Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

FINAL DEGREE PROJECT

TFG TITLE: Design, development and testing of full-stack web service for a trajectory computation algorithm.

DEGREE: Double Bachelor's Degree in Aerospace Systems Engineering and Telematics Engineering

AUTHOR: Maria Cáliz González

DIRECTOR: Xavier Prats Menéndez, David De La Torre Sangrà

DATE: July 6th, 2023

Títol: Disseny, desenvolupament i testing d'un servei web full-stack per a un algorisme de càlcul de trajectòries.

Autor: Maria Cáliz González

Director: Xavier Prats Menéndez, David De La Torre Sangrà

Data: 6 de juliol de 2023

Resum

Aquest document proporciona un informe complet del projecte de final de grau, amb l'objectiu principal de dissenyar, desenvolupar i provar un servei web full-stack complet per facilitar la interacció amigable amb un software de càlcul de trajectòries anomenat Dynamo. El nucli d'aquest projecte rau en prendre un software existent, complex i fer-lo més accessible i senzill d'utilitzar per a una àmplia base d'usuaris a través del desenvolupament d'un servei web.

Aprofitant una àmplia gamma de tecnologies - Python, Flask, Vue.js, Tailwind i MongoDB - i seguint metodologies modernes de desenvolupament de software com Agile, s'ha dissenyat i implementat un servidor backend, una interfície frontend i una estructura de base de dades. A més, s'ha integrat una autenticació segura i un sistema eficient per a la gestió d'errors i validació, per garantir una experiència segura i amigable per a l'usuari.

Un aspecte clau d'aquest projecte ha estat la necessitat de comprendre com funciona Dynamo, tot i que no es va entrar en els detalls operatius del programari en si. L'enfocament principal va restar en el seu procés de configuració, proporcionant la base per al desenvolupament d'un servei web que permet als usuaris configurar, executar i monitoritzar les simulacions de forma intuïtiva. Això facilita una forma més senzilla per als usuaris de treure partit del poder de Dynamo, independentment del seu coneixement del codi. Per això, es va realitzar una captura de requisits del servei web amb investigadors del grup de recerca ICARUS, garantint així que el servei web desenvolupat s'adapti a les seves necessitats.

Tots els aspectes d'aquest projecte, des de la comprensió del procés de configuració de Dynamo fins al desenvolupament del backend i del frontend del servei web, es van abordar sistemàticament i es detallen en aquest document.

Title: Design, development and testing of full-stack web service for a trajectory computation algorithm.

Author: Maria Cáliz González

Director: Xavier Prats Menéndez

Date: July 6th, 2023

Overview

This document delivers a comprehensive report of the final degree project, which had as its primary goal, the design, developing, and testing a full-stack web service to facilitate user-friendly interaction with a trajectory computation software called Dynamo. The crux of this project lies in taking an existing, complex software and making it more accessible and simpler to use for a broad user base through the development of a web service.

By leveraging a wide array of technologies - Python, Flask, Vue.js, Tailwind, and MongoDB - and following modern software development methodologies like Agile, a backend server, frontend interface, and a database structure were designed and implemented. In addition, secure authentication and an efficient system for error handling and validation were integrated to ensure a secure and user-friendly experience.

A key aspect of this project has been the need to understand how Dynamo functions, even though we did not go into the intricate operational details of the software itself. The main focus remained on its configuration process, providing the basis for developing a web service that allows users to intuitively configure, run, and monitor their simulations. This facilitates a more straightforward way for users to leverage the power of the Dynamo software, regardless of their technical proficiency. For this reason, a web service requirements capture was carried out with researchers from the ICARUS research group, thus ensuring that the developed web service adapts to their needs.

All aspects of this project, from understanding Dynamo's configuration process to the development of the backend and frontend of the web service, were tackled systematically and are detailed in this document.

INDEX

INTRODUCTION

This document describes the work developed for the final degree project whose main objective is the design, development and testing of a full-stack web service for a trajectory computation algorithm, Dynamo. The driving force behind this project is to enhance user interaction with the Dynamo software, allowing them to configure and execute simulations through a user-friendly, web-based interface. Consequently, the intricate processes involved in trajectory computation are simplified, making it accessible to a wider user base with varying levels of technical proficiency.

The execution of this project involved a wide range of tasks and sub-tasks, including the development of both a backend server and a frontend interface, the design and implementation of database structures, the establishment of secure authentication procedures, and the management of error handling and validation systems, among others. The project utilizes a comprehensive tech stack, including Python and Flask for the backend, Vue.js and Tailwind for the frontend, along with MongoDB for the database. A deep understanding of these technologies and their successful integration was crucial to build a unified and robust system.

The project adopted contemporary software development methodologies such as Agile, to ensure a structured and efficient development process. Regular meetings and the use of project management tools like Trello were crucial for effective communication, coordination, and tracking of project milestones.

The Dynamo software [1] is a robust and flexible tool, created by the ICARUS research team at Universitat Politècnica de Catalunya (UPC). The primary function of this software is to provide sophisticated computational capabilities for flight trajectory optimization and prediction, forming an integral part of the trajectory-based operations concept implementation. This software's prowess lies in its ability to provide accurate and optimal flight trajectories for various look-ahead times and operational contexts.

This document is organized into six main sections:

- 'Introduction' provides an overview of the project, its objectives, and the technologies and methodologies used.
- Chapter 1, 'Overview and Used Technologies', details the nature of web services, their types, and the technological landscape adopted in this project.
- Chapter 2, 'Analysis of Requirements', outlines the process of gathering project requirements.
- Chapter 3, 'Backend', focuses on the design and implementation of the backend server.
- Chapter 4, 'Frontend', discusses the design and execution of the frontend interface.
- Chapter 5, 'Test and Results', presents the results and outcomes of the project, alongside the identification and handling of system bugs.

- Lastly, Chapter 6, 'Future Work and Conclusions', wraps up the document, outlining potential future work and summarizing the project's conclusions.

In terms of references, this document is divided into two sections for clarity and ease of access. The 'Bibliography' section lists academic and technical references, such as articles, books, and technical documents. These are numbered with square brackets like so: [1], [2], [3], etc. The 'Useful Links' section lists additional online resources that have been consulted during the development of this project. These are also numbered but in a slightly different format: [‡1], [‡2], [‡3], etc. This distinction ensures that both types of resources are easily identifiable throughout the document.

CHAPTER 1. OVERVIEW AND USED TECHNOLOGIES

This chapter aims to provide a comprehensive understanding of web services, with a particular focus on a specific type. It also discusses the different technologies available for building web services and explains why a particular set of languages were chosen for the design and development of the project.

1.1. Understanding Web Services

Web services [‡1] are a standardized way for software applications to communicate with each other over the Internet, no matter what infrastructure or platform they are built on. Essentially, it is a method of communicating between two electronic devices over a network. The main purpose of web services is to ensure that applications can interact with each other, facilitating the exchange of data and processes.

1.1.1. Types of Web Services

The two main categories of web services are SOAP (Simple Object Access Protocol [‡2] and REST (Representational State Transfer) [2].

1.1.1.1. SOAP Web Service

SOAP is a protocol defined by the World Wide Web Consortium (W3C) for exchanging structured information when implementing web services in computer networks. It encodes its HTTP-based [3] calls using XML [4] and has built-in error handling. As a protocol, SOAP is stricter and defines a standard set of rules that must be followed, making it a suitable choice for applications where integrity and confidentiality are critical. SOAP can operate over multiple protocols such as HTTP, SMTP [5] and TCP [6], making it highly versatile. However, it can be more complex and use more resources than REST.

1.1.1.2. RESTful Web Services

On the other hand, REST is an architectural style for building web services. It's not limited to using XML to provide responses—you can also find REST-based web services that return data in Command-Separated Value (CSV) [7], JavaScript Object Notation (JSON) [8], and Really Simple Syndication (RSS) [9]. The point is that besides XML, any media format that can be used to represent data can be used in REST.

REST uses a client-server model, where server applications provide resources or services, and client applications access those resources. The server does not save data between requests, the session is saved on the client. This statelessness and the ability to cache or store responses makes RESTful services faster and more reliable. Thus, it is simpler, uses less bandwidth, and is more flexible. REST uses standard HTTP methods such as GET, POST, PUT and DELETE (see figure 1.1), making it a straightforward choice for web-based interactions. [‡3]

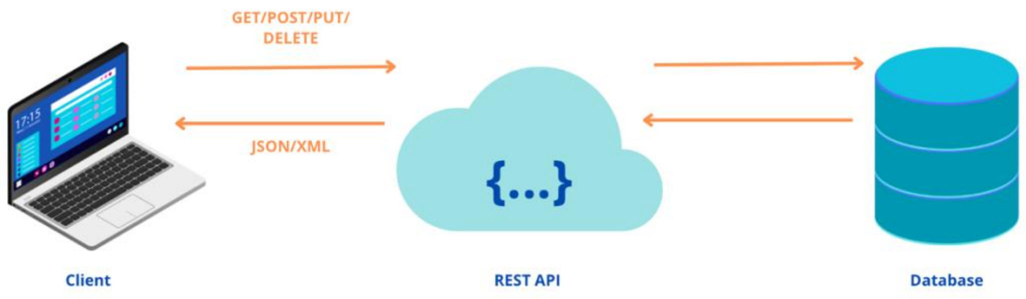


Fig. 1.1 Web Service REST scheme

1.1.2. REST vs SOAP

The choice between SOAP and REST relies on the needs of the application, even though both provide solutions to construct web services. However, due to its simplicity, enhanced efficiency, and superior scalability, REST was chosen over SOAP for this project. [‡4]

- **Simplicity**: RESTful web services are simple to use and comprehend. The interaction between the client and the server is made simpler by the usage of conventional HTTP methods. This ease of use extends to the debugging and testing procedures as well, making it simple to identify and fix problems.
- **Performance**: REST is faster than SOAP since it needs fewer bandwidth and resources. Due to its heavy XML usage and verbose protocol, SOAP uses more processing power and memory.
- **Scalability**: REST is inherently more scalable because it is stateless. As all the information required to process a request is contained within the request itself, RESTful services can be easily distributed across multiple servers to meet high demand.
- **Compatibility**: In terms of flexibility and interoperability with web technologies, REST is thought to be superior. It makes use of common HTTP protocols and is compatible with a variety of data types outside only XML. This makes REST services easier to integrate with already-existing websites or apps since RESTful services may be used by any client that comprehends HTTP, the web's standard protocol.

The choice between REST and SOAP must always be made in the context of the specific project requirements. For instance, if the application demands a higher level of security, SOAP may be a more appropriate choice due to its support for WS-Security. In contrast, if the application needs to be lightweight, flexible, and compatible with the web, REST could be the better choice. In this project, given the need for scalability, flexibility, and speed, **REST was the chosen method.**

1.2. Technology Landscape

Web development comprises a wide variety of technologies, each with its own strengths and use-cases. This section will examine the various backend and frontend languages and database technologies available for web development, providing an overview of their advantages, limitations, and a comparison to provide a clear picture of the current state of the art. We will focus primarily on Python [‡5], Java [10], and JavaScript [11] for backend languages and SQL [12] and NoSQL [13] for database technologies.

1.2.1. Server-Side Technologies

1.2.1.1. Backend languages for Web Development

The backend, or server-side, is the powerhouse of any web service. Although users do not interact with it directly, it's in charge of a web application's logic, server configuration, data administration, and general performance. Backend languages play a role in this. They're the tools that developers use to build and maintain the server-side of web applications, and they handle tasks such as server connections, database interactions, and server logic. Choosing the right backend language is crucial to the efficient operation of your web service, as it can significantly impact speed, scalability, and ease of use. In this section, we will explore several popular backend languages to determine the most suitable one for our project.

- **Python**: Python is well-known for being a flexible language for creating reliable online applications, and it provides outstanding readability, making it a top choice for new programmers and quick deployments. With its enormous library, Python, often referred to as the "Lego" of programming languages, promotes code reuse and significantly lowers the requirement to create individual components from scratch. Its extensive use in data analysis, algorithm development, and machine learning is proof of its adaptability. [‡6]

  Python is open source, expressive, and simple to learn. High-level programming is possible without the use of explicit memory management. Python programs may run on a variety of operating systems with adequate respect for system-specific capabilities thanks to their portability.

Line by line interpretation of Python code eliminates the requirement for compilation. Its broad standard library offers a wide range of modules and functions, and its object-oriented methodology encourages code reuse. Python enables dynamic typing, freeing the programmer from having to declare the data types of variables. [‡7]

We may use some of Python's web development frameworks to extend the capabilities of the basic language [‡8]:

- Django: Django [‡9] is a thorough and reliable framework that is ideal for building complex web applications. The Django programming language is effective and places a strong emphasis on security by offering defense against frequent SQL injection and cross-site request forgery threats. However, Django's extensive features and configurations can lead to a steep learning curve for new users.

- Flask: Flask [‡10] is a micro-framework, not requiring any particular library or tools for web development. It is designed to enable quick and easy development of lightweight applications. Flask is flexible and comfortable for beginners, although its use of third-party modules can potentially lead to security issues.

- FastAPI: FastAPI [‡11], a modern, fast, and robust framework, is used for building APIs with Python (3.6 and above). It's one of the fastest Python frameworks. FastAPI speeds up the creation of programs and reduces bugs. Its autocomplete capability makes creating and debugging applications easier. However, its relatively small community and limited external resources can be a downside.

- **Java**: Java, a high-level, object-oriented, general-purpose language, adheres to the 'write once, run anywhere' philosophy. Because of the Java virtual machine (JVM), Java code does not need to be built for every system type where it will be used. Java uses ideas like polymorphism, abstraction, encapsulation, and inheritance. It is an effective tool for web development because of its security features, robustness, portability, multi-threading, and distributed computing characteristics. [‡12]

- **JavaScript**: JavaScript, particularly its Node.js runtime environment, is another powerful option for backend development. Node.js offers simplicity, freedom in app development, and simultaneous request handling. However, it also presents challenges like frequent API changes, longer development time, and unsuitability for heavy-computing apps. [‡12]

- **PySpark**: PySpark is the Python API for Apache Spark. It enables distributed processing of large data tasks across multiple nodes. One of the key features of PySpark is its 'parallelize' function. This function allows for the splitting of data into an RDD (Resilient Distributed Dataset) [14] -

the primary data structure of Spark, which can then be processed in parallel across a cluster. [‡13]

The core features of Apache Spark, and by extension PySpark, include:

- In-memory computation for faster data processing by storing data in the server's RAM.
- The ability to work with various cluster managers, offering flexibility in the distribution and management of processing tasks.
- Fault tolerance, ensuring that if a node (a computer in the cluster) fails, the system can recover without loss of data or a system-wide failure.
- The creation of immutable data structures (specifically, RDDs), meaning that once an RDD is created, it cannot be modified. Transformations applied to an RDD create a new one, leaving the original unchanged.

1.2.1.2. Database Technologies for Web Development

Database technologies are essential for web development. They ensure efficient data storage, retrieval, and manipulation. In this project we are considering two of the major database technologies available in the market: SQL and NoSQL. The choice largely depends on the requirements of the specific application. [‡14]

- **SQL**: SQL (Structured Query Language) databases are relational, meaning they have predefined relationships between their elements. Examples of SQL databases include DB2 [‡15], Postgres [‡16], Oracle [‡17], and MySQL [‡18]. SQL databases ensure no duplication of records, support a larger community, guarantee atomicity of information, and have a standard system for database operations. Some of the main advantages of these databases are:
    - Greater support as they have been on the market for a long time (larger community)
    - Atomicity of information. When performing any operation on the database, if any problem arises, the operation is not performed.
    - It has a well-defined standard system (SQL) for operations with the database, such as inserts, updates, or queries. This system is easy to understand as it adapts to common language.

- **NoSQL**: NoSQL (Not Only SQL) databases, unlike SQL databases, don't require predefined relationships between their elements. They are non-relational and handle data organized as documents. Examples of NoSQL databases include MongoDB [‡19], Cassandra [‡20], and CouchDB [‡21]. NoSQL databases offer versatility, low cost, horizontal scalability, and the ability to store data of any type at any time. Some advantages of using NoSQL databases are:
    - Versatile databases that allow you to add information or make changes to the system without the need to add extra configurations.

- Open-source NoSQL databases do not require a license fee and do not need very powerful hardware to run.
- Support horizontal growth, that is, by supporting distributed structures, new operating nodes can be installed that balance the workload. Its expansion is easier due to this horizontal scaling.
- Allow saving data of any type, at any time, without requiring verification.

1.2.1.3. Final decision for server side

After thorough research, it is clear that any of the languages and databases discussed above could suit our API development. However, each language and database have unique advantages that make it better suited to specific circumstances. Personal experience also plays a crucial role in technology selection. For instance, if a developer is more comfortable and experienced with Python, they would likely choose it over Java.

Considering all factors, it has been decided to use **Python** for backend development. This decision is based on Python's simplicity, versatility, and my familiarity with it. Among Python's frameworks, we will use **Flask**, as it is flexible and scalable and is also beginner-friendly and makes the development process faster, ideal for a project that requires quick iterations and adaptations. As for the database, **MongoDB**, a NoSQL database, would be a suitable choice given its speed and capacity to handle thousands of user requests per second.

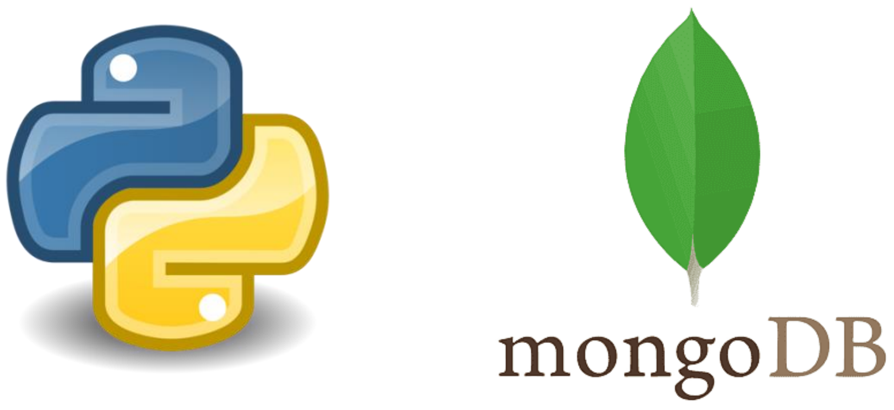


Fig. 1.2 Python and mongo DB logos

1.2.2. Client-Side Technologies

In this section, we'll delve into client-side technologies that are crucial to the design and functionality of any web application. We'll discuss frontend programming languages and make a final decision on which technology best suits our needs.

1.2.2.1. Frontend languages for Web Development

In web development, the frontend is the visible part of the platform where users interact and access content, therefore, the frontend is made up of all client-side technologies. An ideal user experience, immersion, and usability are some of the goals that a good frontend developer strives for, and in the modern era, there are a wide variety of frameworks, preprocessors, or libraries to help you achieve them. That is why, in this section, we will analyze and compare some of the main technologies that currently exist to develop a high-quality frontend. By doing this, we will be able to choose the most appropriate technology according to our case.

Any web page on the Internet is built, at a minimum, by HTML [‡22] (a markup language) and CSS [‡23] (a style language). HTML defines the meaning and structure of the web content while CSS allows you to style the page and build a more pleasant visual interface for the user. However, using only and exclusively HTML and CSS on a page, limits us considerably. Although it is true that with these two languages we can do a wide range of things, there are others that would be totally impossible, or at least much easier to do if we had a programming language. This is where **JavaScript** [15] and **TypeScript** [16] come in, the leading programming languages in web development. These languages are responsible for providing interactivity to development, programming the behavior of the elements to provide dynamism to the interface. Create animations, objects, cookies, data validation in the forms, etc.
In addition to the programming languages that define the "language" and how the code will be written, it is important to highlight the importance of development frameworks. A series of pre-written tools and libraries that make everyday tasks easier for developers to avoid running them from scratch. Therefore, it may be claimed that libraries oversee finding solutions by making the code more readable, whereas frameworks provide a framework for programming in a particular language. Next, we analyze the most used frameworks currently.

We have considered two libraries based on **JavaScript**: React [‡24] and Vue.js [‡25].

- **React**: React, developed, and maintained by Facebook, is a JavaScript library for building user interfaces, especially single-page applications. It allows developers to create large web applications that can change data without reloading the page. One of React's unique features is the virtual DOM which enhances performance by limiting direct manipulation of the DOM and batch updating. However, it often requires additional libraries to develop more complex applications.

- **Vue.js**: Vue.js is a progressive JavaScript framework used to create user interfaces. It is designed to be incrementally adoptable, making it easy to integrate with other libraries or existing projects. Vue.js also emphasizes a flexible architecture that allows the application to scale between a library and a full-featured framework. Its simplicity and smaller size make it quicker and easier to learn than Angular or React.

We have considered Angular [‡26] as the library based on **TypeScript**:

- **Angular**: Angular, developed by Google, is a TypeScript-based open-source framework that enables the development of single-page applications. Angular provides a robust framework that supports a wide array of features like two-way data binding, dependency injection, and declarative templates. Although powerful, Angular has a steep learning curve, and its performance can be slower compared to Vue and React.

An overview table of the benefits and drawbacks of different technologies has been created to aid in thorough comprehension and to help an informed decision-making process.

Table 1.1. Advantages and disadvantages of frontend technologies

Technology	Advantages	Disadvantages
React	**Popularity:** React is one of the most popular JavaScript frameworks on the market, which means there are plenty of resources available online and plenty of experienced developers available. **Integration** with other Facebook products: React is a Facebook product, which means it is integrated with other Facebook products like GraphQL. **Customization** – React offers a great deal of freedom for developers to customize and build apps to fit their specific needs.	**Complexity** – React can be a more complex framework and can have a tougher learning curve than Vue. **Lack of features**: React is a simpler framework and focuses on views, which means that it may be missing some features that are available in other frameworks.
	Ease learning: Vue has a simple and easy to understand syntax, which means that developers can start developing applications quickly.	**Less popular**: Although Vue is a popular framework, it is still not as widely used as React or Angular. This means that there may be fewer online resources available and

Vue.js	**Performance**: Vue is a very lightweight framework and has been optimized for fast and smooth performance. **Community**: The Vue community is very active and constantly growing, which means that developers can easily find solutions to problems, tutorials, and documentation online. **Flexibility**: Vue is highly customizable and flexible, which means that developers can build applications that fit their specific needs.	fewer experienced developers available. **Less maturity**: Since Vue is a newer framework, it may not be as mature as React or Angular and may not have all the features and tools that these frameworks offer.
Angular	**Lots of features**: Angular is a very mature framework and offers a lot of features and tools for developers. **Popularity**: Angular is one of the most popular JavaScript frameworks on the market, which means there are plenty of resources available online and plenty of experienced developers available. **Integration with other Google products**: Angular is a Google product, which means it is integrated with other Google products like Firebase. Although the other frameworks can also be integrated.	**Complexity**: Angular is a very complex framework and can have a harder learning curve than Vue or React. **Performance**: Angular can be heavier and less resource efficient than Vue or React.

While libraries like React, Vue, and Angular form the structure and logic of our front-end, styling frameworks give it the polished, professional look and feel. Styling frameworks provide ready-to-use CSS components that we can use to design our user interface. Some of these frameworks are:

- **TailwindCSS** [‡27]: Is a utility-first CSS framework. Unlike other CSS frameworks that provide pre-designed components, Tailwind allows developers to create custom designs without leaving their HTML file. Tailwind doesn't come with a pre-built set of themes. You can customize your design as you wish. Since you don't have to switch back and forth between different files, using this framework can lead to faster development.

- **Bootstrap** [‡28], on the other hand, is the most popular CSS framework. It provides a series of ready-made components such as navbars, modals, or cards that you can use straight out of the box. Thus, this framework allows developers to ensure consistency regardless of who's working on the project.

There are also alternative CSS frameworks, including Semantic UI [‡29], Foundation [‡30], and Bulma [‡31]. Each has its own unique features and advantages, but they all aim to speed up the development process and make web pages look good and work well.

1.2.2.2. Final decision for client side

The appropriate technological stack must be chosen in order to meet the project's unique requirements. **Vue.js** was consequently selected as the front-end framework because of its flexible and adaptable structure. The learning curve for Vue.js is known for being low, especially when compared to rival frameworks like React and Angular. This results in quicker acceptance, simpler coding, and therefore more effective project development.

Furthermore, Vue.js has a solid reputation for adaptability, enabling it to be smoothly integrated with other libraries or ongoing projects. This adaptability is especially helpful in our case because the Web Service we are developing might need to interface with some software programs to do computations and trajectory improvements.

Moving to styling frameworks, **TailwindCSS** is our choice because it strikes a balance between customization and speed. Tailwind CSS enables the opportunity to make unique designs without leaving the HTML file, in contrast to other CSS frameworks that impose specific design decisions.

By combining Vue.js and TailwindCSS, we are setting up our project with a powerful, yet flexible technological foundation. While TailwindCSS provides the effectiveness and complete flexibility needed to build a specifically designed, responsive user experience, Vue.js offers the adaptable structure and ease-of-use we require.

Fig. 1.3 Vue.js and Tailwind CSS logos

CHAPTER 2. ANALYSIS OF REQUIREMENTS

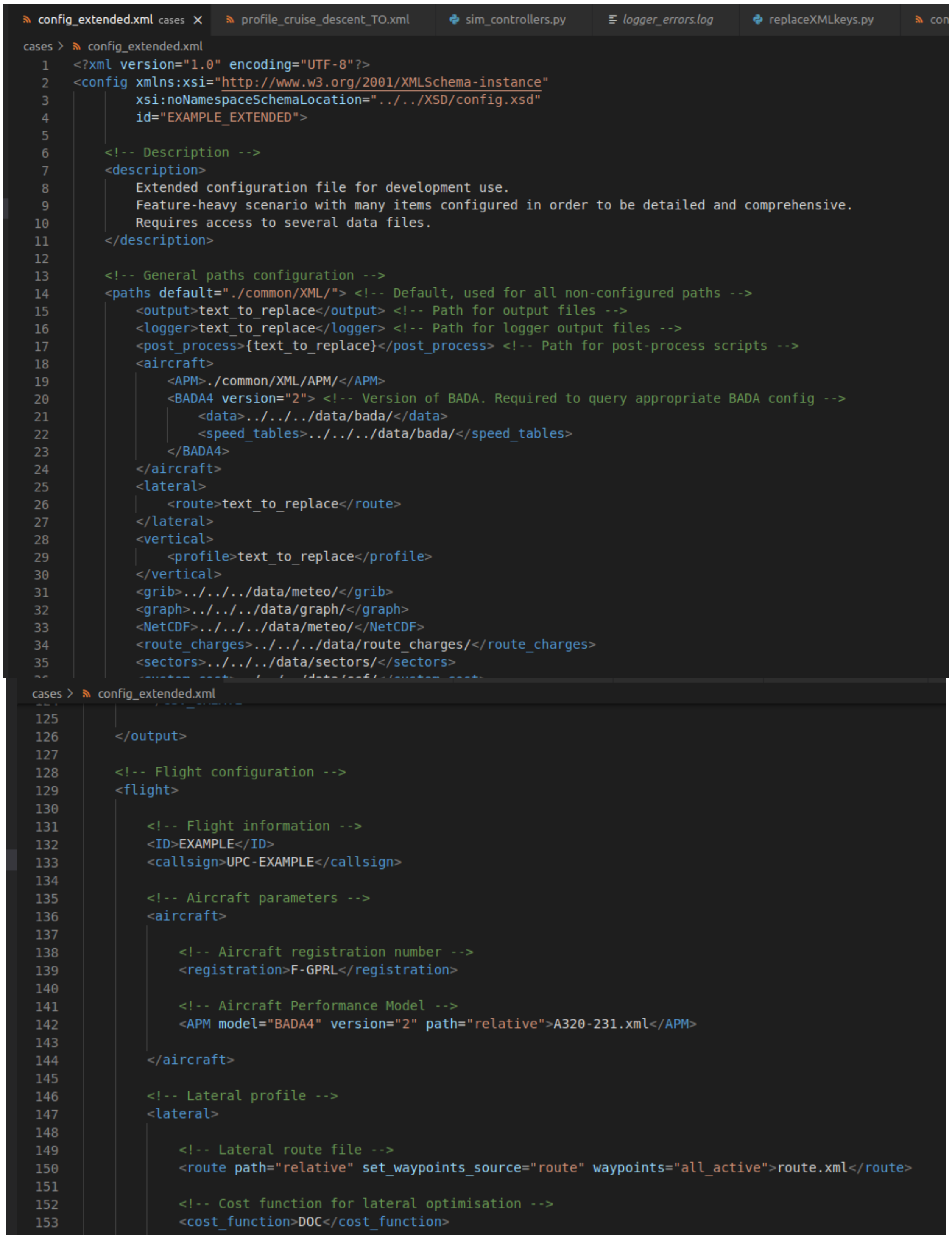
The aim of this chapter is to delve into the process of requirement gathering and analysis that was critical in shaping the Dynamo Web Services project. Understanding the requirements is fundamental to the success of any software development project. It is the foundation upon which the design and development phases rest. Therefore, thorough analysis of the project requirements ensures that the developed software effectively meets the needs it was designed for.

In this chapter, we initially explore the different methods employed for requirements gathering (Section 2.1)**,** highlighting the most important ones. Subsequently, the outcomes from the Requirements Gathering Session (Section 2.2) are presented, followed by a discussion on the emergence of new requirements as the project evolved (Section 2.2.1).
Through the course of this chapter, the readers will gain an understanding of the integral role that requirements analysis plays in software development, and how it has directly contributed to the development of the Dynamo Web Services project.

2.1. Dynamo Software

To manage configuration settings for each simulation, it uses XML files. These XML files offer a powerful yet adaptable system to handle various flight parameters. This project's focus extends to these Dynamo configuration XML files, which contain seven main blocks: **<description>**, **<output>**, **<paths>**, **<logger>**, **<flight>**, **<ATM>**, and **<weather>**. Understanding these blocks is essential as they are the foundation of our project's functionality, which involves adjusting the configuration file values to set up the simulations. Therefore, a deep understanding of how the Dynamo configuration files operate is crucial to the successful implementation of our web service.

