POLITECNICO
MILANO 1863

eMall System

e-Mobility for All

IT

Version 2.0 - 05/02/2023

Alireza Yahyanejad – 10886993

Mohammad Hosein Behzadifard – 10880732

Alireza Azadi - 10888648

Table of Contents

Figures

1 Introduction

1.1 Scope of Document

This document is the Integration Testing Plan Document for EMSP and CPMS Platforms. Integration testing is essential to guarantee that all the different subsystems function according to the requirements laid out in the RASD and without exhibiting unwanted behaviors. The purpose of this document is to describe the preparation of the integration testing activity for all the components, which as a whole build up the product. In the following sections, we are going to introduce the requirements/functions that are actually implemented in the software, adopted development frameworks, the structure of the source code, information on how we performed our testing, and the installation instructions.

1.2 Testing

We have used the python test unit and Django Framework default testing tools to test our application. To test our back-end applications, we have used the unit test method. The goal of unit testing is to isolate each part of the program and show that the individual parts are correct. Running the below command to run the test cases:

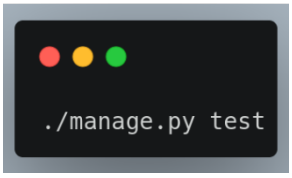

Figure 1 Testing

Designed test cases are as follows:

- Booking:
  - Booking a charging socket with a valid date (no conflict with other bookings on the same charging socket).
  - Booking with invalid data (conflict with other bookings).
- User:
  - Register:
    - Register with a duplicate email
    - Not using a valid password
    - Register with proper information
  - Login:
    - Login with the wrong email/password
    - Login with proper information
  - Profile:
    - Update profile email with a duplicate (used) email

        - Update password
        - Update profile information including name

2 Functions and Features

The list of functions and features implemented in EMSP and CPMS applications are as follows:

2.1 EMSP

- Sign up and Sign in
- Visit and Edit profile (Current user profile)
- Show a list of charging stations
    - Sort them based on the distance from the current user location
    - Sort them based on the offer (discount) and price
    - Retrieve details of a Charing Stations and information about its socket
- Book a Charging Socket for a Specific time frame
- Access to the list of booked items:
    - Filter booking history based on the status
    - Order based on their date, price, and status
    - Get the payment QR code for a successful booking

2.2 CPMS

- Login to the panel
- Retrieve a list of Charging stations
    - Sort them based on the location
    - Receive its statuses:
        - External Status: 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.
        - Internal Status: the amount of energy available in its batteries, if any, the number of vehicles being charged, and, for each charging vehicle, the amount of power absorbed and time left to the end of the charge.
    - Setup its power source: It can be DSO, Battery, or Mixed.
- Manage DSOs
    - Retrieve a list of DSOs
    - Set a DSO as the main DSO for the system to acquire energy
    - Filter them based on their availability
    - Sort them by price
- Managing connected vehicles to the charging sockets: 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 Adopted Development Frameworks

3.1 Programming Languages and Frameworks

3.1.1 Back-end

Both systems' back-ends are implemented with **Python** and **Django Framework**. Back-end applications are connected to the front-ends through the REST API (Django REST Framework). back-end are connected are interconnected with this approach.
The decision to use Python and Django for the back end of the project was driven by several factors. One of the major reasons was our prior experience with these tools. As experienced developers in both Python and Django, we were able to leverage our knowledge to quickly build and implement the required functionality for the project. Additionally, Django is a powerful web framework that provides a comprehensive set of tools for developing robust, scalable, and secure web applications. Its built-in ORM and support for MVC architecture make it ideal for rapidly developing database-driven applications. Furthermore, Python has a large and active community of developers, ensuring that there is always a wealth of support and resources available for any challenges that may arise. Overall, the combination of my experience with Python and Django and the framework's capabilities made it an excellent choice for the back-end of this project.

