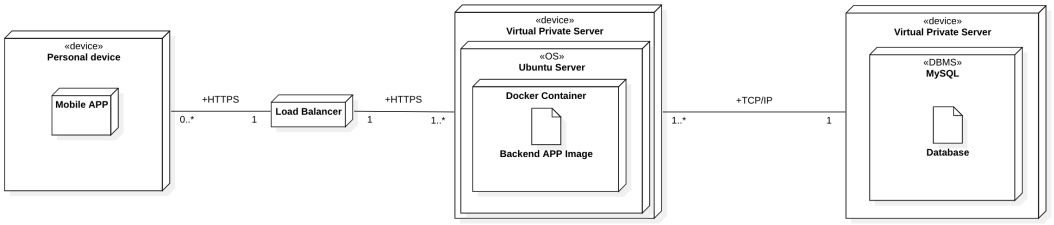

Figure 2.4: Deployment diagram for the System

- **Personal Device**: Mobile device used by users to run the Mobile APP to communicate with the system
- **Load Balancer**: A web server that balances incoming requests between multiple backend servers
- **Backend Server / Virtual Private Server**: Physical device where is installed the Docker Image of the Backend APP. Multiple replicas are possibile and preferable as they will increase both reliability and availability of the system. Keep in mind that without replicating the data layer, there will always be a bottleneck of performances due to the communication between application and data layers.
- **Database Server**: Host the database of the system. Multiple local replicas can improve read performances but also increase complexity.

2.4 Component interfaces

In this section we introduce the main interfaces of the various components of System.

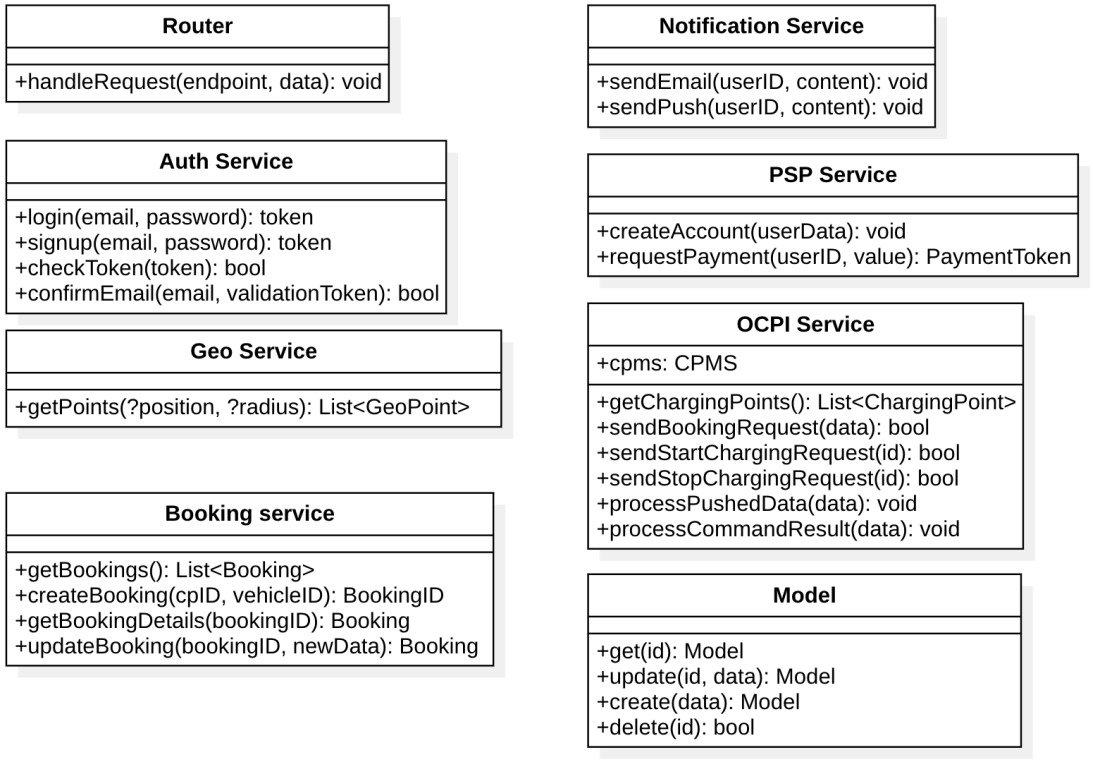

Figure 2.5: Component interfaces diagrams for the system

2.5 Runtime view

In this section are shown the sequence diagrams for the various functionalities of the system. To simplify the description there is an initial generic sequence the summarize the routing and authorization part of the request handling. Also, as all the request go through the router, the actors that sends the request to the router are omitted.

