

ITD
Software Engineering Project
A.Y. 2022-2023

Marco Ronzani, Alessandro Sassi

December 2022 - February 2023

Contents

1	Links to the software	3
2	Introduction	3
2.1	Purpose	3
2.2	Scope	3
2.2.1	Acronyms	3
2.3	Reference Documents	4
3	Implemented Functionalities and Requirements	5
3.1	Functionalities	5
3.2	Requirements	5
4	Adopted development frameworks	9
4.1	Adopted programming languages	9
4.1.1	TypeScript	9
4.1.2	SQL	9
4.1.3	HTML and CSS	9
4.2	Adopted middleware	9
4.2.1	Express.js as a Request Router for the backend	9
4.2.2	Database connection with MySQL2	9
4.2.3	Connection to the CS via WebSocket	10
4.2.4	HTTP requests via the Axios library	10

4.3	Utilized APIs (not present in the DD)	10
4.4	Platforms and Libraries	10
4.4.1	Node.js	10
4.4.2	Authentication with JWT and BCrypt	10
4.4.3	Testing with Mocha and Sinon-Chai	11
4.4.4	Frontend with Vue	11
5	Source Code Structure	11
5.1	Code Structure	11
5.2	Components in the Source Code	12
6	Testing	13
6.1	Integration Testing	13
6.2	Automated Testing	17
6.2.1	Backend Tests	17
6.2.2	Frontend Tests	18
7	Installation instructions	18
7.1	Prerequisites	18
7.2	Preparing the Project	19
7.3	Running the project	20
7.4	Interacting with the running project	20
8	Effort Spent	22
8.1	Ronzani Marco - mat: 224578	22
8.2	Sassi Alessandro - mat: 220837	22
9	References	23

1 Links to the software

The following links point to the resources required to run the project. For installation instruction, see Section 7 of this document.

- **Full Repository:** The GitHub repository can be found at the following link: [GitHub Repository Link](#). In the folder *DeliveryFolder* you can find all the project deliverables (RASD, DD, this document and a compressed archive of the src folder)
- **Source Code:** GitHub Repository src Folder. This folder contains all the source files that make up the project.
- **Compressed Source Code Only Archive:** Can be found in the DeliveryFolder, and the direct link is the following: [Compressed src Archive](#).

2 Introduction

2.1 Purpose

Electric vehicles are starting to grow in number, and their takeover of combustion engines is bound to happen. Consequently, to support such a thriving trend, adequate and easy access to charging stations is of utmost importance. In this landscape the goal of eMall is to allow owners of electric vehicles to easily know where charging stations are and carefully plan their charging process according to their schedules at any such station.

This document will follow up on the RASD and DD documents presenting the result of the development of a demonstrative viable prototype of the eMall software.

2.2 Scope

This IT document starts from what was layed down in the DD document and the requirements discussed in the RASD, presenting the result of the development of a prototype of eMall's software, complete with also the software of a CPO (a CPMS) which eMall relies upon to interact with CSs. Such software consists of a WebApp supported by a set of APIs which is itself reliant on APIs exposed by the software of CPMSs.

2.2.1 Acronyms

Acronym	Full Name
eMall	Electric Mobility for All
eMSP	Electric Mobility Service Provider
CPO	Charging Point Operator
CPMS	Charge Point Management System
CS	Charging Station
DSO	Distribution System Operator
STB	System-To-Be
OS	Operating system
DB	Database

2.3 Reference Documents

1. The provided document describing the project: *Assignment IT AY 2022-2023*.
2. The provided document describing the project: *Assignment RDD AY 2022-2023-v3*.
3. The Software Engineering 2 course held by Prof.s Camilli Matteo, Di Nitto Elisabetta and Rossi Matteo Giovanni.
4. *ISO/IEC/IEEE 29148:2011(E)* standard for Requirement Engineering.
5. Project of last year provided as an assignment.

3 Implemented Functionalities and Requirements

The development of the prototype focused on the set of features that were demanded in the first reference document for a group of two people.

Any requirement for which a justification is not provided regarding its exclusion from the prototype can be assumed to have been disregarded because outside of the following features.

An eMSP shall offer to end users the possibility to:

1. know about the charging stations nearby, their cost, any special offer they have.
2. book a charge in a specific charging station for a certain time frame.
3. start the charging process at a certain station.
4. notify the user when the charging process is finished.

A CPMS shall offer the following main functions:

1. know the location and “external” status of a charging station (number of charging sockets available, their type such as slow/fast/rapid, their cost, and, if all sockets of a certain type are occupied, the estimated amount of time until the first socket of that type is freed).
2. start charging a vehicle according to the amount of power supplied by the socket, and monitor the charging process to infer when the battery is full.

3.1 Functionalities

Refer to the RASD document for detailed descriptions under section 2.2 .

Id	Functionality	Status
1	User registration	Implemented
2	Search for CSs	Implemented
3	Book a recharge at a CS	Implemented
4	Perform a recharge at a CS	Implemented (mocked CS)
5	Acquire price and energy mix information from DSOs	Not Implemented
6	Choose prices, energy sources and battery usage policies for a CS	Not Implemented
7	Allow the CPMS to operate automatically	Not Implemented
8	Allow the CPO to change the “Automatic Mode” policy used by the CPMS	Not Implemented

3.2 Requirements

Refer to the DD document for detailed descriptions under section 4 .