3.1.2 Front-end

Both systems' front-ends are implemented with **JS** (**JavaScript**), **VueJS**, and **NuxtJS**. Front-end applications are connected to the back-ends through the REST API.
The choice to use JavaScript and VueJS for the front-end development of the project was driven by several key factors, including our experience with these technologies. As experienced developers in both JavaScript and VueJS, we were able to leverage our skills to quickly and effectively build out the required user interface and functionality. VueJS is a modern, progressive JavaScript framework that provides a simple and intuitive approach to building user interfaces. Its reactive data bindings, modular architecture, and performance optimizations make it ideal for building fast and dynamic web applications. Furthermore, the widespread use of JavaScript and its growing ecosystem of tools and libraries make it a reliable choice for front-end development. The combination of my experience with these technologies and the capabilities of VueJS made it an excellent choice for the front-end of this project. NuxtJS's goal is to help Vue developers take advantage of top-notch technologies, in a fast, easy, and organized way. It abstracts away most of the complex configuration involved in managing asynchronous data, middleware, and routing. It also helps structure Vue.js applications using industry-standard architecture for building simple or enterprise Vue.js applications.
Despite Nuxt.js version 3 being stable for a few months, we still decided to use version 2 since we had more experience with it. It is also the same case for Vue.js.

3.1.3 Database Service

The back-end services of both databases are **Postgresql**. Due to the need to work with geo-locations, we have used the **Postgis** extension. PostgreSQL is a highly capable and scalable relational database management system that is well-suited for handling complex data structures and relationships. This makes it an ideal choice for building sophisticated web applications, which is a common use case for Django.
Another advantage of using PostgreSQL with Django is its robust feature set. PostgreSQL includes powerful data manipulation and analysis tools, such as full-text search and geospatial indexing, which can be leveraged to build rich and sophisticated applications. These features can be easily accessed from within Django through the use of the PostGIS extension, providing seamless integration between the database and the web framework.
Additionally, PostgreSQL is an open-source technology that is widely adopted and has a large and active community of developers, ensuring that there is always a wealth of support and resources available. This makes it easier to find solutions to any challenges that may arise and helps to ensure that the technology remains up-to-date with the latest advancements.
PostGIS is an extension of the PostgreSQL database that adds support for geographic data, enabling the storage and analysis of spatial information. The PostGIS extension extends the capabilities of PostgreSQL by adding a suite of functions and data types for working with spatial data, such as points, lines, and polygon shapes. It also includes support for advanced geographic analysis and manipulation, such as intersection, union, and difference calculations. This makes PostGIS an ideal choice for applications that require the management and analysis of geographic data, such as location-based services, environmental monitoring, and geographic information systems (GIS). With its flexible data model, rich set of features, and excellent performance, PostGIS is a powerful tool for working with spatial data.

4 The Structure of The Source Code

4.1 Back-end

Due to using the Django Framework, the project structure follows a standard Django MVC structure. In the following, we will discuss the structure of projects.

4.1.1 EMSP

The project has 3 Django apps, app_booking, app_cs, and app_user, each with its own models, serializers, views, admin, apps, and URL configurations. The description of these files are as follows:

- Views: A view is a Python function that takes a web request and returns a web response. It is the heart of the application, where you perform the necessary processing for a given URL. The response can be the HTML contents of a Web page, a redirect, a 404 error, an

XML document, an image... or anything, review is called by the Django framework when a request is made to a specific URL.

- Models: A model is a Python class that defines the structure of an object in your application. This object is mapped to a database table and can be used to retrieve, store and manipulate data in the database.

- Serializers: Serializers are used to convert complex data types, such as Django models, into Python data types that can be easily rendered into JSON, XML, or other content types. Serializers in the Django REST framework work very similarly to Django forms.

- URLs: The URL dispatcher is the component of Django that maps URLs to the view functions that should be called for a given request. It is a simple mapping between URL patterns and Python functions. URLs are defined in a separate file called urls.py. In this file, you map URLs to the views that should be called for a given request.

- admin.py: a module in a Django app that is used to define the admin interface for that app.

    - The admin interface is an automatically generated web-based interface for managing the data in your database tables.
    - By customizing admin.py, you can change how your models are displayed in the admin interface, add extra functionality, and more.
    - You can also use the admin.py file to define custom actions for your models, such as bulk deletion or export.

- apps.py: a module in a Django app that is used to define the configuration for that app.

    - The apps.py file is automatically created for you when you create a new Django app.
    - You can use the apps.py file to configure various settings for your app, such as the default order of models, whether your app is available in the Django admin interface, and more.
    - The apps.py file is also used when you want to create custom application configurations that can be reused throughout your Django project.

Separating these components allows for easier organization, testing, and maintenance of the codebase.