2.5.1 Generic Routing and Authorization

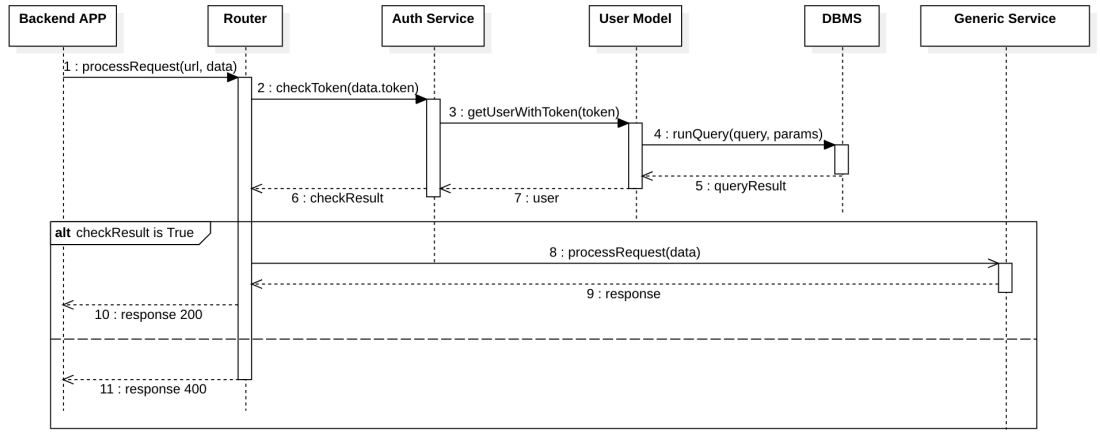

Figure 2.6: Generic Routing and Authorization flow for a generic request

When a request is received by the application web server, it goes through the Router component. It acts as an authorization middleware. As Authorization is needed, the whole Auth Service, Model and DBMS components are envolved. At the end, if authorization has been completed successfully, the request is forward to the dedicated service.

