POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

# ITD: Implementation and Testing Document

Author(s): **Lorenzo Ferretti**
**Lorenzo Manoni**
**Carlo Sgaravatti**

Version: 1.0

Date: 05/02/2023

URL: https://github.com/bighands2304/ManoniSgaravattiFerretti

Academic Year: 2022-2023

# Contents

# 1 | Introduction

## 1.1. Purpose

The purpose of this Implementation and Testing Document is to provide a comprehensive outline of the steps and processes involved in the successful execution of the software project in question. Its aim is to ensure that all team members and stakeholders have a clear understanding of the project's development, and are working towards a common goal. This document serves as a reference for all individuals involved in the implementation and testing phase.

## 1.2. Scope

The scope of this document pertains to the implementation and testing phase of the software project, including all activities related to the development of test cases, the execution of tests, and the resolution of any issues that may arise during the testing phase. The document outlines the responsibilities of each team member involved in the implementation and testing phase. It is important to note that the scope of this document is limited to the implementation, testing, and deploying phase and does not encompass other aspects of the project such as design or requirements.

## 1.3. Glossary

### 1.3.1. Acronyms

- **CPMS**: Charging Point Management System
- **eMSP**: e-Mobility Service Provider
- **CPO**: Charging Point Operator
- **DSO**: Distribution System Operator
- **OCPI**: Open Charge Point Interface

- **OCPP**: Open Charge Point Protocol
- **OSCP**: Open Smart Charging Protocol
- **OpenADR**: Open Automated Demand Response
- **DD**: Design Document
- **DTO**: Data Transfer Object
- **JWT**: Json Web Token

### 1.3.2. Abbreviations

- **CMD**: Command Prompt

## 1.4. Revision History

- February 5, 2023: version 1.0 (first release)

## 1.5. Reference Documents

- Project assignment ad specification document "Assignment RDD AY 2022-2023_v3.pdf"
- Requirements Analysis and Specification Document: "RASD1.1.pdf"
- Design Document: "DD1.pdf"
- OSCP 2.0 Specification.pdf
- SMUD OpenADR Implementation Design Guide v1_0.pdf
- ocpp-1.6.pdf
- OCPI-2.2.1.pdf

# 2 | Development

## 2.1. Implemented functions

### 2.1.1. eMSP

The following list describes all the functions implemented in this version of the product:

1. Know about the charging stations nearby, their cost, and any special offers they have: shows on the map the charging points with their information such as the status of each socket, and the tariffs related to each type;
2. Book a charge in a specific charging station for a certain timeframe: a user can make a reservation of a socket in a charging point from now keeping it valid for the next 20 minutes;
3. Start the charging process at a certain station: Once the station is reached by the user can start a charging session from the previous reservation;
4. Notify the user when the charging process is finished: when the charging process ends, the server will send a push notification to its device.

#### Implemented communication protocols

- **OCPI** communication with the CPMS: a light version of the OCPI protocol with only the needed information was implemented. The modules of the OCPI protocol that were implemented are:
  - The *Locations* module, providing to the CPMS the possibility to send the position and status of the charging points and their relative sockets.
  - The *Sessions* module, is used by CPMS to fetch the current charging sessions' status.
  - The *Tariffs* module, which provides to the CPMS the possibility of sending the tariffs of the charging points.

- The *Commands* module, which is used by the CPMS to send confirmation commands to the eMSP. The commands confirm the making of a reservation, the cancellation of a reservation, the start of a charging session, or the end of a charging session.

### 2.1.2. CPMS

The following list of functionalities has been implemented for the CPMS:

1. Management of the external status of a charging point: the implemented version of the CPMS offers the possibility to the CPO to insert new charging points in the system or remove them and to manage the status of their sockets (in this sense, the CPO can manually make some sockets not available). Regarding the Tariffs, this version of the product includes the possibility to insert, modify or delete tariffs both in manual mode and in automatic mode. In addition, this version of the CPMS implements also the updates that are sent to the eMSP when there are changes in the external status.
2. Charging sessions management and monitoring: the interaction with the charging point in order to start, end and monitor the charging process was implemented. Also, the updates that the CPMS sends to the eMSP when the session is finished were implemented.
3. Management of the internal status of the charging point: this version of the CPMS offers to the CPOs the possibility to monitor the status of the batteries and the charging profiles of the sockets of their charging points. According to the DD, the optimal charging profiles are computed automatically by the CPMS. This functionality was partially implemented, in particular, this version of the CPMS calculates the optimal charging profile with only one scheduling period, according to the current energy sources status.
4. Acquisition of the current price of energy from the DSOs, that is sent from the DSOs using the OpenAdr protocol.
5. Selection of the energy provider (DSO): for this functionality, the implemented CPMS implements both the manual mode and the automatic mode.
6. Management of the energy mix of the charging point: as for the DSO selection, in this version of the CPMS both the manual and the automatic modes are available.