We have tried to keep the app loosely coupled. Hence, each app has its own responsibility as follows:

- app_booking: handle and provide APIs for the charging point booking process.
- app_cs: managing and providing APIs for charging stations and charging points.
- app_user: APIs for user authentication, registrations, and profile management.

- utilities: Common and general modules and functions among other apps. It is not a Django app but a simple python module.

Each registered app that contains at least one implemented model in models.py will have at least one migration file in its migrations folder. Migration files are a way to version control database schema changes in Django. They are used to manage changes to your model classes and keep your database schema in sync with your models. Migrations are used to apply and revert changes to your database schema in a controlled and repeatable way, without the need to manually manipulate the database.
Migration files contain operations for creating, modifying, or deleting tables, fields, and indexes, as well as creating and modifying constraints and other database-specific features. When you make changes to your models, Django automatically generates new migration files, which can then be applied to the database to bring it in line with the latest version of your models.
Migrations can be applied incrementally, so you can easily apply changes to the database schema one step at a time. This makes it easier to track changes to your database and troubleshoot any issues that may arise.
The *emsp* folder contains the project settings module. In the project to separate different environments (including development and production), we use the Django modular settings best practice. The *developments.py* contains all settings for the development of the project and running the project in the development environment and the *production.py* file includes all settings to set up the project in the production environment. One of the major differences between these two different environments is that in the production environment, the *DEBUG* option is set to False. It is not safe to show the debugging information in the production environment. They are some common settings in both environments such as the database settings which exist in the *base.py* file.
It is important to point out that some settings values such as the database are reading from the environments variable. Using environment variables in your settings file is considered good practice because it allows you to separate sensitive information such as passwords, API keys, and other secrets from your codebase. This can help improve the security of your application and make it easier to manage the configuration of different environments, such as development, testing, and production. When you use environment variables, you can store the values in a file outside of your codebase, making it easier to update and manage the values as needed. This can also help you avoid hardcoding sensitive information, which can be a security risk.
We can define commands for an app in Django. We have defined a faker.py command in the app_cs/management/commands path in this project.

4.1.2 CPMS

We have tried to keep the app loosely coupled. Hence, each app has its own responsibility as follows:

- app_admin: handle and provide APIs for the CPMS admin panel. Including CS APIs.
- app_ds: managing and providing APIs for updating DSOs' status and retrieving them.
- app_user: APIs for user authentication, registrations, and profile management.

- utilities: Common and general modules and functions among other apps. It is not a Django app but a simple python module.

4.2 Front-end

4.2.1 CPMS

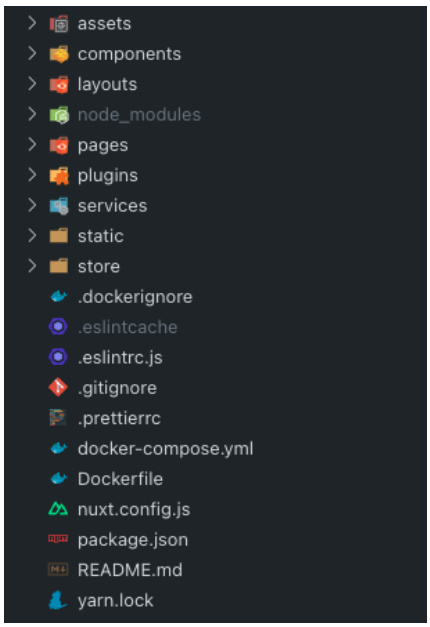

Figure 2 Structure of CPMS - Front-end

The default Nuxt.js application structure is intended to provide a great starting point for both small and large applications. We are free to organize our application however we like and can create other directories when needed.

4.2.1.1 The pages directory

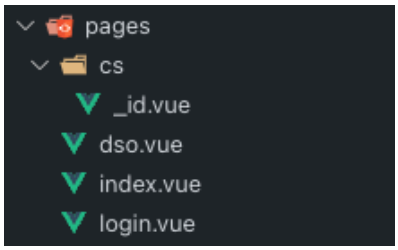

Figure 3 The pages directory

contains our application's views and routes. Nuxt.js reads all the ".vue" files inside this directory and uses them to create the application router. There are four pages in this project:

- The “index.vue” page is used as the main page containing the charging stations list.
- The “login.vue” page displays the login form when the user is not already logged in.
- The “dso.vue” page shows a list of DSOs.
- The “_id.vue” page under the “cs” directory is a dynamic-path page. It accepts an ID in the route. For instance, all of these paths are handled by this page:
    - /cs/123
    - /cs/5
    - /cs/517349

The “id” part can be later used in the code to load the data from the backend API. This page has the table of sockets and external/internal info of the charging station for the given ID.