2.5.2 View Charging Point locations

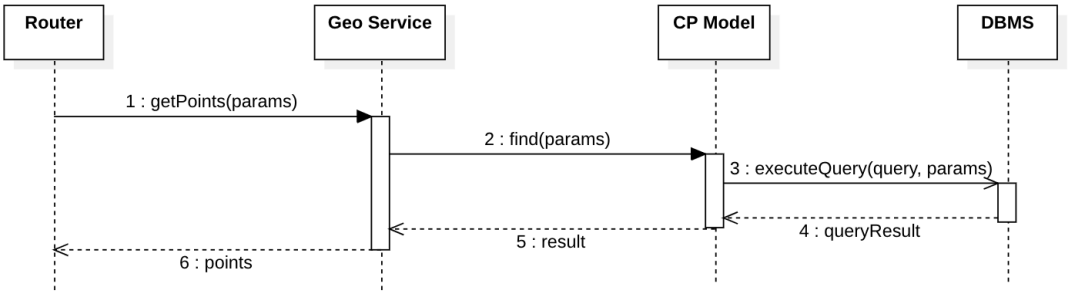

Figure 2.7: View Charging Point locations

2.5.3 Book a charging session

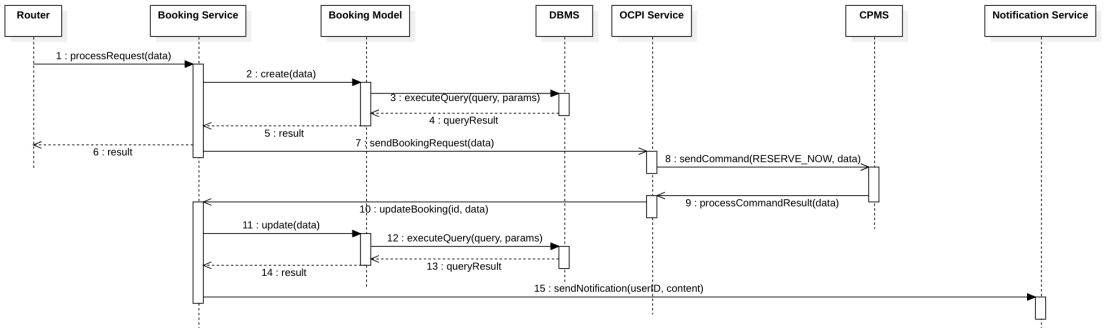

Figure 2.8: Book a charging session

2.5.4 Start a charging session

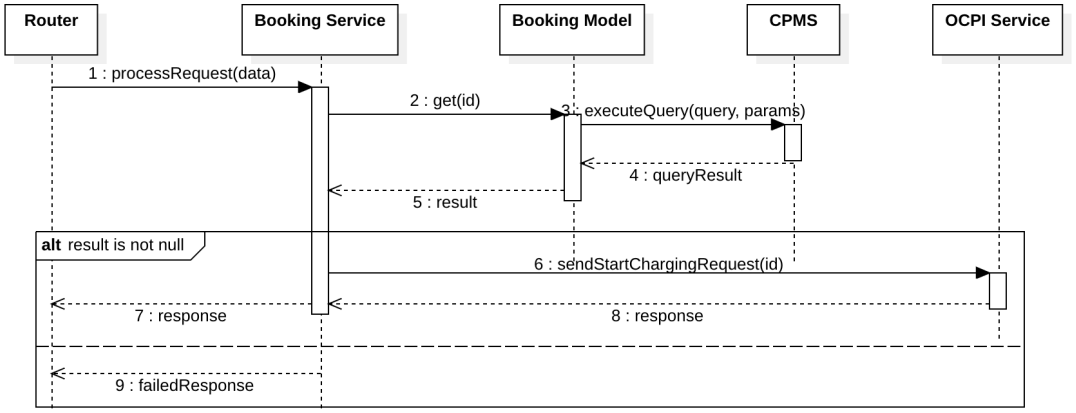

Figure 2.9: Start a charging session

2.5.5 Pay for a completed charging session

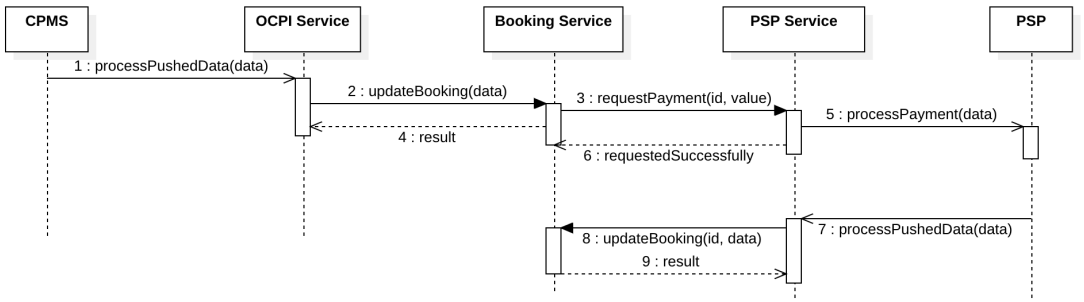

Figure 2.10: Pay for a completed charging session

2.6 Selected architectural style and patterns

- **Three-tier architecture**: As stated before the three-tier architecture was chosen because our system is not the producer of the data, and that there is the need for implementing different functionalities (e.g. payments) and external services.
- **Monolith architecture**: To simplify the development and the deployment of the system, to avoid the so called "vendor lock-in" by choosing a specific service provider for the microservices infrastructure and mainly because no usage spikes are expected, the monolith architecture has been chosen. That means that all the application server replicas contains the whole backend application and not the single services.
- **Authentication with Access Token**: As there are no specific requirements on the authentication functionality, the simply access token pattern can be considered a good suit for the system.