Id	Requirement	Status
R1	The eMSP shall allow a registered user to login	Implemented
R2	The eMSP shall allow an unregistered user to register on the platform	Implemented
R3	The eMSP should report to the user when they sent invalid input data through a form	Implemented
R4	The eMSP should report to the user when they attempted to perform an unauthorized action	Implemented
R5	The eMSP should report to the user when the platform encountered an error while processing the action	Implemented
R6	The eMSP should report to the user any successful action performed	Implemented
R7	The eMSP shall allow a logged-in user to view charging stations nearby	Implemented
R8	The eMSP shall allow a logged-in user to know the cost of a recharge at a CS	Implemented
R9	The eMSP shall allow a logged-in user to know about ongoing special offers present at a CS	Implemented
R10	The eMSP shall allow a logged-in user to book a recharge at a free socket of a CS for a given time slot	Implemented
R11	The eMSP shall allow a logged-in user to cancel any of its booked recharges before their scheduled time	Implemented
R12	The eMSP shall not allow a user to book a recharge at a socket of a CS which has already been booked for the chosen time slot	Implemented
R13	The eMSP shall not allow a user to book multiple recharges with overlapping time slots	Implemented
R14	The eMSP shall allow a logged-in user to see their bookings	Implemented
R15	The eMSP shall allow a logged-in user to start the charging process for one of their booked recharges	Implemented
R16	The eMSP shall not allow a user without a booking to start the charging process	Implemented

Id	Requirement	Status
R17	The eMSP shall not allow a user to start the charging process as long as the socket is vacant	Implemented
R18	The eMSP should notify a logged-in user when their ongoing charging processes is complete	Implemented
R19	The eMSP should notify a logged-in user when their ongoing charging process is terminated due to the expiration of its booked time slot	Implemented
R20	The eMSP shall allow a logged-in user to terminate their charging process earlier	Implemented
R21	The eMSP should charge a user for everyone of his charging processes as soon as they finish	Implemented (mockup)
R22	The eMSP shall charge a user for a fee if they do not start a recharge during their booking's time slot	Not Implemented
R23	The eMSP shall be able to offer its functionalities to multiple users at once	eMSP: Implemented
R24	The CPMS should be able to acquire the location from CSs	Implemented (mocked via a fake CS client, with the data present anyway in teh CPMS's DB)
R25	The CPMS should be able to acquire the internal status from CSs	Not Implemented
R26	The CPMS should be able to acquire the external status from CSs	Implemented
R27	The CPMS should be able to start charging a vehicle connected to a CS socket	Implemented (mocked via a fake CS client)
R28	The CPMS should be able to monitor the charging process of a vehicle	Implemented (mocked via a fake CS client)
R29	The CPMS should be able to acquire from DSOs their current energy prices	Not Implemented
R30	The CPMS should be able to acquire from DSOs their energy mix	Not Implemented
R31	The CPMS shall allow its CPO to authenticate with it	Not Implemented
R32	The CPMS shall allow its CPO to assign a DSO to a CS as its energy provider when operating in "Manual Mode"	Implemented

Id	Requirement	Status
R33	The CPMS should be able to automatically assign a DSO to a CS as its energy provider when operating in “Automatic Mode”	Not Implemented
R34	The CPMS shall allow the CPO to decide the energy source management policies when operating in “Manual Mode”	Not Implemented
R35	The CPMS should be able to automatically choose the current energy source for a CS according to the given policy when operating in “Automatic Mode”	Not Implemented
R36	The CPMS shall allow its CPO to assign a nominal-price, a user-price and an offer reset date to a CS when operating in “Manual Mode”	Not Implemented
R37	The CPMS should be able to automatically assign a nominal-price, a user-price and an offer reset to a CS when operating in “Automatic Mode”	Not Implemented
R38	The CPMS shall allow the CPO to choose its policy for operating in “Automatic Mode”	Not Implemented
R39	The CPMS shall allow the CPO to choose whether it acts automatically or manually	Not Implemented
R40	The CPMS shall never take a decision automatically when it is set in “Manual Mode”	Not Implemented
R41	The CPMS should report to the user when they sent invalid input data	Implemented
R42	The CPMS should report to the user when they attempted to perform an unauthorized action	Implemented
R43	The CPMS should report to the user when the platform encountered an error while processing the action	Implemented
R44	The CPMS should be able to decide whether to start charging the internal batteries of a CS (if present)	Not Implemented

4 Adopted development frameworks

As expected of every modern Web Application supported by a backend API, its software architecture is divided in a Model and Routes for the backend and a framework for the frontend which allows for both a logic and presentation layer in order to offer more advanced functionalities than a plain HTTP page. In the prototype here discussed keep in mind that there is a further distinction between the CPMS's backend and the eMSP's one, when neither is mentioned explicitly, what follows refers to both.

4.1 Adopted programming languages

4.1.1 TypeScript

Both backends and the frontend are primarily written in **TypeScript**, a superset of Javascript that adds to the former static typing and optional classes and interfaces, resulting in better type safety and less bugs. TypeScript unfortunately offers a considerably smaller set of libraries compared to JavaScript, since it mandates that type definitions are included in the library itself or custom-written by the user. Nevertheless, the functionalities of the project were not hindered.

4.1.2 SQL

The interactions with the database that occur within the model are realized by preparing and executing SQL statements, making extensive use of MySQL's dialect of SQL, requiring no redundant storage of data on the database and a simple way to recover from it only what is necessary. MySQL was chosen in favor of other SQL dialects thanks to the prior knowledge that the team had, and NoSQL databases were quickly discarded since the project benefits from the use of relations between entities for many functionalities and does not need to store large binary data. This choice has proven to be successful in many ways, one of which was the extreme ease of creation of native active triggers, which NoSQL solutions like MongoDB are lacking.

4.1.3 HTML and CSS

Wanting to create a web application, the frontend had to be built using HTML and CSS, the two basic languages for frontend creation. However, most of the interface does not consist in hand-written "native" HTML code, but rather it uses custom tags defined by our frontend framework of choice to dynamically render the pages at runtime.

4.2 Adopted middleware

4.2.1 Express.js as a Request Router for the backend

Express is a free and open-source web framework for building APIs with a comprehensive set of features. Its approach to middleware and routing, which involves adding functions to endpoint declarations, helps improve code scalability. The use of functions as middleware also promotes modularity in the application by allowing endpoints to be added through corresponding functions for each route.