4.2.1.2 The components directory

The components directory is where we put all our Vue.js components which are then imported into our pages.
With Nuxt.js, we can create our components and auto-import them into our “.vue” files, meaning there is no need to import them in the script section manually. Nuxt.js will scan and auto-import these for us once we have components set to true in the “nuxt.config.js” file.
In this directory, you can find two main subdirectories. The “auth” directory is used to for all the login and registration components (authentication components). And the “main” directory contains the commonly used components across the project.

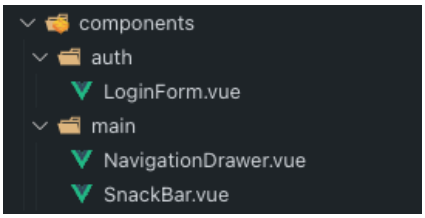

Figure 4 The components directory

The “LoginForm.vue” file under the “auth” directory is responsible for the login form on the “/login” page. It accepts the username and password as inputs. Then it locally validates them and finally sends them to the backend API. It also shows the authentication errors (if any).
The “NavigationDrawer.vue” file under the “main” directory shows the left-side menu bar. It lists different pages (charging stations and DSOs) alongside the logout button.
The “SnackBar.vue” under the same directory shows different messages for different actions. It can have different styles for various needs (success, warning, error, info).

4.2.1.3 The assets directory

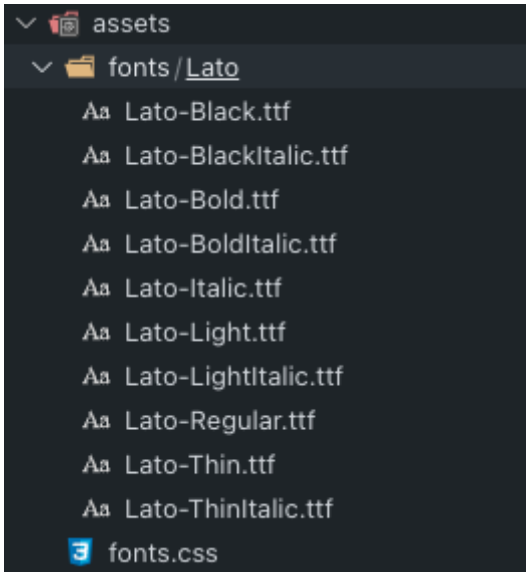

Figure 5 The assets directory

contains our uncompiled assets, such as our styles, images, or fonts.

For this project, we have used the “Lato” font, which the TrueType Font files of it alongside the CSS file for importing it into the project can be found under this directory.
You can also find the “global.scss” file here, primarily used to provide global styling for the projects, including the main background color, the default car box shadows, etc.
The “variables.scss” file is used to define a few CSS variables which will be used later in different places of the project.
Since both of these files are not vanilla CSS files, they need to be first compiled into CSS (using webpack) and then bundled with the final export. This procedure is automatically done when building the project. Finally, the “reset.css” file is used to reset some of the default stylings of browsers' elements.

4.2.1.4 The static directory

The static directory is directly mapped to the server root and contains files that must keep their names (e.g., robots.txt) or likely won't change (e.g., the favicon).

4.2.1.5 The nuxt.config.js file

The “nuxt.config.js” file is the single point of configuration for Nuxt.js. If we want to add modules or override default settings, this is the place to apply the changes.
Various modules and plugins used in this project are all configured within this file. Modules such as:

- Vuetify: The UI library that is used to show inputs, forms, tables, etc.
- Nuxt Auth: The authentication module with handles storing the JWT token, sessions, auth middlewares, etc.
- Axios: The HTTP client that is used to send the HTTP requests to the backend APIs

4.2.1.6 The package.json file

The “package.json” file contains all the dependencies and scripts for our application.