3 User Interface

3.0.1 Search for a Charge Point

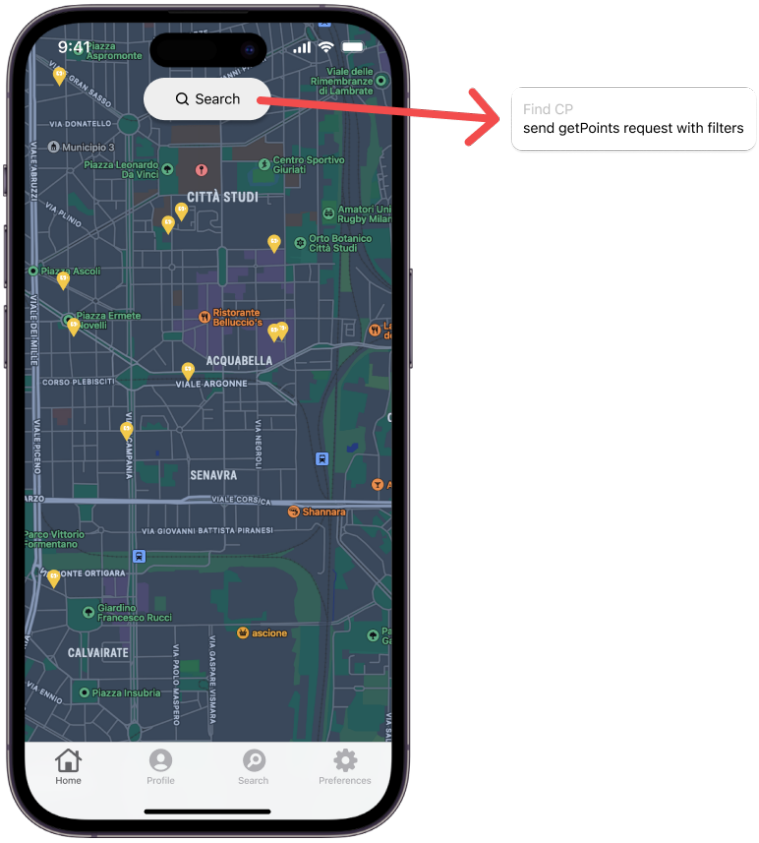

Figure 3.1: Search for a Charge Point

In this first view, the user has the ability to view all the CP nearby via the map. The datapoints are loaded via an API request to the Geo Service that returns all the points. The user can also search for a specific CP using both the "Search" button on the upper part of the map, as also using the tab button on the bottom of the application.