### Implemented communication protocols

According to the DD document, the following communication protocols were considered

- **OCPI** communication with the eMSP: a light version of the OCPI protocol with only the needed information was implemented. The modules of the OCPI protocol that were implemented are:
  - The *Locations* module, providing to the eMSP the possibility to know the position and status of the charging points and their relative sockets.
  - The *Sessions* module, used by eMSP to retrieve the current charging sessions' status.
  - The *Tariffs* module, which provides to the eMSP the possibility of retrieving the tariffs of the charging points managed by this CPMS.
  - The *Commands* module, which is used by the eMSP to send commands to the charging point, passing through the CPMS. This in order to make reservations, cancel reservations, start charging sessions or end charging sessions.
- **OCPP** communication with the charging points: the version of the OCPP protocol that was implemented contains most of the messages, with a subset of the attributes (only those that were needed to implement the previous functionalities).
- A partial version of **OSCP** and **OpenADR** communication with the DSOs:
  - for the OSCP protocol, only the messages that permit the DSOs to publish the current energy capacity that they are able to provide were implemented
  - for the OpenADR protocol, only the messages through which the DSOs can notify the current energy cost were implemented

  Only these aspects of the protocols were implemented since the other parts were not considered important for providing the previous functionalities. For example, the OSCP protocol contains a module for sending metering information to the DSOs that was not implemented.

## 2.2. Adopted frameworks

### 2.2.1. Programming language

## Java EE

For developing our applications we choose to adopt as programming language Java, which offers a wealth of resources and tools to help you build robust and effective applications. Pros:

- **Widely adopted:** Java EE is widely adopted and has a large developer community, making it easier to find resources and help when developing applications.
- **Portability:** Java EE applications can run on any platform that supports Java, making it easier to deploy and maintain applications in different environments.
- **Scalability:** Java EE provides built-in support for scaling applications, making it easier to handle increased traffic and load.
- **Security:** Java EE provides a number of security features, including authentication, authorization, and encryption, to help ensure the security of applications.
- **Built-in components:** Java EE provides a number of built-in components that can be used to build applications quickly and efficiently.

Cons:

- **Complexity:** Java EE can be complex, especially for simple applications that don't require all its features.
- **Resource-intensive:** Java EE applications can require significant resources, such as memory and processing power, which can be a concern for some organizations.
- **Steep learning curve:** The learning curve for Java EE can be steep for new developers, as it involves understanding a number of different technologies and concepts.
- **Slower development:** Java EE can be slower to develop applications compared to other frameworks(we used Spring to avoid this), as it involves more configuration and setup.
- **Vendor lock-in:** Java EE is largely controlled by a single vendor, Oracle, which can make it difficult to switch to another platform or framework in the future.

## React

React is a JavaScript library for building user interfaces. It allows developers to build reusable UI components, manage the state of their applications, and render changes efficiently. React follows a component-based architecture, which makes it easy to write and

test code, and scale applications as they grow. Pros:

- **Virtual DOM:** React uses a virtual DOM, which allows for efficient updates and rendering of components, improving app performance.
- **Reusability:** React components can be easily reused, making it easier to develop complex UI.
- **Components-based architecture:** React has a component-based architecture, making it easier to build, test, and maintain large applications.
- **Strong community:** React has a large and active community, providing a wealth of resources, including libraries, tutorials, and support.
- **Server-side rendering:** React can be used for server-side rendering, improving the performance and SEO of web applications.

Cons:

- **JavaScript fatigue:** React requires a good understanding of JavaScript and its ecosystem.
- **Steep learning curve:** React has a steep learning curve and may need time to become proficient in using it effectively.
- **Dependence on third-party libraries:** React relies on third-party libraries for many features, and there is a risk that these libraries may not be maintained or updated, leading to potential compatibility issues.
- **Unstable API:** React has had a history of making breaking changes to its API, causing compatibility issues and requiring additional effort to maintain older applications.
- **Poor documentation:** React documentation can be inadequate in some areas, making it difficult for developers to find the information they need to complete their projects.

### React Native

React Native is a framework for building mobile apps using React. It allows developers to write code once and run it on both iOS and Android platforms, without the need for separate codebases. Pros:

- **Cross-platform development:** React Native allows developers to write code once and run it on both iOS and Android platforms, reducing development time and cost.

- **Performance:** React Native uses native components, which are compiled into native code, providing a smooth and fast user experience that is comparable to a traditional native app.
- **Reusability:** With React Native, developers can reuse code across platforms, and there are also many pre-built components available in the React Native community that can be used to speed up development.
- **Community support:** React Native has a large and active community, providing a wealth of resources, including libraries, tutorials, and support.
- **Hot Reloading:** React Native allows developers to see the changes they make in real-time, reducing the development cycle and increasing productivity.