A snapshot of the structure of these XML configuration files is given below (see figure 2.1):


config_extended.xml cases
profile_cruise_descent_TO.xml
sim_controllers.py
logger_errors.log
replaceXMLkeys.py
cases > config_extended.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="../../XSD/config.xsd"
        id="EXAMPLE_EXTENDED">

    <!-- Description -->
    <description>
        Extended configuration file for development use.
        Feature-heavy scenario with many items configured in order to be detailed and comprehensive.
        Requires access to several data files.
    </description>

    <!-- General paths configuration -->
    <paths default="./common/XML/"> <!-- Default, used for all non-configured paths -->
        <output>text_to_replace</output> <!-- Path for output files -->
        <logger>text_to_replace</logger> <!-- Path for logger output files -->
        <post_process>{text_to_replace}</post_process> <!-- Path for post-process scripts -->
        <aircraft>
            <APM>./common/XML/APM/</APM>
            <BADA4 version="2"> <!-- Version of BADA. Required to query appropriate BADA config -->
                <data>../../../data/bada/</data>
                <speed_tables>../../../data/bada/</speed_tables>
            </BADA4>
        </aircraft>
        <lateral>
            <route>text_to_replace</route>
        </lateral>
        <vertical>
            <profile>text_to_replace</profile>
        </vertical>
        <grib>../../../data/meteo/</grib>
        <graph>../../../data/graph/</graph>
        <NetCDF>../../../data/meteo/</NetCDF>
        <route_charges>../../../data/route_charges/</route_charges>
        <sectors>../../../data/sectors/</sectors>
```

```xml

    </output>

    <!-- Flight configuration -->
    <flight>

        <!-- Flight information -->
        <ID>EXAMPLE</ID>
        <callsign>UPC-EXAMPLE</callsign>

        <!-- Aircraft parameters -->
        <aircraft>

            <!-- Aircraft registration number -->
            <registration>F-GPRL</registration>

            <!-- Aircraft Performance Model -->
            <APM model="BADA4" version="2" path="relative">A320-231.xml</APM>

        </aircraft>

        <!-- Lateral profile -->
        <lateral>

            <!-- Lateral route file -->
            <route path="relative" set_waypoints_source="route" waypoints="all_active">route.xml</route>

            <!-- Cost function for lateral optimisation -->
            <cost_function>DOC</cost_function>
```

Fig. 2.1 Example of configuration XML file

- <**description**>: Contains a string that provides a brief description of the simulation setup or any other relevant information.
- <**paths**>: This section consists of different paths that Dynamo will use to find the input datafiles and write the outputs. Key elements include:
    - <output>: Path to store the output files of the simulation.
    - <logger>: Location where the logger should write its output files.
    - <aircraft>: It contains paths to two critical components:

        - <APM>: Path to Aircraft Performance Models.
        - <BADA4>: Path to Base of Aircraft Data (BADA), a comprehensive model of aircraft performance. The version attribute here specifies the version of BADA to be used.
    - Other paths specified include lateral and vertical flight plans (<route> and <profile>, respectively), meteorological data (<grib>, <NetCDF>), data for navigation (<graph>, <sectors>) and others.
- <**logger**>: This block provides options for the level of detail to be logged during the simulation. The more detailed the logger configuration, the more insights can be gained about the simulation run:
    - <log_id>: A string identifier for the logger.
    - <max_log_level>: Specifies the maximum log level (FATAL, ERROR, WARNING, INFO, LOG, DEBUG, TRACE).
    - <terminal>: Configuration for logging to the command terminal.
    - <file>: Configuration for file logging, with separate instances for different types of logs.
- <**output**>: Specifies the output files generated by the simulation:
    - Each type of output (e.g., <KML>, <FDR>, <vertical_meteo>) has specific attributes and nested elements to define the output file, whether a post-processing script should run on it, and the verbosity level of the data.
- <**flight**>: Encapsulates flight-specific parameters, such as:
    - <ID>: Unique identifier for the flight.
    - <callsign>: The callsign for the flight.
    - <aircraft>: Contains details about the aircraft used for simulation.
    - <lateral> and <vertical>: Specify the paths to the flight plan files, as well as other options for route and profile.
    - <cost>: Holds parameters used to configure the cost function for the flight.
- <**ATM**>: Details about Air Traffic Management. The nested <ASM> (AirSpace Management) block contains <lateral> and <vertical> blocks to specify air traffic management details for each flight profile phase (climb, cruise, descent).
- <**weather**>: Configures the meteorological conditions for the simulation:
    - <atmosphere>: Specifies the atmospheric model, as well as the associated data sources.
    - <winds>: Specifies the winds model, as well as the associated data sources.
    - <origin> and <destination>: Configure weather conditions (QNH, transition level) at the departure and arrival airports.

These seven blocks constitute the fundamental structure of the Dynamo configuration XML files. It's through the manipulation of these elements that users can set-up the simulations to their exact requirements, giving Dynamo its flexibility and adaptability.

Dynamo requires a set of specific inputs to operate optimally (see figure 2.2). These include aircraft performance data, which consists of mathematical models that describe the forces acting on the aircraft, such as thrust, drag, and fuel flow.

This data can be sourced from various places, like the Base of Aircraft Data (BADA) [17] or directly from flight tests. In addition to performance data, Dynamo utilizes weather data, either in the form of standard atmospheric models like the International Standard Atmosphere (ISA) [18] or real-world weather data in GRIB (GRIdded Binary or General Regularly-distributed Information in Binary form) [19] format.

Operator parameters are also a crucial part of the inputs Dynamo needs. These parameters reflect the relative importance of time and fuel costs and include the cost index, payload, and flight plan. Furthermore, Dynamo requires Concepts of Operation (ConOps) [23] inputs, both horizontal and vertical, which specify how the lateral route and speed, and altitude profiles are to be generated.

Once these inputs are processed, Dynamo's advanced algorithms generate optimized flight trajectories. It employs a lateral and vertical profile prediction/optimization module to calculate the optimal route and vertical profile. The process involves minimizing a cost function that incorporates fuel, time, and route charges.

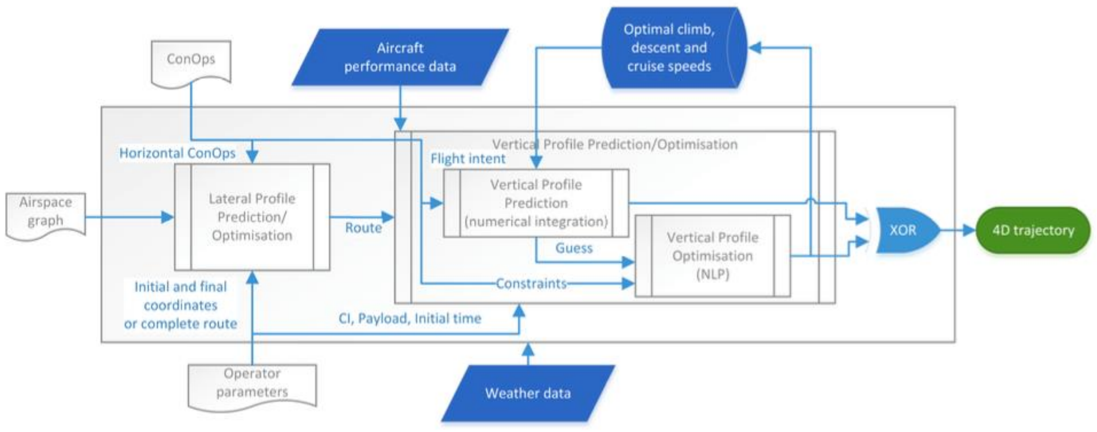

ConOps
Aircraft performance data
Optimal climb, descent and cruise speeds
Horizontal ConOps
Vertical Profile Prediction/Optimisation
Flight intent
Airspace graph
Lateral Profile Prediction/ Optimisation
Route
Vertical Profile Prediction (numerical integration)
Guess
Vertical Profile Optimisation (NLP)
Constraints
XOR
4D trajectory
Initial and final coordinates or complete route
CI, Payload, Initial time
Operator parameters
Weather data

Fig. 2.2 DYNAMO architecture [1]

Dynamo's capabilities have been demonstrated in a variety of applications and assessments. For instance, in the APACHE Project [24], it was used to create realistic traffic scenarios for various vertical and horizontal ConOps using realistic aircraft performance models and weather data. Over one million trajectories were optimized, showcasing Dynamo's scalability and computational efficiency.

In the FASTOP Project, Dynamo was used for real-time on-board optimization during Continuous Descent Operations (CDO), demonstrating its versatility and practical applicability. The software was embedded on-board a research flight management system (FMS), where it computed optimal vertical profiles while satisfying ATC time constraints and standard operational procedures. [25]

Furthermore, an integral part of Dynamo's functioning is its output generation. It provides comprehensive reports containing the predicted and optimized flight trajectories, including altitude and speed profiles, fuel usage, and other essential flight parameters. This data proves to be invaluable for further analysis and decision-making.

In conclusion, Dynamo is a sophisticated and comprehensive tool, developed to optimize and predict flight trajectories. Its ability to handle a vast range of inputs, perform complex calculations, and generate detailed outputs makes it an invaluable asset in air traffic management research.

While this document will not delve into the intricate details of Dynamo's operation, it emphasizes understanding its configuration process. Our goal is to create a web service that allows users to configure, run, and manage their simulations seamlessly and efficiently. The mechanisms of these configuration files, particularly the seven blocks mentioned above, will be discussed in detail throughout this document.

2.2. Requirements gathering methods

Gathering requirements is a crucial phase in software development, since it allows us to identify and understand customer needs to ensure that the final product meets their expectations. Some of the most used methodologies [‡32] for gathering requirements for a software development project are described below. [‡33]

- User Participation Requirements Analysis: This approach focuses on directly involving the user in the requirements gathering process to ensure that their needs are understood and considered.

- Use Case Based Requirements Analysis: This method focuses on describing how a user will interact with the software through detailed scenarios called "use cases".

- Model-Based Requirements Engineering: This methodology is based on the creation of visual models that represent the requirements and allow a better understanding and communication between team members and the client.

- Prototype-Based Requirements Analysis: This approach focuses on creating basic prototypes that illustrate how the software will work and allow the customer to visualize and validate their requirements.

- Brainstorming Requirements Gathering: This methodology is based on the idea that the generation of ideas and solutions can be improved through dialogue and group collaboration. The BRG process typically begins with a brainstorming session where the development team, customers, and stakeholders come together to discuss and brainstorm software requirements. During the session, participants are encouraged to share their ideas without judging or censoring others' suggestions. Once the ideas have been generated, the team categorizes, groups, and prioritizes them to determine the most important and urgent requirements. The team then creates a detailed plan for the development and implementation of each requirement. The BRG is an effective way to involve stakeholders and customers in the requirements capture process, which can improve understanding and satisfaction with the final product. However, it can also be an intensive process in terms of time and resources, and it can be difficult to ensure that all ideas are objectively considered and prioritized. [26]

Focusing on what large development companies use, it's hard to say for sure exactly what requirements capture methodologies are being used, as this can vary depending on the project and company in question. However, we also found another type of methodology, apart from those already described above, that is commonly used in the software development industry: Agile and Scrum methodologies.

The **Agile** [27] methodology is an iterative and adaptive approach to software development that focuses on collaboration and the continuous delivery of small functional pieces of software. The agile methodology was formally introduced in 2001 with the publication of the Agile Manifesto [‡34] and has become increasingly popular in the software development industry.

Agile methodologies are based on the following values and principles:

- Individuals and interactions over processes and tools
- Software working on exhaustive documentation.
- Collaboration with the client on contract negotiation
- Reaction to changes on follow-up to a plan.

In an agile project, the development team works in short, regular sprints, delivering small, functional increments of the software rather than waiting for a complete product before delivery. This allows for greater flexibility and enables the team to adapt to changing customer needs throughout the development process.

Some of the more popular agile methods include Scrum [28], Kanban [29], XP (Extreme Programming) [30], and Lean Software Development [31]. Each agile method has its own unique approaches and practices, but they all share the same basic philosophy of an iterative and adaptive approach to software development.

- Scrum: Scrum is a framework for the management and development of agile projects. It is based on the concept of "sprints" or work cycles of a fixed duration and on constant collaboration and communication between team members. Scrum has well-defined roles, such as Product Owner, Scrum Master, and Development Team, and uses several artifacts, such as the Product Backlog and Sprint Backlog, to organize work and ensure that it is moving in the right direction. correct.

- Kanban: Kanban is a visual management system used to optimize workflow in an agile project. Use visual dashboards that show the current status of tasks and limit the amount of work in progress to reduce wait time and increase efficiency. The development team and the project management team work together to continuously improve the process and eliminate bottlenecks.

- XP (Extreme Programming): XP is an agile software development methodology that focuses on quality and speed. It is based on several practices such as pair programming, test-driven development, continuous integration, and constant refactoring. XP's goal is to deliver high-quality software quickly and consistently.

- Lean Software Development: Lean Software Development is an agile approach based on Lean Manufacturing principles. Focuses on waste reduction and continuous process improvement. It uses techniques such as customer feedback-driven development, workflow optimization, and continuous delivery of small features to improve software quality and increase customer satisfaction.

It is important to note that these methods are not mutually exclusive and are often combined to adapt to the needs and particularities of each project. Thus, many large companies use a combination of requirement capture methodologies and tailor their approach based on the specific needs of each project. It is important to note that the choice of the appropriate methodology depends on various factors such as the type of project, the complexity of the software and the client's preferences.

For our specific case[1], we chose the Brainstorming methodology, we also presented some visual models, and, at the same time, we have been following the Agile methodology. This means, then, that we conducted meetings in which small software increments were presented, to verify that we have a good approach to the product, and if not, we can rectify it in time. Also, during the process of this project, apart from the weekly technical meetings in which progress has been made at the structure and design level, a first meeting was held with most of the team members to capture the general requirements. After some time doing and developing the work based on these requirements, a second meeting was convened with the group to evaluate the project's current status and determine which requirements had been fulfilled to date.

Finally, also comment, that apart from the weekly meet meetings, we also used Trello [‡35], which is a project and task management tool that can be used to implement agile project management methodologies, such as Scrum or Kanban. Therefore, it is not an agile method itself, but rather a tool that can be used to implement agile methodologies.

Trello allows the creation of boards to organize tasks and projects at different stages of the workflow, assign tasks to team members, set deadlines, and communicate through comments. This functionality is very useful for implementing the agile philosophy of continuous iteration and adaptation in software development. Additionally, Trello allows for integration with other

[1] See requirements gathering session section.

software tools, making it easy to automate workflows and track project progress in real time.

2.2.1. Software Requirement Specification (SRS) Document

The Software Requirement Specification (SRS) Document [32] is an indispensable tool in the field of software engineering, acting as a blueprint for the software to be developed. It is a detailed, comprehensive description of the intended purpose, functionality, performance, and interfaces of a software system. Often considered the cornerstone of software documentation, it is primarily used to ensure that the software developer and the customer are on the same page regarding the software's characteristics, functions, and constraints.

The concept of the SRS document can be traced back to the structured analysis movement of the late 1970s and early 1980s, particularly drawing from the work of academics and practitioners such as Edsger W. Dijkstra, Tom DeMarco, and Larry Constantine, who advocated for a more systematic approach to software development. Over time, the SRS document has become a standard practice in the industry and is often seen as an essential part of software development, irrespective of methodology. [33]

The SRS document typically consists of several key sections:

1. **Introduction:** An overview of the software's context, including a brief description, product scope, and its intended audience and use.

2. **Functional Requirements:** Detailed description of system services, capabilities, and actions the software must perform under specific situations.

3. **External Interface Requirements:** This section outlines the software, hardware, user, and communication interfaces requirements.

4. **Non-Functional Requirements:** Covers performance, design constraints, standards compliance, security, and other quality attributes that do not pertain to specific functionalities.

5. **Use Cases:** Detailed accounts of the system’s interactions with users or other systems.

For this project, the SRS document acted as a crucial guide in the design, development, and testing of the Dynamo Web Services software. By articulating the functional and non-functional requirements, external interfaces, and use cases, we were able to capture and understand the intended behavior of the web service. This helped shape our approach to ensure alignment with user needs and expectations. Thus, we used the principles of this document to base the questions and guide the requirements gathering session, as you will see in the next section.

The SRS document for Dynamo Web Services, structured and organized according to the IEEE's recommended practice for Software Requirements Specifications (IEEE Std 830-1998) [34], is available in the annex of this report.

For a more comprehensive understanding of the SRS document and its application in software development, readers can refer to Karl E. Wiegers's book "Software Requirements" (Microsoft Press, 2003) [35], and the IEEE's "Guide to the Software Engineering Body of Knowledge (SWEBOK)" [36].

2.3. Requirements gathering session

To accurately grasp the requirements of the product, specifically for "Dynamo Web Services," a group session was conducted, unfolding in distinct parts. This section presents how the session was prepared and the results we obtained from it:

- **Group Interview**: This comprises a series of inquiries directed at the “customer” group with the aim to straightforwardly clarify some of the basic requirements and functions that will form part of the project. Questions were formulated with the intention of addressing various points of the *Software Requirement Specification Document* (see Section 2.1.1). This document seeks to clarify what the primary objectives and benefits that customers will gain from using the website, to whom the product will be targeted, and to briefly describe what will be the main features and functions it will fulfill. Some of the questions asked were:

    - Who is the intended customer base for the product?
    - What benefits will these customers gain by using Dynamo Web?
    - What kind of permissions would you like to grant to users?
    - What purchase options would you like to provide?
    - Do you want to include a section functioning as a tutorial/demo?
    - Would you like to upload an example? (Such as turbofan type)
    - Will the user be able to upload their own xml file?

- **Brainstorming**: With this segment, the aim was to delve deeper into the requirements that might not have been fully accounted for or clearly defined during the initial interview. It was a dynamic activity in which the entire group participated, using a whiteboard and post-it notes of various colors depending on the topic under consideration. The different themes or "chapters" into which this activity was divided were user and permissions management, simulations management, inputs and visualization, outputs and visualization, and documentation/tutorials.

    - User and permissions management: This involved functionalities related to users and their permissions, such as the ability to create a profile, edit it, delete the account, and purchase "premium" functionalities (this is where user permissions came into play, determining which functionalities would require payment or constitute a Premium user), etc.

- Simulation management: This concerned functionalities such as creating a new simulation, the ability to consult the results of a previous simulation, etc.
- Inputs and visualization: This involved ideas for different ways of introducing inputs, for example via forms, directly uploading an xml file, etc.
- Outputs and visualization: This related to functions concerning the results and their display, for instance, being able to decide if you only wanted a specific result, not all the graphs at once.
- Documentation/tutorials: This involved proposing ideas and deciding whether to include a section on the website dedicated to providing a brief tutorial on how it works, or information sections of some kind.

Participants were, firstly thought, divided into pairs, each of which was given a set amount of time (5-10 minutes) to write down any functionalities they could think of related to the specific topic, before moving on to the next block. This process continued until all pairs had cycled through all the blocks. Figures 2.1-2.3 present an illustrative example of this process:

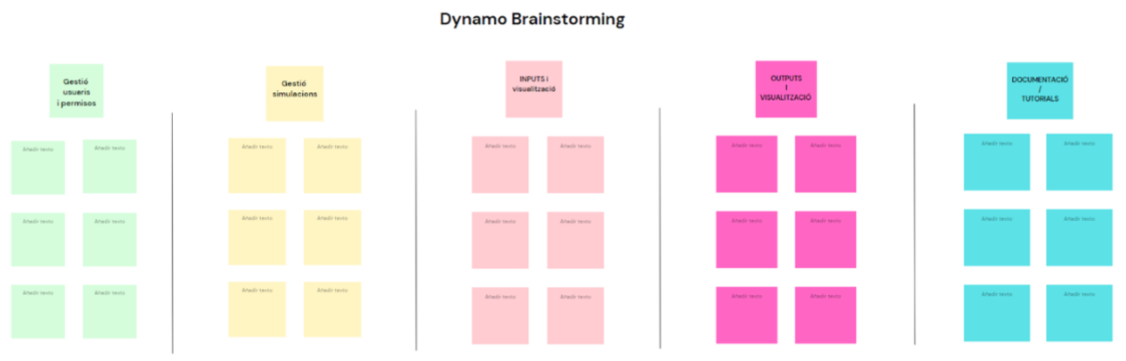


Fig. 2.3. Whiteboard with the different sections

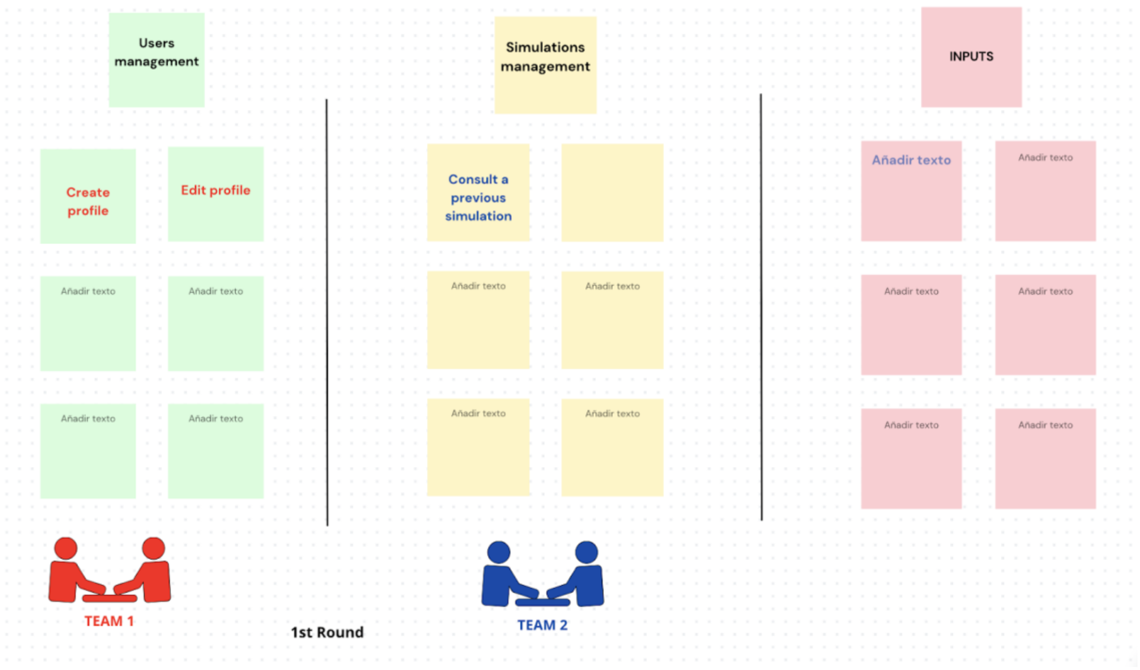


Fig. 2.4. 1st Round example

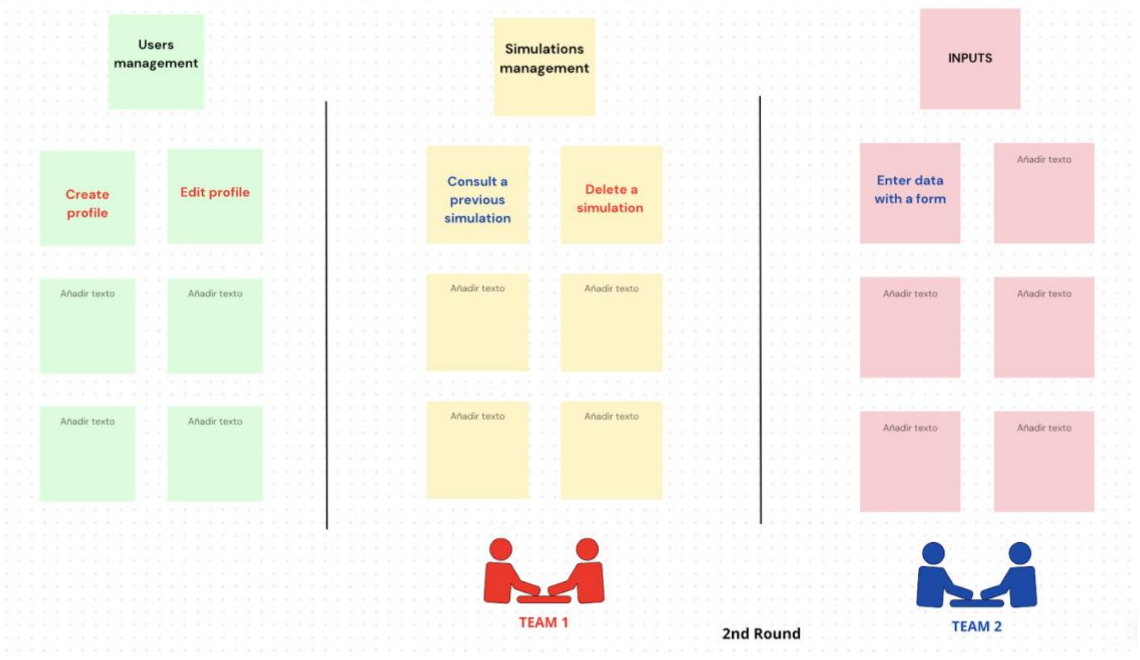
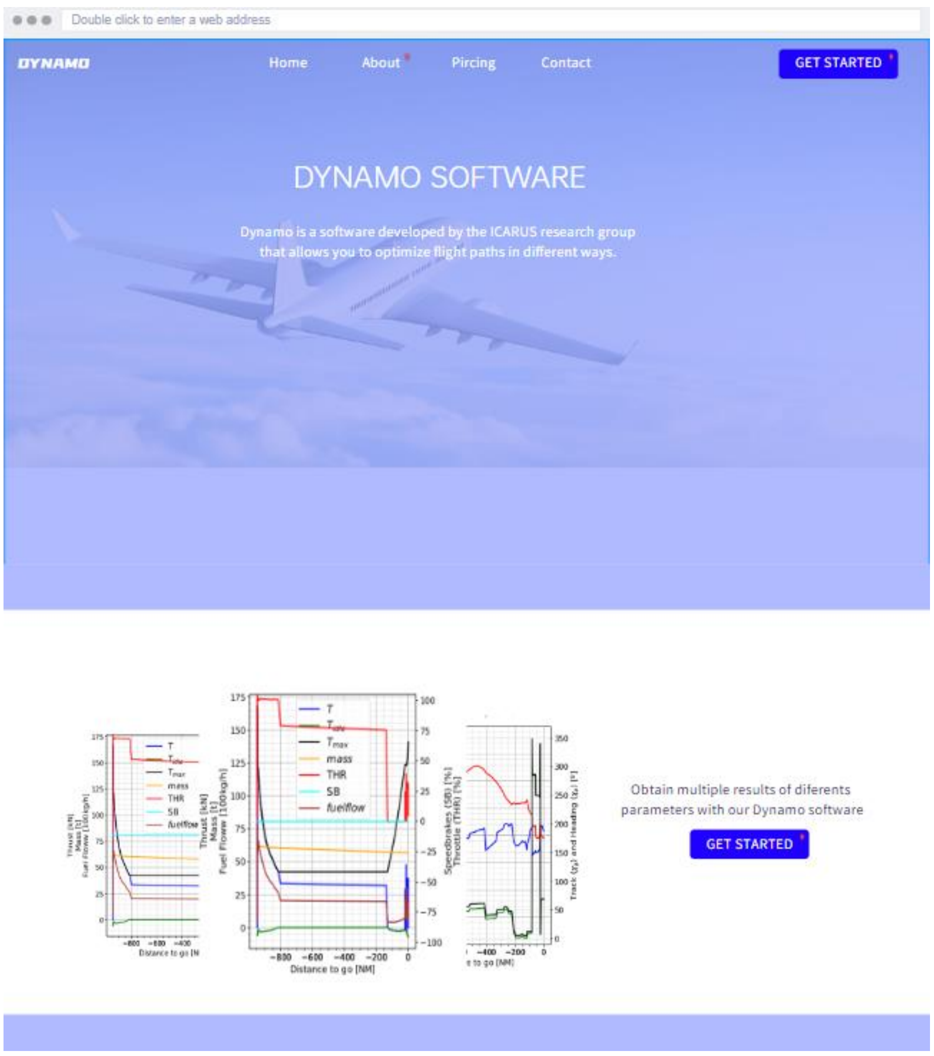


Fig. 2.5. 2nd Round example

The objective of this activity was to enable a brainstorming session among all participants, potentially giving rise to functionalities or needs that may not have occurred to us individually. It was about seeing from different perspectives how the user would like to interact with the website and how the clients/owners of the website wanted it organized and what functionalities they wished to offer.

After all the post-it note sections were filled out, the group moved on to the next phase. This consisted of a group discussion aimed at prioritizing the most important functionalities or characteristics. Consequently, the post-it notes were reorganized from most to least important.

- **Mockup Presentation**: With the aim of visually representing the functionalities mentioned in the previous sections, we showcased mockups of what the organization of the Dynamo website could potentially look like (see figures 2.6 -2.7). Additionally, if necessary, we utilized a whiteboard to create small diagrams of the desired web organization, in terms of both results visualization and all other aspects (user profile, input introduction, visualization of previous simulations, etc.).



Dynamo is a software developed by the ICARUS research group that allows you to optimize flight paths in different ways.

Obtain multiple results of diferents parameters with our Dynamo software

Fig. 2.6. Mockup 1

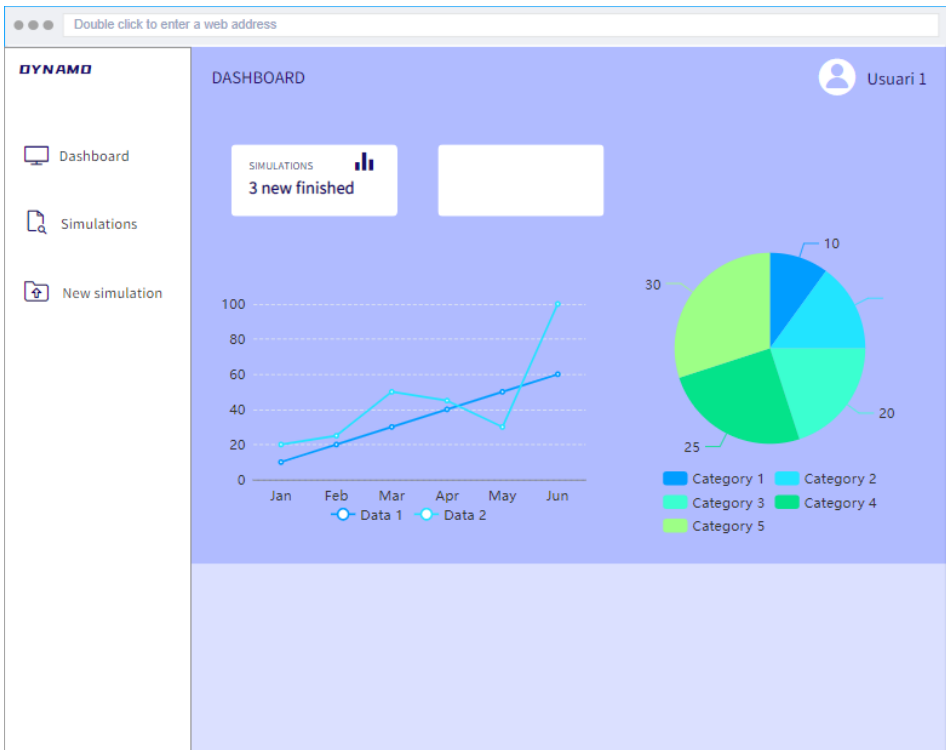


Fig. 2.7. Mockup 2

2.3.1. Results of the session

Following the session, we were able to compile a comprehensive list of various requirements that will be considered and prioritized based on their importance. There were also certain topics that needed further discussion as they were not completely clarified, such as the decision on whether there would be premium features, that is, those functions for which one would need to pay or subscribe.

2.3.1.1. Initial interview

Regarding the initial interview, we garnered several key insights:

- There will be free functions.
- The primary language of the website will be English, although the possibility of translating it into other languages, such as Catalan, was also considered.
- All users will need to register.
- There will be different types of licenses, for example:
  - Providing access for only 200 trajectories.
  - Providing access only for simulations with a specific airplane model.
  - Licenses with a start and end date.
- Help tools, a Help section, or features like hovering the cursor over the data input window to provide basic information will be incorporated.

- There might be some demonstrations through videos.
- When entering the website to create a new simulation, some default example data will be selected.
- There will be basic and advanced users.
- Notifications will be sent to users when their simulations are complete (for example, an email to the user). This option is considered especially for batch processing or when a user has simulated X number of trajectories. Users will be able to configure notifications and decide when to receive them.
- A contact section will be included.
- Users can access previous simulations and, additionally, if they have not finished configuring something, it can be saved so they can complete it when they log back in. This feature does not apply to free users.
- WARNINGS: These would serve to double-check user decisions, for example: "Are you sure you want to conduct this simulation with this airplane?" before executing the simulation.

During the interview/discussion, various themes and requirements emerged, which we later sorted into sections and clarified some ideas during the subsequent part (the brainstorming with post-it notes). Everyone was able to post their personal ideas on the whiteboard, which we then collectively discussed.

2.3.1.2. Brainstorming

Continuing with the session, regarding the brainstorming and the part of the post it (see figure 2.8), these are the requirements that we collected:

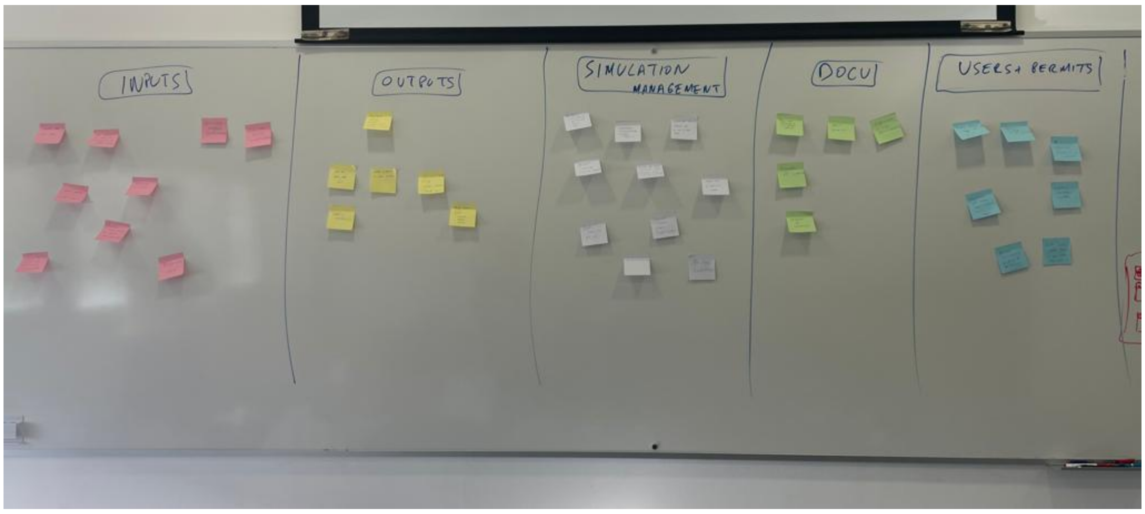


Fig. 2.8. Whiteboard with post its during brainstorming

Inputs:

- A button at the top to switch to META mode (batch simulation mode) -> Individual simulation, batch simulation.This means that at the top of the page, users have the option to choose whether they want to perform an individual simulation, a batch simulation, or an all.ft simulation. Depending on the chosen option, different features will be displayed on the screen.
- Advanced and Simple User: Advanced or simpler settings. Advanced options are not initially displayed; they can be found on a separate tab or by clicking a button.
- Option to upload XML directly (advanced).
- When selecting an airplane model, a picture and relevant information should be displayed.
- Upload any file (weather data, etc.).
- Allow graphs and weather data from different geographic areas.
- Responsive web: adaptable to smartphones, iPads.
- Edit Route.
- Weather files, pre-loaded.

Outputs:

- Select the desired output. Choose which results we want to obtain.
- Route with Google Earth, API: on the same website with tabs.
- Include the usual graphs. (Option to export data)
- Interactive graphs.
- Interactive graphs that can be moved.
- Map with layers for drawing airways, weather data, sectors.

Simulations:

- Modes: dispatch, in-flight (you must enter the airplane's weight), optimize, TP (depending on the choice, different inputs are required).
- META mode (list of flights) [configure warnings, e.g., send me an email every 10k].
- Visual management, errors, fatal, info. Graphically see which errors have occurred.
- Save simulations.
- List of inputs - "MICRO BATCH". Ability to send a list of inputs to simulate different simulations at once.
- View the status of simulations.
- Notifications and email when finished.
- Folder explorer with each user's info
- DEBUG MODE depending on the user (to choose the logger level).

Documentation

- Completed and explained examples: optimization example, other examples (to have a default one).
- Contact section.
- Credits section.
- HELP section manual.
- Educational explanations / potential student users -> have a section on how things are calculated.
- Feedback, bugs, new feature request.

Users

- All users register and login.
- Permissions to see logger level.
- Activate features according to licenses: batch mode, etc.
- Licenses linked to the use of modules: only being able to make 100 trajectories or only being able to use a type of airplane.
- Educational licenses, from one date to another.
- User folder management.
- Certain users can upload their models: APM, weather data (Advanced).
- Video or demo before registration.
- Register and get a free month.

Following the above list of requirements, it's crucial to note that this does not imply that all these ideas will be implemented as part of the final product. This session was an open forum for brainstorming, where everyone could share their ideas to provide a comprehensive vision of possible features. The goal was not to commit to every proposal, but to stimulate creative thinking and explore a broad range of possibilities.

It was expected that some of these initial ideas will change over time, reflecting the dynamic nature of product development. This reflective process was made explicit during a second session that we held with the entire Dynamo team several months later.

This follow-up session served as a critical milestone in our project timeline, providing us with a clear sense of our accomplishments and highlighting the objectives that still needed to be addressed. Interestingly, this session also sparked the emergence of new objectives that were not identified in the initial brainstorming.

Furthermore, it's worth mentioning that our weekly technical meetings have been instrumental in this ongoing development process. They have not only allowed for continuous feedback but also provided an opportunity to bring up new requirements that were not initially considered. Conversely, these discussions have also led to the reevaluation and sometimes dismissal of some of the initial requirements.

Besides, a dedicated section detailing our results will be provided in the Results chapter. This section will specifically enumerate the requirements that have ultimately been met, serving as evidence of what we have achieved throughout our project development.

2.3.2. Emergence of new requirements

Over the course of a software development project, it is typical for new requirements to emerge as the project evolves, and the team becomes more familiar with the user needs and the capabilities of the software. This is a natural part of the Agile development process, which encourages the ongoing refinement of requirements based on feedback and insights gained throughout the project.

After several months into the development of Dynamo Web Services, and following many weekly technical meetings, the entire team convened for a comprehensive review session. This meeting was designed to assess the progress made, review the implemented requirements, and discuss any necessary adjustments to the project's direction. A demonstration of the current system was conducted to provide a tangible understanding of its functionalities and interface, and it was during this demo when new requirements started to surface. These were essentially enhancements and modifications that were not originally anticipated but were recognized as valuable additions to improve the system's usability, efficiency, and overall user experience.

The newly identified requirements that emerged from this interactive session were as follows:

1. **Stop Simulation Button:** The user's need for control over the simulation process was underscored with the requirement of a function to interrupt a running simulation. The ability to halt the computational operation at any time gives the user additional flexibility and authority over the process, enhancing the interactive nature of the system.
2. **Asynchronous Simulation Execution:** Another pivotal requirement that emerged was the capability for asynchronous execution of simulations. It became evident that such a feature would significantly augment the system's computational efficiency and would allow for better utilization of resources. It was suggested that the system should support running up to five simulations simultaneously, a number arrived at considering the trade-off between computational load and performance.
3. **Dynamo Execution Option:** An interactive element for choosing the desired execution option for Dynamo - c3po or r2d2 - was identified as a necessary feature. By providing this choice through a simple user interface component like a radio button, users can customize their simulations according to their specific needs without wrestling with complicated configurations.
4. **Mandatory Form Fields:** To enhance the robustness of the simulation configuration process, it was suggested that all form fields should be marked as required. This feature ensures that simulations are only initiated

when all necessary information is provided, thereby avoiding potential errors or inconsistencies due to incomplete or incorrect data.

5. **Email Notifications:** The final new requirement identified during this session was the ability for the system to send out email notifications upon completion of simulations. This feature, intended to enhance user convenience, would allow users to be notified as soon as their simulations are ready, eliminating the need to continuously check the system for results. Users could choose to enable or disable these notifications as per their preference via a checkbox in the application.

These emergent requirements underscored the importance of regular review sessions, as they provided an opportunity to reassess the project's direction based on practical, hands-on experience with the system. In response to these newly identified needs, the project team had to adapt their approach and incorporate these additional features into the development process.

As a closing remark, it is important to note that the aforementioned points represent proposals for new requirements and do not mean that the final result of the project has fulfilled 100% of these proposed requirements. The extent to which these proposed requirements have been met will be evaluated throughout the course of this document and finally assessed in the Results chapter.

CHAPTER 3. BACKEND

This chapter focuses on the design and implementation of the backend architecture for Dynamo. The backends' primary role is to manage simulation creation and configurations, handle user interactions, and maintain a specific file structure for storage and retrieval of simulations.

3.1. Backend Design

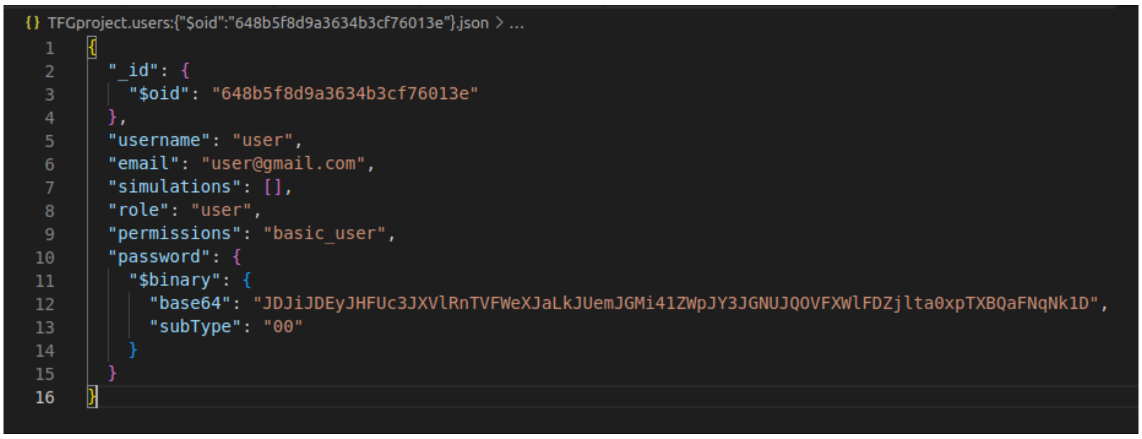
The design of the backend is primarily focused on effectively managing users and simulations. These two integral aspects of the system function in tandem to provide the necessary services to the Dynamo tool.

3.1.1. User Management System

The User Management System is designed around a specific structure in the MongoDB database (see figure 3.1). For each user, the following details are stored:

- **User ID**: A unique identifier automatically generated for each user.
- **Username**: The chosen username of the user, a unique field that differentiates each user in the system.
- **Email**: The user's email address for potential communication and notification purposes.
- **Simulations**: An array storing the unique identifiers of the simulations associated with the user, aiding in efficient retrieval and linkage of user-specific simulations.
- **Role**: Indicates the user's role within the system. This could be 'user' for a regular user or 'admin' for an administrator who has additional privileges, such as managing other users.
- **Permissions**: Defines the user's system permissions, which could be 'basic_user' or 'advanced_user'. A basic user has access to the standard functionality of the system, while an advanced user enjoys additional features such as uploading their own XML configuration files.
- **Password**: The user's password, securely hashed and stored for ensuring privacy and security.

The structure of the User Management System allows the application to handle all user-related actions efficiently and maintains the integrity and security of the user data.



{} TFGproject.users:{"$oid":"648b5f8d9a3634b3cf76013e"}.json > ...