3.0.2 Details of a CP and booking of a session

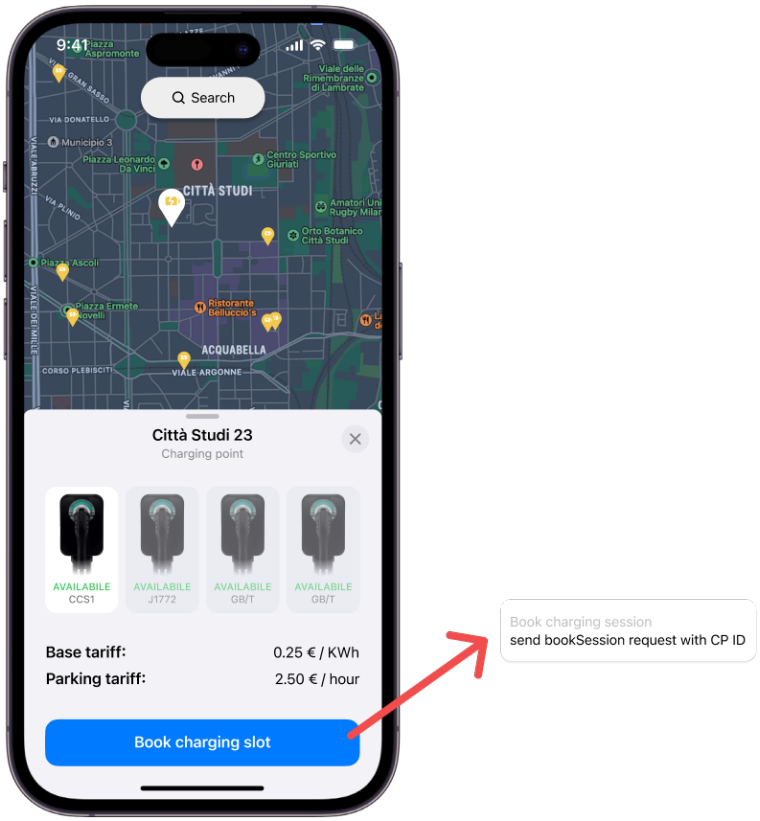

Figure 3.2: Details of a CP and booking of a session

After tapping on a CP icon on the map, the user will be presented with the details of the selected CP, with all the plugs available and the tariffs. Then, the user can decide to book the charging slot by selecting the connector type that he needs and by tapping onto the "Book charging slot" button. A bookSession request is forwarded to the Booking Service, than the OCPI Service is used to send the correct request to the selected CPMS. After some time the CPMS will push the session to the eMSP system and the Notification Service will send a push notification to the user and will update the UI

3.0.3 Starting the charging process

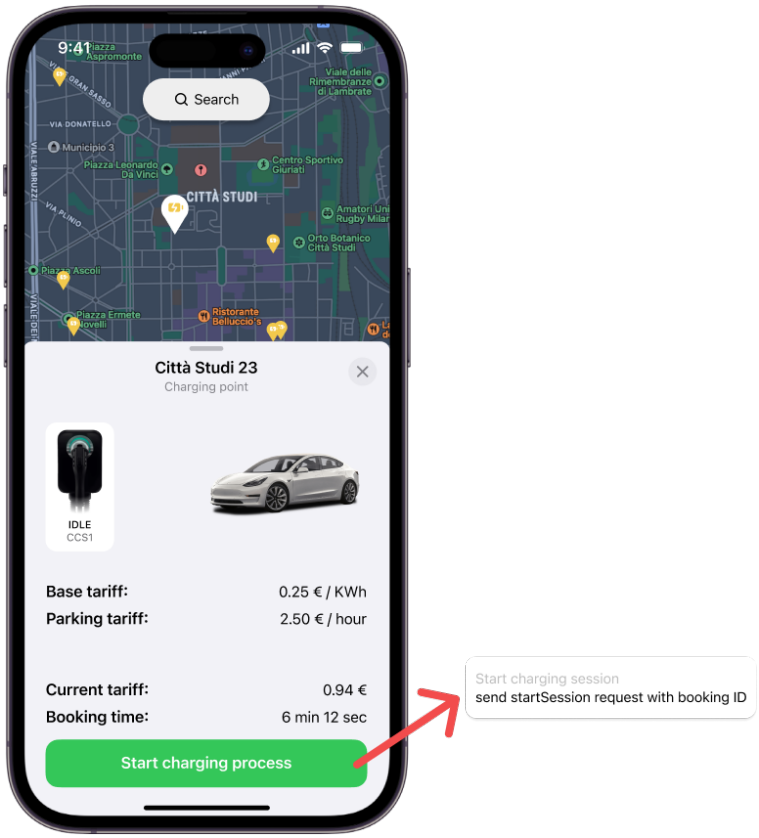

Figure 3.3: Starting the charging process

After the update received by the CPMS, a session will start and the booking details will be updated to show the current tariff that will be applied and the booking time already passed. The user, once he's reached the CP and has connected the EV to the station, can start the charging process via the "Start charging process" button. The startSession request will be sent to the Booking Service that will forward the request to the CPMS by using the OCPI Service. As the previous request, the CPMP will push the update on the booking session.