4.2.2 Database connection with MySQL2

MySQL2 is a typescript version of the popular MySQL package and consists in a set of modules for working with the MySQL database. This library was chosen for its excellent pooling system and the possibility of preparing statements before executing queries, where the query string is passed directly to the database and parameters are replaced there, preventing the risk of SQL injection attacks that can result from string concatenation on the server.

4.2.3 Connection to the CS via WebSocket

The WebSocket library for JavaScript (ws) enables real-time, two-way communication between a client and a server, with a simple and efficient API for creating and managing connections, supporting both server-side and client-side use.

4.2.4 HTTP requests via the Axios library

Axios is JavaScript middleware library for making HTTP requests in Node.js. It is promise-based and enables sending HTTP requests to access REST APIs and other web services. Axios abstracts away some of the complexities of making HTTP requests and provides a simple, straightforward API for handling requests and responses.

4.3 Utilized APIs (not present in the DD)

As shown in the design document, the project interacts with an external service for obtaining map data and to get the current position of the user. The service used is **OpenStreetMaps**, coupled with the **OpenLayers** library to interact with it and display maps. Such library and map provider were chosen for two main reasons:

- **Ease of use:** before choosing OpenLayers we tried to integrate multiple other libraries, including Apple Maps and Google Maps. However, all other solutions required complex setup with our frontend framework (Vue), and were intended to be used in a JavaScript-only environment, making them unfit for our purposes;
- **No Key required:** since the project had to be hosted on a GitHub repository and can potentially be downloaded by other individuals, we decided not to use other frameworks since they all required our frontend system to have a private token linked to one of the developers' account. As this posed some privacy and security concerns, we opted for OpenStreetMaps since it does not require any API token.

4.4 Platforms and Libraries

4.4.1 Node.js

Node is an open-source, cross-platform, back-end JavaScript (and by extension TypeScript) runtime environment which offers many stable and well known libraries (ex: Express). It allows the execution of JavaScript outside of a web browser and as act as a server, is well-suited for building responsive applications, such as eMall, due to its ability to handle multiple simultaneous connections with low latency. Its only drawback is its single-threaded model, resulting in poor performances when dealing with CPU-intensive tasks, but that is never the case with the present prototype.

4.4.2 Authentication with JWT and BCrypt

JWT (JSON Web Token) is a javascript implementation of the JSON Web Token standard, which provides a secure and URL-safe method of transferring user information between two parties.

BCrypt is the Javascript implementation of the bcrypt hashing function, and is a standard to hide passwords in a web environment. It provides a simple and secure way to hash passwords and compare them during authentication.

4.4.3 Testing with Mocha and Sinon-Chai

Mocha is a JavaScript testing framework that works with Node.js. It provides a simple, flexible, and fast environment for writing and executing test cases for applications and libraries.

Sinon-Chai is comprised of the two libraries Sinon and Chai, the first allows for testing via stubs and mocks of objects and functions, the second complements it with an easy way to write assertions.

4.4.4 Frontend with Vue

The frontend was built using the Vue framework, a performant and easy-to-use JavaScript/TypeScript framework for building web applications. Its main advantage is its compositional API, which allowed us to split the UI into modules, each one corresponding to a page, and each module into multiple reusable components, which are organized according to page-specific templates. It also proved to be very elegant and versatile for building interactive applications, since each component is dynamically bound to JavaScript objects and automatically updates whenever changes to model objects occur either due to user interactions or API calls.

5 Source Code Structure

5.1 Code Structure

As mentioned in the DD document, the structure of the code is made to reflect the Model View Controller (MVC) pattern. Briefly:

- **Model**, it contains the logic that retrieves from the database the data that needs to be shown to the user, as well as classes used to encapsulate the data retrieved. Ad-hoc versions of the model exist for the frontend and backend, when the former is present. More precisely there are 3 different models, one for the eMSP's backend, one for its frontend, and one for the CPMS's backend.
- **View**, its goal is to show the data of the model to the user, formatted in a human-friendly way. The view is realized with the aforementioned Vue framework and is present only for the eMSP, since it is the only user-facing component of the system.
- **Controller**, logic in charge of using the Model to feed the View and respond to user inputs. It is comprised of the client-side modules used to asynchronously request and send data to the backend, but also the Server-side components handling all API routes.

Reflecting such structure is the hierarchy of directories within the source code, at least for what concerns the backend of the CPMS and eMSP, with the added paths being for tests and helper functions:

- *src/routes* are the express routes realizing the aforementioned controller.
- *src/model* contains all the classes for data representation and retrieval from the database.
- *src/test* contains, following the same hierarchy as the **src** directory, the tests for the entire backend.
- *src/helper* consists of some miscellaneous utility functions, each commented in detail.

The frontend follows a structure not so different from the two backends, with the notable additions of:

- *src/components* stores Vue components, which are reusable user interface elements.
- *src/controllers*, contains JavaScript files that define the behavior and logic of the web application, interacting with the Vue instance and the backend.

- *src/views* contains components that define the layout and structure of the frontend's web pages, as well as which methods are executed based on user interactions.

5.2 Components in the Source Code

What follows is a brief description of the realized components (please refer to the DD for a more complete overview of the components and modules involved).

The only component present here, not previously discussed as such in the DD is the "bank" component, it simply simulates the "payment provider" external module mentioned in the DD.