```
{
  "_id": {
    "$oid": "648b5f8d9a3634b3cf76013e"
  },
  "username": "user",
  "email": "user@gmail.com",
  "simulations": [],
  "role": "user",
  "permissions": "basic_user",
  "password": {
    "$binary": {
      "base64": "JDJiJDEyJHFUc3JXVlRnTVFWeXJaLkJUemJGMi41ZWpJY3JGNUJQOVFXWlFDZjlta0xpTXBQaFNqNk1D",
      "subType": "00"
    }
  }
}
```

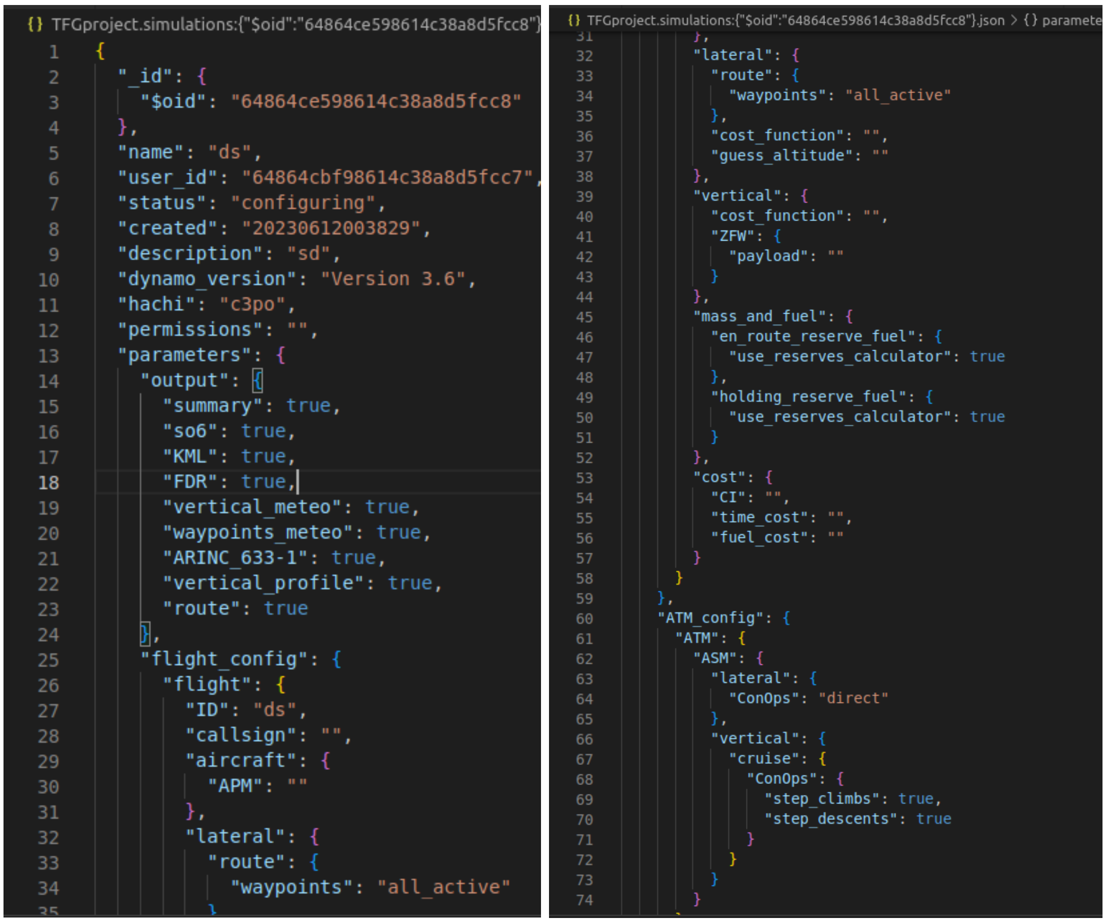
Fig. 3.1. A user on the database

3.1.2. Simulation Management System

In parallel to user management, the Simulation Management System plays a vital role in handling the core functionality of the Dynamo tool. Each simulation is represented in the MongoDB database with the following structure (see figure 3.2):

- Simulation ID: A unique identifier assigned to each simulation.
- Name: The name assigned to the simulation by the user.
- User ID: The unique identifier of the user who initiated the simulation, linking the simulation to the specific user.
- Status: Indicates the current status of the simulation, which can be 'configuring', 'running', 'completed', etc.
- Creation Date: The timestamp indicating when the simulation was created.
- Description: A brief user-generated description of the simulation.
- Dynamo Version: The version of Dynamo used for the simulation.
- Hachi: The identifier for the specific version of Dynamo's internal solver used in the simulation.
- Permissions: Any permissions associated with the simulation.
- Parameters: Various parameters essential for the configuration and execution of the simulation.

The parameters include specific settings and options required by Dynamo for accurately configuring and running the simulation. These parameters define output preferences, flight and ATM configurations, weather, and route configurations, as well as initial conditions. These are captured from user inputs and are used to generate the necessary XML configuration files that Dynamo requires to perform its calculations and optimizations.


{} TFGproject.simulations:{"$oid":"64864ce598614c38a8d5fcc8"}

```
{
  "_id": {
    "$oid": "64864ce598614c38a8d5fcc8"
  },
  "name": "ds",
  "user_id": "64864cbf98614c38a8d5fcc7",
  "status": "configuring",
  "created": "20230612003829",
  "description": "sd",
  "dynamo_version": "Version 3.6",
  "hachi": "c3po",
  "permissions": "",
  "parameters": {
    "output": {
      "summary": true,
      "so6": true,
      "KML": true,
      "FDR": true,
      "vertical_meteo": true,
      "waypoints_meteo": true,
      "ARINC_633-1": true,
      "vertical_profile": true,
      "route": true
    },
    "flight_config": {
      "flight": {
        "ID": "ds",
        "callsign": "",
        "aircraft": {
          "APM": ""
        },
        "lateral": {
          "route": {
            "waypoints": "all_active"
```

{} TFGproject.simulations:{"$oid":"64864ce598614c38a8d5fcc8"}.json > {} paramete

```
        "lateral": {
          "route": {
            "waypoints": "all_active"
          },
          "cost_function": "",
          "guess_altitude": ""
        },
        "vertical": {
          "cost_function": "",
          "ZFW": {
            "payload": ""
          }
        },
        "mass_and_fuel": {
          "en_route_reserve_fuel": {
            "use_reserves_calculator": true
          },
          "holding_reserve_fuel": {
            "use_reserves_calculator": true
          }
        },
        "cost": {
          "CI": "",
          "time_cost": "",
          "fuel_cost": ""
        }
      }
    },
    "ATM_config": {
      "ATM": {
        "ASM": {
          "lateral": {
            "ConOps": "direct"
          },
          "vertical": {
            "cruise": {
              "ConOps": {
                "step_climbs": true,
                "step_descents": true
              }
            }
          }
        }
```

Fig. 3.2. A simulation on the database

3.1.3. File Management System

Our project relies on a well-organized directory structure to manage simulations, user data, code, configuration files, and necessary utilities for the Dynamo software (see figure 3.3). The structure of directories and subdirectories is of utmost importance for efficient data management and easy access to necessary files. The following provides a detailed description of this hierarchy:

The root directory of our project is called 'web-icarus'. Within web-icarus, there are several subdirectories designed to separate different types of data and functionality.

1. The **'admin'** directory: Currently, this directory is reserved for future functionalities. Its primary purpose is to store files or utilities that may be useful for administrative tasks.

2. The **'cases'** directory: This directory is used to store and manage the simulations created by the users. For each user, a unique directory labeled as 'USER_{username}' is created. Within this user-specific directory, individual simulation directories are created with the naming convention

'{creation_date}_{simulation_id}'. Each of these simulation directories further contain three subdirectories:
  - 'config': This subdirectory stores the configuration file for each specific simulation. The user will modify this file to set up the simulation as per their requirements.
  - 'log': This is where Dynamo's log files generated during a simulation run are stored.
  - 'output': Upon completion of a simulation, Dynamo generates output files that are stored here.

This 'cases' folder also contains the config xml template files that will be used to create the simulations of these users.

3. The **'code'** directory: This directory houses the code repositories for both the frontend and backend of the application, namely 'kokyaku' for frontend and 'oshiri' for the backend.

4. The **'config'** directory: This directory stores the configuration files for the backend, frontend, and database. These include details like paths and ports for connections.

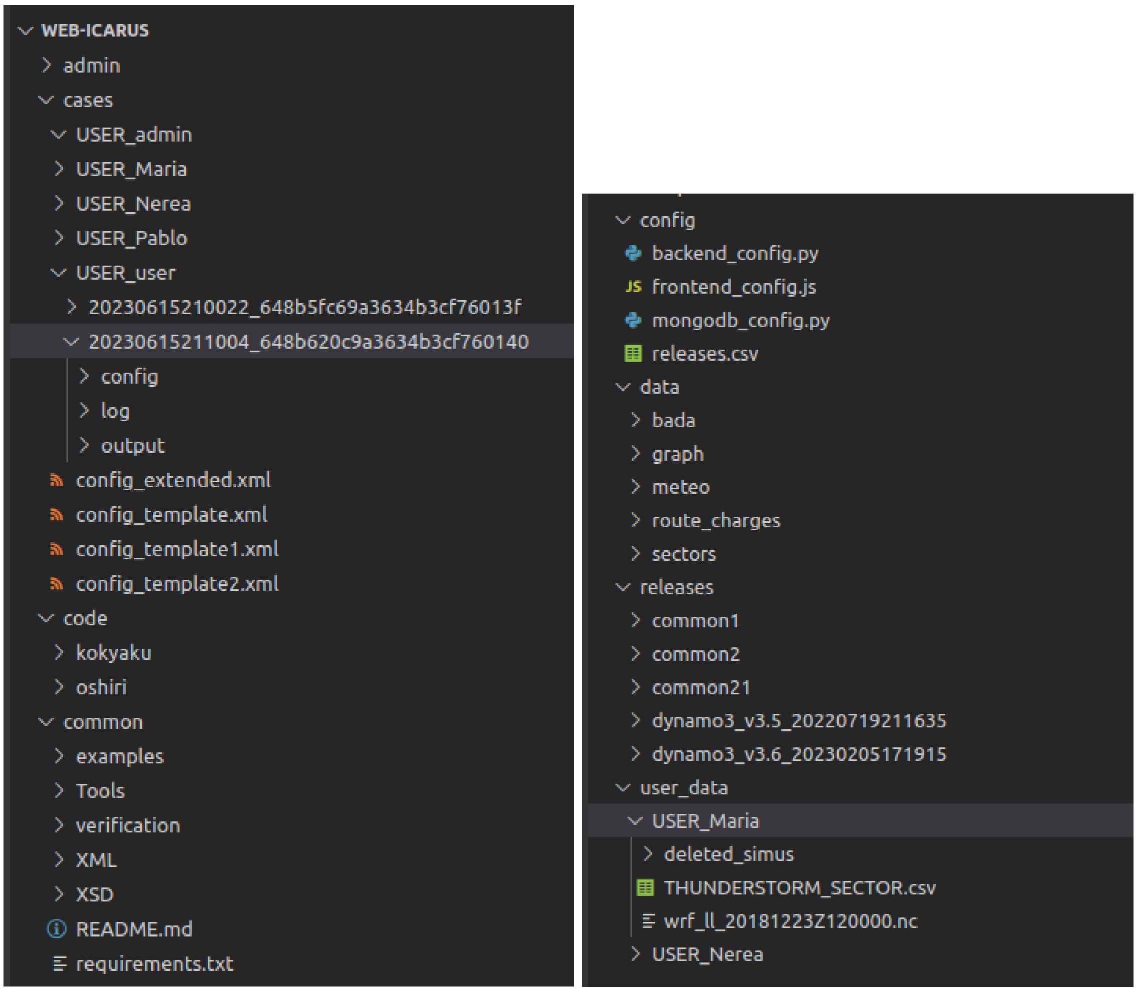
5. The **'data'** directory: This directory is used to store data files necessary for Dynamo's simulations. These can include meteorology files, aircraft APM files, among others.

6. The **'releases'** directory: This directory stores the release files for both Dynamo and additional utilities for running Dynamo, located in 'common'.

7. Finally, the **'user_data'** directory: This directory is specifically for storing files uploaded by the users and moving the deleted simulations of the users.

In summary, our directory structure is designed to compartmentalize different types of data and files required for user simulations and the functioning of our application, while ensuring easy access and efficient management.

The subsequent sections will dive deeper into how these design decisions were realized into a fully functional backend system. They will explore the practical application of the design, discussing how the chosen technologies were employed to create an efficient backend system.



```
WEB-ICARUS
  > admin
  v cases
    v USER_admin
    > USER_Maria
    > USER_Nerea
    > USER_Pablo
    v USER_user
      > 20230615210022_648b5fc69a3634b3cf76013f
      v 20230615211004_648b620c9a3634b3cf760140
        > config
        > log
        > output
    config_extended.xml
    config_template.xml
    config_template1.xml
    config_template2.xml
  v code
    > kokyaku
    > oshiri
  v common
    > examples
    > Tools
    > verification
    > XML
    > XSD
    README.md
    requirements.txt
```

```
  v config
    backend_config.py
    frontend_config.js
    mongodb_config.py
    releases.csv
  v data
    > bada
    > graph
    > meteo
    > route_charges
    > sectors
  v releases
    > common1
    > common2
    > common21
    > dynamo3_v3.5_20220719211635
    > dynamo3_v3.6_20230205171915
  v user_data
    v USER_Maria
      > deleted_simus
      THUNDERSTORM_SECTOR.csv
      wrf_ll_20181223Z120000.nc
    > USER_Nerea
```

Fig. 3.3. Web Service directory structure

3.2. Backend Implementation

The backend implementation of our project plays a pivotal role in ensuring efficient management of user data and simulations, smooth communication between the frontend and the database, and proper functioning of the Dynamo software. The backend of our system was developed using Python, leveraging the capabilities of Flask as a web framework and PyMongo as a MongoDB driver.

In essence, our backend serves as a bridge that connects the user interface with the MongoDB database and the Dynamo software. It processes requests from the frontend, retrieves or modifies data in the database as needed, and manages the simulations based on user actions.

Throughout this chapter, we will delve into the specifics of how this backend system has been built and set into operation. We will break down the implementation process into several subsections that will cover two primary components of the backend: the User Management System and the Simulation Management System. For each of these, we will discuss how we set up the database models, the endpoint design, and the interaction with the file system and Dynamo software.

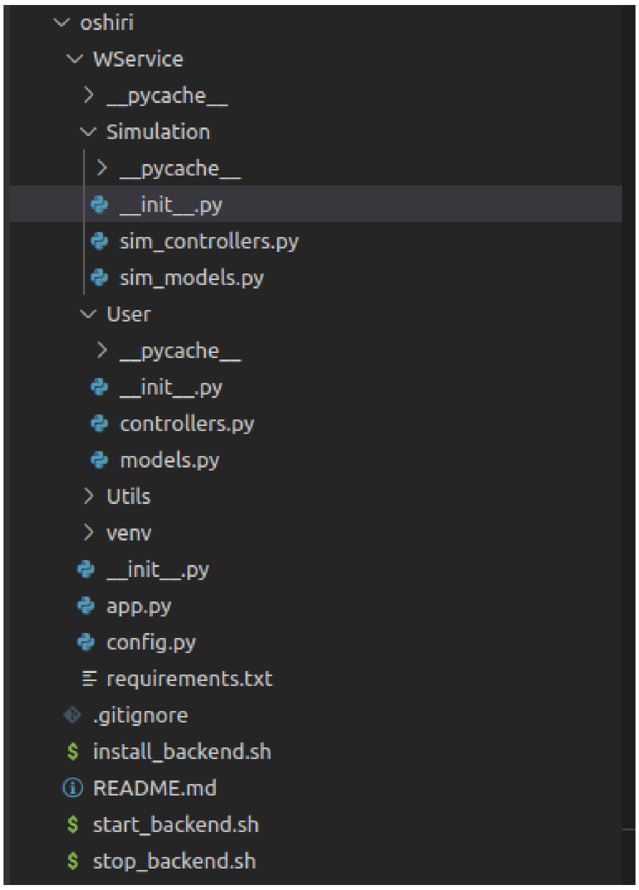
It is important to note that the implementation of the backend goes hand in hand with the design that we've discussed in the previous chapter. We will see the practical realization of the design considerations and get to understand how each element plays an important role in our system.

3.2.1. Code structure

The backend of our project, residing in the **WService** folder, is divided into three main directories: **Users**, **Simulations**, **Utils**, and config.py and app.py files (see figure 3.4). In this section, we will delve into an overview of the content and functionality of each directory and file.

The WService directory is the core of our backend, where all the main functionalities reside. This directory contains two main subdirectories, simulation, and users, and two important Python scripts, app.py and config.py.

- **Users**: This directory is dedicated to handling user-specific operations. It contains two Python files: controllers.py and models.py.
    - **controllers.py** is responsible for handling HTTP requests concerning users. This includes creating new users, checking user existence, saving user details to the database, and updating user permissions.
    - **models.py** is where the User class is defined, including its initialization, methods for hashing passwords, converting users to dictionaries, and saving them to the MongoDB database.
- **Simulations**: This directory would be analogous to the user's directory but oriented towards handling simulation-specific operations.
- **Utils**: This directory is where we keep helper functions like *assignCommonRelease*, *releasesSearcher*, which help assign the correct dynamo release with its corresponding common release, and *replaceXMLkeys*, which helps modify XML files.
- **app.py**: This file is where our Flask application is set up. It imports necessary modules, enables CORS (Cross-Origin Resource Sharing), sets up the Flask-RESTX API with our users and simulations namespaces, configures the JWT (JSON Web Tokens) extension, and sets up the secret keys necessary for session encryption and token verification.
- **config.py**: Contains the configuration settings for the Flask application. This includes secret keys needed for the app, as well as the MongoDB URI and database name. The secret keys are used to sign and verify JSON Web Tokens (JWT), encrypt session data, and other sensitive information. The MongoDB URI and database name are used to connect to the MongoDB database where our user and simulation data are stored.



```
oshiri
  WService
    __pycache__
    Simulation
      __pycache__
      __init__.py
      sim_controllers.py
      sim_models.py
    User
      __pycache__
      __init__.py
      controllers.py
      models.py
    Utils
    venv
    __init__.py
    app.py
    config.py
    requirements.txt
  .gitignore
  install_backend.sh
  README.md
  start_backend.sh
  stop_backend.sh
```

Fig. 3.4. Backend code directory structure

You can also notice that three scripts have been added (**install_backend.sh**, **start_backend.sh**, **stop_backend.sh**). These scripts are used to install all dependencies and execute the project.

In the next sections, we will go into more detail about how these files and directories work together to implement the backend functionality of our project (see figure 3.5).

3.2.1.1. Setting up and Configuring Flask

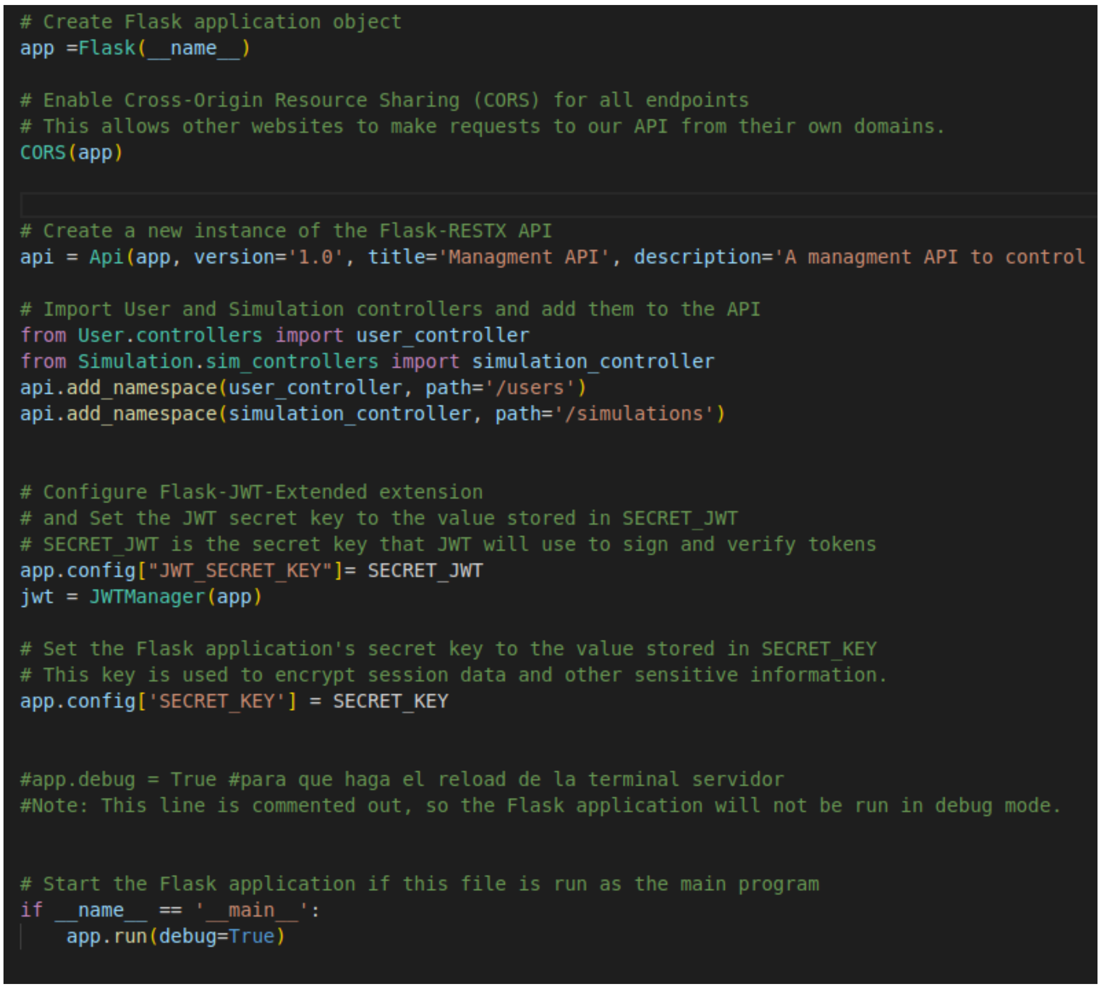
The **app.py** file serves as the entry point for the Flask application and is responsible for setting up and coordinating the main elements of the project. Here is an in-depth walkthrough:

These are the libraries necessary for the functionality of the Flask application.

- **flask**: This is the main Flask library that provides the core framework for building web applications in Python. Here it's used to create the application.
- **flask_restx**: This Flask extension is used for building RESTful APIs, it includes tools for easy creation of routes, parsing arguments and generating API documentation.

- **config**: This file (usually a config.py at the project's root) contains sensitive data such as secret keys and database URI. The keys SECRET_KEY and SECRET_JWT are imported, which will be used later for configuring the Flask application and the JWT extension.
- **flask_cors**: This stands for "Cross-Origin Resource Sharing" and allows requests to be made from different domains. This is necessary in many modern web applications where the front-end and the back end are hosted on different servers or domains.
- **flask_jwt_extended**: This library is used to secure routes and handle user authentication and authorization using JSON Web Tokens (JWT).

After importing the needed libraries our code continue as follows:



```
# Create Flask application object
app =Flask(__name__)

# Enable Cross-Origin Resource Sharing (CORS) for all endpoints
# This allows other websites to make requests to our API from their own domains.
CORS(app)


# Create a new instance of the Flask-RESTX API
api = Api(app, version='1.0', title='Managment API', description='A managment API to control

# Import User and Simulation controllers and add them to the API
from User.controllers import user_controller
from Simulation.sim_controllers import simulation_controller
api.add_namespace(user_controller, path='/users')
api.add_namespace(simulation_controller, path='/simulations')


# Configure Flask-JWT-Extended extension
# and Set the JWT secret key to the value stored in SECRET_JWT
# SECRET_JWT is the secret key that JWT will use to sign and verify tokens
app.config["JWT_SECRET_KEY"]= SECRET_JWT
jwt = JWTManager(app)

# Set the Flask application's secret key to the value stored in SECRET_KEY
# This key is used to encrypt session data and other sensitive information.
app.config['SECRET_KEY'] = SECRET_KEY


#app.debug = True #para que haga el reload de la terminal servidor
#Note: This line is commented out, so the Flask application will not be run in debug mode.