3.0.4 Completing the charging process

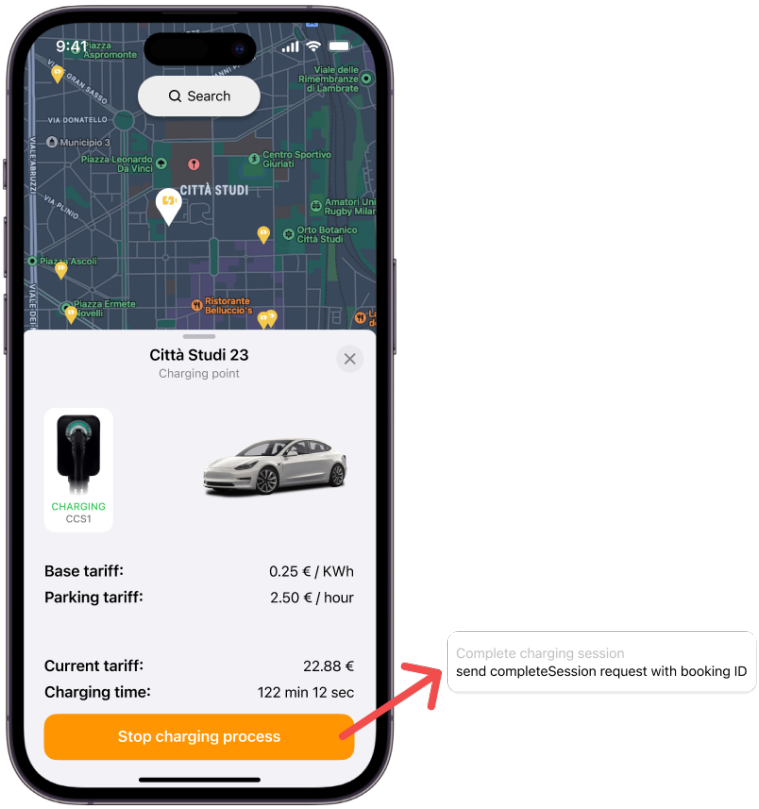

Figure 3.4: Completing the charging process

At any time the user can view the status of the booking, updated with the latest info pushed by the CPMS. He can also complete the charging process by tapping the "Stop charging process" button. A completeSession request will be forwarded via the Booking Service to the OCPI Service that will send the correct command to the CPMS. After the confirmation from the CPMS, the user will be charged automatically via the PSP Service with the complete tariff communicated by the CPMS.

3.0.5 Push notification from Charging Point

Figure 3.5: Push notification from Charging Point when the charging session has been completed

When the CPMS push an update to the system to notify the end of the charging process, the System will send a push notification to the user, reminding them that the parking fares might be applied.

4 Requirements Traceability

4.1 Functional requirements

In the following table are summarized the requirements extracted from the RASD.

Requirement	Description
R1	Allow Guest User sign-up
R2	Allow Login for Active User
R3	Require verification for email
R4	Require verification for payment method
R5	Allow to get a list of Charge Point
R6	Allow to book a charging session on a specific Charge Point
R7	Allow to start a charging session for a specific booking
R8	Allow to view the list of bookings
R9	Allow to complete a charging session
R10	Forward the payment request to the PSP
R11	Allow to get a list of Charge Point based on the location
R12	Allow to filter the list of Charge Point based on the location
R13	Be able to send push notifications to the user device
R14	Periodically poll the various CPMS to cache the data

4.2 Components mapping

To keep the mapping simple, we define some abbreviations for the components.

	Component
AS	Application Server
APP	Mobile APP
RT	Router
AT	Auth Service
GE	Geo Service
BS	Booking Service
NS	Notification Service
PS	PSP Service
OC	OCPI Service
OR	ORM
MD	Models (User, CP, Booking Model...)

4.3 Components mapping on Requirements

Comp.	Requirements	Reason
AS	R*	Application server is needed for all the functionalities as it is the foundation for the system
APP	R1-9,R11-13	Mobile APP is the medium for the communication between the user and the system
RT	R1-3,R5-9,R11-12	Router is the middleware component for the "frontend" functionalities.
AT	R1-4	Authentication Service allows to check the state of the user (e.g. Active, Pending or Guest)
GE	R5,R11-12	Geo Service is the service that allows to show the items on the map and apply filters based on the user's location
BS	R6-10,R13	Booking Service manages the "state machine" and the related effects for the charging sessions.
NS	R13	It allows to communicate with the external service providers
PS	R10	It allows to communicate with the Payment Service Provider
OC	R5-12,R14	It allows the bidirectional communication with the various CPMS
OR	R*	It allows to easily access the DBMS APIs and to manage the relationships between the models.
MD	R*	It allows to access the ORM interfaces and to trigger functions on models' events (e.g. send a notification when charging is completed)