- **The CPMS component** is comprised of the Request Router (Web Server) for the CPMS realized with Express and the CPMS Application Server, the latter containing every module realizing API endpoints and Data Access Models.
The CPMS accepts connections via WebSockets from the CSClients and allows known eMSPs to interact with its charging stations, from simply searching for them to managing a charging process ongoing at a CS. To simulate a real CS, we realized a mockup which connects to the CPMS as expected, but instead of allowing a vehicle to be charged we exposed a CLI to manually simulate such actions.
- **The EMSP component** is comprised of the Request Router (Web Server) for the EMSP realized with Express, the EMSP Application Server, the latter containing every module realizing API endpoints and Data Access Models, and the EMSP Web Application, which is the fronted of the component, seen by the end user. The EMSP allows users to register and login, as well as reserve and operate (start/end charge) bookings with any of the CSs known to the EMSP. To know about CSs and manage them on behalf of the user the EMSP knows about multiple CPMSs and reaches their APIs to fulfill its tasks. Furthermore the EMSP can reach the external BANK component to verify credit cards validity and charge users for the offered service.
- **The BANK component** was only mentioned in the DD as an external component, here it exists only as a mockup to allow a simplified simulation of how a payment would happen on the EMSP's end. It is comprised of a Request Router realized with Express and a simple Application Server, exposing a simple endpoint which allows to both verify the correctness of payment information and to charge the owner of the provided credit card with a certain amount. Since this component is expected to be an external service, and due to its extreme simplicity (it only performs database lookups/updates) it has not been tested.

6 Testing

See the DD for a more detailed description of the following process.

6.1 Integration Testing

Following what was discussed in the DD document, integration testing has been performed during each phase of the development, which proceeded in parallel for both the CPMS and eMSP backends, with their final integration with one another occurring only when both were already fully ready.

Internally each backend had its development start from the database, then data access modules (the previously discussed Models) followed by the rest of the control logic (the Routes). After the routes went through testing and were complemented with the authentication capabilities, the final set of integration tests could be done before moving on to the frontend development and finally full integration tests.

In more detail every component's API has been tested with Postman (<https://www.postman.com/>) and MySQL WorkBench (<https://www.mysql.com/products/workbench/>), in order:

1. After constructing of each component's database and Model, the utilized queries were tested with MySQL WorkBench by:

- Running the queries directly from WorkBench, with dummy DB entries and query parameters.

- Running the queries from the Model, ran withing Node.js, using dummy entrypoints. The following is an example of such tests for the "CS.ts" model file within the CPMS's backend:

```
CS.getCSList([30, 140], 20, [2, 20]).then( async (result) => { console.log(result);  
await DBAccess.closePool();}
```

This has been done together with active logging within the database instance to ensure the correct format of parameters inserted into prepared statements:

```
SET GLOBAL log_output = 'TABLE';  
SET GLOBAL general_log = 'ON';  
SELECT * FROM mysql.general_log;
```

2. Each realized API interface has been tested during and after construction, together with the Model it relied upon, with Postman, by sending to it HTTP requests and verifying both the correctness of responses and the tolerance of the API to malformed requests. The following are some examples of the performed tests:

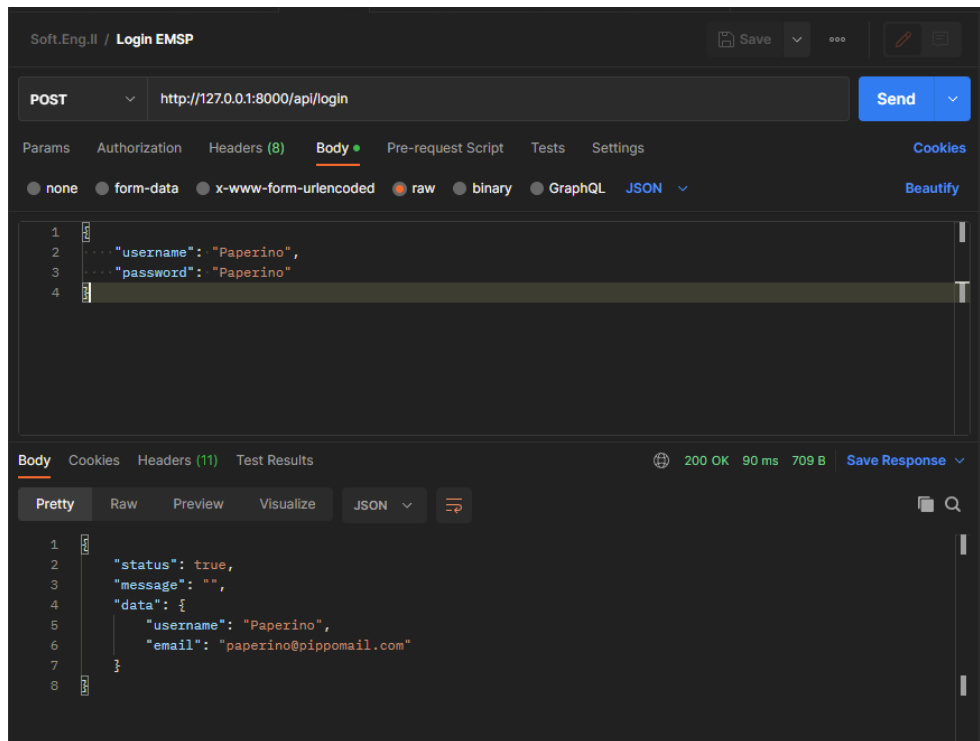


Figure 1: Testing of the eMSP Login endpoint, resulting in a success

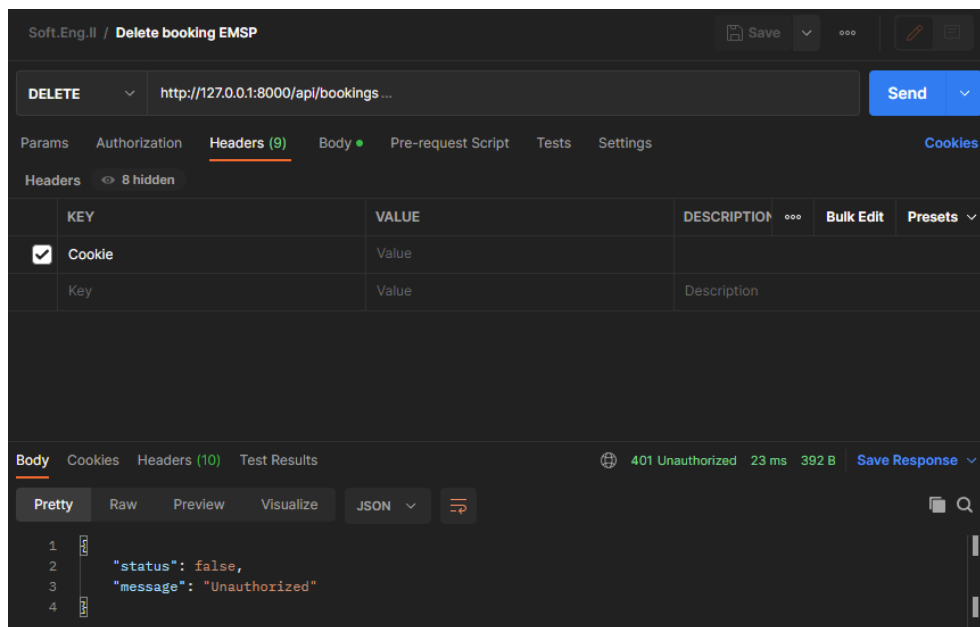


Figure 2: Testing of the eMSP Bookings endpoint, an attempt to access any booking before being logged in fails

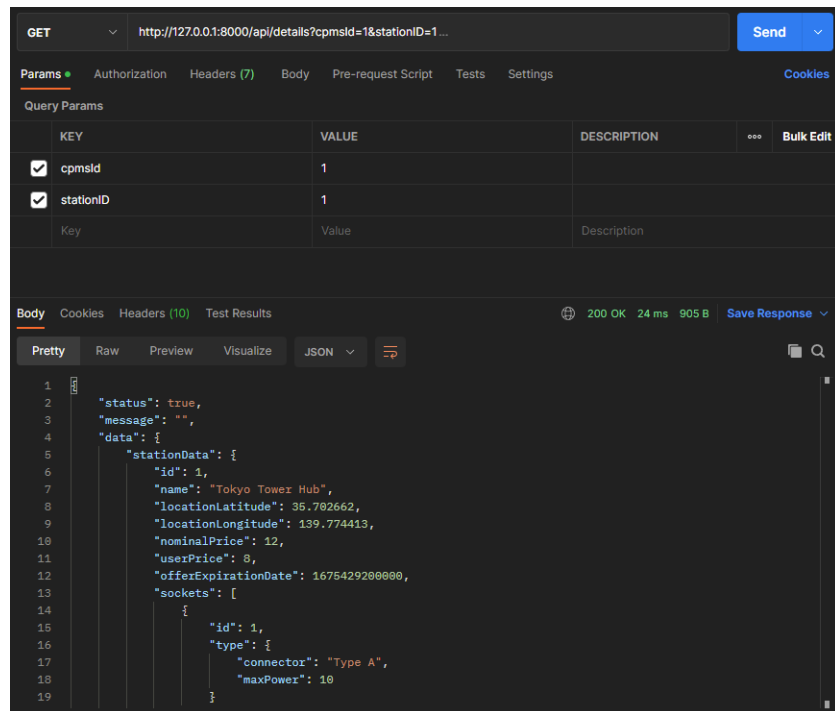


Figure 3: Testing of the eMSP Details endpoint, a logged in user received an object describing the requested CS

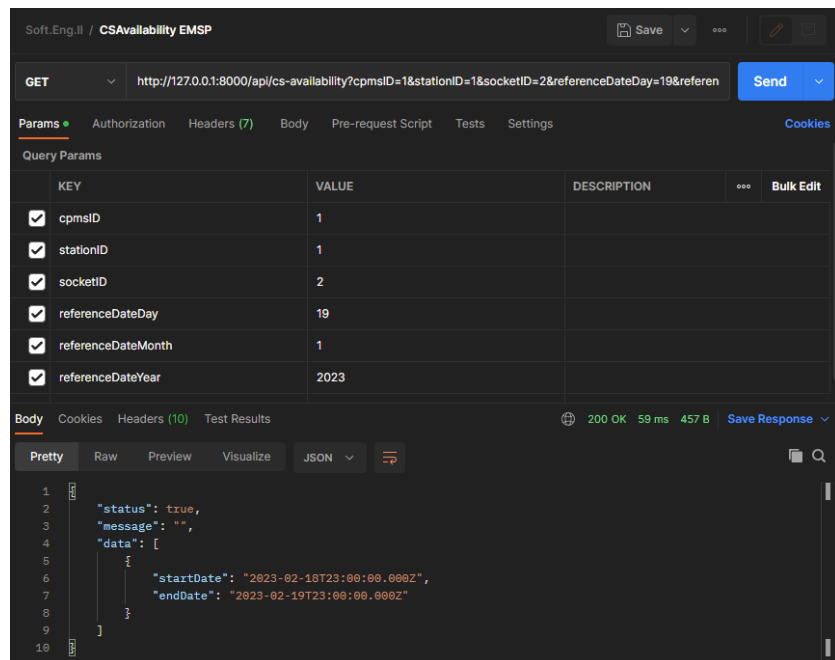


Figure 4: Testing of the eMSP Availability endpoint, it correctly responds to a logged in user with the intervals in which the specified CS is available for the provided date