4.2.1.7 The plugins directory

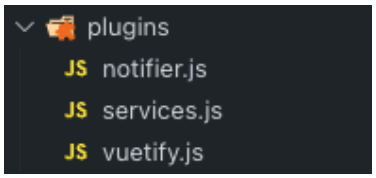

Figure 6 The plugins directory

contains the JavaScript plugins we want to run before instantiating the root Vue.js application. There are three plugins used in this project:

- notifier.js: This plugin provides an API for the entire project to interact with the “SnackBar.vue” component found under the “components/main” directory. It has a few simple helper functions to show various messages with different types (success, error, etc.)
- services.js: This plugin injects the CS and DSO service files into the main context object (accessible by “this” keyword in JavaScript), which can later be used to communicate with the backend APIs.
- vuetify.js: Since we are using material design icons in this project, the CSS file for that is imported into this plugin.

4.2.1.8 The services directory

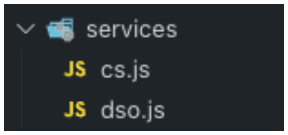

Figure 7 The services directory

You can find every service file used within this project under the services directory. Service files are used to provide a simple interface (API) for the rest of the project to communicate with the backend API, aka sending the HTTP requests and parsing the response data.
These service files accept a few or no inputs for each of their methods and send the HTTP requests to the predefined backend endpoints (APIs) using the Axios HTTP client.
The “cs.js” and the “dso.js” are used to communicate with charging stations and DSO backend endpoints, respectively.

These service files are later injected into the main context object (accessible by the “this.$services” keyword) by the “services.js” plugin under the “plugins” directory (dependency injection).

4.2.1.9 The store directory

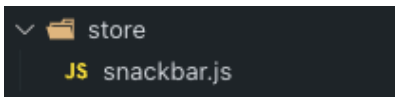

Figure 8 The store directory

contains our Vuex.js store files. The Vuex.js store comes with Nuxt.js out of the box. Vuex.js is a state management pattern + library for Vue.js applications. It serves as a centralized store for all the components in an application, with rules ensuring that the state can only be mutated predictably.
You can find a single “snackbar.js” file under this directory which is used to globally store the message that is used to show the snack bar in the project.

4.2.1.10 The layouts directory

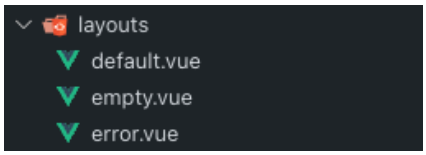

Figure 9 The layouts directory

Nuxt.js provides a customizable layouts framework we can use throughout our application, ideal for extracting common UI or code patterns into reusable layout components. Layouts are placed in the “layouts” directory and will be automatically loaded via asynchronous import when used.
We have three layout files for this project:

- default.vue: This is the default layout that is used for all pages. It contains the “NavigationDrawer.vue” and the “SnackBar.vue” components. All of these components, alongside the main “Nuxt” component (“<Nuxt />” tag) are inside a “v-container” component to introduce some default margins and stylings.
- empty.vue: This layout is like the default layout with the difference that it does not include the “NavigationDrawer.vue” component and also does not put them inside the “v-container” tag to have the default margins.
- error.vue: This is the layout used to show any internal error (404, 500, etc.).

Note: there are some other files in the project which are not directly being used by us and are only there to help us develop code with better style and format (e.g, .eslintrc.js)

4.2.2 EMSP

4.2.2.1 The pages directory

There are four pages in this project.

- The “index.vue”: is used as the main page containing the “select date and time” form to find the charging stations (displays the list of charging stations for the given start and end date times).
- The “login.vue”: displays the login/signup form when the user is not already logged in.
- The “index.vue”: page under the “panel” directory shows the profile page, in which the user can see his/her info (such as first name or last name) and also update his/her profile.
- The “reservation_history.vue”: under the same directory shows a list of past charging station reservations that the user has.

4.2.2.2 The components directory

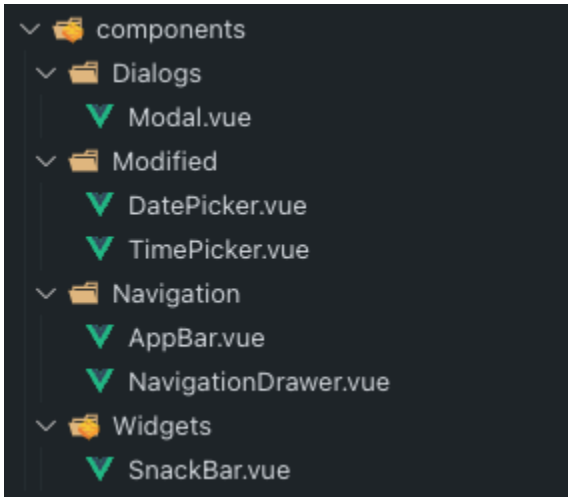

Figure 10 The components directory

In this directory, you can find four main subdirectories.

