AY 2022/2023

Politecnico di Milano

ITD: Implementation Document

Marcello De Salvo Riccardo Grossoni
Francesco Dubini

Professor
Elisabetta Di Nitto

Version 1.0
February 3, 2023

Contents

1 Introduction

The code can be found in the official project repository on GitHub at the link: https://github.com/MarcelloDeSalvo/DeSalvoDubiniGrossoni.git.

1.1 Purpose

The objective of this document is the realization of a full technical description of the system presented in the RASD document. Here we will analyze both hardware and software architectures, focussing on the interaction between components that constitute the system. Additionally, we will also delve into the implementation, testing and integration process. This document will use technical language as it's aimed for programmers, but stakeholders are also invited to read as it can be useful to understand the characteristics of the development.

1.2 Definitions, acronyms, abbreviations

Acronyms

- **RASD**: Requirement Analysis and Specification Document
- **DD**: Design Document
- **ITD**: Implementation Document
- **API**: Application Programming Interface
- **DBMS**: Database Management System
- **OCPP**: Open Charge Point Protocol
- **OCPI**: Open Charge Point Interface
- **CPOW**: Charge Point Operator Worker
- **UML**: Unified Modeling Language
- **GPS**: Global Positioning System
- **UI**: User Interface

- **HTTPS**:HyperText Transfer Protocol Security
- **CSRF**: Cross Site Request Forgery
- **HTML**: HyperText Markup Language
- **CSS**: Cascade Style Sheet
- **JS**: JavaScript
- **MVVM**: Model View View-Model
- **MVC**: Model View Controller
- **REST**: Representational State Transfer
- **JSON**: JavaScript Object Notation
- **JWT**: JSON Web Token
- **URL**: Uniform Resource Locator
- **TS**: TypeScript
- **DRF**: Django REST Framework
- **ACID**: Atomicity-Consistency-Isolation-Durability
- **IDE**: Integrated Development Environment

1.3 Revision history

- Version 1.0: first release

1.4 References

- Django Framework: `https://www.djangoproject.com/`
- REST Framework: `https://www.django-rest-framework.org/`
- Nuxt 3 Framework: `https://nuxt.com`
- Vue.js: `https://vuejs.org/`

- PostgreSQL: `https://www.postgresql.org/docs/14/index.html`
- Vercel: `https://vercel.com/docs`
- Digital Ocean: `https://www.digitalocean.com`
- Tailwindcss: `https://tailwindcss.com/docs`

2 Development

2.1 Implemented Functionalities

We implemented the following functionalities for emsp users and cpo's operators:

User

- Visualization of nearby stations and their status
- Make a booking
- Delete a booking
- Visualization of all bookings
- Start charging session

CPOW

- Access informations on the stations
- Change the power Source of the stations (battery and DSO)
- Add and remove discounts

2.2 Functionalities not implemented

For what concerns the eMSP, we didn't implement neither the payment nor the suggestion system as they weren't required. We didn't fully implement the charging process initiated by start charge, as many features were impossible for us to implement (the user gets notified after a bit while inside the page as a prototype). We also couldn't implement the connection to multiple CPMSs, as the only one we had available was ours. To emulate it, we created a CPMS app that would save their various CPMS links to the database and would fix the posts and gets basing on the saved urls.
In the CPMS subsytem a lot of functions required a real communication with a DSO and the ChargingStation, so it was not possible to truly implement them. We inserted mocked data relative to the DSOs and the batteries

in the database based on our discretion. We also didn't fully implement the discount activation feature, as it was hard to test how the discounts and daily checks would behave with days passing. We thought of a daily check in pseudocode the machine would run, placed inside Discount/Management/Commands, that in an ideal setting would be run every day by the server. This could be done by setting a scheduled 24 hour run of "manage.py check_for_discounts".
A similar reasoning was applied to the dynamic DSO decision, given that our data was static. We implemented a button that displays the funcionlity on the CPMS portal, but in reality it would be activated by a trigger placed on the DSO price database.

2.3 Implemented requirements

Here it follows the list of requirements taken from the RASD. The implemented ones are checked with a checkmark.