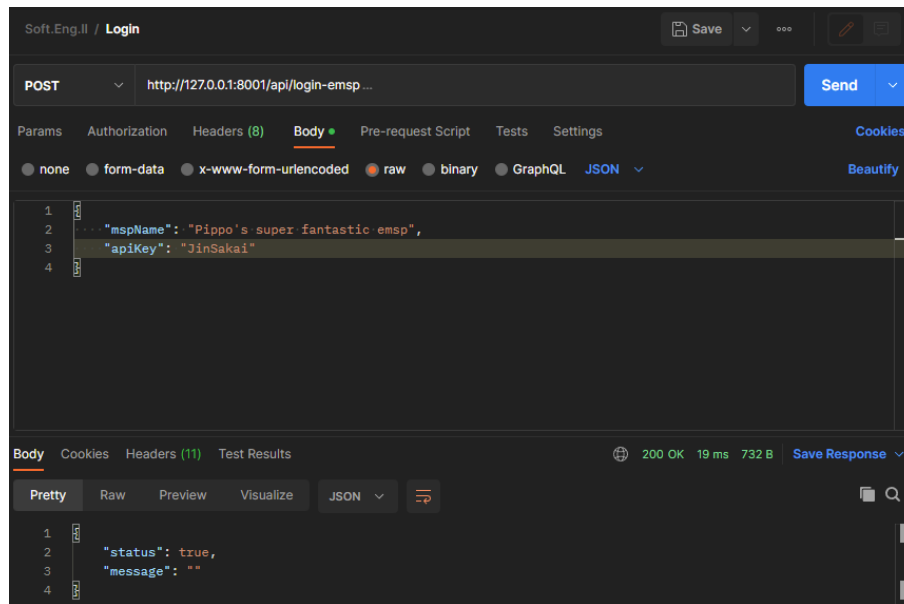


Figure 5: Testing of the CPMS Login-eMSP endpoint, resulting in a success

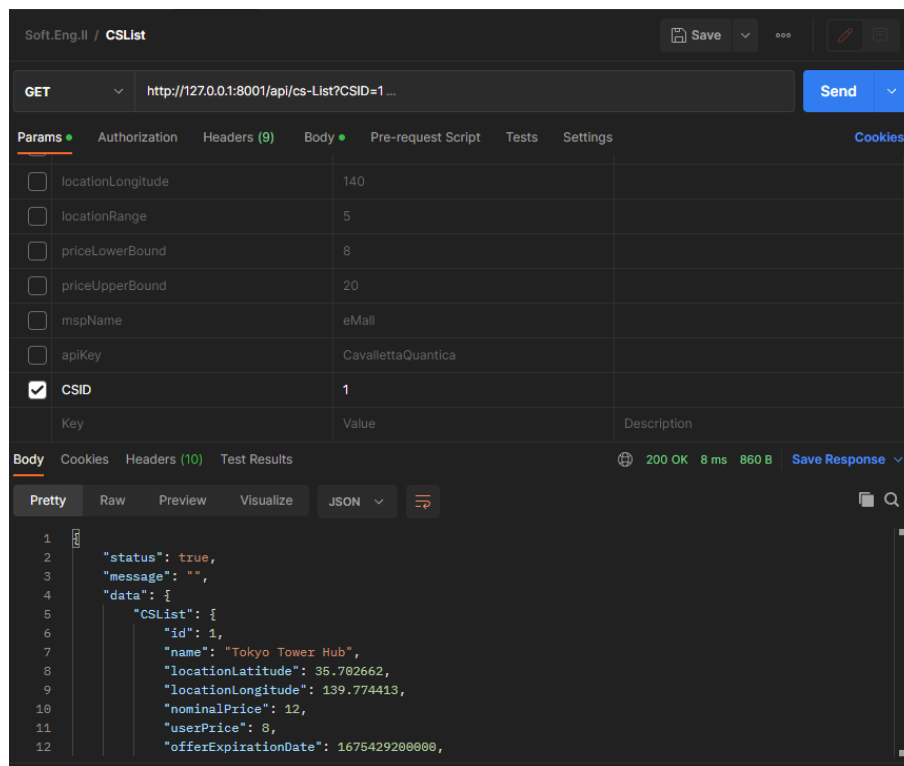


Figure 6: Testing of the CPMS CS-list endpoint, that returns to an authenticated eMSP a filtered list of charging stations information

6.2 Automated Testing

6.2.1 Backend Tests

Automated tests have been written for every route with the aid of Sinon-Chai. During those tests the interactions with the database are always stubbed with SinonStub instances that replace every connections to the DB, returning predefined results that consequently not only test the behaviour of the routes, but of the Model as well. Every route of the CPMS and eMSP has been fully tested, as well as the one (and only) API endpoint for the dummy bank.

To run the tests, assuming that you already followed the **Installation instructions** in the section below, you can follow those instructions:

1. **Run the CPMS's tests** by reaching its location, `YourProjectRoot/cpms/backend`, with a console and running the command: `npm run test`.
2. **Run the eMSP's tests** by reaching its location, `YourProjectRoot/emsp/backend`, with a console and running the command: `npm run test`.
3. **Run the dummy bank tests** by reaching its location, `YourProjectRoot/bank/backend`, with a console and running the command: `npm run test`.

These will run the npm scripts to execute the tests.

You can also produce coverage reports (a pre-evaluated instance is already included in the project) with the following commands:

1. **Evaluate the CPMS's coverage** by reaching its location, `YourProjectRoot/cpms/backend`, with a console and running the command: `npm run coverage`.
2. **Evaluate the eMSP's coverage** by reaching its location, `YourProjectRoot/emsp/backend`, with a console and running the command: `npm run coverage`.
3. **Evaluate the dummy bank coverage** by reaching its location, `YourProjectRoot/bank/backend`, with a console and running the command: `npm run coverage`.

The coverage reports can be viewed by opening `YourProjectRoot/<module>/backend/coverage/index.html`, where `module` is either `emsp`, `cpms` or `bank`.