# Start the Flask application if this file is run as the main program
if __name__ == '__main__':
    app.run(debug=True)
```

Fig. 3.5. app.py file

The **Flask** class is instantiated with the name of the module (__name__) passed as the argument. This object, **app**, is the core of the Flask application. The **CORS** function is used to apply CORS to the application. This allows requests from any domain, enhancing the flexibility and compatibility of your API.

It is also setting up Flask-restx, An **Api** object is created from the **Flask** application object. This is the core of the Flask-RestX extension, which aids in the building of RESTful APIs. The User and Simulation controllers are imported and added as namespaces to the **Api** object. This is useful for grouping related routes and helps in maintaining a clean and organized codebase.

The application is configured to use JWT for user authentication. The secret key for JWT is set as **SECRET_JWT**, which is used to sign and verify tokens. The **SECRET_KEY** is set for the Flask application. This key is used by Flask and its extensions to keep data secure. For example, it is used to sign session cookies for protection against cookie data tampering.

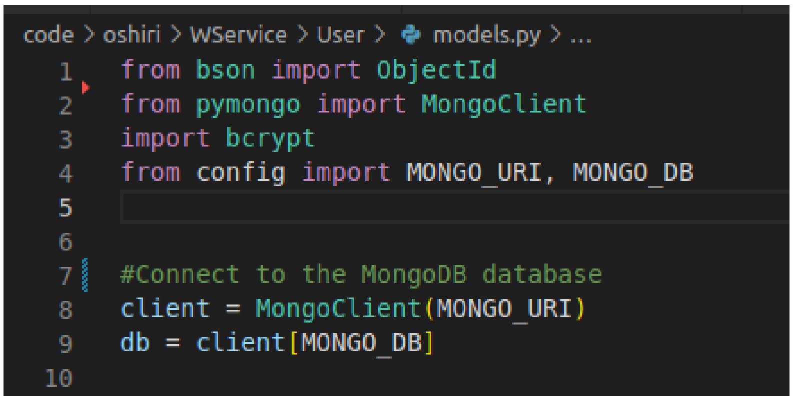
The last block of code checks if this file is the main program (i.e., it's not being imported by another Python file). If it is, it starts running the Flask application with debugging enabled. The debugging mode helps during the development stage by providing detailed error pages when something goes wrong.

3.2.2. User Management System

The user management system plays a crucial role in our application. It allows for the registration, authentication, and management of users, alongside providing the necessary permissions. Users can be registered as either an "admin" or a "user" depending on their role, with each having different levels of access.

3.2.2.1. Models.py

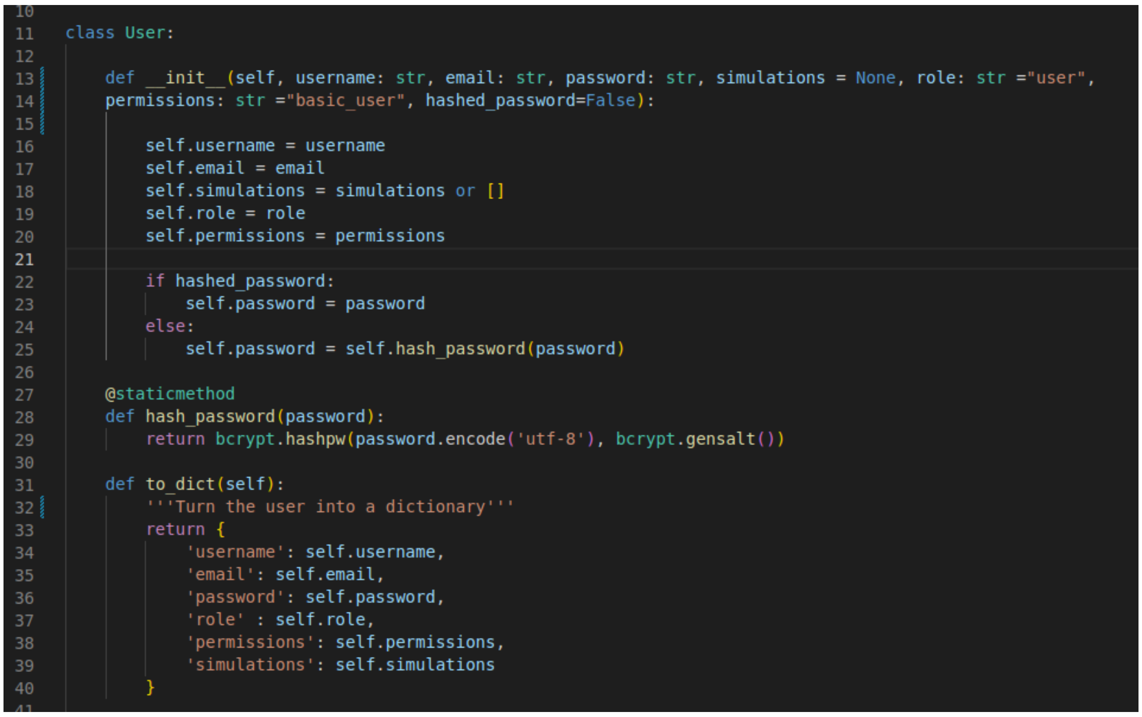
We now present our **models.py** file (see figures 3.6-3.9), where the User class is defined and the interaction with the MongoDB database is handled.



```
from bson import ObjectId
from pymongo import MongoClient
import bcrypt
from config import MONGO_URI, MONGO_DB


#Connect to the MongoDB database
client = MongoClient(MONGO_URI)
db = client[MONGO_DB]

```

Fig. 3.6. User’s models.py file

The above lines import the required libraries. MongoDB is used for data storage and bcrypt is used for password hashing to improve security. MONGO_URI and MONGO_DB are imported from the config.py file, these are the details necessary to connect with the MongoDB database. The connection to the MongoDB database is established using the MongoClient object. The MONGO_URI and MONGO_DB, from the config.py file, are used to specify the database details.

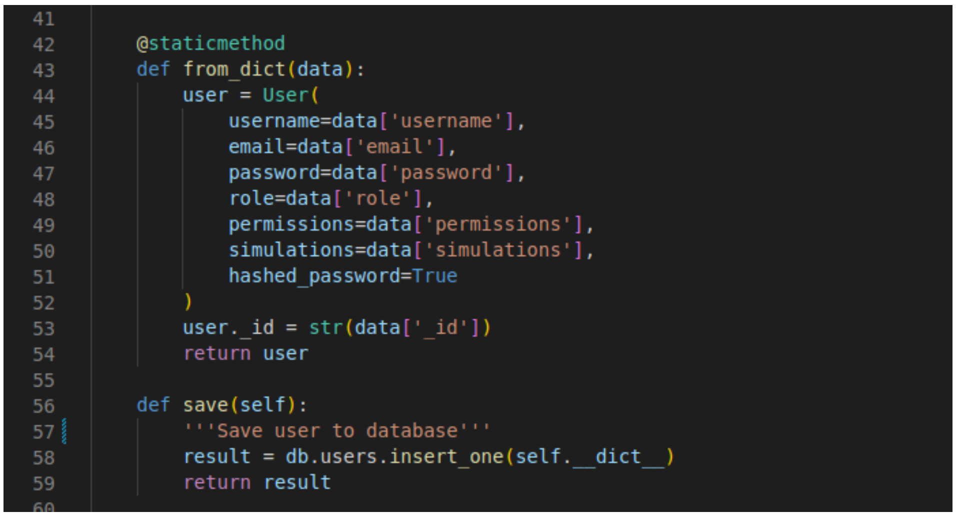


```python
class User:

    def __init__(self, username: str, email: str, password: str, simulations = None, role: str ="user",
    permissions: str ="basic_user", hashed_password=False):

        self.username = username
        self.email = email
        self.simulations = simulations or []
        self.role = role
        self.permissions = permissions

        if hashed_password:
            self.password = password
        else:
            self.password = self.hash_password(password)

    @staticmethod
    def hash_password(password):
        return bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

    def to_dict(self):
        '''Turn the user into a dictionary'''
        return {
            'username': self.username,
            'email': self.email,
            'password': self.password,
            'role' : self.role,
            'permissions': self.permissions,
            'simulations': self.simulations
        }
```

Fig. 3.7. User’s models.py file. Part 2

The User class is defined to represent a user in the system. Each user has a username, email, password, simulations, role, and permissions. The **__init__** method initializes a new instance of the User class. The hashed_password parameter is used to differentiate between a password that is already hashed and one that is not. The **hash_password** method hashes the password using bcrypt, providing additional security for user data.

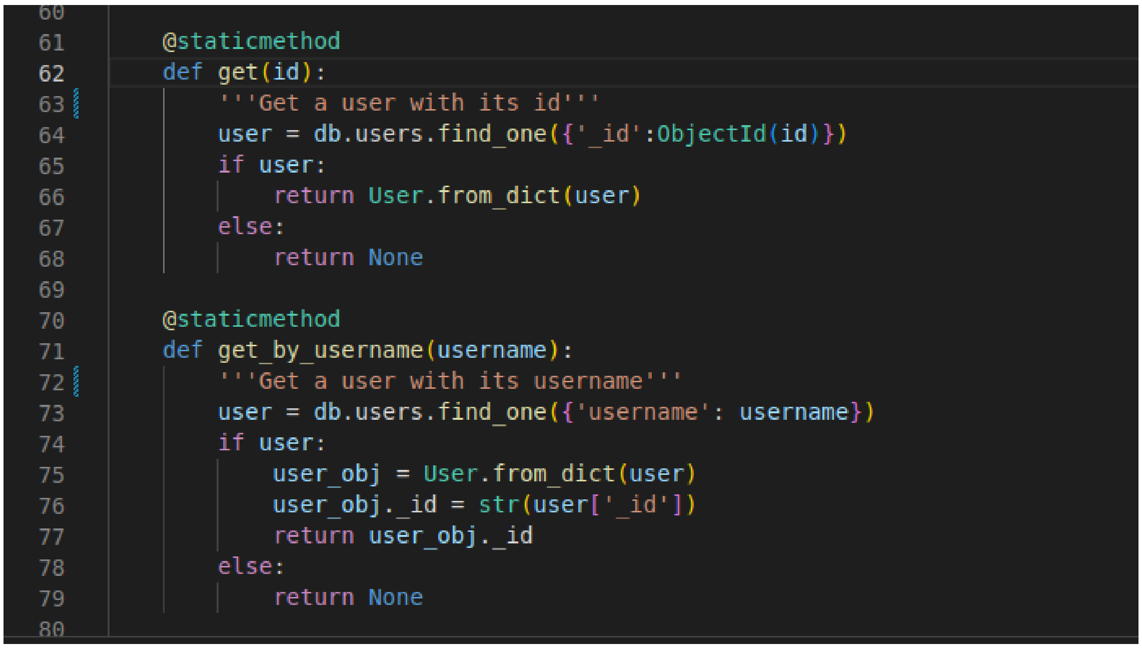
The **to_dict** method converts the User object into a dictionary. This is used when storing the user in the database or when returning the user's data in a response.



```python
    @staticmethod
    def from_dict(data):
        user = User(
            username=data['username'],
            email=data['email'],
            password=data['password'],
            role=data['role'],
            permissions=data['permissions'],
            simulations=data['simulations'],
            hashed_password=True
        )
        user._id = str(data['_id'])
        return user

    def save(self):
        '''Save user to database'''
        result = db.users.insert_one(self.__dict__)
        return result
```

Fig. 3.8. User’s models.py file. Part 3

The **from_dict** method creates a user object from a dictionary. This is used when retrieving user data from the database. Finally, the **save** method saves the User object to the MongoDB database.



```
    @staticmethod
    def get(id):
        '''Get a user with its id'''
        user = db.users.find_one({'_id':ObjectId(id)})
        if user:
            return User.from_dict(user)
        else:
            return None

    @staticmethod
    def get_by_username(username):
        '''Get a user with its username'''
        user = db.users.find_one({'username': username})
        if user:
            user_obj = User.from_dict(user)
            user_obj._id = str(user['_id'])
            return user_obj._id
        else:
            return None
```

Fig. 3.9. User's models.py file. Part 4

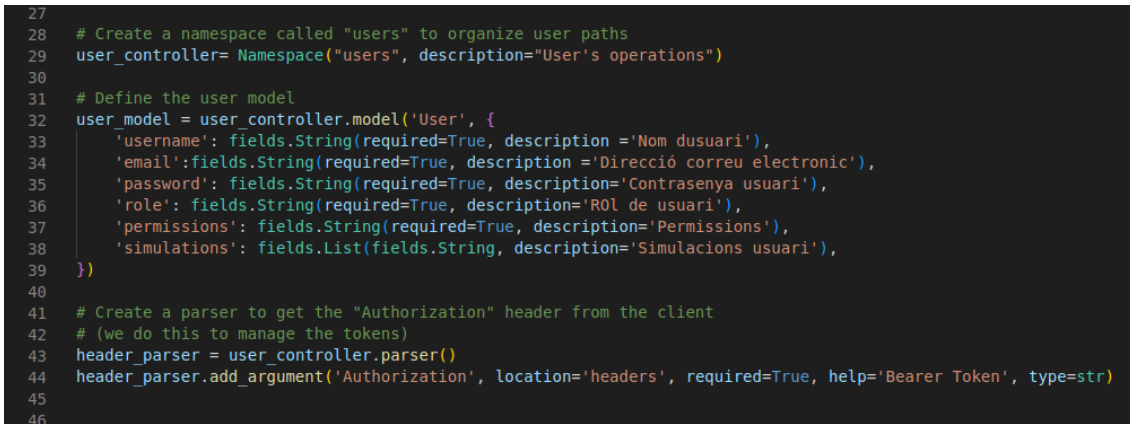
The **get** method retrieves a user from the database using their **_id** and the **get_by_username** using their username. We have also **get_by_email** and **get_all** users.

Finally, this file also has methods for deleting the user from the database, the **check_password** method, which checks whether the provided password matches the hashed password in the database and the **get_user_id** method retrieves the **_id** of the user from the database, which is used as a unique identifier for each user.

3.2.2.2. Controllers.py

The controllers in our application handle the interactions between the user and the system, taking user inputs, manipulating the data as required, and then sending a response back to the user. For our user management system, the controller primarily provides the routes for user registration, authentication, and updates, as well as managing user permissions.

The controllers.py file is where the application routes and their corresponding handlers are defined. Flask-RESTx, an extension for Flask that adds support for quickly building REST APIs, is being used to manage these routes and responses. Here we go through the controllers.py file (see figure 3.10).



```
# Create a namespace called "users" to organize user paths
user_controller= Namespace("users", description="User's operations")

# Define the user model
user_model = user_controller.model('User', {
    'username': fields.String(required=True, description ='Nom dusuari'),
    'email':fields.String(required=True, description ='Direcció correu electronic'),
    'password': fields.String(required=True, description='Contrasenya usuari'),
    'role': fields.String(required=True, description='ROl de usuari'),
    'permissions': fields.String(required=True, description='Permissions'),
    'simulations': fields.List(fields.String, description='Simulacions usuari'),
})

# Create a parser to get the "Authorization" header from the client
# (we do this to manage the tokens)
header_parser = user_controller.parser()
header_parser.add_argument('Authorization', location='headers', required=True, help='Bearer Token', type=str)
```

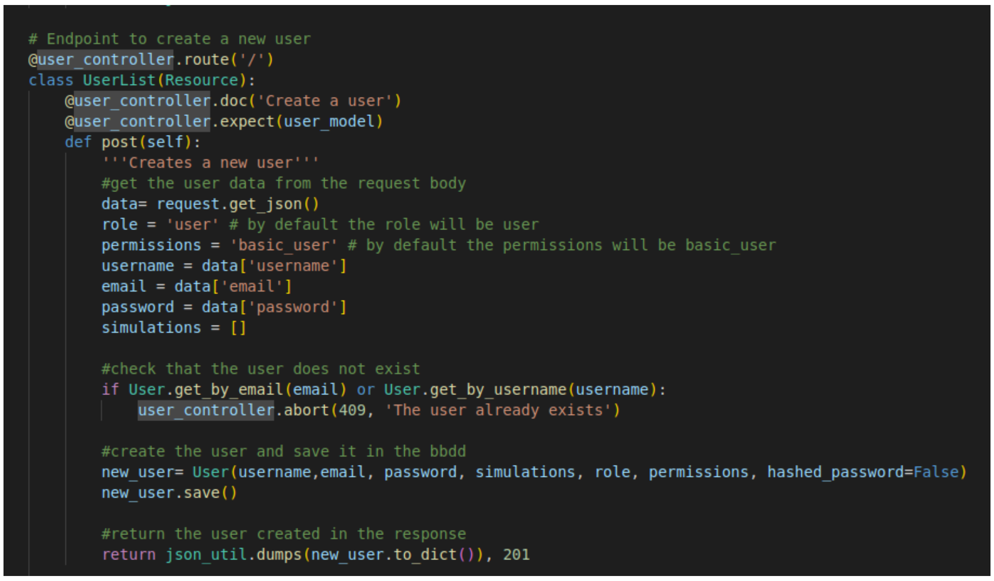
Fig. 3.10. User’s controller.py file.

A namespace is created for our user-related routes. This makes it easier to organize and manage routes and allows for logical grouping of related routes. The **user_model** object is defined, which represents the structure of a user. The fields of the **user_model** correspond to the properties of our **User** class.

A parser is created to get the "Authorization" header from incoming requests. This is used for token-based authentication.

Thereafter, various routes are defined:

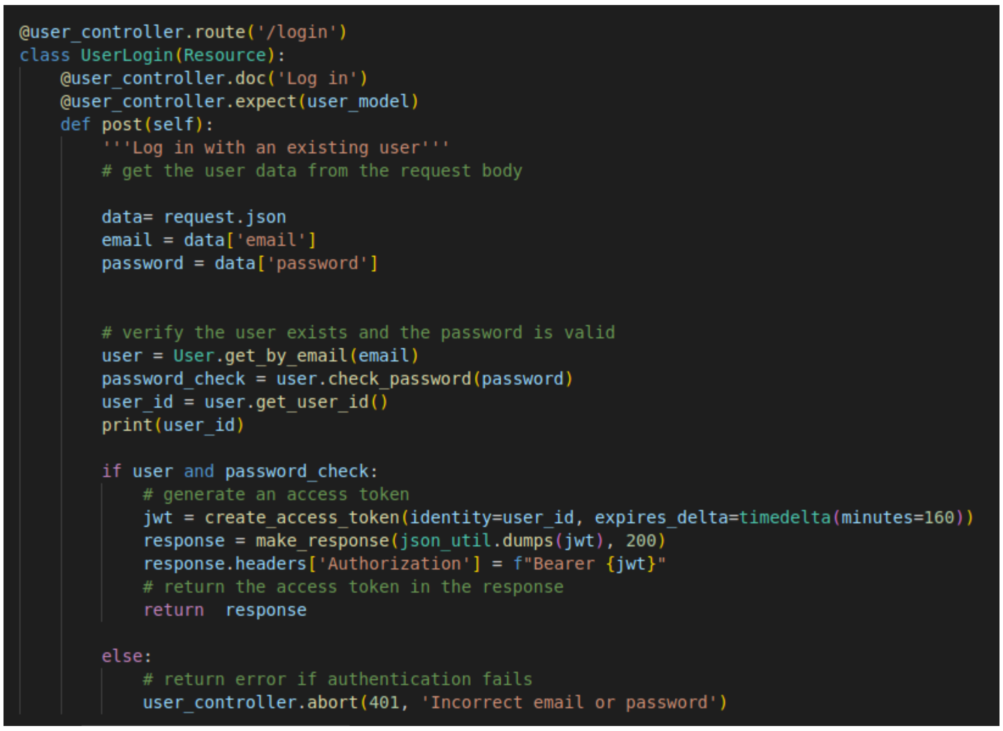
1. **User Registration:** A POST route at the **/** endpoint is created to handle user registration. This route receives a request with the new user's details, creates a **User** object with those details, and saves the object to the database. The same endpoint but using a GET route, list all registered users (see figure 3.11).



```
# Endpoint to create a new user
@user_controller.route('/')
class UserList(Resource):
    @user_controller.doc('Create a user')
    @user_controller.expect(user_model)
    def post(self):
        '''Creates a new user'''
        #get the user data from the request body
        data= request.get_json()
        role = 'user' # by default the role will be user
        permissions = 'basic_user' # by default the permissions will be basic_user
        username = data['username']
        email = data['email']
        password = data['password']
        simulations = []

        #check that the user does not exist
        if User.get_by_email(email) or User.get_by_username(username):
            user_controller.abort(409, 'The user already exists')

        #create the user and save it in the bbdd
        new_user= User(username,email, password, simulations, role, permissions, hashed_password=False)
        new_user.save()

        #return the user created in the response
        return json_util.dumps(new_user.to_dict()), 201
```

Fig. 3.11. User’s controller.py file. Endpoint to create a new user

2. **Admin Registration:** A POST route at the */admin* endpoint is used to create an administrator user.
3. **User Permissions Updating:** A PUT route at the */[username]/permissions* endpoint is used to update the permissions of a user.
4. **User Detail Retrieval:** A GET route at the */[user_id]* endpoint is used to retrieve details of a specific user.
5. **User Login:** A POST route at the */login* endpoint is used to authenticate a user and provide a token for further access to protected routes (see figure 3.12).



```python
@user_controller.route('/login')
class UserLogin(Resource):
    @user_controller.doc('Log in')
    @user_controller.expect(user_model)
    def post(self):
        '''Log in with an existing user'''
        # get the user data from the request body

        data= request.json
        email = data['email']
        password = data['password']


        # verify the user exists and the password is valid
        user = User.get_by_email(email)
        password_check = user.check_password(password)
        user_id = user.get_user_id()
        print(user_id)

        if user and password_check:
            # generate an access token
            jwt = create_access_token(identity=user_id, expires_delta=timedelta(minutes=160))
            response = make_response(json_util.dumps(jwt), 200)
            response.headers['Authorization'] = f"Bearer {jwt}"
            # return the access token in the response
            return  response

        else:
            # return error if authentication fails
            user_controller.abort(401, 'Incorrect email or password')
```

Fig. 3.12. User's controller.py file. Login endpoint.

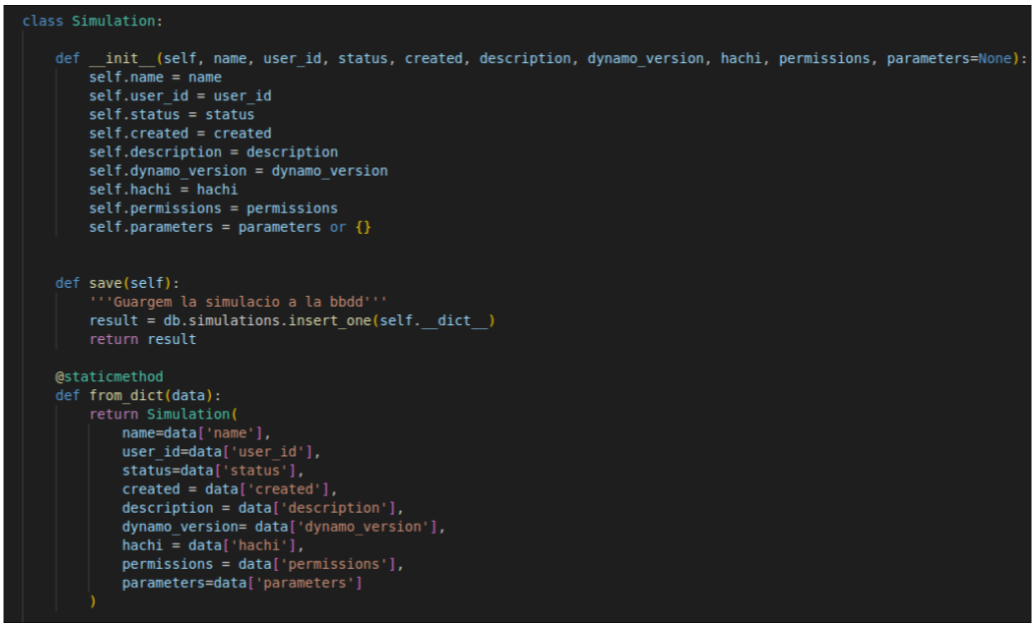
6. **User Update & Deletion:** PUT and DELETE routes, at the */[username]* endpoint, are used to update and delete a user, respectively.
7. **User Authentication:** A GET route at the */auth* endpoint is used to verify a user's authentication.
8. **Simulation List Retrieval:** A GET route at the */[user_id]/simulations* endpoint is used to retrieve a list of a user's simulations.
9. **Protected Route Access:** A GET route at the */protected* endpoint checks if the current user is authenticated.

3.2.3. Simulation Management System

The Simulation Management System is responsible for handling the creation, storage, retrieval, and manipulation of simulation data. This part of the system would manage different simulations that users might create and store in the system, including their various parameters, statuses, and related information.

3.2.3.1. Models.py

In our simulation management system, each simulation is represented as an instance of the Simulation class, defined in our models.py file (see figure 3.13).



```
class Simulation:

    def __init__(self, name, user_id, status, created, description, dynamo_version, hachi, permissions, parameters=None):
        self.name = name
        self.user_id = user_id
        self.status = status
        self.created = created
        self.description = description
        self.dynamo_version = dynamo_version
        self.hachi = hachi
        self.permissions = permissions
        self.parameters = parameters or {}


    def save(self):
        '''Guargem la simulacio a la bbdd'''
        result = db.simulations.insert_one(self.__dict__)
        return result

    @staticmethod
    def from_dict(data):
        return Simulation(
            name=data['name'],
            user_id=data['user_id'],
            status=data['status'],
            created = data['created'],
            description = data['description'],
            dynamo_version= data['dynamo_version'],
            hachi = data['hachi'],
            permissions = data['permissions'],
            parameters=data['parameters']
        )
```

Fig. 3.13. Simulation’s model.py file.

The Simulation class is initialized with several attributes: name, user_id, status, created, description, dynamo_version, hachi, permissions, and parameters. These attributes store information about the simulation.

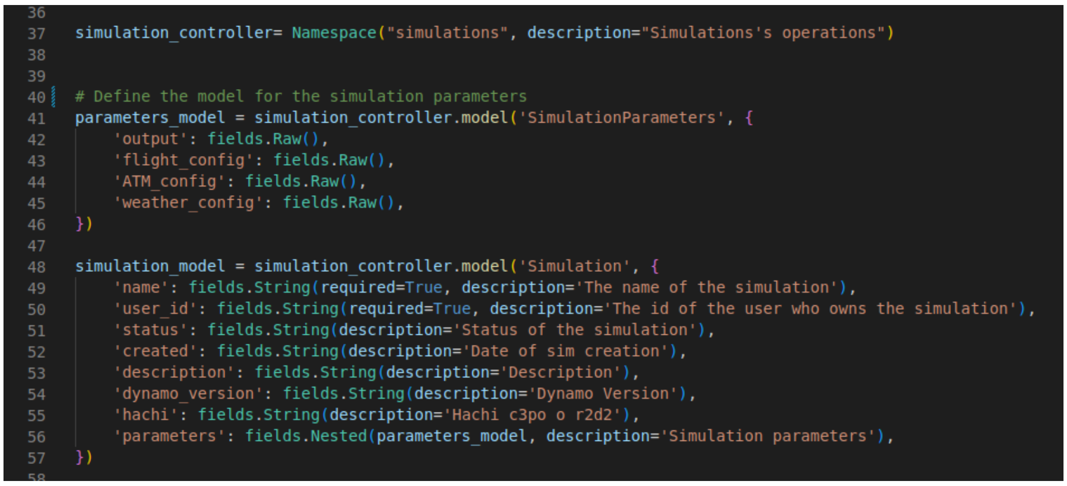
Similar as we do with the users’ controllers, we have several methods:

- The **save** method is used to save a simulation to the database. The **insert_one** function from PyMongo is used to insert the simulation's dictionary representation (**self.__dict__**) into the 'simulations' collection in the database.
- The **from_dict** method takes a dictionary and returns a **Simulation** object with attributes set based on the dictionary.
- The **get** method retrieves a simulation from the database using its ID.
- The **get_by_user_id** method retrieves all simulations for a specific user from the database using the user's ID.
- The **get_all** method retrieves all simulations from the database.

- The **update_status** method updates the status (configuring, running, completed) of a simulation in the database using the simulation's ID and the new status.
- The **update_parameters** method updates the parameters of a simulation in the database. This is thought to be used when a user is configuring a simulation and modifying some parameters, such as the Flight configuration, ATM configuration, etc.
- The **delete_simulation** method deletes a simulation from the database using the simulation's ID.
- The **to_dict** method converts a **Simulation** object to a dictionary.
- The **get_simu_id** method returns the simulation's ID as a string.

3.2.3.2. Controllers.py

The simulation controller (see figure 3.14) is responsible for managing the creation, retrieval, and deletion of simulations. More generally, the business logic associated with simulations is handled here. Each method within this controller interacts with the database and carries out CRUD (Create, Read, Update, Delete) operations on simulation data.



```python

simulation_controller= Namespace("simulations", description="Simulations's operations")


# Define the model for the simulation parameters
parameters_model = simulation_controller.model('SimulationParameters', {
    'output': fields.Raw(),
    'flight_config': fields.Raw(),
    'ATM_config': fields.Raw(),
    'weather_config': fields.Raw(),
})

simulation_model = simulation_controller.model('Simulation', {
    'name': fields.String(required=True, description='The name of the simulation'),
    'user_id': fields.String(required=True, description='The id of the user who owns the simulation'),
    'status': fields.String(description='Status of the simulation'),
    'created': fields.String(description='Date of sim creation'),
    'description': fields.String(description='Description'),
    'dynamo_version': fields.String(description='Dynamo Version'),
    'hachi': fields.String(description='Hachi c3po o r2d2'),
    'parameters': fields.Nested(parameters_model, description='Simulation parameters'),
})
```

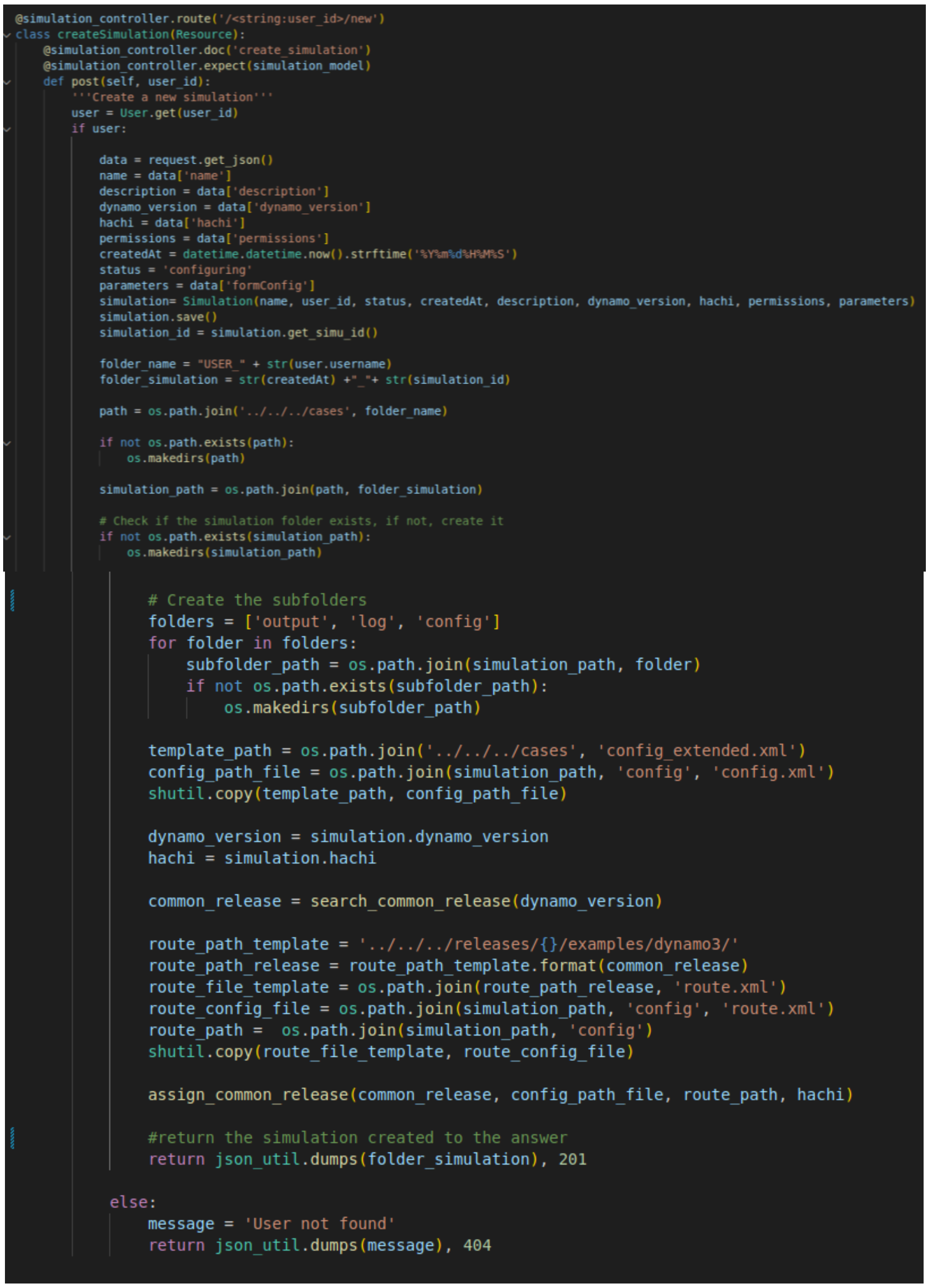
Fig. 3.14. Simulation's controller.py file.

This sets up a **namespace** for all simulation-related routes under "/simulations". All routes in this namespace will be prefixed with "/simulations". Simulation and parameters model definitions are schemas for simulation objects and their parameters. They are used to validate incoming data when creating a new simulation.

In the same way as with the user controllers here we also have different resources to manage the routes.

1. **SimulationList** resource: This class is used to handle GET requests to "/simulations". It retrieves and returns all simulations from the MongoDB database.

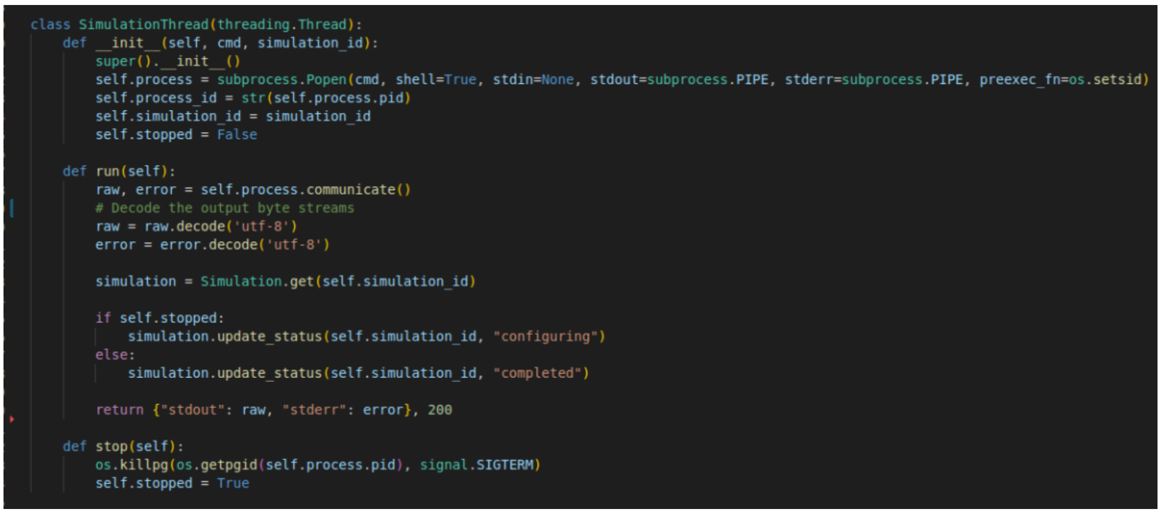
2. **createSimulation** resource (see figure 3.15): This class is used to handle POST requests to "/simulations/string:user_id/new", where **<string:user_id>** is the ID of the user who owns the simulation. It receives data from the request, validates it against the simulation model, creates a new Simulation object, and saves it to the database. It also creates necessary folders and files for the simulation, assigns a dynamo version and a common release, and finally returns the created simulation's ID. This folder creation follows the structure defined in section 3.1.3.