Shared requirements

R1. The system must allow registered and logged-in users to use the app. ✓

R2. The system must respect the GDPR and the user's privacy. ✓

R3. The system must allow registered users to log-in using their e-mail. ✓

R4. Both the CMPS and the eMSP subsystem must abide to the OCPI 2.2.1 protocol. ∼

EMSP requirements

R5. The eMSP subsystem must show the user the nearby charging stations through an interactive map. ∼

R6. The eMSP subsystem must be allowed to use the user's GPS location in order to view the nearby charging stations, if given permission. ×

R7. The eMSP subsystem must notify the user when the charge has finished. ✓

R8. The eMSP subsystem must show the user the discounted prices. ✓

R9. The eMSP subsystem must show the user the prices of the charging stations. ✓

R10. The eMSP subsystem must access the User calendar schedule in order to suggest the best charging timeframes, if given permission. ×

R11. The eMSP subsystem must suggest the user on which charging stations to go based on the EV battery and his daily schedule. ×

R12. The eMSP subsystem must communicate with the CPMSs in order to exchange all needed information about the charging stations. ✓

R13. The eMSP subsystem must allow the user to book a charging spot for a future date. ✓

R14. The eMSP subsystem must allow the user to cancel a future reservation for a charging spot. ✓

R15. The eMSP subsystem must allow the user to pay for the charge through an external payment service. ×

R16. The eMSP subsystem must allow the user to start or stop the charge via the application. ✓

R17. The eMSP subsystem must notify the user of the upcoming booked sessions. ×

CPMS requirements

R18. The CPMS subsystem must communicate with the DSO according to a standard protocol. ×

R19. The CPMS subsystem must retrieve the DSO current energy price. ✓

R20. The CPMS subsystem must automatically decide from which DSO to acquire energy. ∼

R21. The CPMS subsystem must retrieve the charging station's battery status. ✓

R22. The CPMS subsystem must dynamically change the energy source depending on the internal station's battery status and the available DSOs' prices. $\sim$

R23. The CPMS subsystem must communicate the location of the charging stations to all connected eMSPs. ✓

R24. The CPMS subsystem must retrieve the internal status of the sockets through the OCPP standard protocol. $\sim$

R25. The CPMS subsystem must retrieve and communicate to all connected eMSPs the external status of the sockets through the OCPP and the OCPI standard protocols. ✓

R26. The CPMS subsystem must allow to start or stop a charging session through the OCPP standard protocol. ✓

R27. The CPMS subsystem must retrieve the battery status of the EV through the DIN/ISO specifications. $\times$

R28. The CPMS subsystem must allow the CPOW to add or change the current special offer. ✓

R29. The CPMS subsystem must allow the CPOW to change the energy provider of a charging station. ✓

R30. The CPMS subsystem must unlock the reserved charging spot if the user doesn't show up. ✓

2.4 Design Choices

By using TailwindCSS, we efficiently created a UI that is both scalable and optimized for mobile devices. However, during the development and testing process, the focus was primarily on the desktop version of the web application.

2.5 Adopted Development Frameworks

Framework selection was based on factors such as ease of use, availability of support, and alignment with commonly used design patterns in current real-world applications. For the front-end we opted for Nuxt 3 and Vue, which

utilizes the Model-View-ViewModel (MVVM) design pattern (an evolution of the traditional MVC). For the back-end we chose Django, which enforces a standard Model-View-Controller (MVC) pattern for the API.

MVC

In the MVC pattern, the View handles user requests and returns a response, the Model manages the data access and manipulation logic, and the Controller acts as an intermediary between the Model and View to process user requests and manage the flow of data.

MVVM

As previously said, the MVVM is and extension of MVC mainly used in web development. MVVM separates the different components of the development process into three categories, model, view and ViewModel. The ViewModel serves a similar role as the Controller in MVC, but with some key differences: the ViewModel is responsible for exposing data from the Model in a form that can be easily consumed by the View thanks to a binder that automates their communication. With respect to the MVC, the MVVM pattern makes it easier to manage and test large applications, but it lacks standardization and increase the complexity of the architecture.