Cons:

- **Limited native modules:** Some native features may not be available in React Native, and custom native modules need to be built, which can be time-consuming and complex.
- **Fragmentation:** React Native has multiple versions, and updates to the framework can cause compatibility issues, requiring additional effort to maintain compatibility.
- **Debugging:** Debugging React Native apps can be more challenging compared to native apps and may require additional time and effort.
- **Learning curve:** React Native requires a good understanding of both React and native platform development, and developers may need time to become proficient in using it effectively.
- **Dependence on third-party libraries:** React Native relies on third-party libraries for some features, and there is a risk that these libraries may not be maintained or updated, leading to potential compatibility issues.

### 2.2.2. Frameworks

The following list describes all the frameworks used in this version of the product in order to satisfy the implementation of the above-described functions:

- **Spring:** an open-source application framework that provides infrastructure support for developing Java applications. Spring helps developers create high-performing applications using plain old Java objects (POJOs). This framework is composed by the following components:

  - *Dispatcher Servlet:* is the front-controller in the Spring MVC application that handles all incoming requests and delegates them to the appropriate controllers;
  - *Controller:* is a component in Spring MVC that handles incoming requests and maps them to specific handler methods based on the request URI and HTTP method.
  - *Model:* represents the data that is to be displayed on the view, it acts as a data transfer object between the Controller and the View;
  - *Service:* is a component in the application that performs business logic. Services are usually used by Controllers to perform the necessary operations.
  - *Repository:* is a component in the application that is responsible for accessing and manipulating data from a database or any other data source.
  - *Entity:* is a model that represents a table in a database or a business object in the application. It typically maps to a single row in a database table and is used to persist data to and from a database.

- **Spring Data JPA:** is a module in the Spring framework that provides a simplified approach for data access using Java Persistence API (JPA). It enables the development of JPA-based data access layers by eliminating the need for writing boilerplate code and reducing the amount of coding required. Spring Data JPA provides a number of convenient and consistent abstractions for data access operations, including CRUD operations, pagination and sorting, dynamically defining queries, and integrating with transactions. The module integrates with Hibernate, allowing developers to use a common interface to access databases and persist data, while still taking advantage of the underlying JPA provider's features.

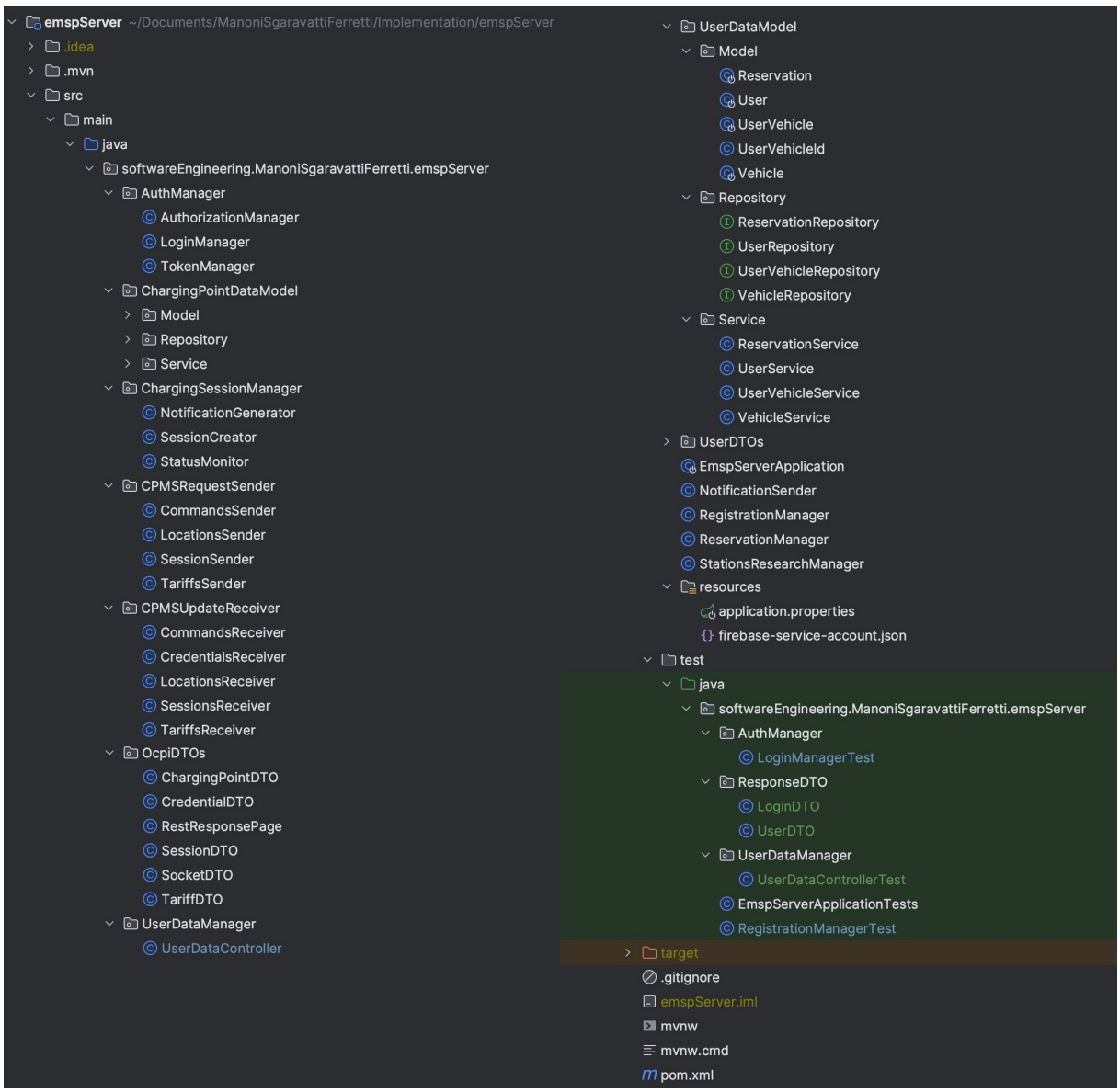
- **Spring Security:** is a powerful and highly customizable authentication and access-control framework for Java applications. It is a module of the Spring framework that provides a comprehensive security solution for both web-based and method-level security. Spring Security allows developers to secure their applications with minimal configuration, while still providing the flexibility to customize security policies as needed.
  Spring Security provides authentication and authorization services through a variety of authentication mechanisms such as LDAP, database authentication, and OAuth. It also provides a wide range of access-control options, including role-based access control, URL-based security, method-level security, and expression-based security.
  It provides a robust set of APIs for customizing and extending the security features,