- Dialogs: This contains the “Modal.vue” component and is used to show various dialogs.
- Modified: This has two components:
    - DatePicker.vue: This is used to get the start/end dates when looking for a charging station.
    - TimePicker.vue: This is used to get the start/end time when looking for a charging station.

- Navigation: This has two components:

    - AppBar.vue: This is the app bar for the website and shows the title and user's full name.
    - NavigationDrawer.vue: This shows the left-side menu bar. It lists different pages (home, booking history, profile) alongside the logout button.
- Widgets: This contains the commonly used components throughout the project. In this case, it only has one component with is the "SnackBar.vue" component. It is used to show different messages for different actions. It can have different styles for various needs (success, warning, error, info).

4.2.2.3 The services directory

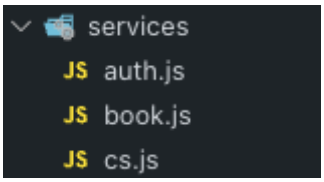

Figure 11 The services directory

The "auth.js" is used to send the "signup" and "update profile" requests. The "book.js" is used to book a socket of a charging station or to get the history of users' past reservations. The "cs.js" is used to communicate with charging station backend endpoints. It can get the list or the details of a specific charging station.

5 Installation Instruction

Both back-end and front-end projects are fully dockerized therefore they can be set up easily. Both applications can be run on any machine that installed docker. To run the projects, you can refer to the *Readme.md* file in the projects folder for details. In general, to run the back-end projects, run the below command in the project's folder in the terminal:

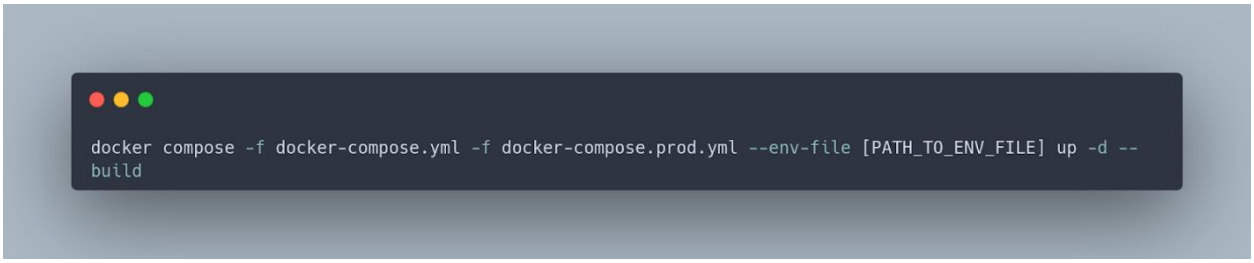

Figure 12 Installation Instruction - back-end

For the front-end projects:

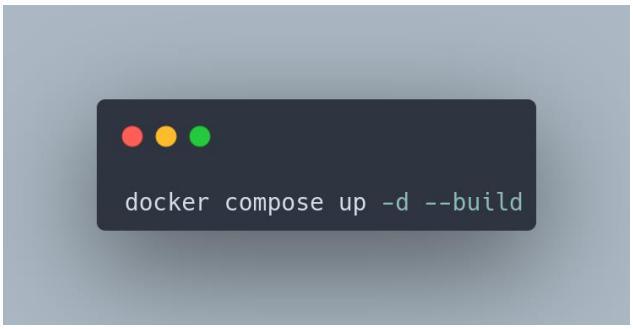

Figure 13 Installation Instruction - front-end

All the required services including the database will be installed through the docker.

6 Effort Spent

Name	Section	Time
Alireza Azadi	Back-end (CPMS and EMSP)	40h
Alireza Yahyanejad	CPMS front-end	36h
Mohammad Hosein Behzadifard	EMSP front-end	30h

7 References

- Specification Document: “Assignment RDD AY 2022-2023.pdf”
- Course slides
- https://evroaming.org/app/uploads/2021/11/OCPI-2.2.1.pdf