2.6 Programming languages

The programming languages used in the project are Python and JavaScript. For both the backends (CPMS and EMSP) Python was chosen since it is a very versatile language that is easy to learn and use. For the front-end, TypeScript was chosen since it is a superset of JavaScript that adds static typing and object-oriented programming to the language.

2.6.1 Django Framework

We used Django as the backend framework for the CPMS and EMSP subsystems. Django is a widely-used high-level Python web framework that provides a clean and pragmatic design. It has several advantages that make it a great choice for web development projects:

- **Rapid Development:** Django has a lot of built-in functionality, including a powerful admin interface, which makes it possible to develop web applications very quickly.
- **Scalability:** Django is designed to handle high traffic, which makes it a great choice for large-scale projects. It provides a solid foundation for scaling up your application as your needs change.
- **Security:** Django provides robust security features, including protection against cross-site scripting (XSS), cross-site request forgery (CSRF), and SQL injection attacks.
- **Ease of use:** Django has a straightforward syntax and follows the Model-View-Controller (MVC) architectural pattern, making it easy for developers to understand and work with.
- **Large Community:** Django has a large and active community of developers who contribute to the development of the framework, provide support and share best practices.
- **Third-party Packages:** Django has a large number of third-party packages, which makes it easy to add additional functionality to your application, such as authentication, payment processing, and more.

Overall, Django is a robust and reliable framework that provides a lot of built-in functionality, security, scalability, and ease of use, making it a good choice for web development projects.

Django Middlewares

To manage the communication between the two backends and frontend, we mainly use the following Django middlewares:

- *SecurityMiddleware*
- *SessionMiddleware*
- *CsrfViewMiddleware*
- *AuthenticationMiddleware*
- *MessageMiddleware*
- *CorsMiddleware*

2.6.2 Django REST Framework

We decided to pair REST framework to our Django backends since REST allows even more security features. Django Rest Framework (DRF) is a powerful and flexible toolkit for building Web APIs. It is built on top of the Django web framework, which makes it easy to integrate with existing Django applications. DRF provides a number of features that make it a good choice for building RESTful APIs, including:

- **Serialization:** DRF provides a simple and flexible way to serialize and deserialize data, which helps to convert complex data structures into a format that can be easily transmitted over the web.
- **Authentication and Permissions:** DRF provides a range of authentication and permission classes, which can be easily configured to secure your APIs. This allows you to control who has access to your data and what actions they can perform. We chose *JWTauthentication* and *IsAuthenticated* as the authentication and permission classes.
- **URL Routing:** DRF provides a simple and powerful URL routing mechanism, which makes it easy to map URLs to views and handle different HTTP methods (such as GET, POST, PUT, and DELETE).
- **Extensibility:** DRF is highly extensible, which means that you can easily add custom functionality to meet the specific needs of your project. There are a wide range of third-party packages available that can be used to add additional features to your APIs.

2.6.3 Nuxt 3 Vue

Nuxt.js is a JavaScript framework built on top of the popular Vue.js framework. Nuxt.js is often used to build modern and powerful web applications, including Single Page Applications (SPAs) and Server-Side Rendered (SSR) applications. Some of the benefits of using Nuxt.js include:

- **Easy setup and deployment**: Nuxt.js makes it easy to set up and deploy your applications, with a simple configuration file and easy-to-use command-line tools.

- **Automatic code splitting**: Nuxt.js automatically splits your code into smaller chunks, making it easier for users to load your application quickly.
- **Built-in support for Vue.js**: Nuxt.js is built on top of Vue.js, so you can take advantage of the powerful features and tools that Vue.js provides, such as reactive data bindings and a component-based architecture.
- **Modular and reusable code**: Nuxt.js allows you to easily write modular and reusable code, making it easier to maintain and scale your application.
- **SSR capabilities**: Nuxt.js provides built-in support for server-side rendering, making it easy to create fast and scalable applications that are optimized for search engines.
- **Integration with other tools and technologies**: Nuxt.js can be easily integrated with other tools and technologies, such as APIs and databases, making it a versatile and flexible choice for web development.