making it a popular choice for securing Java applications of all sizes and complexity.

- **Spring Data MongoDB**: is a library that provides a high-level abstraction for working with MongoDB databases within the Spring framework. It helps to simplify the development of MongoDB-based applications by reducing the amount of boilerplate code needed and providing a consistent, familiar interface for accessing and manipulating data.
- **Spring Boot WebSocket**: is a module in the Spring Boot framework that allows developers to easily integrate WebSocket functionality into their applications. It provides a simple and convenient way to handle incoming WebSocket connections and process messages, allowing developers to build real-time, interactive web applications.
- **AstraDB**: a cloud-native, globally distributed database service provided by Astra, a subsidiary of DataStax. It is built on top of Apache Cassandra, one of the most popular NoSQL databases, and is designed to make it easy for developers to quickly create and manage Cassandra databases in the cloud. Astra DB provides a fully managed, highly available, and scalable database service that eliminates the need for manual database administration and maintenance tasks.
- **Junit**: a widely-used, open-source framework for writing and running tests in Java applications. It is designed to simplify the process of writing, testing, and maintaining software by providing a consistent, repeatable testing environment. JUnit provides a set of annotations and assertions that make it easy to define test cases, test methods, and expected results. Tests written with JUnit can be run automatically, making it easy to identify and fix problems early in the development process. JUnit tests can be run in isolation, which makes it easy to write tests that focus on a single class or module.
- **Firebase:** is a Backend-as-a-Service (BaaS) platform that provides developers with a suite of tools and services to help them develop high-quality apps, in our specific case it is used for sending a push notification to the device of the user;

# 3 | Source code

## 3.1. eMSP structure



- **AuthManager:** Is the package containing the classes responsible for the authentication and authorization;
  - *AuthorizationManager:* This class implements SpingSecurity;
  - *LoginManager:* This class implements the login API and the controls of the login;
  - *TokenManager:* This class implements the JWT management;
- **ChargingPointDataModel:** This package contains the model, the repositories, and the services of the MongoDB DataBase;
- **ChargingSessionManager:** This package contains the function for managing the charging sessions;
  - *NotificationGenerator:* This class sets the structure of the notifications to send;
  - *SessionCreator:* This class implements the start session API that creates a session from a reservation;
  - *StatusMonitor:* This class implements the APIs for the management and monitoring of the sessions;
- **CPMSRequestSender:** This package encloses all the OCPI requests that are sent to the CPMS;
- **CPMSUpdateReceiver:** This package contains all APIs for the OCPI communication with the CPMSs;
- **OcpiDTOs:** This package contains all the DTOs of the OCPI requests and responses;
- **UserDataManager**/*UserDataController:* This class implements all the APIs that affect the users in the database;
- **UserDataModel:** This package contains the model, the repositories, and the services of the MySQL DataBase;
- **UserDTOs:** This package contains all the DTOs of the users' requests and responses;
- **NotificationSender:** This class implements the Firebase notification sender;
- **RegistrationManager:** This class implements the registration API;

- **ReservationManager:** This class implements all the APIs related to the reservation;
- **StationResearchManager:** This class implements all the APIs that fetch the CPs and related information;
- **resources**/*application.properties:* This file contains all the information related to the connection and management of the databases through spring.