```python
@simulation_controller.route('/<string:user_id>/new')
class createSimulation(Resource):
    @simulation_controller.doc('create_simulation')
    @simulation_controller.expect(simulation_model)
    def post(self, user_id):
        '''Create a new simulation'''
        user = User.get(user_id)
        if user:

            data = request.get_json()
            name = data['name']
            description = data['description']
            dynamo_version = data['dynamo_version']
            hachi = data['hachi']
            permissions = data['permissions']
            createdAt = datetime.datetime.now().strftime('%Y%m%d%H%M%S')
            status = 'configuring'
            parameters = data['formConfig']
            simulation= Simulation(name, user_id, status, createdAt, description, dynamo_version, hachi, permissions, parameters)
            simulation.save()
            simulation_id = simulation.get_simu_id()

            folder_name = "USER_" + str(user.username)
            folder_simulation = str(createdAt) +"_"+ str(simulation_id)

            path = os.path.join('../../../cases', folder_name)

            if not os.path.exists(path):
                os.makedirs(path)

            simulation_path = os.path.join(path, folder_simulation)

            # Check if the simulation folder exists, if not, create it
            if not os.path.exists(simulation_path):
                os.makedirs(simulation_path)

            # Create the subfolders
            folders = ['output', 'log', 'config']
            for folder in folders:
                subfolder_path = os.path.join(simulation_path, folder)
                if not os.path.exists(subfolder_path):
                    os.makedirs(subfolder_path)

            template_path = os.path.join('../../../cases', 'config_extended.xml')
            config_path_file = os.path.join(simulation_path, 'config', 'config.xml')
            shutil.copy(template_path, config_path_file)

            dynamo_version = simulation.dynamo_version
            hachi = simulation.hachi

            common_release = search_common_release(dynamo_version)

            route_path_template = '../../../releases/{}/examples/dynamo3/'
            route_path_release = route_path_template.format(common_release)
            route_file_template = os.path.join(route_path_release, 'route.xml')
            route_config_file = os.path.join(simulation_path, 'config', 'route.xml')
            route_path =  os.path.join(simulation_path, 'config')
            shutil.copy(route_file_template, route_config_file)

            assign_common_release(common_release, config_path_file, route_path, hachi)

            #return the simulation created to the answer
            return json_util.dumps(folder_simulation), 201

        else:
            message = 'User not found'
            return json_util.dumps(message), 404
```

Fig.3.15. Create Simulation Resource

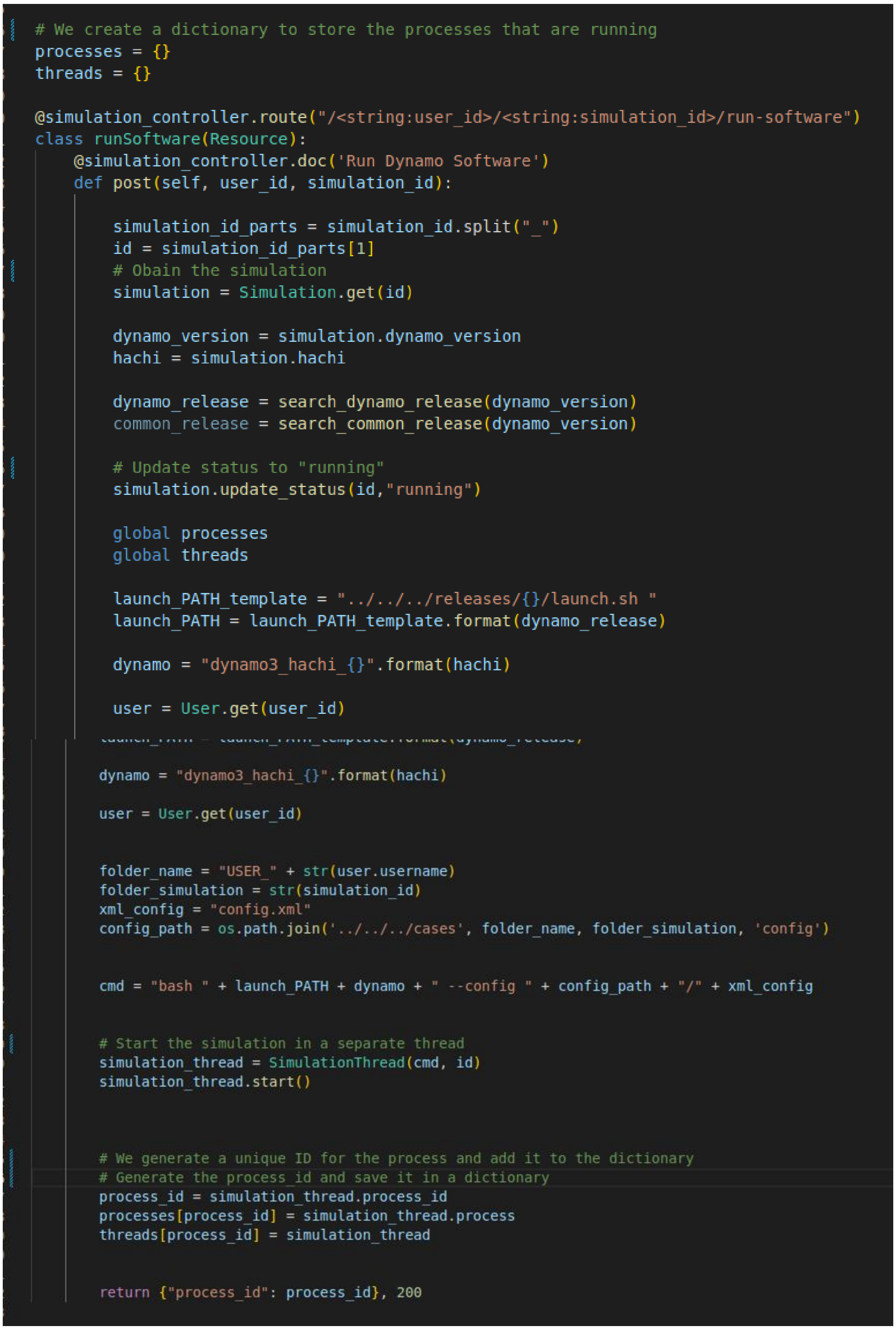
1. **getSimulation** resource: This class is used to handle GET requests to "/simulations/string:user_id/string: simulation_id/get_one", where **<string:user_id>** and **<string: simulation_id>** are the IDs of the user and the simulation respectively. It retrieves and returns the specified simulation.
2. **UserSimulationList** resource: This class is used to handle GET requests to "/simulations/string:user_id/simulations", where **<string: user_id>** is the ID of the user. It retrieves and returns all simulations owned by the specified user.
3. **deleteSimulation** resource: This class is used to handle DELETE requests to "/simulations/string: user_id/string: simulation_id/string: created", where **<string: user_id>**, **<string: simulation_id>** and **<string: created>** are the IDs of the user, the simulation, and the created timestamp respectively. It moves the simulation's directory to a "deleted_simulations" directory and then deletes the simulation from the database.
4. **SimulationThread** class (see figure 3.16): This class inherits from **threading.Thread** and overrides its **run** and **stop** methods to allow the simulation to run in a separate process. It uses the **subprocess** module to execute the simulation command, and provides a method to terminate the process. This class is especially handy when simulations take a long time to complete and you don't want to block the server.



```python
class SimulationThread(threading.Thread):
    def __init__(self, cmd, simulation_id):
        super().__init__()
        self.process = subprocess.Popen(cmd, shell=True, stdin=None, stdout=subprocess.PIPE, stderr=subprocess.PIPE, preexec_fn=os.setsid)
        self.process_id = str(self.process.pid)
        self.simulation_id = simulation_id
        self.stopped = False

    def run(self):
        raw, error = self.process.communicate()
        # Decode the output byte streams
        raw = raw.decode('utf-8')
        error = error.decode('utf-8')

        simulation = Simulation.get(self.simulation_id)

        if self.stopped:
            simulation.update_status(self.simulation_id, "configuring")
        else:
            simulation.update_status(self.simulation_id, "completed")

        return {"stdout": raw, "stderr": error}, 200

    def stop(self):
        os.killpg(os.getpgid(self.process.pid), signal.SIGTERM)
        self.stopped = True
```

Fig. 3.16. Simulation’s controller.py file. SimulationThread class

5. **runSoftware** route (see figure 3.17): This endpoint allows the user to run the Dynamo software. This involves locating the correct Dynamo release, updating the simulation status to "running", and starting the simulation in a separate thread. The unique process_id is then returned as a response.



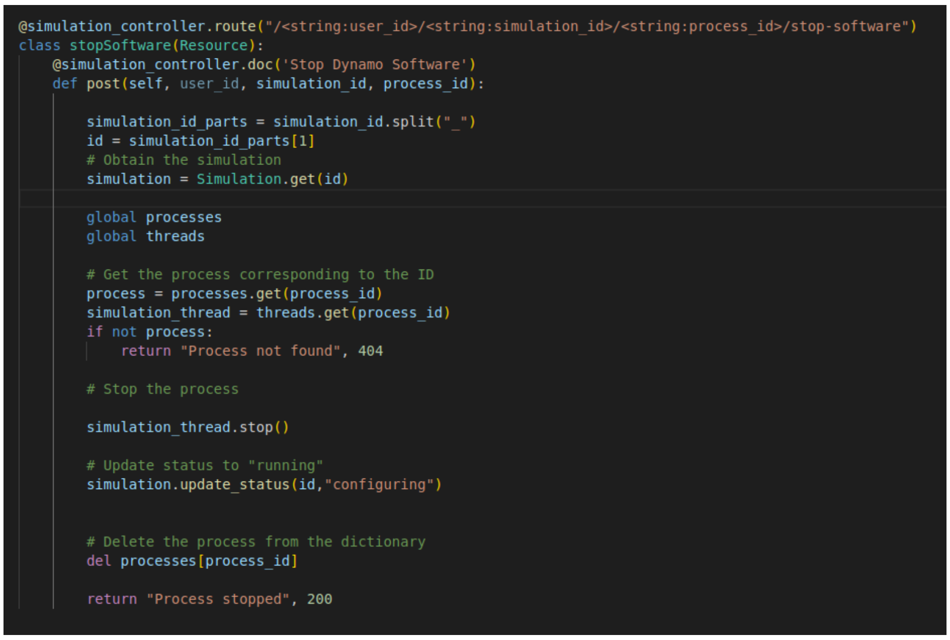
```python
# We create a dictionary to store the processes that are running
processes = {}
threads = {}

@simulation_controller.route("/<string:user_id>/<string:simulation_id>/run-software")
class runSoftware(Resource):
    @simulation_controller.doc('Run Dynamo Software')
    def post(self, user_id, simulation_id):

        simulation_id_parts = simulation_id.split("_")
        id = simulation_id_parts[1]
        # Obain the simulation
        simulation = Simulation.get(id)

        dynamo_version = simulation.dynamo_version
        hachi = simulation.hachi

        dynamo_release = search_dynamo_release(dynamo_version)
        common_release = search_common_release(dynamo_version)

        # Update status to "running"
        simulation.update_status(id,"running")

        global processes
        global threads

        launch_PATH_template = "../../../releases/{}/launch.sh "
        launch_PATH = launch_PATH_template.format(dynamo_release)

        dynamo = "dynamo3_hachi_{}".format(hachi)

        user = User.get(user_id)
```

```python
        dynamo = "dynamo3_hachi_{}".format(hachi)

        user = User.get(user_id)
```

```python

        folder_name = "USER_" + str(user.username)
        folder_simulation = str(simulation_id)
        xml_config = "config.xml"
        config_path = os.path.join('../../../cases', folder_name, folder_simulation, 'config')


        cmd = "bash " + launch_PATH + dynamo + " --config " + config_path + "/" + xml_config


        # Start the simulation in a separate thread
        simulation_thread = SimulationThread(cmd, id)
        simulation_thread.start()



        # We generate a unique ID for the process and add it to the dictionary
        # Generate the process_id and save it in a dictionary
        process_id = simulation_thread.process_id
        processes[process_id] = simulation_thread.process
        threads[process_id] = simulation_thread


        return {"process_id": process_id}, 200
```

Fig.3.17. Run Software Resource

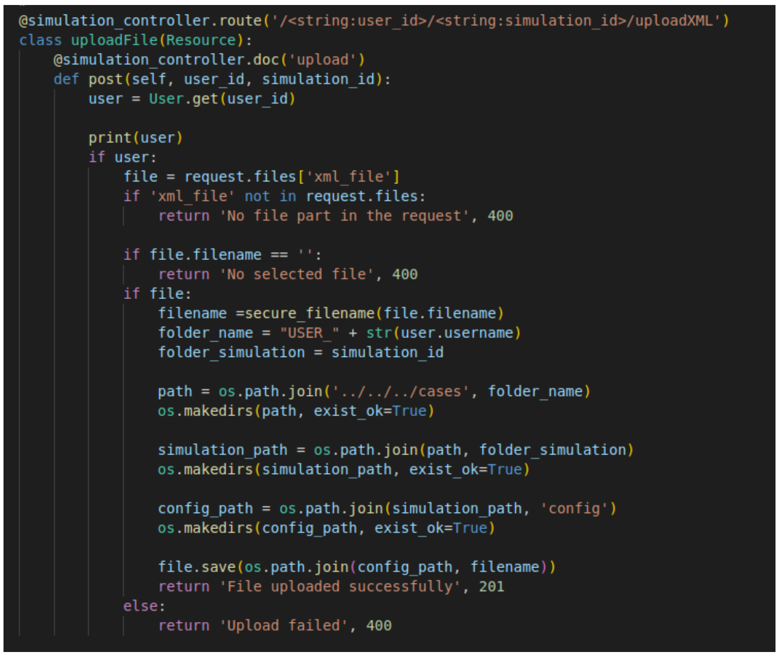
6. **stopSoftware** route (see figure 3.18): This allows the user to stop a running Dynamo software process by providing the process_id. It fetches the corresponding process and stops it, also updating the simulation status back to "configuring". The process is then removed from the processes dictionary.



```python
@simulation_controller.route("/<string:user_id>/<string:simulation_id>/<string:process_id>/stop-software")
class stopSoftware(Resource):
    @simulation_controller.doc('Stop Dynamo Software')
    def post(self, user_id, simulation_id, process_id):

        simulation_id_parts = simulation_id.split("_")
        id = simulation_id_parts[1]
        # Obtain the simulation
        simulation = Simulation.get(id)

        global processes
        global threads

        # Get the process corresponding to the ID
        process = processes.get(process_id)
        simulation_thread = threads.get(process_id)
        if not process:
            return "Process not found", 404

        # Stop the process

        simulation_thread.stop()

        # Update status to "running"
        simulation.update_status(id,"configuring")


        # Delete the process from the dictionary
        del processes[process_id]

        return "Process stopped", 200
```

Fig. 3.18. Stop Software Resource

7. **SimulationStatus** route: This route provides the current status of a simulation.
8. **uploadXML** route (see figure 3.19): This route lets the user upload an XML configuration file for the simulation. The file is saved to a specific location.



```python
@simulation_controller.route('/<string:user_id>/<string:simulation_id>/uploadXML')
class uploadFile(Resource):
    @simulation_controller.doc('upload')
    def post(self, user_id, simulation_id):
        user = User.get(user_id)

        print(user)
        if user:
            file = request.files['xml_file']
            if 'xml_file' not in request.files:
                return 'No file part in the request', 400

            if file.filename == '':
                return 'No selected file', 400
            if file:
                filename =secure_filename(file.filename)
                folder_name = "USER_" + str(user.username)
                folder_simulation = simulation_id

                path = os.path.join('../../../cases', folder_name)
                os.makedirs(path, exist_ok=True)

                simulation_path = os.path.join(path, folder_simulation)
                os.makedirs(simulation_path, exist_ok=True)

                config_path = os.path.join(simulation_path, 'config')
                os.makedirs(config_path, exist_ok=True)

                file.save(os.path.join(config_path, filename))
                return 'File uploaded successfully', 201
        else:
            return 'Upload failed', 400
```

Fig. 3.19. Upload XML Files Resource

9. **uploadMultipleFiles** route: This endpoint provides the functionality to upload multiple files at once. The files are stored in a dedicated directory.
10. **replaceXML** route: This route is used to replace specific keywords in an XML configuration file with the user-defined parameters.
11. **downloadZIP** route: This route enables users to download a ZIP file containing the results of a simulation. The results are zipped up into a single file which can be downloaded by the user.
12. **getImages** route: This endpoint is used to fetch all images resulting from a simulation. These images are returned as base64-encoded strings that can be decoded and displayed on the frontend.
13. **MeteoFiles** route: This route returns a list of all the meteorological files available in the "meteo" directory.
14. **get_config_files** route: This route allows users to fetch the configuration files for a simulation. The filenames are returned as a list.

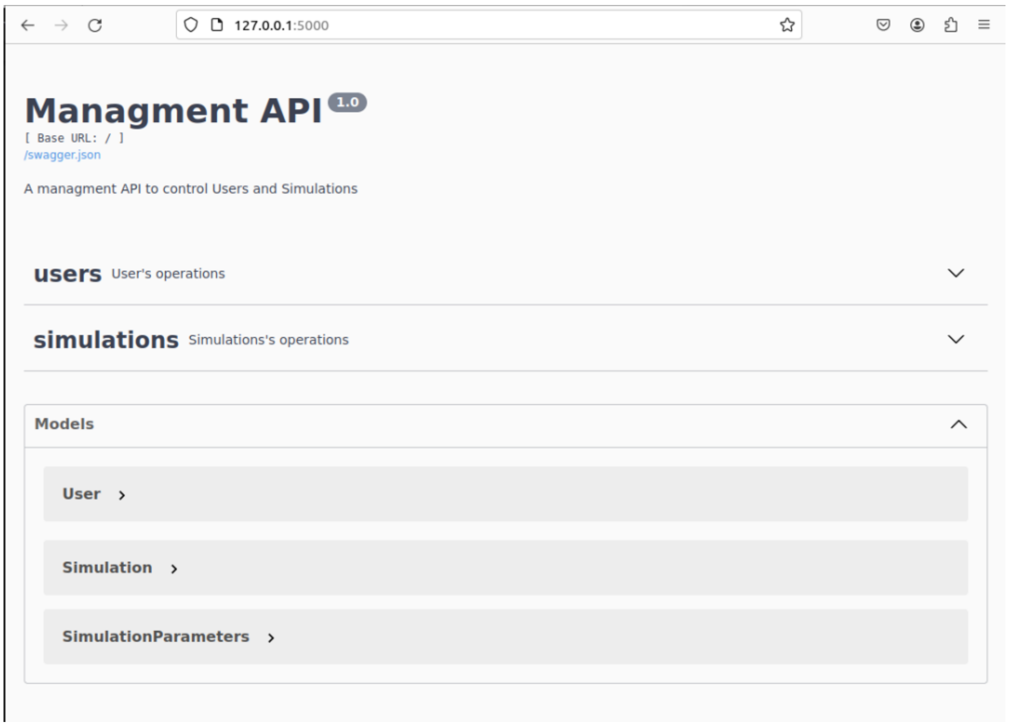
In summary, this simulation controller provides a comprehensive set of features to manage and control complex simulations. It provides users with the ability to initiate and stop simulations, check the status, upload necessary configuration files, replace parameters, and finally fetch and download the results.

3.3. Swagger

Swagger is a powerful tool for designing, building, documenting, and using RESTful web services. It can automatically generate an interactive user interface that represents your application's API documentation. This documentation includes all information about the routes, methods, parameters, responses, and other details of your APIs.

Thus, to conclude the backend section, it's important to note that Dynamo Web Services provides the added convenience of a built-in Swagger UI. This allows for a streamlined, interactive exploration of the APIs exposed by the backend.

In order to access the swagger, once the application is running locally, simply navigate to `http://localhost:5000/`. This will open up the Swagger interface (see figure 3.20), a window into the inner workings of the application's APIs, straight from your browser.



127.0.0.1:5000

Managment API 1.0

[Base URL: /]

/swagger.json

A managment API to control Users and Simulations

users User's operations

simulations Simulations's operations

Models

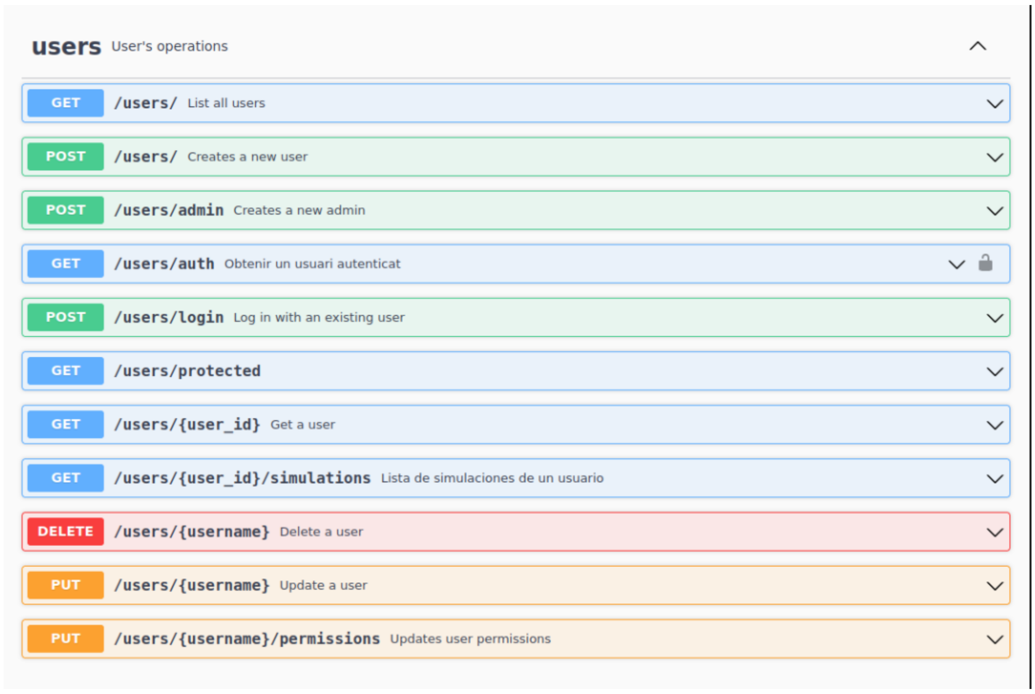
User ›

Simulation ›

SimulationParameters ›

Fig. 3.20. Swagger Interface

Within the Swagger UI, you will find all the API routes the application has to offer (see figure 3.21). Each of these routes comes with an explanation of what they do, their HTTP methods (GET, POST, etc.), and any parameters they might require. This is particularly useful for understanding the structure and utility of each API.



users User's operations

GET /users/ List all users

POST /users/ Creates a new user

POST /users/admin Creates a new admin

GET /users/auth Obtenir un usuari autenticat

POST /users/login Log in with an existing user

GET /users/protected

GET /users/{user_id} Get a user

GET /users/{user_id}/simulations Lista de simulaciones de un usuario

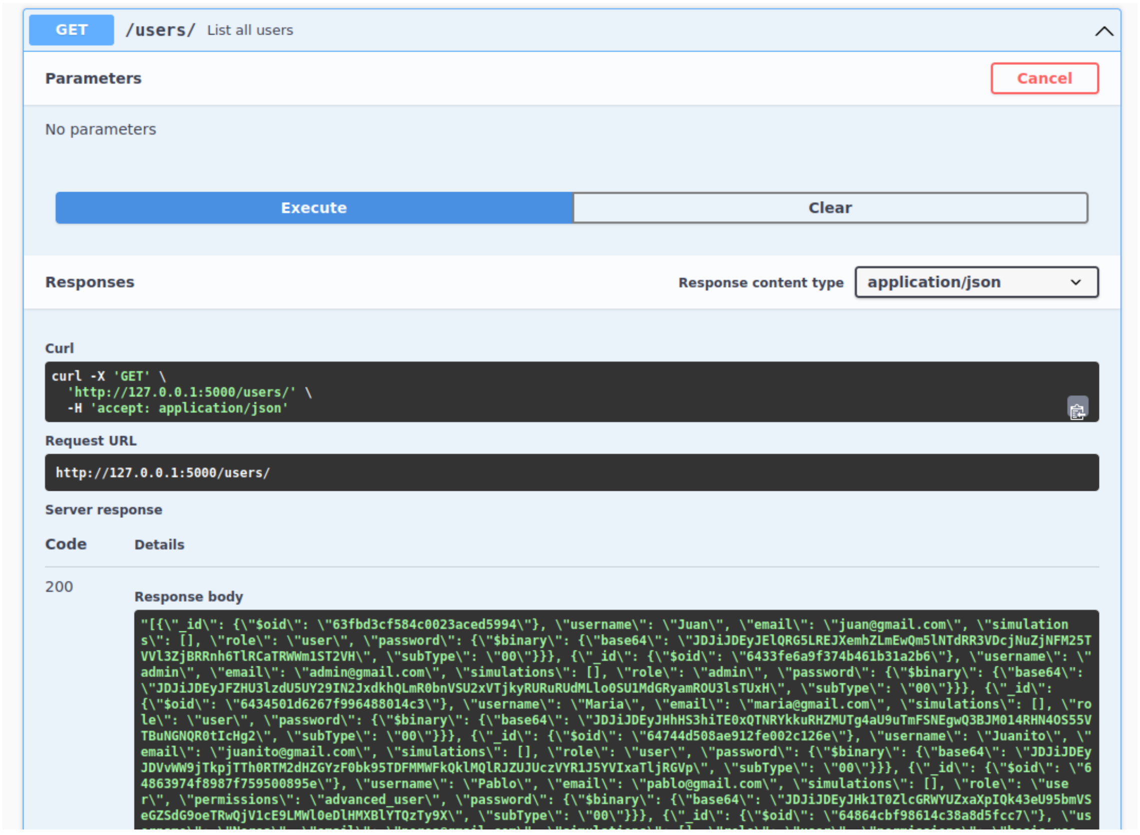
DELETE /users/{username} Delete a user

PUT /users/{username} Update a user

PUT /users/{username}/permissions Updates user permissions

Fig. 3.21. Swagger Interface. User Endpoints

What sets Swagger apart is its interactive nature. From the UI, it's possible to send requests to your APIs directly. Every API route has a 'Try it out' button, which, when clicked, allows you to provide necessary parameters and make a request (see figure 3.21). The server's response can be viewed immediately, right on the same interface.



```
curl -X 'GET' \
  'http://127.0.0.1:5000/users/' \
  -H 'accept: application/json'
```

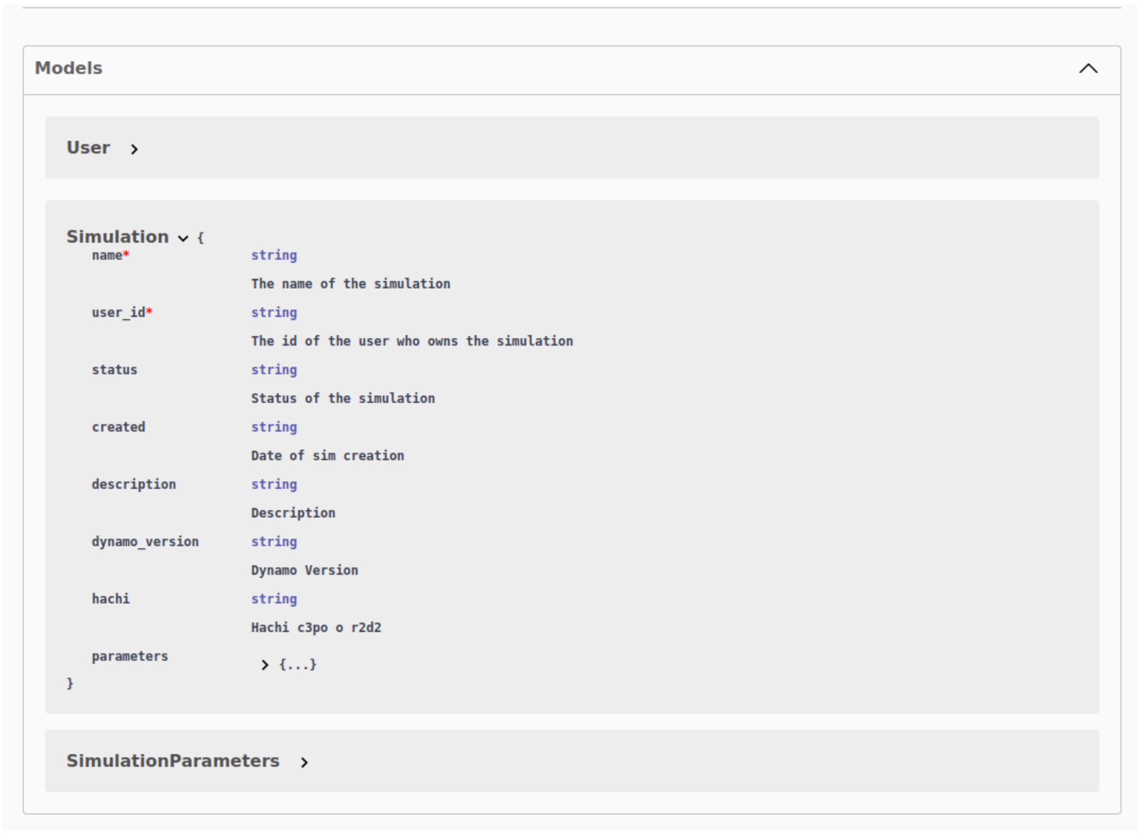
```
http://127.0.0.1:5000/users/
```

```
"[{\"_id\": {\"$oid\": \"63fbd3cf584c0023aced5994\"}, \"username\": \"Juan\", \"email\": \"juan@gmail.com\", \"simulation
s\": [], \"role\": \"user\", \"password\": {\"$binary\": {\"base64\": \"JDJiJDEyJElQRG5LREJXemhZLmEwQm5lNTdRR3VDcjNuZjNFM25T
VVl3ZjBRRnh6TlRCaTRWWm1ST2VH\", \"subType\": \"00\"}}}, {\"_id\": {\"$oid\": \"6433fe6a9f374b461b31a2b6\"}, \"username\": \"
admin\", \"email\": \"admin@gmail.com\", \"simulations\": [], \"role\": \"admin\", \"password\": {\"$binary\": {\"base64\":
\"JDJiJDEyJFZHU3lzdU5UY29IN2JxdkhQLmR0bnVSU2xVTjkyRURuRUdMLlo0SU1MdGRyamROU3lsTUxH\", \"subType\": \"00\"}}}, {\"_id\":
{\"$oid\": \"6434501d6267f996488014c3\"}, \"username\": \"Maria\", \"email\": \"maria@gmail.com\", \"simulations\": [], \"ro
le\": \"user\", \"password\": {\"$binary\": {\"base64\": \"JDJiJDEyJHhHS3hiTE0xQTNRYkkuRHZMUTg4aU9uTmFSNEgwQ3BJM014RHN4OS55V
TBuNGNQR0tIcHg2\", \"subType\": \"00\"}}}, {\"_id\": {\"$oid\": \"64744d508ae912fe002c126e\"}, \"username\": \"Juanito\", \"
email\": \"juanito@gmail.com\", \"simulations\": [], \"role\": \"user\", \"password\": {\"$binary\": {\"base64\": \"JDJiJDEy
JDVvWW9jTkpjTTh0RTM2dHZGYzF0bk95TDFMMWFkQklMQlRJZUJUczVYR1J5YVIxaTljRGVp\", \"subType\": \"00\"}}}, {\"_id\": {\"$oid\": \"6
4863974f8987f759500895e\"}, \"username\": \"Pablo\", \"email\": \"pablo@gmail.com\", \"simulations\": [], \"role\": \"use
r\", \"permissions\": \"advanced_user\", \"password\": {\"$binary\": {\"base64\": \"JDJiJDEyJHk1T0ZlcGRWYUZxaXpIQk43eU95bmVS
eGZSdG9oeTRwQjV1cE9LMWl0eDlHMXBlYTQzTy9X\", \"subType\": \"00\"}}}, {\"_id\": {\"$oid\": \"64864cbf98614c38a8d5fcc7\"}, \"us
```

Fig. 3.21. Swagger Server Response. Code 200.

As you can see in the figure above, Swagger UI also offers insight into the response formats for each API. This is illustrated through example responses, providing an understanding of what to expect when integrating the APIs into other parts of the application or when troubleshooting.

In addition, Swagger UI provides comprehensive definitions for the data models (see figure 3.22). This greatly helps in understanding the type of data the APIs is expecting or what they may return.