2.7 API Integration

For what concerns the integration of APIs, we ended up not implementing any of them. Car API, Google calendar and Payment API weren't implemented as they weren't required.
Google Calendar and the Car API were key in the suggestion module and the payment API was likely unpractical for us to implement. The only one that was actually feasible was the Maps API, but we decided to invest our time into more meaningful features as the live position of the user could be later implemented at no cost.

2.8 DataBase

For our database we opted for PostgreSQL, a well known object-relational database that provides a reliable, robust and ACID-compliant system and that guarantees high performance and support with Django.

2.9 Vercel

We chose to host our front-end with Vercel, a platform as a service (PaaS) that allows to build, run, and host applications entirely in the cloud for free.

- `de-salvo-dubini-grossoni.vercel.app`

2.10 Digital Ocean

In order to host both the CPMS and the eMSP backends in a simple way, we decided to use Digital Ocean, another platform as a service (PaaS) that allows to host and run django applications and postgres databases on the cloud. We created a domain for each backend system that can be used to make HTTP request and access django rest framework API views:

- `https://sea-lion-app-4fmmp.ondigitalocean.app`
- `https://sea-lion-app-4fmmp.ondigitalocean.app/cpms/`

3 Source Code

3.1 Backend Structure

Following the project structure described in the RASD and DD documents, we opted to make two subsystems for the eMSP and the CPMS. By doing so, we had to separate the two subsystems' backends inside two different Django projects, both with the same structure:

- **mysite**: root folder
- **__init.py__**: it tells the Python interpreter that the directory is a Python package
- **settings.py**: main setting file for the Django project, used to configure all the applications and middleware, it also handles the database settings
- **urls.py**: URL mapping declaration for the API. Maps each View to an url endpoint

- **wsgi.py**: is the entry point for WSGI-compatible web servers to serve your Django application. It sets up the environment for the Django application to run and serves as a bridge between the web server and Django.
- **asgi.py**: entry-point for ASGI-compatible web servers to serve your project, ASGI works similar to WSGI but comes with some additional functionality
- **migrations**: folder that contains all the operations that should be applied to the database in order to create models' tables, updates and inserted data
- **admin.py**: used for registering the Django models into the Django administration, it allows to display them in the Django admin panel
- **apps.py**: common configuration file for all Django apps, used to configure the attributes of the app
- **models.py**: defines the structure of your database tables by creating Django model including the relationships between them, as well as any custom methods you want to add to your models to encapsulate business logic
- **tests.py**: it defines all the unit tests that can be run to test the application's functionalities
- **views.py**: provide an interface through which a user interacts with a Django website, it contains the business logic of the app
- **manage.py**: command-line utility for executing Django commands for debugging, deploying, running and testing

3.1.1 Apps

3.2 eMSP backend

- **User**: contains all the models and views related to the user, from logging to authentication and account creation.
- **Booking**: contains all the models and views related to bookings, from booking creation to booking cancellation.

- **Cpms**: contains all the models and views related for saving the different CPMS the user will be able to interact with.
- **OCPI**: contains the views relative to the OCPI protocol, it allows to send and receive OCPI messages to and from the CPMSs.

3.3 CPMS backend

- **User**: contains model and views related to the CPOW, from logging to account creation.
- **Booking**: contains all the models and views related to bookings, so the stations can see their associated bookings.
- **ChargingStation**: contains all the models and views related to bookings, allowing it to interact with the OCPI and showing the CPOW useful information.
- **Socket**: contains all the models and views related to the sockets, allowing the CPOW to create them and assign them to a specific station.
- **EnergyProvider**: contains all the models and views related to both the internal BSS and the DSOs, handles all the requests related to the energy providers.
- **OCPI**: handles all the views relative to the OCPI protocol, allowing it to send and receive OCPI messages to and from the eMSP.

3.4 Frontend Structure

Here is represented the structure of the web app:

- **.nuxt**: contains everything needed to generate your vue application. It's hidden by default and should not be touched since it will be generated after e build.
- **.output**: holds all build files when building your Nuxt application to production (nuxt build). It's hidden by default and should not be touched since it will be generated after e build.