At the time of writing the coverage results are as follow:






src		62.24%	61/98	36.66%	11/30	44.44%	8/18	62.5%	60/96
src/helper		84.39%	146/173	58.22%	46/79	72.41%	21/29	84.88%	146/172
src/model		94%	204/217	85.54%	71/83	89.39%	59/66	94.14%	193/205
src/routes		100%	123/123	92.3%	72/78	100%	8/8	100%	118/118
src/routes/eMSP Authentication		100%	25/25	100%	6/6	100%	4/4	100%	24/24

Figure 7: Coverage of the CPMS - 87.96% Lines Overall





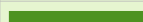
src		77.14%	54/70	50%	12/24	63.63%	7/11	76.81%	53/69
src/helper		94.57%	157/166	77.01%	67/87	92%	23/25	94.51%	155/164
src/model		92.16%	153/166	85.18%	46/54	85.29%	29/34	93.25%	152/163
src/routes		94.97%	321/338	87.5%	161/184	100%	19/19	94.91%	317/334
src/routes/authentication		100%	71/71	100%	34/34	100%	6/6	100%	69/69

Figure 8: Coverage of the eMSP - 93.36% Lines Overall

For a more detailed look at the coverage we suggest opening the `index.html` files directly and navigating the report.

src	<div><div></div></div>	73.43%	47/64	33.33%	7/21	63.63%	7/11	73.01%	46/63
src/helper	<div><div></div></div>	90.62%	58/64	63.33%	19/30	81.81%	9/11	90.62%	58/64
src/model	<div><div></div></div>	100%	9/9	100%	2/2	100%	2/2	100%	9/9
src/routes	<div><div></div></div>	100%	28/28	100%	8/8	100%	2/2	100%	26/26

Figure 9: Coverage of the dummy Bank - 85.8% Lines Overall

Note that the coverage for the `src` folder is reported as lower than 80% due to the presence of the `DBAccess` class, a simple wrapper of MySQL2 calls to help with the creation and handling of MySQL2 connections in pools. As such, testing its behavior would have ultimately resulted in testing the behavior of the MySQL2 library itself, which was out of the scope of the testing process.

6.2.2 Frontend Tests

Some automated tests were also written for the frontend, in order to verify the correct behaviour of Vue controllers. These test were written using the same libraries as Backend tests, but were run using a frontend-specific command called *Vitest*. During these tests, API calls made through Axios were stubbed, in order to verify the behavior of the controllers both in normal and error cases. Model objects have not been tested, since they are only data structures with no additional methods exposed.

Coverage reports have been generated for the frontend as well. These test reports are present in the .zip archive and can be found at `YourProjectRoot/ems/frontend/coverage/index.html`, but you can produce new coverage reports following the steps below.

Assuming that you have already followed the **Installation instructions** in the section below, you can run those tests as well as produce a coverage report for the frontend with them with the following steps:

- **Run the frontend tests** by reaching its location, `YourProjectRoot/emsp/frontend`, with a console and running the command: `npm run test`.
- **Evaluate the frontend coverage** by reaching its location, `YourProjectRoot/emsp/frontend`, with a console and running the command: `npm run coverage`.

Npm scripts will then take care of running the tests and the coverage reports can be viewed by opening `YourProjectRoot/emsp/frontend/coverage/index.html`.

At the time of writing the coverage results are as follows:

controllers	<div><div></div></div>	96.81%	577/596	90.65%	97/107	97.36%	37/38	96.81%	577/596
helpers	<div><div></div></div>	82.75%	24/29	100%	4/4	75%	3/4	82.75%	24/29
model	<div><div></div></div>	97.91%	188/192	89.65%	26/29	100%	22/22	97.91%	188/192

Figure 10: Coverage of the Frontend - 96.57% Lines Overall

7 Installation instructions

7.1 Prerequisites

- **MySQL Server** must be installed on your system, you can download it from here: <https://dev.mysql.com/downloads/installer/>.
For installation instructions for Windows you can refer to this guide: <https://dev.mysql.com/doc/refman/8.0/en/windows-installation.html>.
Make sure to take note of your root password during installation, it will be useful later.

- **MySQL WorkBench** is not necessary, but recommended to allow you to import the provided database dump into your instance of MySQL Server (which is needed in order to properly test the behavior of the system). You can download it from here:
<https://dev.mysql.com/downloads/workbench/>
 After installing WorkBench, connect it to your local instance of MySQL Server (which must be up and running for this to work): first choose to create a new connection (+ button near "My Connections"), then set its host to "localhost" and port to "3306". Your username for the connection is recommended to be "root", and its password the one chosen during the installation of MySQL Server. Note that you can use other user accounts as long as you give them high enough privileges to import a schema and operate on it at will.
- **Node.js** must be present on your system. The project was built and tested using version 18.12.0, which you can download here:
<https://nodejs.org/download/release/v18.12.0/>
 Make sure that node is present in your system path.

7.2 Preparing the Project

1. **Download the Project:** download the ".zip" file provided with the project (It is located under "/DeliveryFolder" on GitHub) and extract it in a known location. As an alternative, you can also download the entire project repository using the "Clone" button provided by GitHub.
2. **Import the schemas in the DB:** open MySQL Workbench and open the DB connection you previously configured. In the left-side menu select "Administration" and then "Import Data/Restore". The screen to import database dumps will open and you will need to select "Import from Self-Contained File" and choose as the file to import `YourProjectRoot/DBs_dump.sql`, and then choose to "Start the Import".
 This will create in your MySQL local instance of the `emsp_db`, `cpms_db` and `bank_db` schemas.
 If you can't access the newly imported schemas after "Schemas" on the left menu, close and reopen WorkBench.
3. **Run NPM** to download all the node packages required by the project. To do this open a terminal and reach each of the following paths within the extracted project folder, then in each path run:
`npm install`
 List of paths:
 - `YourProjectRoot/cpms/backend`
 - `YourProjectRoot/bank/backend`
 - `YourProjectRoot/emsp/backend`
 - `YourProjectRoot/emsp/frontend`
 This should fetch all node modules listed in "package.json" for each of the 4 directories containing (respectively) the CPMS's backend, the dummy bank's backend, the eMSP's backend and the eMSP's frontend.
4. **Configure the ".env" files**, there are 3 ".env" files you need to configure to connect to your MySQL installation, they are located in:
 - `YourProjectRoot/cpms/backend`
 - `YourProjectRoot/bank/backend`
 - `YourProjectRoot/emsp/backend`

The only needed change is to the field "DB_PASSWORD", which should be set to the root password you chose while installing MySQL Server.

If you also are using a different user than "root", you must also change the "DB_HOST" field accordingly.

Note: quotes are not required.

7.3 Running the project

Run each of the following programs, in the specified order (not a must, but highly suggested).