```
Models

User >

Simulation v {
   name*            string
                    The name of the simulation
   user_id*         string
                    The id of the user who owns the simulation
   status           string
                    Status of the simulation
   created          string
                    Date of sim creation
   description      string
                    Description
   dynamo_version   string
                    Dynamo Version
   hachi            string
                    Hachi c3po o r2d2
   parameters        > {...}
}

SimulationParameters >
```

Fig. 3.22. Swagger Simulation Model

To sum it up, the integration of Swagger UI within this backend simplifies the process of understanding and interacting with the API. This tool serves as a handy, intuitive interface that aids in effective debugging and testing of the application. In essence, it acts as a bridge between a person and the application, making backend API interaction a seamless process.

CHAPTER 4. Frontend

As we shift focus from the application's backend to its frontend, it is important to recognize that the visual aspect of a software is just as important to its success as its underlying functionality. The frontend, or application's "face," is a representation of the user interface (UI) that users interact with. It is a potent synthesis of design, functionality, and user experience. The design and implementation of the frontend are covered in detail in this chapter, where we also go over some of its key components.

My frontend journey started with a course from Zero to Mastery Academy [‡36] on Vue 3 and Tailwind CSS, which served as an introduction to this progressive JavaScript framework. Vue 3 was chosen for its ease of integration, scalability, and its reactivity system, which all contribute to an efficient development process and a high-performing end product.

Together, Vue 3 and Tailwind provide the foundation for our frontend, contributing to an efficient and effective user experience. As such, the frontend was designed with an emphasis on clarity, simplicity, and ease of use.

Users are greeted by a registration and login screen when they first use the application. They have access to a dashboard after successful authentication, where they may keep track of their prior simulations and start new ones. This user-friendly interface was shaped by a carefully chosen web template, resulting in a smooth user experience.

In the following sections, we will delve deeper into the specifics of the frontend design and its implementation, discussing in detail the User Management System and Simulation Management System.

4.1. Frontend Design

The frontend design forms the critical point of interaction between the user and the backend processing machinery. It is what the users see, touch, and interact with directly. Thus, the effectiveness of a frontend design can be gauged by its ability to provide a seamless, intuitive, and enjoyable user experience.

The process of designing our frontend started with creating wireframes and design mock-ups (see figures 4.1-4.2) during the requirements gathering phase. Mockplus RP [‡37], a renowned web-based design tool, was used for this purpose. This tool is particularly suited for designing interactive prototypes swiftly and with high precision, helping to capture the application's overall flow and functionality from the early design stages.

The original idea was to create a simple area where users could sign up and log in. Users would be welcomed with a comprehensive dashboard after getting access. The center of user activity would be this dashboard, which was made to be simple to use and intuitive.

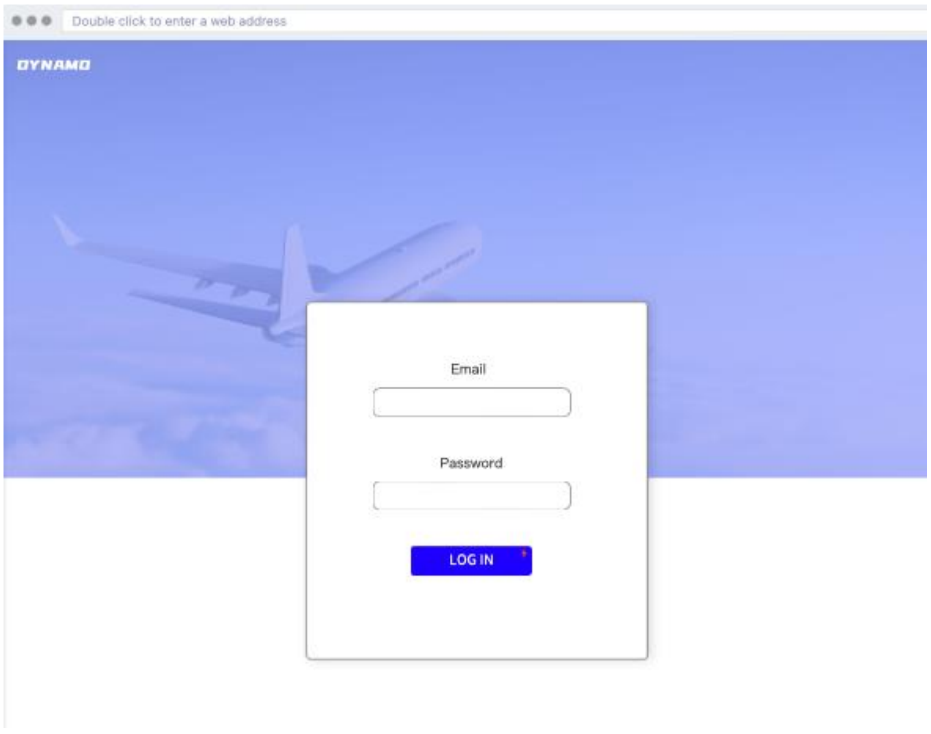


Fig. 4.1. Design for the Login page

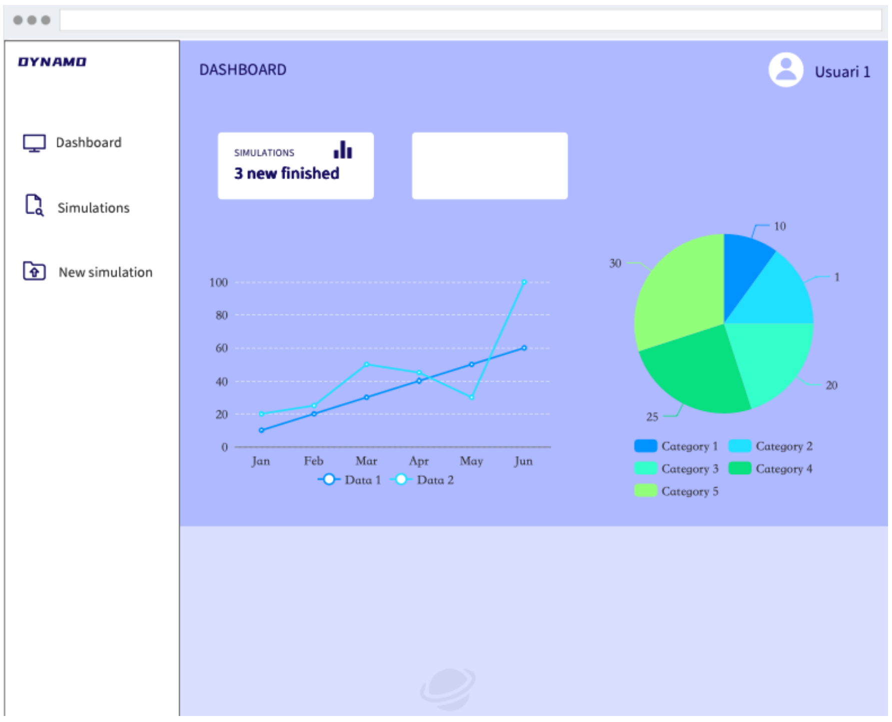


Fig. 4.2. Design for the initial Dashboard

The design of the dashboard was rooted in principles of simplicity and ease of navigation. The aim was to create an intuitive, user-friendly environment where information could be easily found, and new simulations could be conveniently started. The layout was thoughtfully designed to display simulations and a dedicated section for creating new ones.

For the new simulations, we adopted a form-based approach (see figure 4.3). The form is broken down into different sections, or blocks, each corresponding to the distinct components of the XML configuration files, as Flight Configuration, ATM Configuration, and Weather.

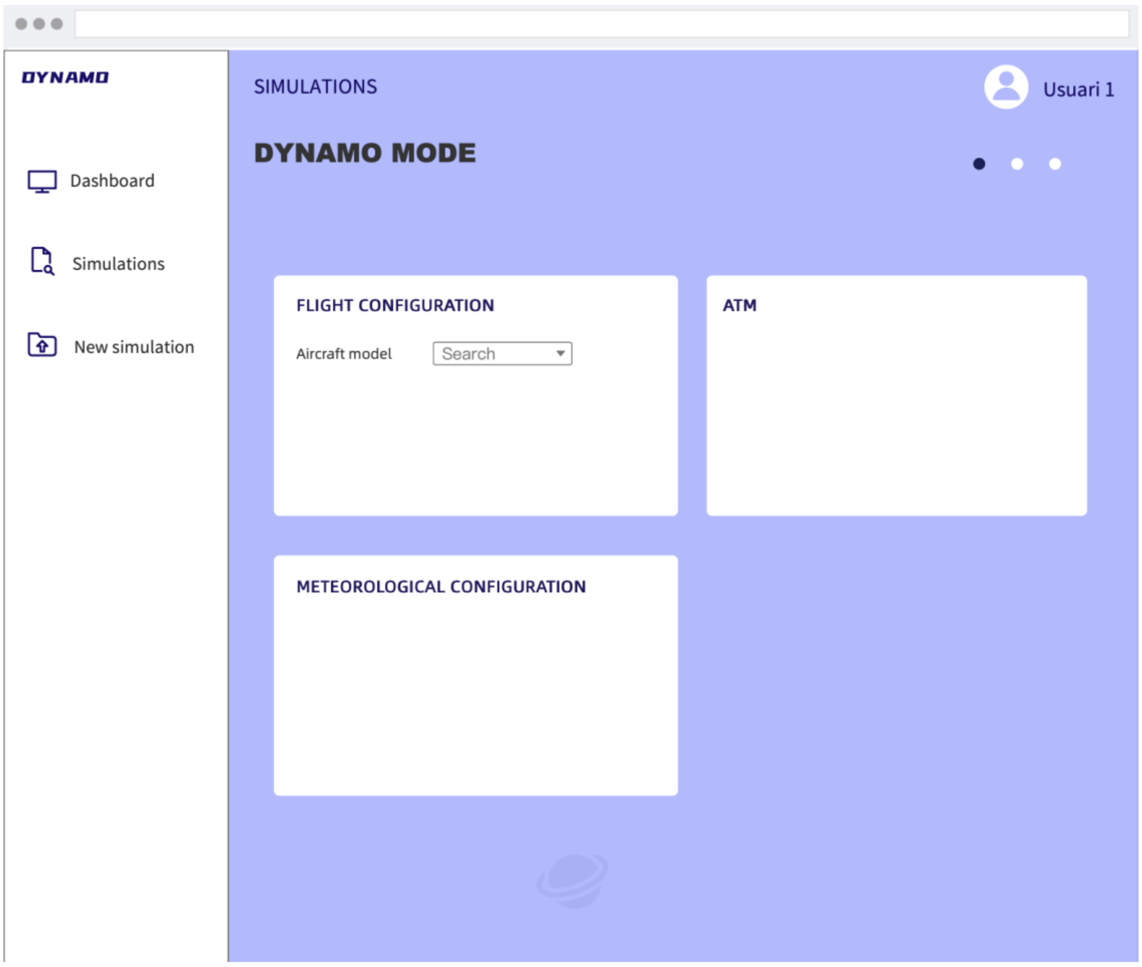


Fig. 4.3. Design for the creation of new simulations page

The design idea behind this block-structure was to make the simulation creation process as streamlined as possible. This breakdown not only mirrored the structure of our configuration files but also allowed users to focus on one aspect of the simulation at a time, enhancing the overall user experience.

In essence, the frontend design was focused on a seamless user experience, utility, and simplicity in addition to aesthetics. We were able to visualize and refine the design using tools like Mockplus RP, making sure it was user-centered and in line with the general objectives of our application. As we move forward into the implementation details in the next sections, we'll see how these design considerations materialized into a functional user interface.

4.2. Frontend Implementation

As we transition from the design phase into the implementation one, we turn our attention to the means and methods used to bring the design concepts to life. The frontend implementation is where the aesthetics meet functionality, and our design concepts materialize into a tangible, interactive platform for our users.

For this project, I used a template provided by JustBoil.me as a foundation. JustBoil.me is renowned for its high-quality, modern, and responsive templates that provide an excellent starting point for web projects. The particular template used for this project came bundled with a host of pre-designed components, such as buttons and cards, easing the process of development by providing a head start.

This template is available under the MIT License, a permissive free software license. This means that it can be freely used, modified, and shared, allowing us flexibility in tailoring the template to meet our specific requirements and ensuring that no licensing issues hinder our project's progression.

In the following sections, we will go into more detail about the specifics of how this template, along with the power of Vue 3 and Tailwind CSS, were used to build the User Management System and the Simulation Management System. We will also explore how these tools and resources allowed us to implement the design considerations discussed earlier, ultimately leading to a frontend that is both pleasing to the eye and practical to use.

4.2.1. Code structure

The frontend codebase for Dynamo Web Services, residing in the root directory, is composed of a systematic hierarchy of files and folders (see figure 4.4). This structured organization facilitates a clear understanding of the system's flow and contributes to the maintainability and scalability of the software.

At the root level of the frontend directory, known as 'kokyaku', we encounter the following significant elements:

- **node_modules**: This directory hosts the Node.js modules - the building blocks of the application, installed via npm (Node Package Manager). These are third-party modules that our application depends upon.
- **public**: This folder contains the static assets that are directly accessible to the public. These are files that do not undergo any kind of processing and are served as they are.

- **src**: The 'src' or 'source' folder is where the core application logic resides. It comprises Vue components, routes, stores, and other pertinent code files that define the application.

At the top level, the project also contains configuration files like '.editorconfig', '.eslintrc.cjs', 'jsconfig.json', 'postcss.config.js', 'tailwind.config.js', 'vite.config.js', as well as package handling files such as 'package-lock.json' and 'package.json'. Additionally, scripts for installing dependencies and executing the frontend ('install_frontend.sh', 'start_frontend.sh', 'stop_frontend.sh') are placed at this level.

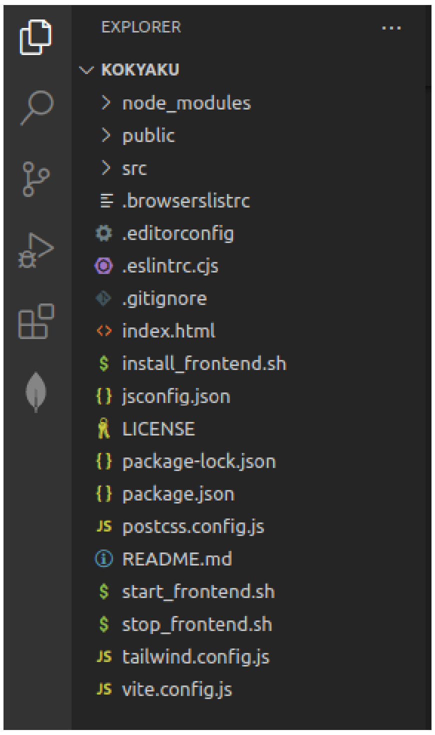

```
EXPLORER                ...
∨ KOKYAKU
  > node_modules
  > public
  > src
    .browserslistrc
    .editorconfig
    .eslintrc.cjs
    .gitignore
    index.html
    install_frontend.sh
    jsconfig.json
    LICENSE
    package-lock.json
    package.json
    postcss.config.js
    README.md
    start_frontend.sh
    stop_frontend.sh
    tailwind.config.js
    vite.config.js
```

Fig. 4.4. Frontend directory structure

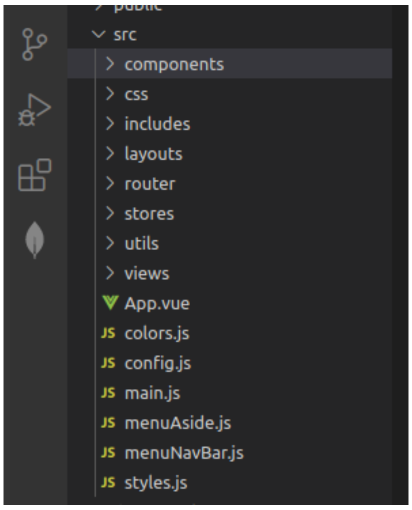
Delving into the 'src' directory, we find several key files and directories (see figure 4.5):

- **App.vue**: The root Vue component that serves as a starting point for our application.
- **colors.js, config.js, menuAside.js, menuNavBar.js, styles.js**: These files manage various aspects of the application, from colors and styles to configurations and menu structures.
- **main.js**: This is the entry point of our Vue application.

Subdirectories within the 'src' directory include:

- **components**: This folder contains reusable Vue components that make up the different parts of the application.
- **css**: All the style sheets related to the application are stored here.
- **includes**: This folder houses any included modules or libraries.

- **layouts**: This directory contains layout components that structure the application's UI.
- **router**: This is where we manage application routes using Vue Router.
- **stores**: Here we manage the application's state using Vuex.
- **views**: This folder contains Vue components that correspond to the different views or pages in the application.



```
> public
v src
  > components
  > css
  > includes
  > layouts
  > router
  > stores
  > utils
  > views
  App.vue
  colors.js
  config.js
  main.js
  menuAside.js
  menuNavBar.js
  styles.js
```

Fig. 4.5. Source directory

This structure forms the backbone of the frontend code for Dynamo Web Services, providing a comprehensive and organized map of our codebase that fosters efficient navigation and development.

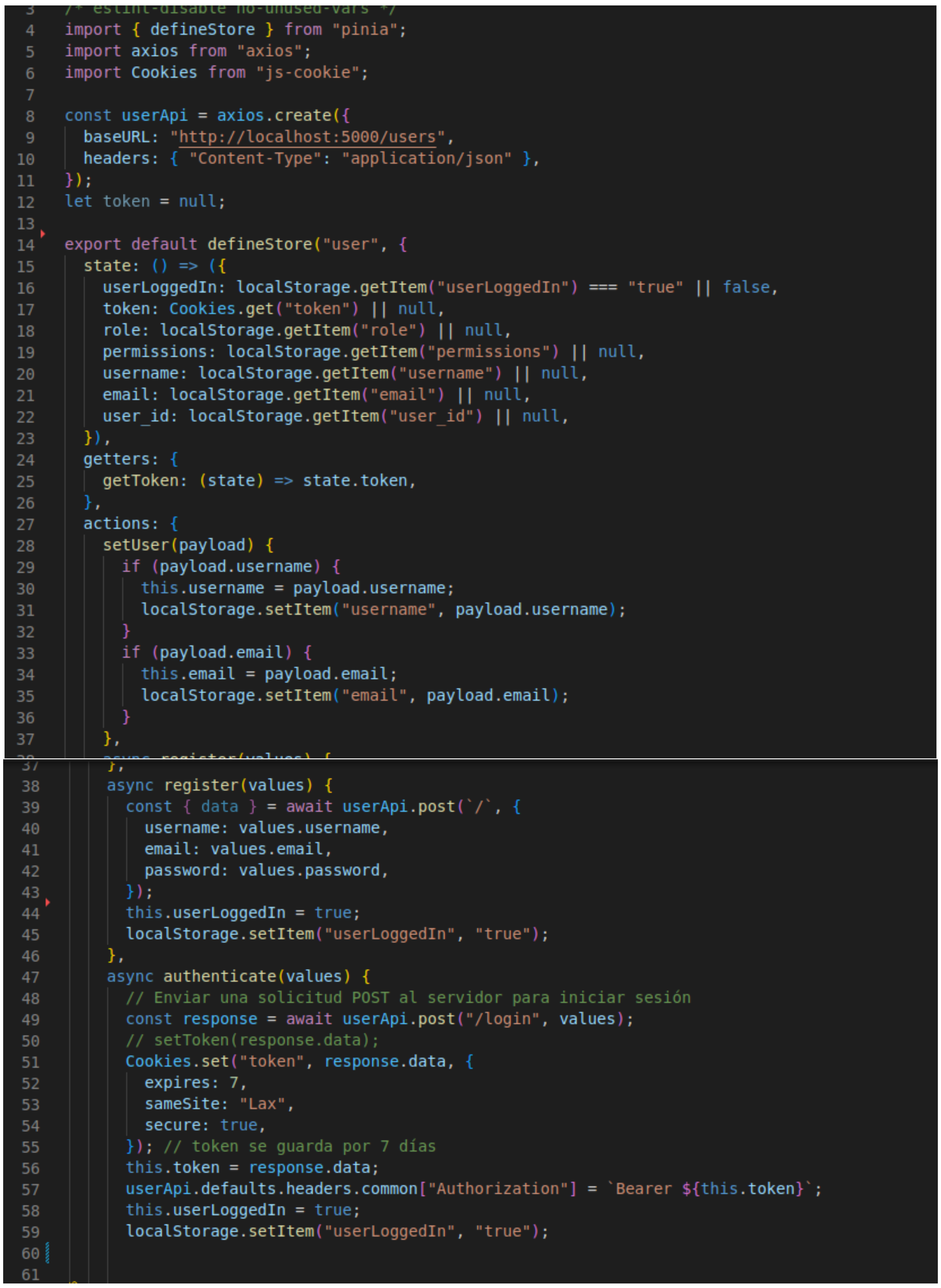
4.2.2. User Management System

4.2.2.1. User Authentication process

The User Management System is a crucial aspect of any web application requiring user registration and access control. Especially in the context of an API-driven application, the implementation of authentication mechanisms is vital to ensure that sensitive data is protected and that only authorized users have access to it.

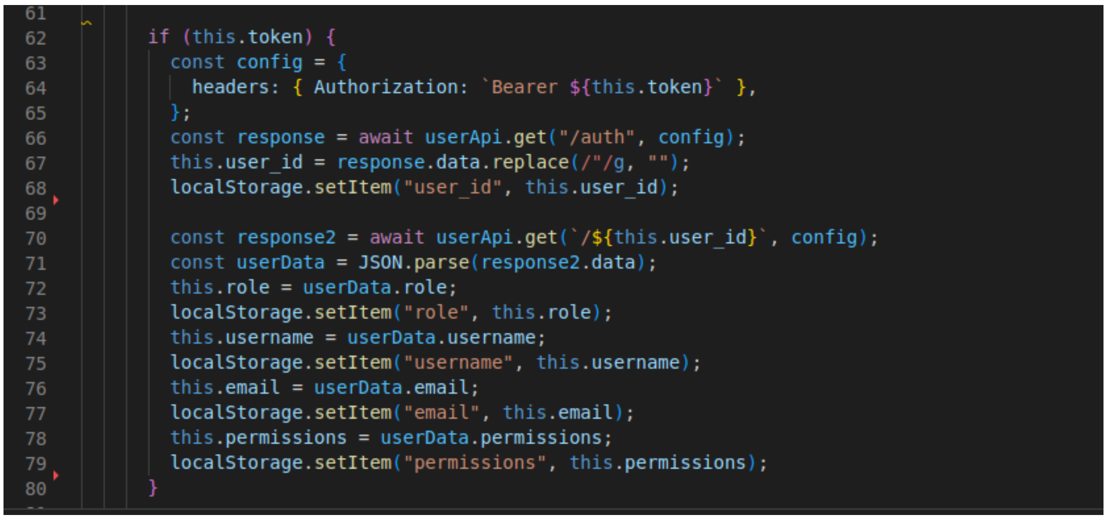
The entire process revolves around the principle of API authentication, which fundamentally involves verifying the identity of a user or application attempting to access the API. This typically involves the exchange of credentials, such as a username and password, for a unique identifier known as an access token. This token is then used to authorize the user for subsequent interactions with the API.

In our application, we have employed the "pinia" package to manage our state and "axios" for handling HTTP requests. To maintain persistence across sessions, we save the user's information in local storage and the user's token in cookies (see figures 4.6-4.7). This allows us to authenticate the user even when the page is refreshed, thus improving the user experience.



```
/* eslint-disable no-unused-vars */
import { defineStore } from "pinia";
import axios from "axios";
import Cookies from "js-cookie";

const userApi = axios.create({
  baseURL: "http://localhost:5000/users",
  headers: { "Content-Type": "application/json" },
});
let token = null;

export default defineStore("user", {
  state: () => ({
    userLoggedIn: localStorage.getItem("userLoggedIn") === "true" || false,
    token: Cookies.get("token") || null,
    role: localStorage.getItem("role") || null,
    permissions: localStorage.getItem("permissions") || null,
    username: localStorage.getItem("username") || null,
    email: localStorage.getItem("email") || null,
    user_id: localStorage.getItem("user_id") || null,
  }),
  getters: {
    getToken: (state) => state.token,
  },
  actions: {
    setUser(payload) {
      if (payload.username) {
        this.username = payload.username;
        localStorage.setItem("username", payload.username);
      }
      if (payload.email) {
        this.email = payload.email;
        localStorage.setItem("email", payload.email);
      }
    },
    async register(values) {
      const { data } = await userApi.post(`/`, {
        username: values.username,
        email: values.email,
        password: values.password,
      });
      this.userLoggedIn = true;
      localStorage.setItem("userLoggedIn", "true");
    },
    async authenticate(values) {
      // Enviar una solicitud POST al servidor para iniciar sesión
      const response = await userApi.post("/login", values);
      // setToken(response.data);
      Cookies.set("token", response.data, {
        expires: 7,
        sameSite: "Lax",
        secure: true,
      }); // token se guarda por 7 días
      this.token = response.data;
      userApi.defaults.headers.common["Authorization"] = `Bearer ${this.token}`;
      this.userLoggedIn = true;
      localStorage.setItem("userLoggedIn", "true");
```

Fig. 4.6. Frontend User store



```
      if (this.token) {
        const config = {
          headers: { Authorization: `Bearer ${this.token}` },
        };
        const response = await userApi.get("/auth", config);
        this.user_id = response.data.replace(/"/g, "");
        localStorage.setItem("user_id", this.user_id);

        const response2 = await userApi.get(`/${this.user_id}`, config);
        const userData = JSON.parse(response2.data);
        this.role = userData.role;
        localStorage.setItem("role", this.role);
        this.username = userData.username;
        localStorage.setItem("username", this.username);
        this.email = userData.email;
        localStorage.setItem("email", this.email);
        this.permissions = userData.permissions;
        localStorage.setItem("permissions", this.permissions);
      }
```

Fig. 4.7. Frontend User store. Part 2

The "axios" library simplifies making requests to the API. We create an instance of axios, userApi, which is configured to communicate with our user management backend service, as shown by the base URL of `http://localhost:5000/users`.

Our user store incorporates several state variables to hold user data, such as username, email, role, permissions, user_id, and token. Most of this data is stored in local storage to preserve the user's login state across multiple sessions. The token, which is received from the server upon successful login, is stored as a cookie. This token is included in the Authorization header of subsequent axios requests to the backend.

The actions **register** and **authenticate** make POST requests to the backend for registering a new user and logging in an existing user, respectively. Upon successful login, the backend returns a **JWT token** which is stored in a cookie. This token is also set as the default Authorization header for **userApi** axios instance, thus facilitating secure communication with the backend for subsequent requests.

The **signOut** action clears all stored user data from local storage and removes the JWT token from the cookie. Additionally, it removes the Authorization header from userApi axios instance, effectively logging out the user from the application.

JWT (JSON Web Tokens) tokens are used for authentication and user management. JWT is a standard for securely transmitting information between parties as a JSON object. It consists of three parts: a header, a payload, and a signature.

The header contains information about the type of token and the algorithm used for signing it. The payload contains the claims, which are statements about an entity (typically, the user) and additional metadata. The signature is used to verify the message wasn't changed along the way and, in the case of tokens signed with a private key, to verify that the sender of the JWT is who it says it is and to ensure the message wasn't tampered with.

So, here a small schema (see figure 4.8) of how the procedure will be, being the browser our Vue client:

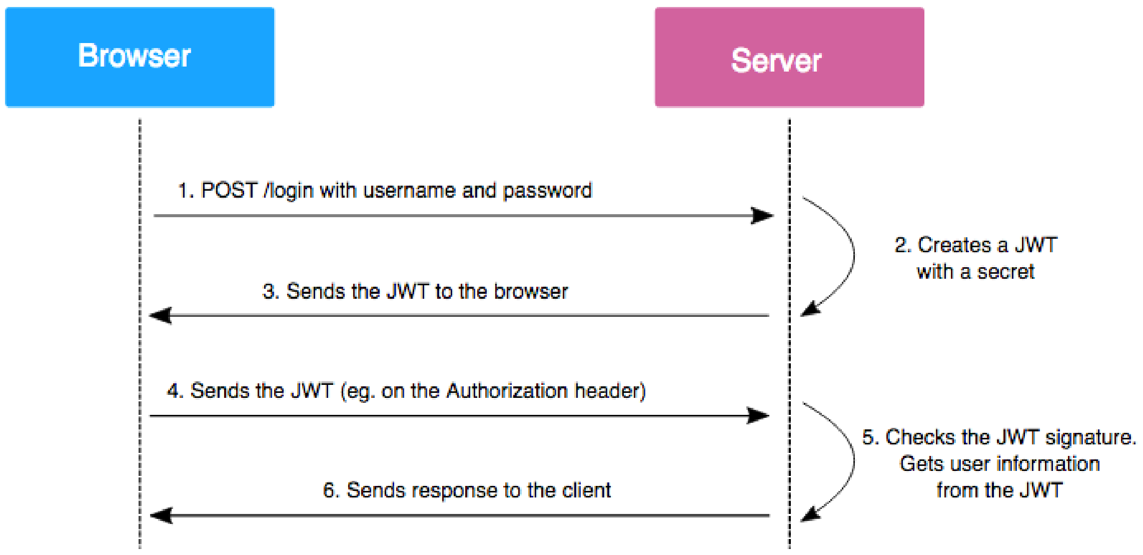
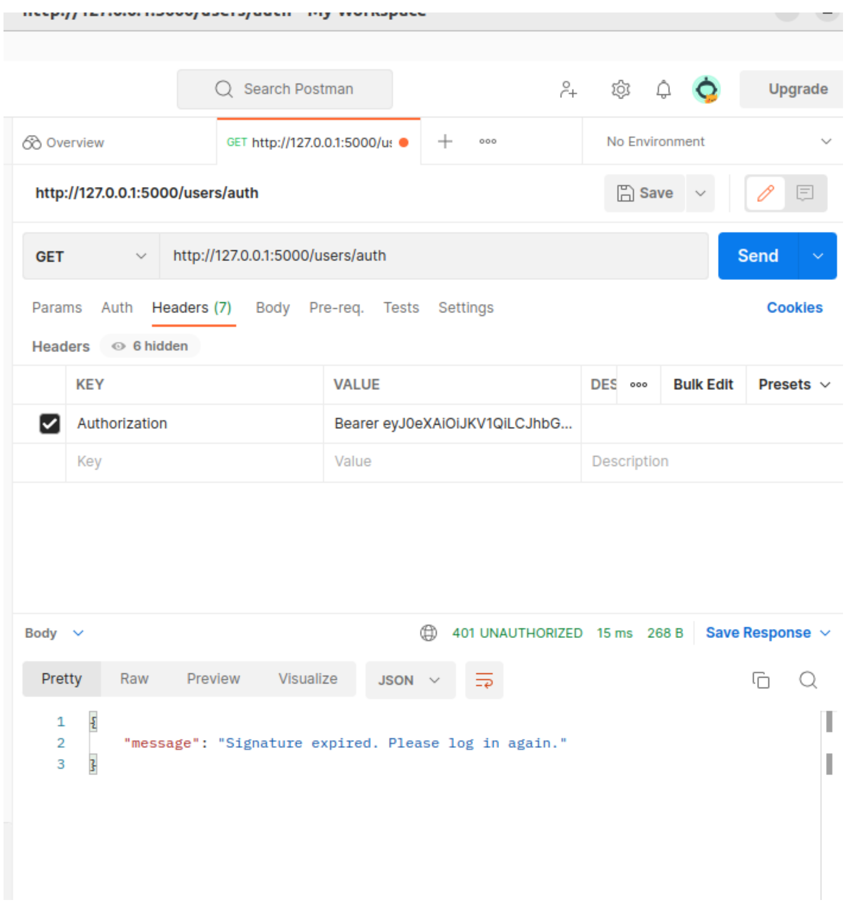

Browser
Server
1. POST /login with username and password
2. Creates a JWT with a secret
3. Sends the JWT to the browser
4. Sends the JWT (eg. on the Authorization header)
5. Checks the JWT signature. Gets user information from the JWT
6. Sends response to the client

Fig. 4.8. Browser-Server communication schema

The user only can access our website with a valid token. To verify whether a user is authenticated correctly, we have a route in the API that can be accessed by sending a request with the token inserted in the authentication header.

When the token is invalid or has expired (see figure 4.9), this information reaches the Frontend and then the user will not be able to access any more of the application's functionalities. If the token has expired, the user should log in again with their correct credentials to regenerate a valid token. If the token is invalid, surely the user has entered his credentials incorrectly or does not exist in the database, therefore he will not be able to access the web page.



http://127.0.0.1:5000/users/auth

GET http://127.0.0.1:5000/users/auth

Headers 6 hidden

	KEY	VALUE	DESCRIPTION
✓	Authorization	Bearer eyJ0eXAiOiJKV1QiLCJhbG...	
	Key	Value	Description

401 UNAUTHORIZED 15 ms 268 B