- **node_modules**: contains all installed dependencies of the project, listed in the *package.json* file. It's generated automatically every time the project is set up.
- **assets**: contains by convention every asset that you want the build tool (Vite or webpack) to process.
- **components**: contains all the vue components that can be imported inside the pages.
- **layouts**: contains all vue layouts that can be applied to the pages.
- **middleware**: contains middlewares' scripts that can intercept the users requests before serving a page. For instance, contains the authorization script that blocks access to unauthorized pages.
- **pages**: holds every application's View. Nuxt reads all the .vue files inside this directory and automatically creates the router configuration.
- **public**: stores static files (images, audio etc.) that should not be processed by the build tool (Vite or webpack).
- **utils**: holds useful javascript files that can be imported to avoid code repetition.

There are other several files needed by Nuxt 3 to run the project. The most important are:

- **package.json**: it contains the list of all the dependencies installed in the project.
- **tailwind.config.js**: it contains themes and configurations used by TailwindCSS.
- **.env.example**: example file to know what environment variables KEY=VALUE pairs you need for the project to run.

4 Testing

While devepoling, we noticed many component functionalities were codependent on the other server running. As a result we decided to emphasize system testing over unit testing.

4.1 Unit Testing

For unit testing we used django built-in tests framwork that allows to run every test cases inside each app's tests.py file.

Emsp Booking

- Tested if booking insertion was done correctly.
- Tested if booking deletion was done correctly.
- Tested if *getBookings* endpoint returned correctly the requested bookings.
- Tested if method *get_ bookings_ by_ user* returned correctly the requested bookings.

CPMS Charging Station

- Tested if charging station insertion was done correctly.
- Tested if *requestChargingStationById* endpoint returned correctly the requested charging station.
- Tested if *getChargingStations* endpoint returned correctly all the charging stations.

CPMS Socket

- Tested if Socket insertion was done correctly, including the relation with its charging station.
- Tested if *getSocket* endpoint returned correctly the requested Socket.
- Tested if *resetSocket* endpoint returned a correct status.

CPMS Discount

- Tested if Discount insertion was done correctly, including the relation with its charging stations.
- Tested if Discount deletion worked correctly.
- Tested if *requestChargingStationById*, which includes the applied discounts in the data returned, returns all the valid discounts for that charging station.

4.2 System Testing

The testing phase included manual testing of the app through a web browser and with Reqbin, an online API testing tool for REST and SOAP APIs, for forging custom HTTP GET/POST requests and evaluating the server's response. This approach allowed us to verify the authentication process and all other backends' views.

4.3 Post-deployment Testing

The deployment of the application was followed by an extensive testing phase. This comprehensive testing approach allowed for a thorough examination of the entire system, including both the front-end and back-end components, and their interactions with each other. The testing was conducted in order to identify any potential issues and ensure that the system was functioning as intended.

5 Installation

Since the web application is already deployed on Verecel at:
`https://de-salvo-dubini-grossoni.vercel.app` we suggest to use that instead of proceeding with the local installation. If you instead want to proceed with a local set-up you can follow below instructions.

5.1 Requirements

In order to install every needed dependency and have a working **PostgreSQL** databse you have to install:

- **Node.js** >= 16.0
  - Download the latest version of Node.js `https://nodejs.org/it/download/`
- **Python** >= 3.9
  - Download Python 3.10 or equivalent from `https://www.python.org/downloads/`
- **PostgreSQL**
  - Download the installer of the latest version from `https://www.postgresql.org/download/` for your OS, that comes also with **PGadmin**, a web-based GUI used to communicate easily with Postgre
- **Git**
  - Download the installer of the latest version from `https://git-scm.com/downloads`

5.2 Initial installation

The repository was subdivided in four main branches, one for each required module and the documentation.

1) main
   - contains the project's documentation, including RASD, DD and the ITD.
2) front-end
   - web application, used for both the eMSP and the CPMS.
2) emsp-back-end
   - contains the Django backend project for the eMSP.
2) cpms-back-end
   - contains the Django backend project for the CPMS.