1. **Run the CPMS's backend** by reaching its location, `YourProjectRoot/cpms/backend`, with a console and running the command: `npm run run`.
2. **Run the CS mockup**, for any CS you wish to emulate of the 9 available ones, by reaching its location, `YourProjectRoot/cpms/backend`, with a console and running the command: `npm run run-cs [ID]`, where "ID" is a number from 1 to 9 (extremes included) indicating the charging station ID. Refer to the following figure for which ID corresponds to which charging station.

	id	name	locationLatitude	locationLongitude	nominalPrice	userPrice	offerExpirationDate	imageUrl
1	1	Tokyo Tower Hub	35.782662	139.774613	12.00	8.00	1675429200000	http://www.widest.com/wp-content/uploads/Tokyo-Tower-In-Tokyo-Japan.
2	2	Akiba Bolt	35.782662	139.776447	11.00	11.00		https://www.japan-guide.com/g18/3003_01.jpg
3	3	Daib	45.478611	9.232778	5.00	5.00		https://lh3.googleusercontent.com/p/AF1Qip0YUQwspC8FtuBFeJ70r-0-hh
4	4	Building 26	45.475833	9.234444	5.50	5.50		https://lh3.googleusercontent.com/p/AF1Qip0-0rEIQ-alw_0327T_g07jHcuo
5	5	Linate 1	45.468278	9.278611	8.00	8.00		https://external-content.duckduckgo.com/iu/?u=https%3A%2F%2Fcdn.busi
6	6	Linate 2	45.458056	9.282222	8.00	8.00		https://external-content.duckduckgo.com/iu/?u=https%3A%2F%2Fwww.want
7	7	Duomo	45.464167	9.191667	3.00	3.00		https://external-content.duckduckgo.com/iu/?u=https%3A%2F%2Flive.sta
8	8	CityLife	45.477778	9.155833	6.00	6.00		https://external-content.duckduckgo.com/iu/?u=https%3A%2F%2Fmedia-cd
9	9	Sempione	45.475833	9.172222	6.00	6.00		https://external-content.duckduckgo.com/iu/?u=https%3A%2F%2Fak.jogu

You can run multiple mockups by opening multiple console instances and repeating the above procedure, however do not run two mockups on the same charging station ID.

Also note that the project uses live geolocation services to display the stations nearby, therefore it is highly suggested to emulate stations that are "closer" to your location. This is, however, not a requirement, since you will be able to move around the globe to search for charging stations and pick whichever you prefer.

3. **Run the bank's backend** by reaching its location, `YourProjectRoot/bank/backend`, with a console and running the command: `npm run run`. This is required for new user signups, since it is used to simulate billing operations as well as for credit card validity checks.
4. **Run the eMSP's backend** by reaching its location, `YourProjectRoot/emsp/backend`, with a console and running the command: `npm run run`.
5. **Run the eMSP's frontend** by reaching its location, `YourProjectRoot/emsp/frontend`, with a console and running the command: `npm run dev`.

7.4 Interacting with the running project

1. **Start the WebApp** by opening the URL printed during the frontend's startup, which should be `http://localhost:5173/`.
2. **Create a new account** by following the register link on the login page. You will be asked to enter some user profile data (such as username and password), plus payment details, which will be checked against the ones saved by the dummy bank. A valid set of such details is:
 - **Card Number:** 4365875436666669
 - **Owner:** MARIO LUIGI
 - **Expiration Date:** 11/24

- **CVV:** 123

Those are obviously stand-ins allowed by the "bank" we simulated, more can be added by interacting with the "bank_db" schema in WorkBench.

3. **Login** by simply going back to the login page and inserting the credentials you just created.
4. **Navigate the WebApp**, you can now freely navigate the WebApp by using the menu on the top right, comprising of:
 - **The Bookings Tab**, where you can see all your current bookings and delete some, as well as manage any booking that is currently active (start/stop charge).
 - **The Stations Tab** allows you to search for CSs with the aid of a map. Click on any CS available from the list on the side of the map to see its details and create a new booking if you desire. The map also uses live, external geolocation services to show stations near you, but it will default to a known location in case such services fail to report your accurate position or you denied consent to access it.
5. **Manually operate the CS mockup** by utilizing the following commands:
 - `connectCar [socketId]` : simulates a car connecting to the given socket.
 - `disconnectCar [socketId]` : simulates a car disconnecting from the given socket.
 - `fullyCharge [socketId]` : simulates a car charging completely over the given socket.
 - `quit` : closes the program and quits the emulated station.
 - `help` : to print the instructions above in your terminal window.

8 Effort Spent

8.1 Ronzani Marco - mat: 224578

Task	Time spent
Backend (APIs and Models)	40 <i>h</i>
Frontend	0 <i>h</i>
Tests	16 <i>h</i>
Documentation	4 <i>h</i>
This document	9 <i>h</i>
Total	70 <i>h</i>

8.2 Sassi Alessandro - mat: 220837

Task	Time spent
Backend (APIs and Models)	15 <i>h</i>
Frontend	80 <i>h</i>
Tests	6 <i>h</i>
Documentation	1 <i>h</i>
This document	2 <i>h</i>
Total	104 <i>h</i>

9 References

1. ChargeLab - operating system for EV charges
2. Platform for Electromobility. EV Charging: How to tap in the grid smartly?.
3. F. Campos, L. Marques, and K. Kotsalos, Electric Vehicle CPMS and Secondary Substation Management.
4. Shu Su, Hui Yan, and Ning Ding. 2018. Machine Learning-Based Charging Network Operation Service Platform Reservation Charging Service System. In Proceedings of the 2018 International Conference on Signal Processing and Machine Learning (SPML '18). Association for Computing Machinery, New York, NY, USA, 1–5.
5. Jan Mrkos, Antonín Komenda, and Michal Jakob. 2018. Revenue Maximization for Electric Vehicle Charging Service Providers Using Sequential Dynamic Pricing. In Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS '18). International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 832–840.