```
{
    "message": "Signature expired. Please log in again."
}
```

Fig. 4.9. Response when token is expired

To sum up, our User Management System, developed with the help of modern libraries such as axios, and pinia, ensures the persistent and secure handling of user data. Moreover, by efficiently integrating frontend and backend through the use of JWT tokens, we achieve seamless data transmission, thereby providing a smooth and secure user experience.

4.1.1.1. Form Handling and User Profile

Moving on to the user interaction part of our application, here we will see form handling and user profile management.

The forms in our application are not just simple input fields. They are intuitive, user-friendly, and designed to ensure the integrity of the information provided by the user. To achieve this, the forms implement various levels of error handling and data validation. Fields that are required to create a new user or to authenticate an existing one are marked as 'required', and the form submission process ensures these fields are filled before proceeding. If a user attempts to submit a form with empty required fields, an error message will be displayed, and the form will not be submitted (see figure 4.10). This handling of form errors helps guide the users through the process, reduces the likelihood of failed submissions, and enhances overall user experience.

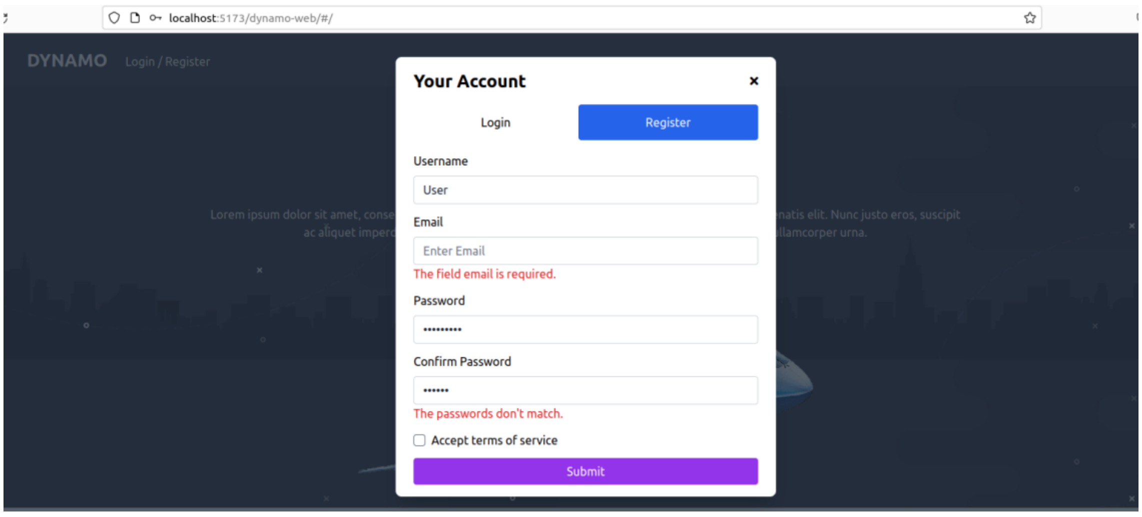


Fig. 4.10. Register form

Additionally, there are checks in place to prevent the creation of multiple accounts with the same username. When a user attempts to register with a username that is already taken, an error message is triggered (see figure 4.11). This preemptive error management ensures each user has a unique identity in the system and minimizes potential confusion or data discrepancies down the line.

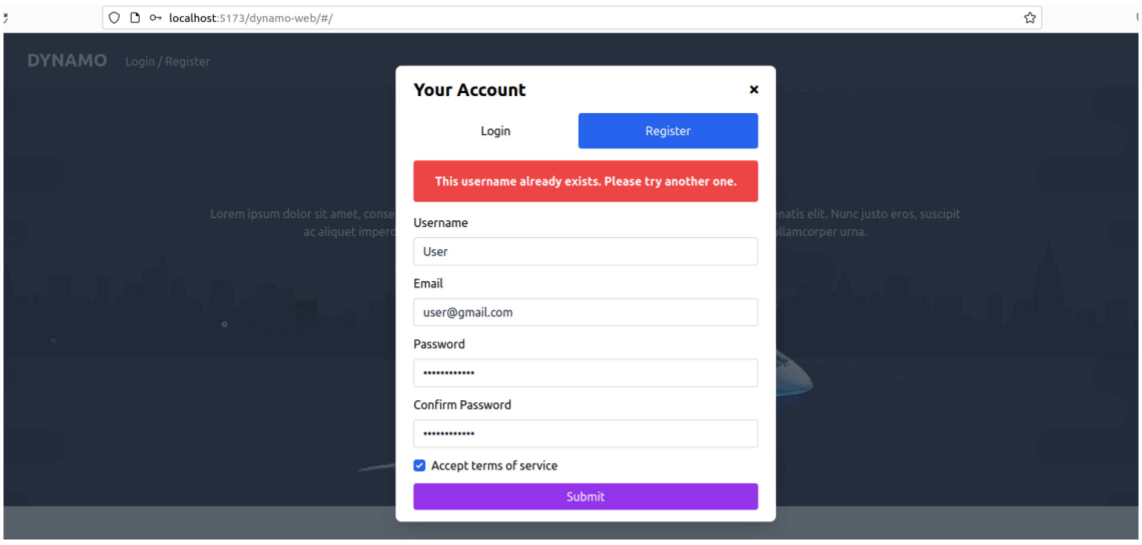
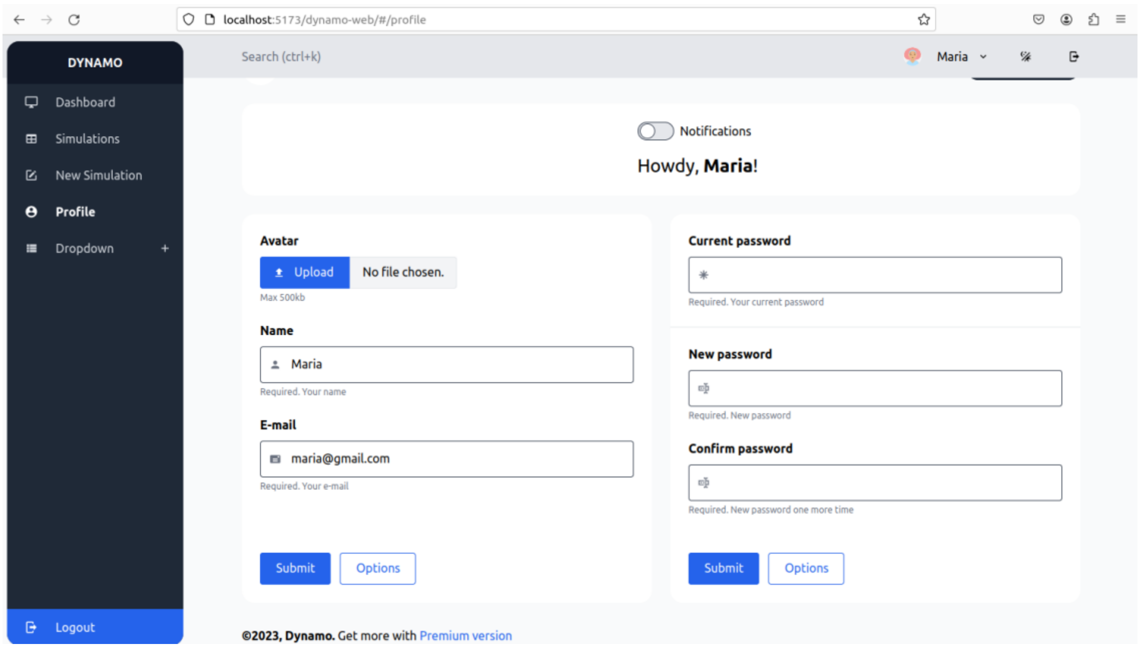


Fig. 4.11. Login form

A User Profile section is also available to users (see figure 4.12), where they can view their personal data stored in the application, such as their username and email. This data is pulled from local storage, where it was securely stored during the authentication process. The user profile offers a personalized experience for users and promotes transparency by letting them view the data associated with their account. There, users can also change or update their username or email.



Howdy, **Maria**!

Avatar
No file chosen.
Max 500kb

Name
Maria
Required. Your name

E-mail
maria@gmail.com
Required. Your e-mail

Current password
Required. Your current password

New password
Required. New password

Confirm password
Required. New password one more time

©2023, Dynamo. Get more with Premium version

Fig. 4.12. Profile page

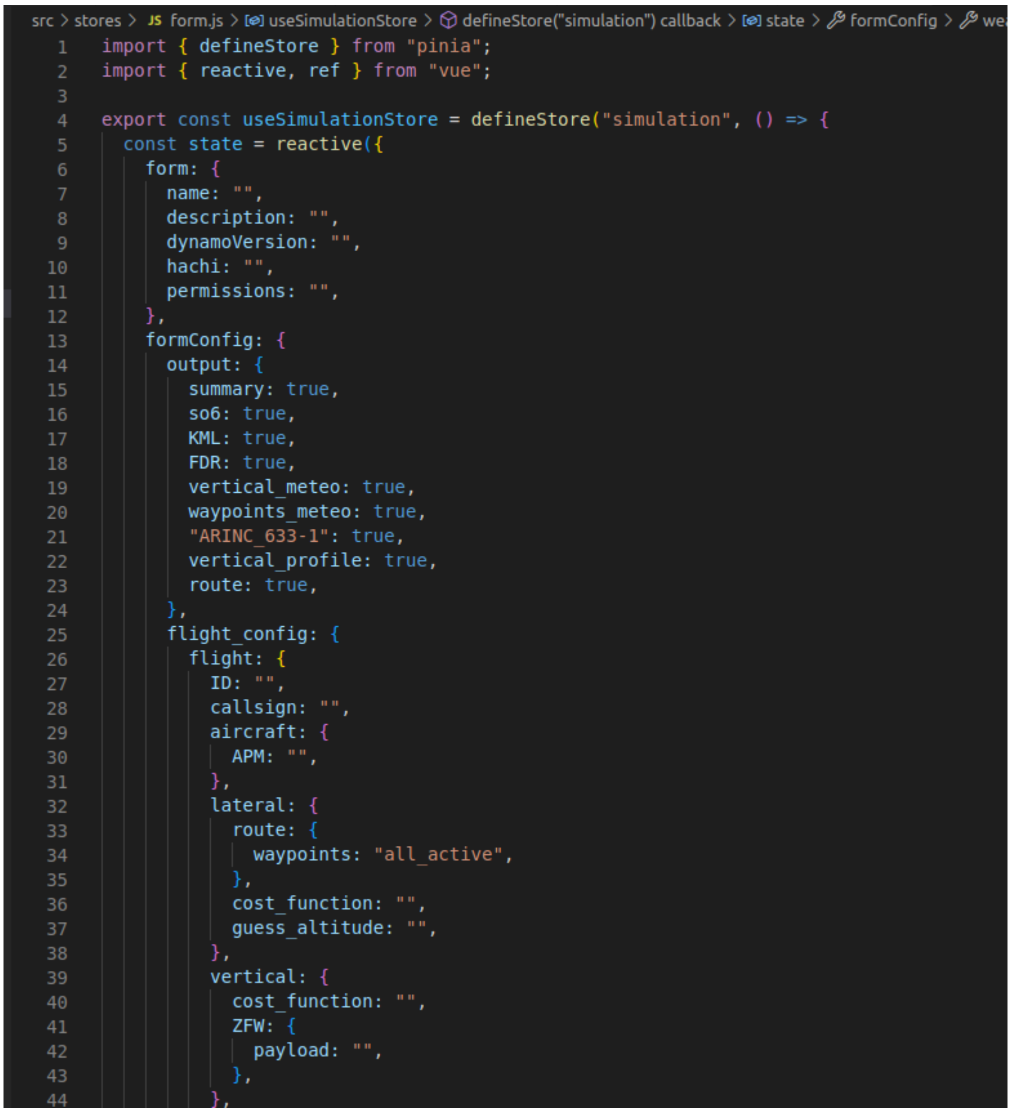
By integrating intuitive forms with thorough error handling and a comprehensive user profile section, our frontend not only ensures smooth user interaction but also adheres to best practices in user data management. Up next, we will explore another essential aspect of our frontend application, the Simulation Management System.

4.2.3. Simulation Management System

4.2.3.1. Code

Given the comprehensive nature of the frontend code for the Simulation Management System, only a specific segment is highlighted here to explain the core logic and behavior. This particular segment focuses on how the system handles the state management of simulation data. The full scope of the frontend code extends beyond this, encompassing the creation and behavior of various interface elements such as buttons, tables, images, and more.

This system employs a well-structured state management pattern, leveraging the capabilities of Vuex - a state management library tailored for Vue.js applications. Utilizing the defineStore function from Pinia, a store has been created (see figure 4.13) where the state of the simulation form and its configuration data is maintained.



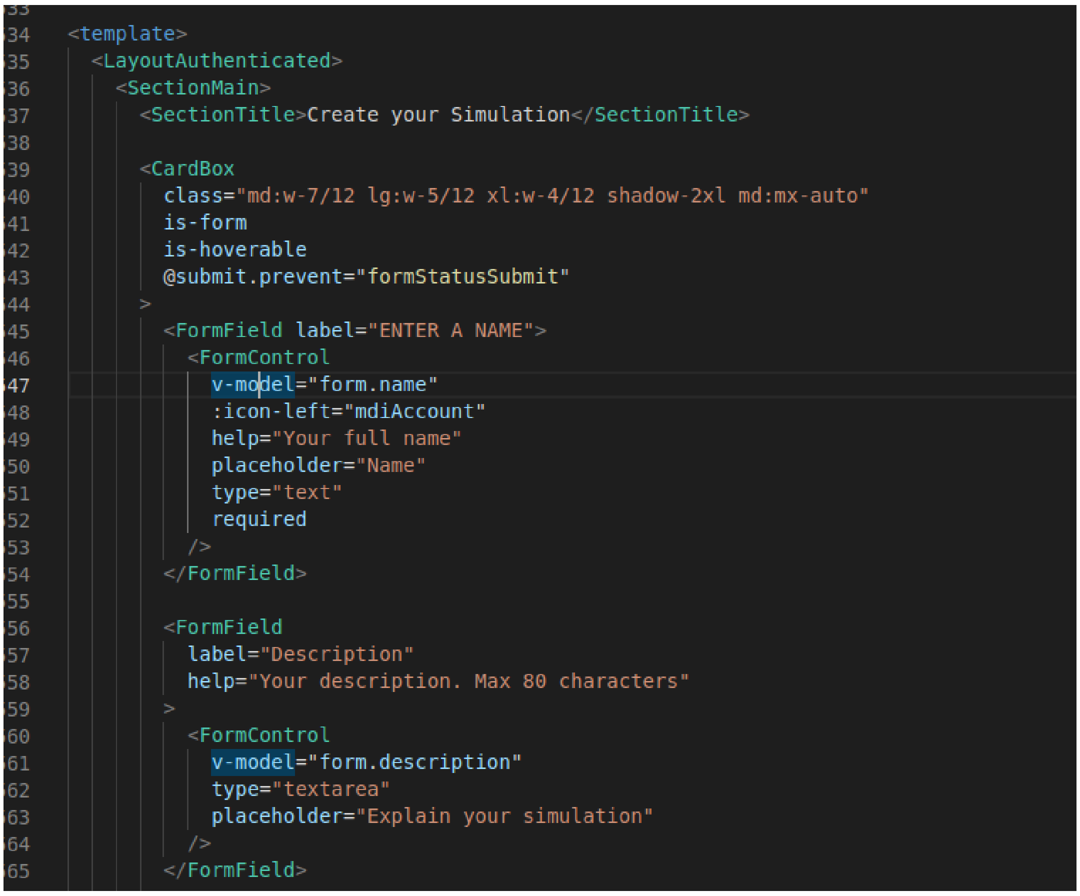
src > stores > JS form.js > useSimulationStore > defineStore("simulation") callback > state > formConfig > we

```
import { defineStore } from "pinia";
import { reactive, ref } from "vue";

export const useSimulationStore = defineStore("simulation", () => {
  const state = reactive({
    form: {
      name: "",
      description: "",
      dynamoVersion: "",
      hachi: "",
      permissions: "",
    },
    formConfig: {
      output: {
        summary: true,
        so6: true,
        KML: true,
        FDR: true,
        vertical_meteo: true,
        waypoints_meteo: true,
        "ARINC_633-1": true,
        vertical_profile: true,
        route: true,
      },
      flight_config: {
        flight: {
          ID: "",
          callsign: "",
          aircraft: {
            APM: "",
          },
          lateral: {
            route: {
              waypoints: "all_active",
            },
            cost_function: "",
            guess_altitude: "",
          },
          vertical: {
            cost_function: "",
            ZFW: {
              payload: "",
            },
          },
```

Fig. 4.13. Simulation store

This state includes a form object which stores general attributes such as the simulation's name, description, selected Dynamo version, and other related properties. Apart from this, there is a formConfig object that encapsulates a more intricate configuration setup for the simulation. This object is divided into multiple sections, each detailing a particular aspect of the simulation configuration such as flight_config for flight specifics, ATM_config for ATM details, and more.

On the Vue component responsible for the creation of the forms (see figure 4.14), two-way data binding is facilitated between the store state properties and form input fields using the **v-model** directive. This ensures a synchronous update between the Vuex store state and form fields, providing an efficient and consistent way to track changes. Whenever a field's value is updated, it gets reflected in the Vuex store state and vice versa. This mechanism, thus, ensures that the state of the simulation form remains synchronized with the Vuex store and can be accessed uniformly across the application.



```
<template>
  <LayoutAuthenticated>
    <SectionMain>
      <SectionTitle>Create your Simulation</SectionTitle>

      <CardBox
        class="md:w-7/12 lg:w-5/12 xl:w-4/12 shadow-2xl md:mx-auto"
        is-form
        is-hoverable
        @submit.prevent="formStatusSubmit"
      >
        <FormField label="ENTER A NAME">
          <FormControl
            v-model="form.name"
            :icon-left="mdiAccount"
            help="Your full name"
            placeholder="Name"
            type="text"
            required
          />
        </FormField>

        <FormField
          label="Description"
          help="Your description. Max 80 characters"
        >
          <FormControl
            v-model="form.description"
            type="textarea"
            placeholder="Explain your simulation"
          />
        </FormField>
```

Fig. 4.14. Part of the template for the New Simulation Page

4.2.2.2. *Simulation interface*

The Simulation Management System is a key component of our frontend application. It provides users with an intuitive and flexible interface to create and manage their simulations.

Upon navigating to the 'Create Simulation' page, users are presented with an initial form that gathers basic information about the upcoming simulation (see figure 4.15). Users are asked to provide a unique name for the simulation, a brief description, the Dynamo version to be used, and a choice between executing 'c3po' or 'r2d2'.

Once the user clicks the 'Create' button, a series of forms are displayed for configuring the simulation. The number and type of forms depend on the user's previous selections. If the user opted for 'r2d2', an additional block appears to input 'Initial Conditions'. Each form is structured to align with the structure of the XML configuration files, making the transition between form-based configuration and file-based configuration smoother for advanced users.

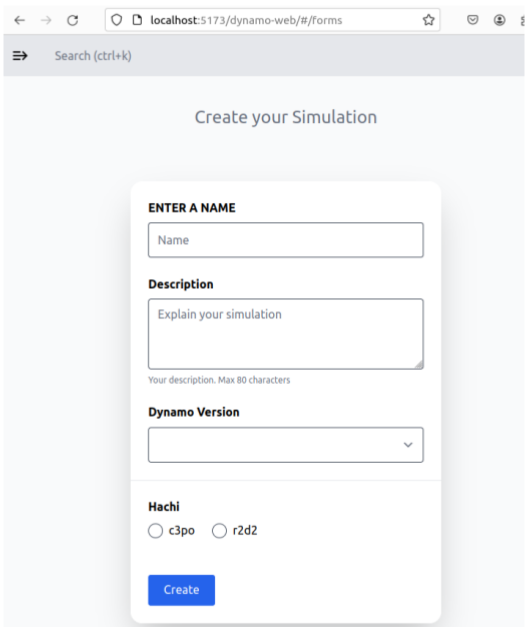
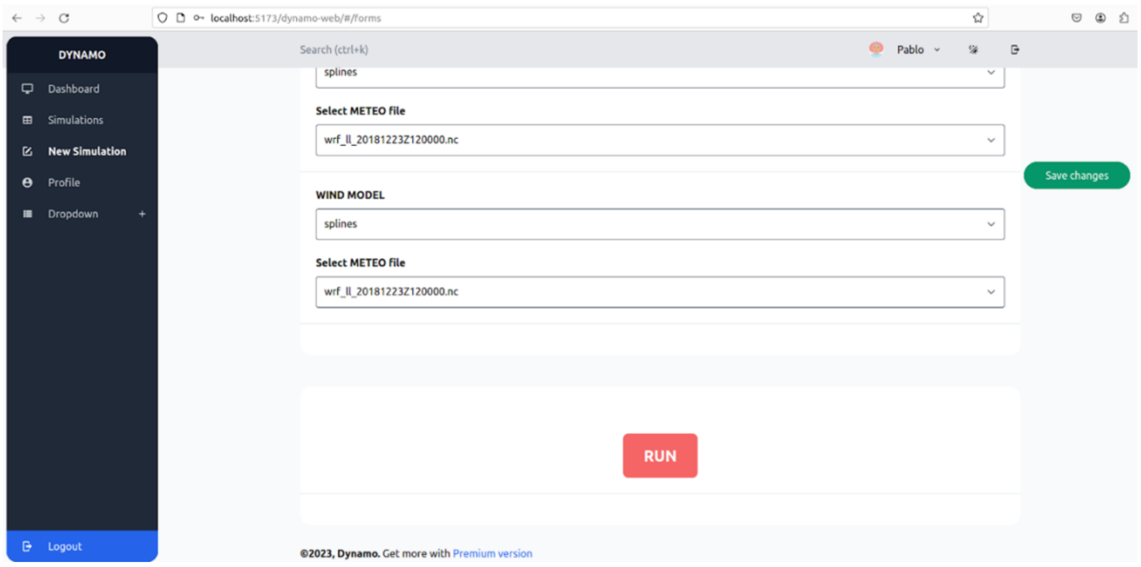


Fig. 4.15. Initial form configuration

In this setup, users are allowed to save their progress and return to it later. This allows for flexibility and convenience - users can work on setting up their simulations in multiple sessions without losing previous configurations. Once all forms have been filled, the user can then click 'RUN' to execute the simulation (see figure 4.16).



splines

Select METEO file
wrf_ll_20181223Z120000.nc

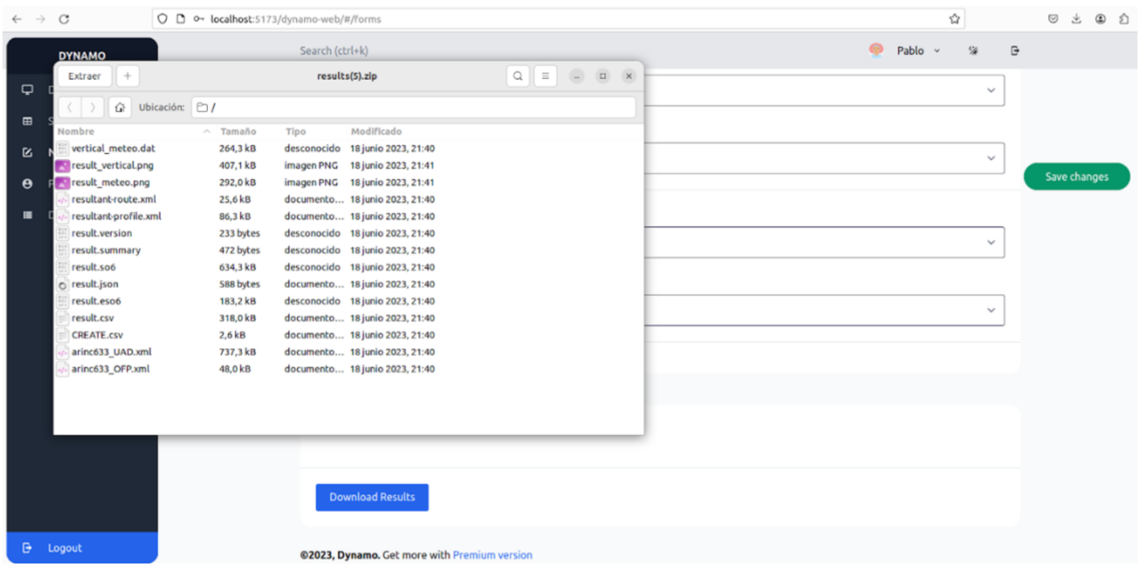
WIND MODEL
splines

Select METEO file
wrf_ll_20181223Z120000.nc

Fig. 4.16. End of the configuration page

In our user-centric design, we have also incorporated the option for users to select the outputs they wish to receive in the configuration forms. This ensures that the simulation generates only the desired data, tailored to the specific needs and preferences of each user. With this feature, users have complete control over the output of their simulations, ensuring they receive only the most relevant data for their requirements.

Certainly, an additional feature of the Simulation Management System is the ability to download the results once the simulation is complete (see figure 4.17). This feature provides the user with a comprehensive set of data that includes all the generated files such as XML, CSV, and others, encapsulating all the outputs from the executed simulation.

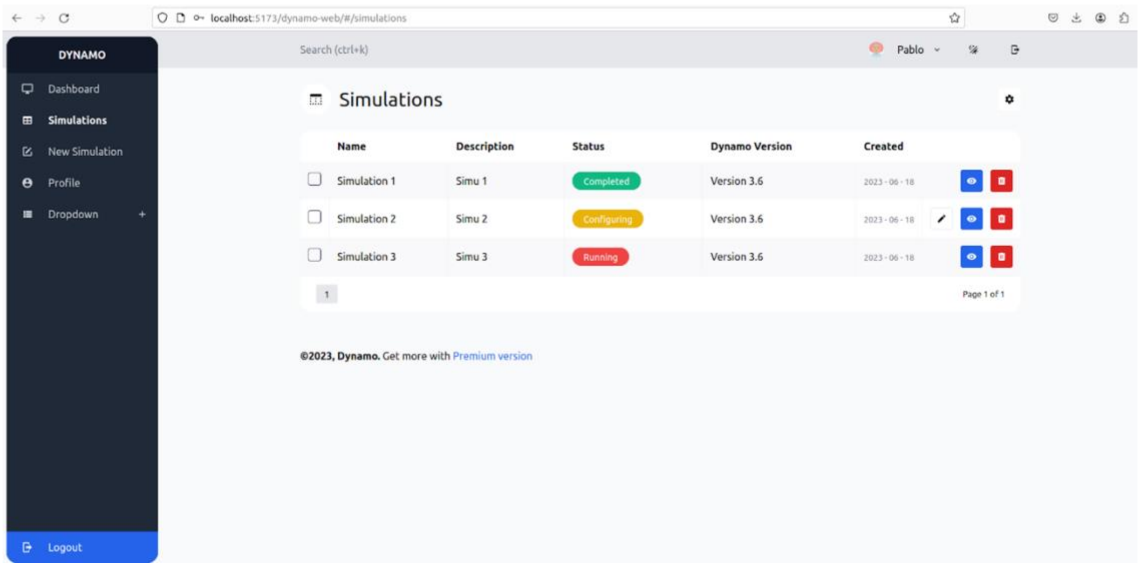


Nombre	Tamaño	Tipo	Modificado
vertical_meteo.dat	264,3 kB	desconocido	18 junio 2023, 21:40
result_vertical.png	407,1 kB	imagen PNG	18 junio 2023, 21:41
result_meteo.png	292,0 kB	imagen PNG	18 junio 2023, 21:41
resultant-route.xml	25,6 kB	documento...	18 junio 2023, 21:40
resultant-profile.xml	86,3 kB	documento...	18 junio 2023, 21:40
result.version	233 bytes	desconocido	18 junio 2023, 21:40
result.summary	472 bytes	desconocido	18 junio 2023, 21:40
result.so6	634,3 kB	desconocido	18 junio 2023, 21:40
result.json	588 bytes	documento...	18 junio 2023, 21:40
result.eso6	183,2 kB	desconocido	18 junio 2023, 21:40
result.csv	318,0 kB	documento...	18 junio 2023, 21:40
CREATE.csv	2,6 kB	documento...	18 junio 2023, 21:40
arinc633_UAD.xml	737,3 kB	documento...	18 junio 2023, 21:40
arinc633_OFP.xml	48,0 kB	documento...	18 junio 2023, 21:40

Fig. 4.17. Download of the results of a simulation

Also, an integral part of the Simulation Management System is the ability to track and manage your simulations. After initiating a simulation, or even just configuring one, you can navigate to the 'Simulations' page. This page provides an overview of all your simulations and their current state (figure 4.18).

There are three primary status a simulation can be in: ‘completed’, 'configuring', and 'running'. For simulations marked as 'Complete', you can access the results by clicking on the eye button. Once in the results page, users can view the PNG files that represent graphics generated by Dynamo (see figure 4.19). However, it's important to note that other file types are only accessible when you download the full results.



Simulations

	Name	Description	Status	Dynamo Version	Created
☐	Simulation 1	Simu 1	Completed	Version 3.6	2023 - 06 - 18
☐	Simulation 2	Simu 2	Configuring	Version 3.6	2023 - 06 - 18
☐	Simulation 3	Simu 3	Running	Version 3.6	2023 - 06 - 18

1 Page 1 of 1

©2023, Dynamo. Get more with Premium version

Fig. 4.18. Table with all the simulations of the user

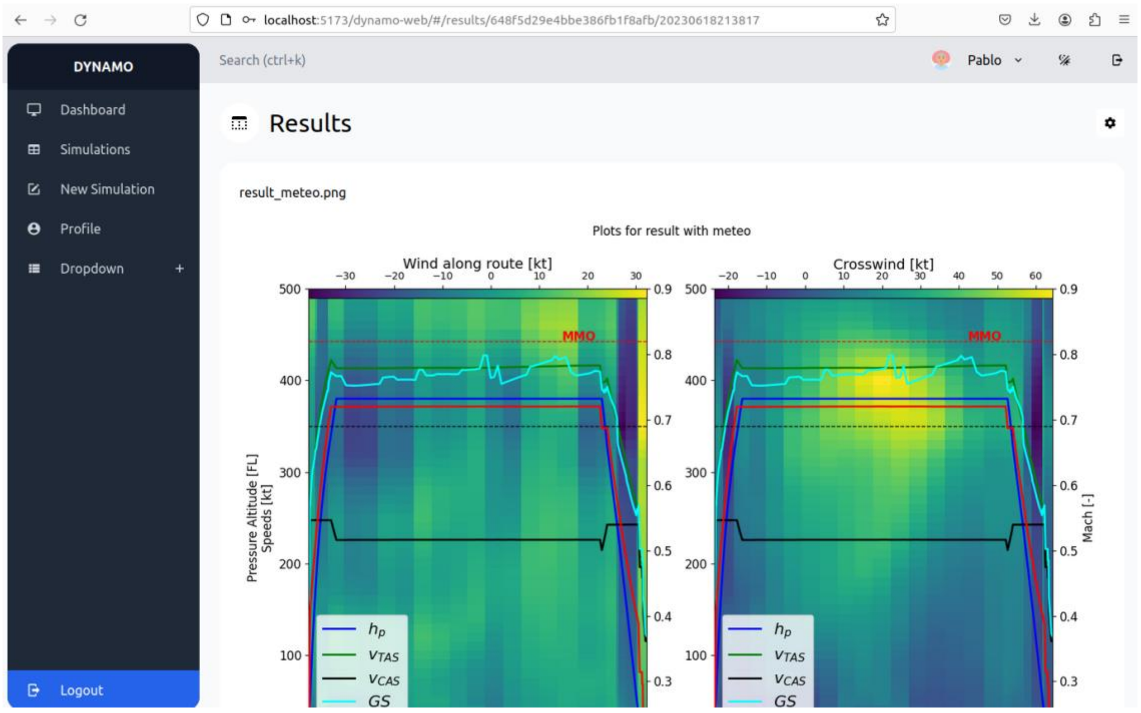
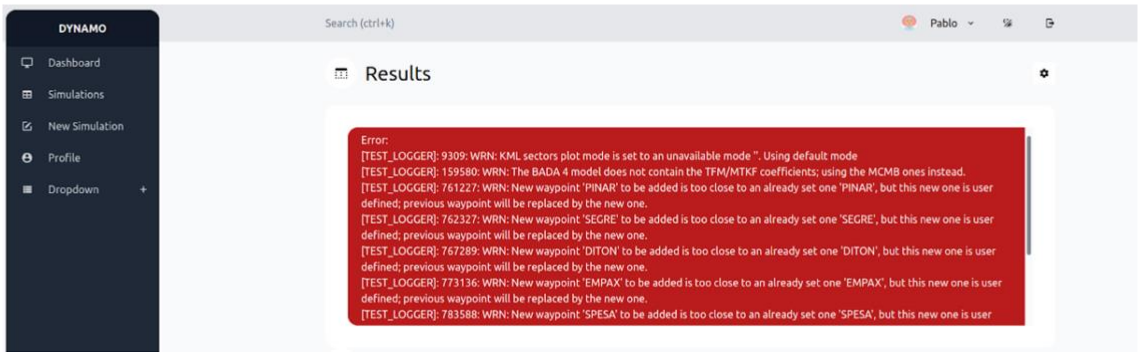


Fig. 4.19. Results page

For those simulations still in the 'Configuring' state, clicking on the pencil button takes you back to the configuration page. The state of this page reflects the exact progress you made while setting up the simulation, with all the previously filled fields preserved. This feature allows you to continue from where you left off, making the process more user-friendly and efficient.

Additionally, an extra feature has been implemented to enhance user experience and understanding in case of failed simulations. Once a simulation is completed, if something has gone wrong, users can navigate to the results section. Here, they will be presented with a comprehensive error message generated by the Dynamo software (see figure 4.20). This feature ensures transparency, allowing users to understand the issues that might have occurred during the simulation.