Instructions:

1) Clone the git repository at `https://github.com/MarcelloDeSalvo/DeSalvoDubiniGrossoni.git`

2) Open a terminal and cd into the root folder of the project

3) Execute these git worktree commands. At the end of the process you should have four folders, one for each branch, that can be opened independently inside your favourite IDE.

```
git worktree add ../emall_frontend front-end
git worktree add ../emsp_backend emsp-back-end
git worktree add ../cpms_backend cpms-back-end
```

Now you can open each folder inside your IDE and start following the next instructions.

5.2.1 Backend

Optional: create a virtual environment to install all the required dependencies. You can use any virtual environment managers like venv, conda or virtualenv. For this example we will use venv:

1) create a virtual environment named "django"

```
python -m venv /path/to/new/virtual/
   environment/django
```

2) activate the environment in WINDOWS

```
path\django\Scripts\activate.bat
```

or in LINUX

```
source path/django/bin/activate
```

Then, remember to link the path to the environment inside your IDE settings.

5.2.2 CPMS backend

Once inside the cpms backend folder, you can install the requirements by running

```
pip install -r requirements.txt
```

Make sure to make a .env file inside mysite/mysite containing the following informations:

- **SECRET_KEY** key used by Django to manage authentication and hashing messages. You can set your own.
- **DATABASE_PORT**, **DATABASE_USER**, **DATABASE_PASSWORD** credentials for the PostgreSQL database
- **DATABASE_URL** credentials to the database if deployed on the cloud. You can set a random placeholder or igore it if the database is running locally.
- **DEVELOPMENT_MODE** tells Django if your database is running locally or not, and selects the correct credentials for the database. Could also be used for other checks.
- **DEBUG** checks if debug mode is enabled
- **ALLOWED_HOST** a list of strings representing the host/domain names that this Django site can serve. This is a security measure to prevent HTTP Host header attacks.

Example:

```
SECRET_KEY=secret_key
DATABASE_PORT=5432
DATABASE_USER=postgres
DATABASE_PASSWORD=password
DATABASE_URL=link
DEVELOPMENT_MODE=True
DEBUG=True
ALLOWED_HOSTS="*"
```

To then setup your database, install postgresSQL and make a DB with

- **Name** Your_db_name, must match with the one in the .env file
- **Passwrod** The passord with which you will access your db
- **User** Name of the user that is going to access the db
- **Host** Url of the local database
- **Port** Port the database is listening to

Example:

```
name = your_db_name
password = your_db_password
user = your_db_user_name
host = your_db_local_url
port = your_db_port
```

Finally, to run the server

```
cd /mysite
manage.py makemigrations
manage.py migrate
python manage.py runserver
```

5.2.3 EMSP backend

To correctly setup the eMSP backend, repeat all the steps of the CPMS backend, but inside the emsp backend folder and changing the db name. Every step can also be found inside the README.md files.
For the EMSP setup an additional argument in the .env is needed:
CPMS_URL the url for the CPMS backend.
Example:

```
CPMS_URL = your_cpms_url
```

5.2.4 Frontend

1) Open the 'emall_frontend' folder inside your IDE

2) Create a .env file following .env.example

   - **EMSP_URL**, is the url of the eMSP backend.
   - Ex: EMSP_URL=`http://127.0.0.1:8000`
   - **CPMS_URL**, is the url of the CPMS backend.
   - Ex: CPMS_URL=`http://127.0.0.1:8001`

3) Open a terminal and execute

```
npm install --global yarn
```

This will install the `yarn` package manager that will be used to install and manage our web application. Alternatively you can just use `npm`.

```
yarn
```

This will install all the dependencies listed inside the **package.json** file.
Finally you can run:

```
yarn run dev
```

to start the development server at `http://127.0.0.1:3000`

6 Effort Spent

Student	Time for implementation
Marcello De Salvo	20000h
Francesco Dubini	0 h (licenziato)
Riccardo Grossoni	3 KW/h

7 References

References

[1] MDN Web Docs Glossary: Definitions of Web-related terms -> MVC `https://developer.mozilla.org/en-US/docs/Glossary/MVC`