## 3.2. CPMS structure

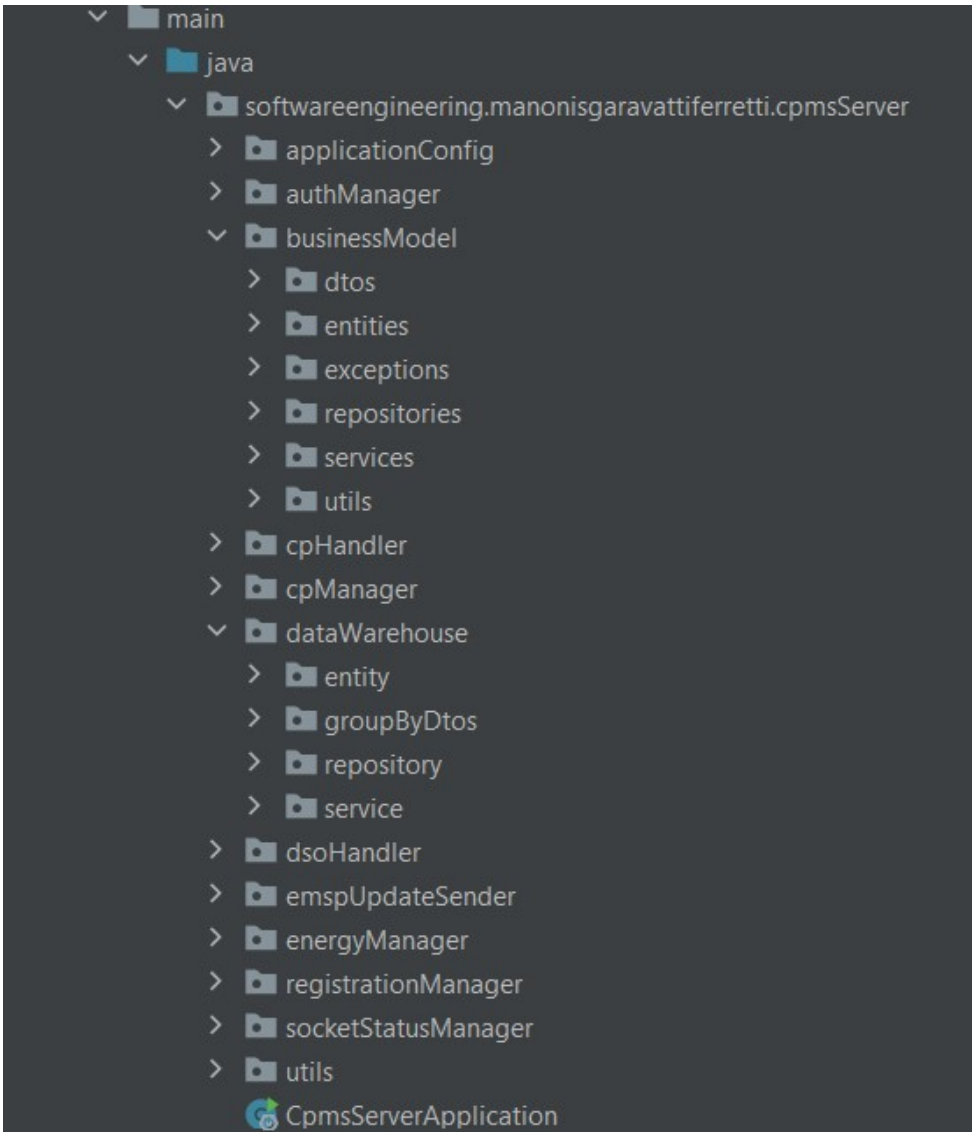



Figure 3.1: The packages of the CPMS server

The structure of the CPMS server directly follows the component diagrams present in the DD. In particular, there are the following packages:

- The **applicationConfig** package contains all the configuration classes for Spring
- The **authManager** package contains the following classes:
  - *AuthorizationManager*: implements the security of the application
  - *LoginManager*: handles the login of the CPO
  - *TokenManager*: handles the JWT management and generation
  - *EMSPCredentialsManager*: handles the insertion of an eMSP from the CPO, that will be added to the eMSPs that communicate with the CPMS
  - *UserDetailsServiceImpl*: retrieves from the persistent storage the details of the registered CPOs and eMSPs
- The **businessModel** package contains all the model classes that represent the data in the MongoDB Business Database of the CPMS. The dtos subpackage contains all the DTOs that are exchanged with the user interface and with the eMSPs. The entities are a representation of what the DB contain. The repositories contains the methods to retrieve data from the DB and the services uses the repositories in order to expose the model to the other components of the server.
- The **cpHandler** package contains all the classes that handle the communication with the charging point using the WebSocket and all the DTOs that are exchanged with the charging point.
- The **cpManage** package contains the classes to handle requests from the CPOs and to manage the tariffs of the charging points. The most important classes in this package are:
  - *ChargingPointsManager*: receives all the requests from the CPO and passes them to the correct component
  - *PriceManager*: manages the tariffs of the charging points
  - *PriceOptimizer*: optimizes the tariffs of the charging point when it is set on automatic mode
- The **dataWarehouse** package contains a representation of the Cassandra Data Warehouse. In particular, the entity subpackage contains the classes that represent Cassandra table, the groupByDto subpackage contains classes that are the result of

some aggregation queries. The repository and service sub packages are the equivalent of the ones of the businessModel package.

- The **dsoHandler** package handles the acquisition of the DSO information through the OSCP and OpenADR protocols.
- The **emspUpdateSender** contains all the methods to send OCPI requests to the eMSP.
- The **energyManager** package handles the energy management of the charging points. The most important classes in this package are:
  - *EnergyMixManager*: handles the energy mix changes in the charging points
  - *EnergyMixOptimizer*: optimizes the energy mix of the charging point when set on automatic mode
  - *DSOManager*: handles the changes in the DSO selection
  - *DSOSelectionOptimizer*: optimizes the selection of the DSO when is set in automatic mode
- The **registrationManager** contains the *RegistrationManager* class that handles the registration of the CPO
- The **socketStatusManager** contains the classes that handles the status of the sockets and that exposes the CPMS OCPI interface to the eMSP. The most important classes are:
  - *SocketStatusController*: handles the OCPI requests from the eMSP in order to return to them the status of the sockets of a charging point and their tariffs
  - *ReservationController*: handles the commands from the eMSP in order to make a new reservation or to cancel an existing reservation
  - *ChargingSessionController*: handles the commands from the eMSP in order to start a charging session from an existing reservation and to end a charging session before it is finished

# 4 | Testing

## 4.1. eMSP

For the eMSP server, we wrote some test cases using JUnit5. Figure **??** shows the test classes for the eMSP Server in the 'test' package. The part tested with Junit are the one that affects MySQL DataBase:

- RegistrationManagerTest: Test the API that handles the registration of the users;
- LoginManagerTest: Test the API that handles the login of the users;
- UserDataControllerTest: Test the API that handles the addition, modification, and deletion of the vehicles;

## 4.2. CPMS

For the CPMS server, we wrote some test cases using JUnit5 and Mockito. Figure 4.1 shows the test classes for the CPMS Server. In particular, two types of testing classes are shown:

1. **Unit testing**: using Mockito we have built some test cases for testing for the most important and complex modules of the system. These are:
    - The automatic mode choices, whose tests are in the following classes: DSOSelectionOptimizerTest, ChargingProfileOptimizer, EnergyMixOptimizer, PriceOptimizer. These test were important to verify that the optimizers were making the correct choices.
    - The tariff application when a charging session is finished (PriceManagerTest class). These tests were done to verify that the PriceManager was calculating the cost of the charging session in the proper way
    - The manual selection of DSO provider (in the DSOManagerTest class). This test was done in order to make sure that all the necessary checks are done when

the CPO selects the DSO from which the energy is taken.

2. **Integration testing**: We have done two types of integration tests

   - The integration between the model classes and the DBMS. We have written some test cases in order to check the execution of some custom queries in the database. In particular:

     – ChargingPointRepositoryTest, ReservationRepositoryTest, and EmspRepositoryTest are testing some custom MongoDB update queries.

     – EnergyConsumptionServiceTest, that is testing the execution of a custom aggregation query.

     Not all the queries in the repositories were executed using the query methods of the Spring Data MongoDB and Spring Data Cassandra, so these tests were checking that these queries were correctly written.

   - The integration between the controllers, which are the outer part of the CPMS, and the other classes. In particular:

     – **RegistrationManagerTest**: this class tests the CPO registration process in the system.

     – **LoginManagerTest**: this class tests the login of a registered CPO in the system.

     – **CharginPointsManagerTest**: this class tests all the APIs that the CPMS exposes to the client application.

Further details on the integration test cases are provided in the next subsection. In total, considering both unit and integration tests, 45 test cases were written. Figure 4.2 shows that all the tests were successful.