5 Implementation, Integration and test Plan

5.1 Plan details

To implement the system components we've decided to follow a planned structure, dividing the components into various groups in order to be able to develop them in parallel, using a more complex workforce. Given the limited number of functionalities, we think that a progressive release strategy (i.e. with multiple smaller releases) is not really worth for the final user, that means that the only way to bring value to the user will be to have the system released all at once.
This assumption allows us to follow different strategies, both bottom-up and top-down.
To keep aligned the tests set and the actual codebase, we decided to follow the famous TDD (Test Driven Development) strategy, where unit/integration tests are written before the code and then the actual codebase is developed to fulfil those tests.
To develop the best user experience possible, we decided to follow the top-down approach, splitting the development of the various components by functionalities viewed from the user's point of view. We've can use the goals defined in the RASD as a starting point.

We've defined the following functionalities:

1. Signup, Login, Email confirmation and payment method registration
2. View the CP locations, filter them and view the details of a location
3. Book a charge session at a specific location for a specific connector
4. Start and complete a charge session from the app for a specific booked session
5. View the list and the details of the booked sessions
6. Be notified when the charging process has been completed
7. Be charged with the correct amount after the charging process

Based on those functionalities we can define the following development path:

1. **Application scaffolding**
   In this phase we set up the servers (from a logical point of view, through docker), the frameworks and the connections with the DBMS. We use the Router component given by the framework, so it can be considered as being developed/integrated in this phase.

2. **Authentication and notification scaffolding**
   In this phase we develop the Authentication Service, the Notification Service and we connect with the chosen Email Provider to send the email confirmation link to the user. The related UI and Models are also developed in this phase, alongside with the needed database migrations.

3. **Payment method verification**
   As we delegate the verification to the PSP, we only need to develop the PSP Service on the backend (with the related models, tables and DB migrations) and integrate the UI on the mobile APP.

4. **OCPI service integration**
   We use some open source libraries that implements the needed interface for the OCPI (both PULL and PUSH methods), and wrap it with a utility service (i.e. the OCPI Service) to expose the chosen interfaces. As we are following the top-down approach we only set up the libraries and the base service.

5. **CP list and details**
   We start to develop the needed interfaces for the OCPI service and then we can start implementing the Geo Service to allow the user to view all the needed information.

6. **Booking Service**
   In this phase we start to develop the functionalities to book a charge from a specific CP, we need to expose those functionalities from the OCPI Service and then we can implement the needed models alongside with the Booking Service and the Mobile APP UI.

7. **Charging functionalities**
   In this phase we add the charging management functionalities to both OCPI Service and Booking Service.

8. **Booking status and list**
   In this phase we use the PUSH interfaces of the OCPI libraries to update our booking model data. We also add the needed functionalities to the Booking Service that will allow the user to view the bookings status.

9. **Notifications**
   We implement a event callback on the Booking Model that will trigger the Notification Service to send a "charging completed" notification to the user devices.

10. **PSP integration**
    We implement another event callback on the Booking Model to trigger the payment from the PSP Service to charge the user for the service.

5.2 Additional testing

After development and integration we will run some system-level tests. First test to run is on non-functional requirements, developers need to test accessibility features and base performance, doing so we can easily find the most important bottlenecks of the application. Another test that is useful to run is the so called "chaos test": various components of the system are disconnected on purpose to try to verify the behaviour of the system, the goal here is to check that the system fails safely, without losing or leaking data. The final test that will be run is the acceptance test: the system will be tested with real users in the production environment to verify that is really useful to them.

6 Effort spent

Task	Time spent
Introduction	2 h
Architectural Design	25 h
User interface	4 h
Requirements Traceability	2 h
Implementation, Integration and Test Plan	5 h
Revision	2 h

7 References

- **TDD (Test Driven Development)** - https://it.wikipedia.org/wiki/Test_driven_development
- **Three-Tier Architecture** - https://www.ibm.com/topics/three-tier-architecture
- **OCPI specs** - https://evroaming.org/app/uploads/2020/06/OCPI-2.2-d2.pdf
- **Chaos Testing** - https://www.ibm.com/garage/method/practices/manage/practice_chaotic_testing/
- **Acceptance Testing** - https://en.wikipedia.org/wiki/Acceptance_testing#User_acceptance_testing