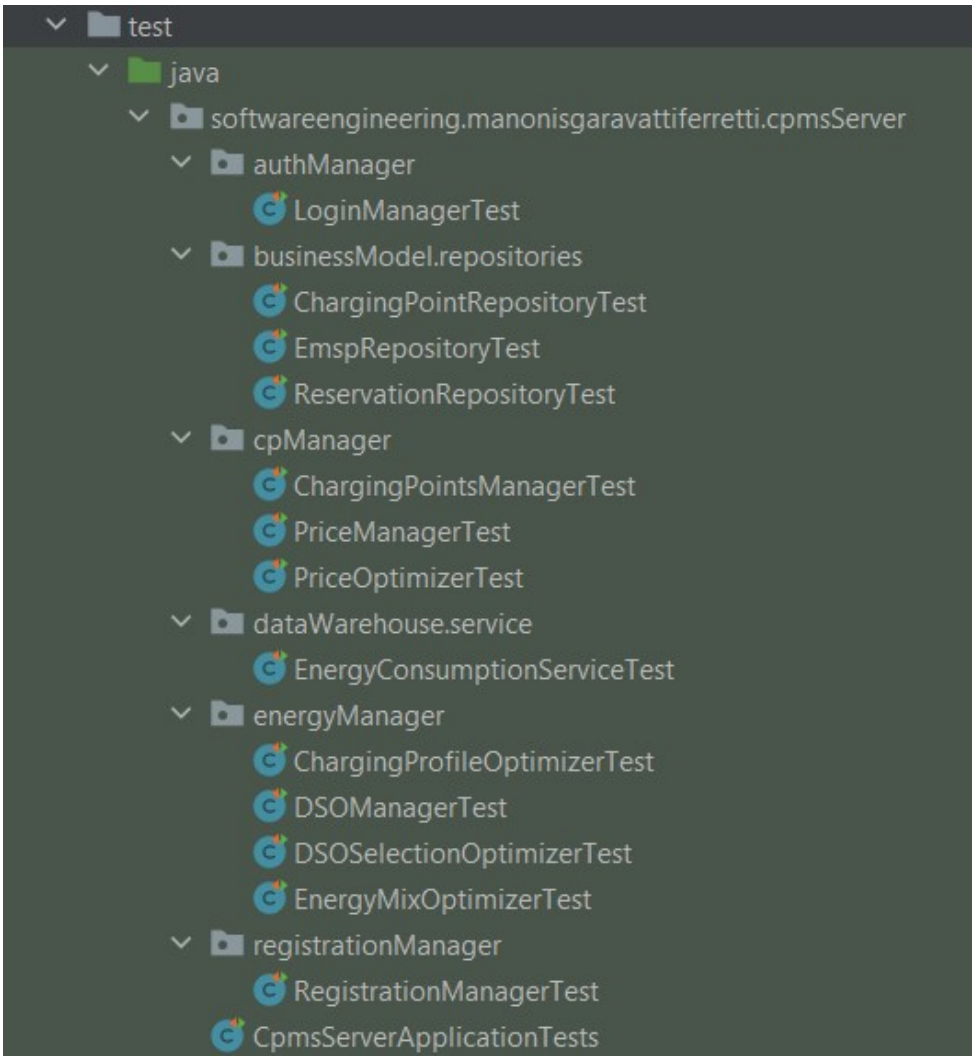



Figure 4.1: The testing classes of the CPMS

```
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 45, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  01:20 min
[INFO] Finished at: 2023-02-05T22:10:50+01:00
[INFO] ------------------------------------------------------------------------
```

Figure 4.2: The testing report of the CPMS

## Integration testing details

The following tests were made for integration tests of ReservationManager:

1. Registration with all valid data (CPO code, password and IBAN).
2. Attempted registration with invalid data (missing password). This test should not allow registering with missing data

For the LoginManager, the following integration tests were made:

1. Login with valid cpo code and password. This test aims to both verify the authorization and the generation of the token that the CPO will use for next requests.
2. Attempted login with a wrong password. This test should not allow a CPO to login without a password

Finally, the following tests were made for the ChargingPointsManager:

1. Selecting a charging point that does not exist. This test verifies that an error is sent to the user interface when the CPO tries to get information about a charging point that does not exist.
2. Retrieving the information about the sockets of an existing charging point. This test ensures that the sockets of an existing charging point are returned when requested.
3. Adding a new charging point with all correct information
4. Deleting an existing charging point from the system.
5. Changing the energy mix of a charging point. This test verify the inclusion of the battery in the energy sources (specifying the minimum and maximum levels of storage and the percentage of energy to take from the battery) when all information provided is correct. This test should also verify that the manual mode is set (if not already done) for the energy mix management as a consequence of the manual choice of the CPO.
6. Attempting to change the energy mix with wrong parameters (minimum level higher than the maximum).
7. Changing the availability of a socket.
8. Changing the availability of a battery.
9. Retrieving all the tariffs of a certain charging point.
10. Retrieving a specific tariff of a charging point.

11. Deleting a tariff from a charging point.
12. Adding a new tariff to the charging point. This test is also verifying that the price optimizer is correctly set to manual mode, if not already done.
13. Modifying an existing tariff of a charging point. As for the previous test, also this one is checking that the price optimizer is correctly set to manual mode, if not already done.
14. Attempting to modify a not existing tariff.
15. Retrieving the offers of the DSOs for a certain charging point.
16. Changing the DSO provider of a charging point. This test also checks that the DSO selection optimizer is set to manual mode, if not already done.
17. Setting the price optimizer in manual mode
18. Setting the energy mix optimizer in manual mode
19. Setting the DSO selection optimizer in manual mode

## 4.3. Ocpp and Oscp Driver

To test the WebSocket OCPP communication with the charging points a small Charging Point Driver was written in python in order to test the OCPP messages and to be able to send commands to it. In particular, this driver always responds in a positive way and, for the charging session builds a series of messages sent in sequences (with pauses between each) that simulates a charging session.

The same driver send also OSCP and OpenADR requests to the CPMS server in order to simulate the updates from the DSO for both the capacity they provide and the price of the energy of their offers.

## 4.4. System Testing

A part of the testing process was also made by simulating client requests using Postman, an easy-to-use API development and testing tool that allows developers to create, test, and manage APIs by sending HTTP requests and analyzing the responses. All the HTTP methods exposed by both the eMSP and the CPMS were tested using Postman.

The communication between the CPMS and the eMSP was the most important aspect that these tests were trying to evaluate. In particular, the following tests, that simulate

the interaction with the user, were done:

- Making a new reservation, through the following steps:

  1. User login into the eMSP;
  2. Get the Charging Points in a certain area;
  3. Sending a "make reservation" request to the eMSP server.

  By analyzing the responses and the logs in the server we were able to check that the reservation was passed to the CPMS server and that was successful.

- Deleting a reservation, through the following steps:

  1. User login into the eMSP;
  2. Get the Charging Session in a certain area;
  3. Sending a "make reservation" request to the eMSP server;
  4. Deleting the reservation.

  By analyzing both the responses and the logs we were able to detect that the reservation was correctly eliminated.

- Start a charging session, through the following steps:

  1. User login into the eMSP;
  2. Get the Reservation;
  3. Sending a "start charging session" request to the eMSP server;
  4. Starting a charging session.

  By analyzing both the responses and the logs we were able to detect that the charging session was correctly started.

- End a charging session, through the following steps:

  1. User login into the eMSP;
  2. Get the Reservation;
  3. Sending an "end charging session" request to the eMSP server;
  4. Ending a charging session.

  By analyzing both the responses and the logs we were able to detect that the charging session was correctly ended.

# 5 | Installation

## 5.1. eMSP Server and CPMS Server

Both eMSP and CPMS servers are already hosted on Railway (an infrastructure platform where you can provision infrastructure, develop with that infrastructure locally, and then deploy to the cloud). There is no need of installing any environment or DataBase locally. The applications are hosted in the following link:
https://emspserver.up.railway.app
https://cpmsserver.up.railway.app

## 5.2. eMSP Client

This section outlines the steps to run the eMSP Client:

1. Open the cmd/terminal;
2. Ensure that you have installed the latest version of *Yarn*;
3. Download the directory from the git (/Implementation/empsClient/MealsToGoProject);
4. Open the project directory and run the *yarn* command to initialize the project;
5. Once all the dependencies have been installed, you can run the project using the *yarn start* command;
6. - **Computer:** This is a mobile application, you'll need a simulator for the mobile device, you could use either Xcode (iOS) or install Android SDK (Android).
   - **iOS:** Open the camera and scan the QR code shown after the 'Yarn start'.
   - **Android:** Download the 'Expo Go' application from the Play Store and scan the QR code shown after the 'Yarn start'.

## 5.3. CPMS Client

This section outlines the steps to run the CPMS Client:

1. Open the cmd/terminal;
2. Ensure that you have installed the latest version of **Yarn**;
3. Download the directory from the git (/Implementation/cpmsClient);
4. Open the project directory and run the '*yarn*' command to initialize the project;
5. Once all the dependencies have been installed, you can run the project using the '*yarn start*' command;
6. The application will automatically run on '*http://localhost:3000*'.

# 6 Effort Spent

## 6.1. Lorenzo Ferretti

| Task | Hours Spent | Software |
|---|---|---|
| Implementation | 120 | eMSP Client, CPMS Client |

## 6.2. Lorenzo Manoni

| Task | Hours Spent | Software |
|---|---|---|
| Implementation | 120 | eMSP Server |

## 6.3. Carlo Sgaravatti

| Task | Hours Spent | Software |
|---|---|---|
| Implementation | 120 | CPMS Server |