```
Error:
[TEST_LOGGER]: 9309: WRN: KML sectors plot mode is set to an unavailable mode ''. Using default mode
[TEST_LOGGER]: 159580: WRN: The BADA 4 model does not contain the TFM/MTKF coefficients; using the MCMB ones instead.
[TEST_LOGGER]: 761227: WRN: New waypoint 'PINAR' to be added is too close to an already set one 'PINAR', but this new one is user
defined; previous waypoint will be replaced by the new one.
[TEST_LOGGER]: 762327: WRN: New waypoint 'SEGRE' to be added is too close to an already set one 'SEGRE', but this new one is user
defined; previous waypoint will be replaced by the new one.
[TEST_LOGGER]: 767289: WRN: New waypoint 'DITON' to be added is too close to an already set one 'DITON', but this new one is user
defined; previous waypoint will be replaced by the new one.
[TEST_LOGGER]: 773136: WRN: New waypoint 'EMPAX' to be added is too close to an already set one 'EMPAX', but this new one is user
defined; previous waypoint will be replaced by the new one.
[TEST_LOGGER]: 783588: WRN: New waypoint 'SPESA' to be added is too close to an already set one 'SPESA', but this new one is user
```

Fig. 4.20. Results page with errors

Our Simulation Management System is designed to make the process of creating and managing simulations straightforward and user-friendly, regardless of the user's level of experience with the underlying technologies. This commitment to usability, combined with flexibility in configuration options, enhances the user experience, and facilitates efficient use of the simulations.

4.2.2.3. Simulation interface for advanced users

Advanced users have additional options at their disposal. In the initial form when creating a simulation, they are presented with the option to configure the simulation via forms or to directly upload their configuration file (see figure 4.21). This advanced feature provides a higher level of customization, enabling experienced users to exercise precise control over their simulation setup.

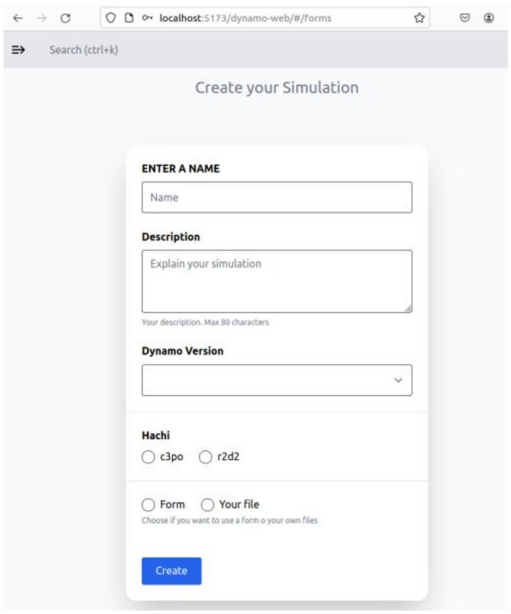


Fig. 4.21. Initial form configuration for advanced users

On choosing to upload their configuration file, users are directed to a page where they can upload their config.xml file as well as any other supporting files such as weather, graph, or restricted sectors files (see figure 4.22). This page represents a core feature of the interface for advanced users, allowing them to bypass the standard form-based configuration and use their pre-prepared configuration files instead.

When the advanced user selects the RUN button in this page, the system will accept the uploaded files and begin the simulation process. The resulting workflow ensures a quick and efficient process for experienced users, while still preserving all the comprehensive features of the Simulation Management System.

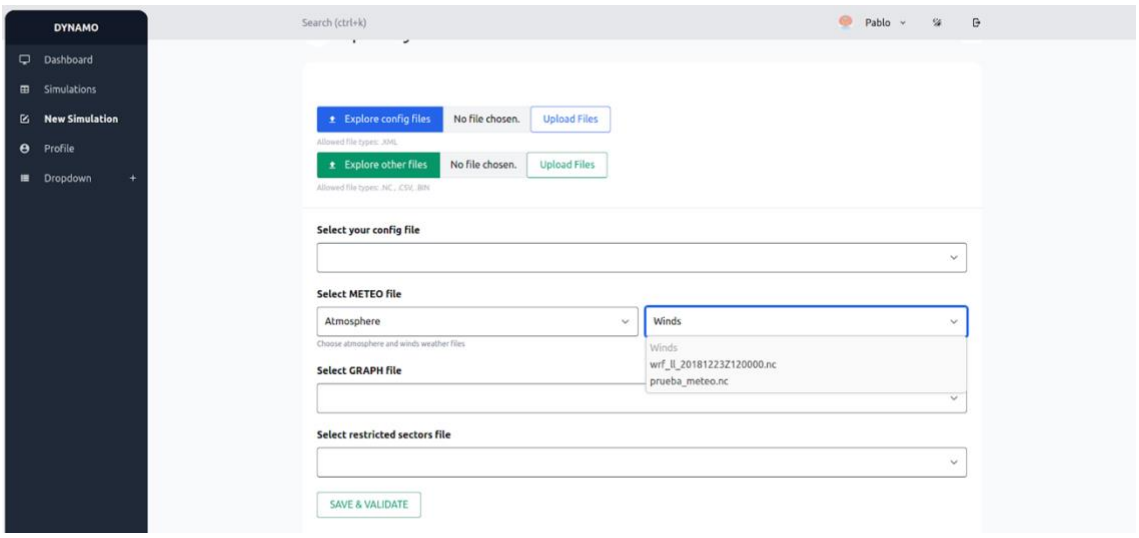


Fig. 4.22. Upload files for advanced users

CHAPTER 5. TEST AND RESULTS

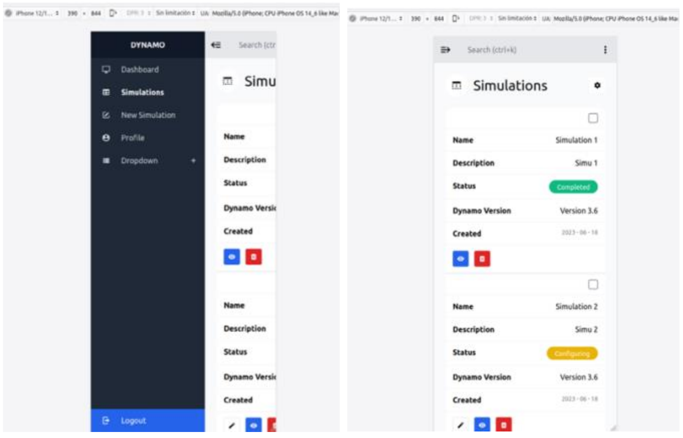
This chapter provides a detailed analysis of the final results and performance of the system, following the completion of both the backend and frontend development. It outlines the significant outcomes of our development efforts and offers a thorough review of the finished system, focusing on the user interface and experience, responsiveness across various devices, and user-customizable features. Additionally, this chapter discusses the system's alignment with the initial requirements and highlights some of the challenges and bugs encountered during the development process.

5.1. System outcomes

In this chapter, we delve into the culmination of all the aforementioned development efforts and discuss the final outcomes of the system implementation, while also addressing the system's adaptability for different devices and screen sizes. In addition, we present the incorporation of user interface customizations such as a 'dark mode' and explore some of the challenges and bugs encountered throughout the development process.

Upon successful completion of the frontend and backend development, the Simulation Management System offers a comprehensive dashboard to facilitate the creation and management of simulations. The dashboard provides a seamless and intuitive user experience, empowering users to configure simulations, execute them, and examine the results all within a unified platform.

As a modern web application, a considerable emphasis has been placed on ensuring that the interface is responsive, meaning that it adapts to different screen sizes for optimal viewing and interaction (see figure 5.1). Whether accessed from a desktop, a tablet, or a mobile device, the Simulation Management System maintains a user-friendly interface and consistent functionality across all devices.



Name	Simulation 1
Description	Simu 1
Status	Completed
Dynamo Version	Version 3.6
Created	2023 - 06 - 18
Name	Simulation 2
Description	Simu 2
Status	Configuring
Dynamo Version	Version 3.6
Created	2023 - 06 - 18

Fig. 5.1. Responsive web

Another aspect of the interface design is the provision for theme customization, specifically, a 'dark mode' (see figure 5.2). This feature, increasingly popular in contemporary digital platforms, offers an alternative color scheme that can improve visual comfort for users in low-light conditions or those who simply prefer a darker aesthetic for their applications.

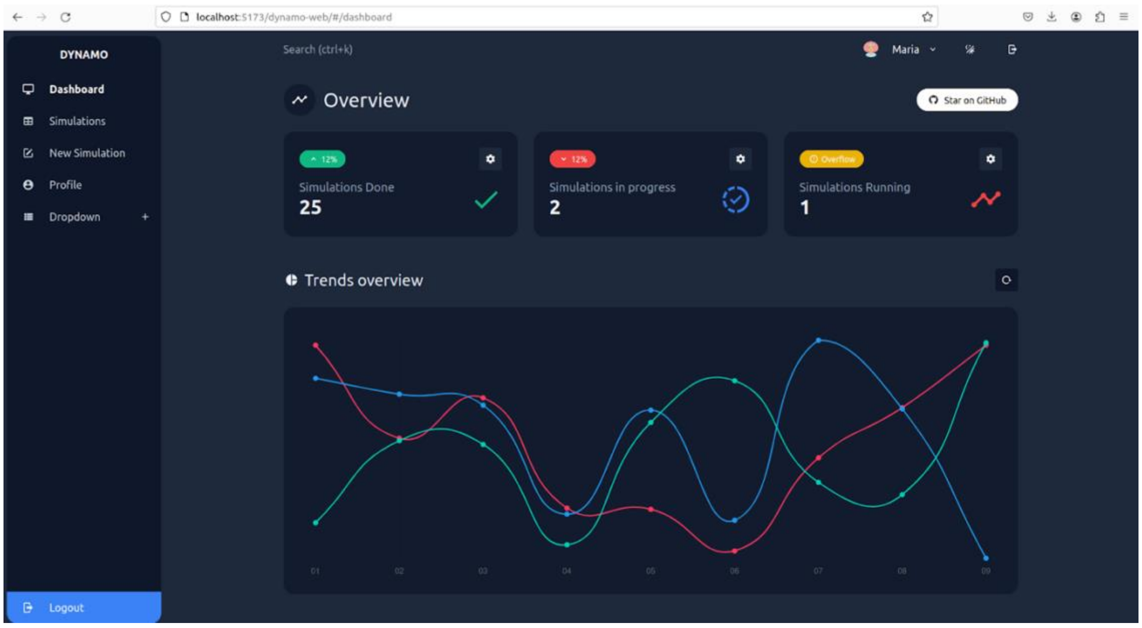


Fig. 5.2. Dark mode dashboard

In the development process, we aimed to strike a balance between functional requirements, user interface design, system responsiveness, and customization features. The outcomes underline the success of the approach taken, with a system that is not only efficient and functional but also adaptable and user centric.

5.2. Fulfilled requirements

During the course of this project, the primary focus was to address and fulfill as many requirements as were initially established during our requirements gathering sessions. It is crucial to mention that while our ultimate goal was to meet all of these requirements, some challenges and time constraints inevitably arose that prevented us from achieving this goal in its entirety. However, significant strides were made towards satisfying the majority of the stipulated requirements, resulting in a product that we believe encapsulates the key objectives initially set out for the Dynamo Web Services platform.

Here is a summary of the requirements fulfilled:

- One of the essential requirements we managed to fulfill was the authentication process for users. It was crucial for us to ensure that each

user had to register and log in to gain access to the system. By doing so, we managed to establish a secure environment, one that provides a personalized and unique user experience.
- Another core requirement that was effectively addressed was the ability for users to create, manage, and save their simulations. This feature was designed with the goal to allow users to save their progress and resume their work at their convenience, providing an optimal balance between flexibility and functionality.
- We have implemented different permissions. For advanced users, the option to upload files directly was implemented successfully. This function bypasses the necessity of configuring simulations through form-based inputs, marking a significant accomplishment towards our aim to cater to diverse user needs and preferences.
- The Dynamo Web Services platform was also designed to be fully responsive.
- Is possible for users to select the simulation outputs they desire to view.
- The system is designed to notify users of any errors during simulations (showing the log errors if something fails), and they can visually track the status of their simulations, fostering transparency and user control.

At this point, it is important to note that the above statements provide a high-level summary of what has been achieved. The preceding sections in this document offer a more in-depth look at how these accomplishments were realized and the challenges encountered along the way.

While we have managed to fulfill a considerable number of the initial requirements, there were a few that we could not accomplish within the project timeframe:

- Notification of completed simulations via email.
- Incorporation of a 'Help' or 'Tutorial' section within the platform
- META/BATCH modes (list of flights)
- Route with Google Earth
- Interactive graphs
- Different types of licenses
- Website available in several languages

Despite the unfulfilled requirements, it is essential to emphasize the current state of the Dynamo Web Services platform: a functional and an effective tool. The platform, as it stands, significantly eases user interaction with the Dynamo software, enabling trajectory simulation management with a high degree of accessibility and intuitiveness.

While the initial project vision was ambitious, the unmet requirements in no way undermine the value or usability of the system. The platform is already providing substantial assistance to its users, thereby fulfilling its core mission.

5.3. System bugs and issues

The current implementation of the Dynamo Web Services platform, while operational and efficient, has demonstrated some minor issues during its testing phase. It is important to note that these detected bugs do not critically impact the application's performance or its overall usability. Instead, they reflect the natural process of continuous refinement that is inherent to any software development project. Below are the most significant bugs or issues identified, alongside proposed improvements for future development iterations.

1. **User Registration Flow**: Currently, when a user completes the registration process, they are not immediately redirected to the dashboard, but instead, they are taken to the login form. For an enhanced user experience, we propose an improvement where upon successful registration, users are directly redirected to the dashboard. This approach removes the need for users to manually navigate to the login form, streamlining their onboarding process.

2. **New Simulation Page Access**: There is a minor navigation issue where users, while already in the process of creating a new simulation, cannot refresh the 'New Simulation' page directly from the sidebar to start a different simulation. To mitigate this, users must navigate to another section of the platform, such as 'Simulation List', before they can access a fresh 'New Simulation' page. Future development efforts could focus on implementing a mechanism that refreshes the 'New Simulation' page whenever the corresponding sidebar button is clicked, regardless of the current page.

3. **Token Expiration Handling**: Lastly, there is an issue related to user token expiration. When the user's token expires, they are no longer able to access the data, and tables appear empty. However, they still remain on the dashboard. For improved user experience and system security, we suggest redirecting users to the login page or an error page informing them of the necessity to re-login when their session token expires. This would provide users with clear information about their session status and the actions they need to take.

Each of these identified bugs presents opportunities for further refinement and improvement in the Dynamo Web Services platform. Addressing these issues in future development cycles will ensure a more seamless and user-friendly platform and underscores our commitment to continuous improvement and user satisfaction.

CHAPTER 6. FUTURE WORK AND CONCLUSIONS

The conception of the Dynamo Web Services project was motivated by an initial challenge - the inherent complexity in the usage of the Dynamo software. As an advanced trajectory computation tool, Dynamo's intricate operation and user interface presented a steep learning curve for users, particularly those lacking an elevated level of technical expertise. Despite its powerful capabilities, Dynamo's reach was restricted. The software needed a transformation that could enhance its accessibility and intuitiveness. This need marked the starting point of our project - to encapsulate the rich functionality of Dynamo within an intuitive web-based application.

Beyond the software's complexity, a key need that prompted the creation of the web service was the demand for remote execution of Dynamo. Users needed to be able to conduct simulations without having to install the software on their own machines, nor access the source code.

The project's primary objectives were clearly defined to address this challenge. The first objective was to develop a robust and secure backend server. We realized that a reliable, efficient, and secure processing environment was needed, and Python, with its simplicity and expressiveness, seemed to be an ideal choice. We combined Python with the Flask framework, known for its flexibility and minimalism, to ensure the backend server was not only powerful, but also adaptable and lightweight.

Parallel to the backend development, the second objective was to create an intuitive and user-friendly frontend interface. The frontend interface served as a medium to translate the complex functionalities of Dynamo into a format that users can easily understand and interact with. For this crucial task, Vue.js and Tailwind were chosen. Vue.js offered simplicity and versatility, while Tailwind supplemented the interface design with its modern aesthetics and responsiveness, ensuring a pleasant user experience.

Next, we needed to design and implement an effective database structure, considering the need to store, manage, and retrieve vast amounts of data. MongoDB, a NoSQL database known for its scalability, flexibility, and performance advantages, emerged as the suitable choice for our project's database requirements.

Finally, the establishment of secure authentication and comprehensive error-handling systems were essential. We aimed to integrate robust security measures, including user authentication and authorization, to ensure data protection. A comprehensive error-handling system was also established to assure the reliability and robustness of the system.

Reflecting on these objectives and the final outcome, it's clear that the project has been successfully accomplished. The choices of Python and Flask for the backend, Vue.js and Tailwind for the frontend, and MongoDB for the database proved to be effective, leading to the creation of an operational and full-stack web application.

Throughout the development process, we adhered to Agile methodologies. This iterative and user-centric approach ensured regular feedback and swift adjustments, fostering an environment of continuous delivery of usable software. This project reiterated the importance of clear communication, precise requirements gathering, and user feedback, leading to an application that is not only technically sound but also effective and user-friendly.

In conclusion, the journey from the initial problem to the accomplished objectives signifies the successful transformation of Dynamo into a more accessible, web-based application. The project underscored the value of technical growth, effective development choices, and the application of appropriate methodologies. The result is Dynamo Web Services, a solution that successfully bridges the gap between complex trajectory computation and the wide-ranging user base that can now leverage this tool with ease.

However, the nature of software development is an ongoing evolution towards a new set of challenges and opportunities. There are always new horizons to explore, capabilities to enhance, and bugs to resolve. It is in this context that we consider our future work.

In terms of system enhancements, we have identified a number of valuable features that would further elevate the user experience. Firstly, the implementation of various user licenses, such as access for a specific number of simulations or access restricted to certain types of airplanes, would allow for greater flexibility and customization of the service. Secondly, the introduction of the META batch mode, enabling users to manage a list of flights instead of a single one, has been recognized as a crucial addition. Another significant area of development lies in the integration of interactive graphs, a feature that has already sparked interest for future research and it is being developed by other student in its research project.

Addressing system bugs and improving user experience also form an integral part of the future work agenda. Streamlining the user registration process, refining the navigation system within the application, and improving the handling of token expiration are all areas requiring dedicated attention in the forthcoming phases of development.

Turning our attention to unfulfilled requirements, we recognize that not all initial objectives have been fully actualized within the current phase of this project. While the system is fully operational and provides substantial utility, aspects such as comprehensive user notifications and advanced file management remain to be incorporated. These represent areas for ongoing development, contributing to the continuous evolution and enhancement of the system.

Reflecting on the personal growth and learnings throughout this project, it has been a journey of significant value. It has offered a platform to apply and hone a diverse set of skills from working with modern frontend and backend technologies to handling databases and user authentication, managing errors, and implementing security measures. Beyond the technical learnings, the project also emphasized the importance of other critical aspects of software development, such as the role of clear and precise requirements gathering, user feedback, rigorous testing, and iterative development in delivering an effective and user-friendly software solution.

In conclusion, the Dynamo Web Services project serves as a testament to the immense potential of leveraging technology to simplify complex tasks and widen access to powerful tools. While this project phase has witnessed significant accomplishments, the path forward presents exciting opportunities for further growth and enhancements. As the technological landscape continues to evolve, the 'Dynamo Web Services' project will adapt and grow to better serve its users, always striving to offer a more accessible, intuitive, and user-friendly platform for trajectory computation.

BIBLIOGRAPHY

[1] Dalmau, R., Melgosa, M., Vilardaga, S., Prats, X. “A Fast and Flexible Aircraft Trajectory Predictor and Optimiser for ATM Research Applications.” *International Conference on Research in Air Transportation. ICRAT 2018 - 8th International Conference for Research in Air Transportation*, 26-29 (2018)

[2] Richardson, L., & Ruby, S., "RESTful Web Services", *O'Reilly Media*, Sebastopol, CA, USA (2007).

[3] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., & Berners-Lee, T., "Hypertext Transfer Protocol -- HTTP/1.1", IETF RFC 2616, The Internet Engineering Task Force (1999).

[4] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., & Yergeau, F., "Extensible Markup Language (XML) 1.0", *World Wide Web Consortium Recommendation* REC-xml-19980210 (1998)

[5] Klensin, J., "Simple Mail Transfer Protocol", IETF RFC 5321, The Internet Engineering Task Force (2008).

[6] Postel, J., "Transmission Control Protocol", IETF RFC 793, The Internet Engineering Task Force (1981).

[7] Shafranovich, Y., "Common Format and MIME Type for Comma-Separated Values (CSV) Files", IETF RFC 4180, The Internet Engineering Task Force (2005).

[8] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", IETF RFC 4627, The Internet Engineering Task Force (2006).

[9] Winer, D., "RSS 2.0 Specification", RSS Advisory Board (2002).

[10] Gosling, J., Joy, B., Steele, G., Bracha, G., and Buckley, A., "The Java Language Specification, Java SE 8 Edition", Addison-Wesley Professional (2014).

[11] Flanagan, D., "JavaScript: The Definitive Guide: Activate Your Web Pages", O'Reilly Media (2011).

[12] ISO/IEC 9075-1:2016 Information technology -- Database languages -- SQL -- Part 1: Framework (SQL/Framework), International Organization for Standardization (2016).

[13] Pokorny, J., "NoSQL databases: a step to database scalability in web environment", International Journal of Web Information Systems, Vol. 9, No. 1, pp. 69-82 (2013).

[14] Zaharia, M. et al., "Spark: Cluster Computing with Working Sets", *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, pp. 10-10, (2010).

[15] IEEE, "IEEE Recommended Practice for Software Requirements Specifications - IEEE Std 830-1998", (1998).

[16] Karl E. Wiegers, "Software Requirements", 2nd edition, Microsoft Press, (2003).

[17] IEEE, "Guide to the Software Engineering Body of Knowledge (SWEBOK)", (2004).

[18] Flanagan, D. "JavaScript: The Definitive Guide: Master the World's Most-Used Programming Language", O'Reilly Media, (2020).

[19] Rosen, N. "Pro TypeScript: Application-Scale JavaScript Development", Apress, (2017).

[20] A. Nuic, D. Poles, and V. Mouillet, “Bada: An advanced aircraft performance model for present and future atm systems,” *Adaptative Control and Signal Processing*, vol. 24, pp. 850–866, 2010.

[21] International Civil Aviation Organization (ICAO), "Manual of the ICAO Standard Atmosphere: extended to 80 kilometres (262 500 feet)", *ICAO Doc 7488-CD*, Third Edition, International Civil Aviation Organization, Montreal, Canada (1993).

[22] World Meteorological Organization, "The WMO Format for the Representation of Meteorological Data", *Manual on Codes (WMO-No. 306)*, World Meteorological Organization, Geneva, Switzerland (2015).

[23] "IEEE Guide for Information Technology - System Definition - Concept of Operations (ConOps) Document," in *IEEE Std 1362-1998* , vol., no., pp.1-24, 22 Dec. 1998

[24] APACHE Consortium, “Report on the availability of the APACHE Framework,” Tech. Rep., June 2018, Deliverable D4.1. v01.00.00.

[25] X. Prats, M. Pe ́rez-Batlle, C. Barrado, S. Vilardaga, I. Bas, F. Birling, R. Verhoeven, and A. Marsman, “Enhancement of a time and energy management algorithm for continuous descent operations,” in *14th Aviation Technology, Integration, and Operations (ATIO) Conference*, Atlanta, GA, 2014.

[26] I. Sommerville, "Software Engineering," 10th ed. Boston, MA, USA: Pearson, (2016).

[27] Larman, C., & Vodde, B. “Scaling Lean & Agile Development: Thinking and Organizational Tools for Large-Scale Scrum”. *Pearson Education*, (2008).

[28] Schwaber, K., & Sutherland, J.”Scrum guide”. *Scrum Alliance*. (2017).

[29] Anderson, D. J. “Kanban: Successful evolutionary change for your technology business.” *Blue Hole Press*. (2010).

[30] Beck, K. “Extreme programming explained: embrace change.” *Addison-wesley professional.* (2000).

[31] Poppendieck, M., & Poppendieck, T. “Lean software development: An agile toolkit.” *Addison-Wesley Professional.* (2003).

[32] IEEE, "IEEE Guide to Software Requirements Specifications (Std 830-1998)", IEEE, New York, NY, (1998).

[33] DeMarco, T, "Structured Analysis and System Specification", Yourdon, (1978).

[34] IEEE, "IEEE Recommended Practice for Software Requirements Specifications - IEEE Std 830-1998", (1998).

[35] Karl E. Wiegers, "Software Requirements", 2nd edition, Microsoft Press, (2003).

[36] IEEE, "Guide to the Software Engineering Body of Knowledge (SWEBOK)", (2004).

USEFUL LINKS

[‡1] World Wide Web Consortium (W3C), "Web Services Architecture", W3C, Cambridge, MA, USA (2004). [online] Available in: https://www.w3.org/TR/ws-arch/ [10-10-2022]

[‡2] World Wide Web Consortium (W3C), "SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)", W3C, Cambridge, MA, USA (2007). [online] Available in: https://www.w3.org/TR/soap12-part1/ [10-10-2022]

[‡3] W3schools Tutorials, “REST Methods”. [online] Available in: https://www.w3schools.in/restful-web-services/rest-methods [10-10-2022]

[‡4] SOAP vs REST: Difference Between Web Services [online] Available in: https://www.guru99.com/comparison-between-web-services.html[10-10-2022]

[‡5] Python Software Foundation, "Python Language Reference, version 3.x". [online] Available at: https://docs.python.org/3/reference/ [15-10-2022]

[‡6] Applications for Python. [online] Available in: https://www.python.org/about/apps/ [15-10-2022]

[‡7] What Is Python? [online] Available in: https://aws.amazon.com/es/what-is/python/ [15-10-2022]

[‡8] Choosing between Django, Flask, and FastAPI [online] Available in: https://www.section.io/engineering-education/choosing-between-django-flask-and-fastapi/ [15-10-2022]

[‡9] Django Software Foundation, "Django: The web framework for perfectionists with deadlines", Django Software Foundation, 2023. [Online]. Available in: https://www.djangoproject.com/. [15-10-2022]

[‡10] Pallets, "Welcome to Flask — Flask Documentation (1.1.x)", Pallets, 2023. [Online]. Available: https://flask.palletsprojects.com/en/1.1.x/ [15-10-2022]

[‡11] Starlette, "FastAPI", FastAPI, 2023. [Online]. Available: https://fastapi.tiangolo.com/ [15-10-2022]

[‡12] 10 principales lenguajes de programación backend [online] Available in: https://blog.back4app.com/es/lenguajes-de-programaciobackend/#JavaScript [15-10-2022]

[‡13] PySpark Overview [online] Available in: https://spark.apache.org/docs/latest/api/python/ [17-10-2022]

[‡14] NoSQL vs SQL [online] Available in: [19-10-2022] https://pandorafms.com/blog/es/nosql-vs-sql-diferencias-y-cuando-elegir-cada-una/

[‡15] IBM, "Db2 - Database Software - IBM", IBM, 2023. [Online]. Available: https://www.ibm.com/products/db2-database [19-10-2022]

[‡16] PostgreSQL Global Development Group, "PostgreSQL: The world's most advanced open source relational database", PostgreSQL, 2023. [Online]. Available: https://www.postgresql.org/ [19-10-2022]

[‡17] Oracle, "Database Software and Technology | Oracle", Oracle, 2023. [Online]. Available: https://www.oracle.com/database/ [19-10-2022]

[‡18] Oracle, "MySQL :: The world's most popular open source database", MySQL, 2023. [Online]. Available: https://www.mysql.com/ [19-10-2022]

[‡19] MongoDB, Inc., "MongoDB: The most popular database for modern apps", MongoDB, 2023. [Online]. Available: https://www.mongodb.com/ [19-10-2022]

[‡20] The Apache Software Foundation, "Apache Cassandra", Apache Cassandra, 2023. [Online]. Available: https://cassandra.apache.org/ [19-10-2022]

[‡21] The Apache Software Foundation, "Apache CouchDB", Apache CouchDB, 2023. [Online]. Available: https://couchdb.apache.org/ [19-10-2022]

[‡22] World Wide Web Consortium, "HTML: The Living Standard", W3C, 2023. [Online]. Available: https://html.spec.whatwg.org/ [22-10-2022]

[‡23] World Wide Web Consortium, "Cascading Style Sheets", W3C, 2023. [Online]. Available: https://www.w3.org/Style/CSS/ [22-10-2022]

[‡24] React - A JavaScript library for building user interfaces, [Online]. Available: https://reactjs.org/ [3-11-2022]

[‡25] Vue.js - The Progressive JavaScript Framework, [Online]. Available: https://vuejs.org/ [3-11-2022]

[‡26] Angular - One framework. Mobile & desktop, [Online]. Available: https://angular.io/ [3-11-2022]

[‡27] Tailwind CSS - Rapidly build modern websites without ever leaving your HTML. [Online]. Available: https://tailwindcss.com/ [10-11-2022]

[‡28] Bootstrap - The most popular HTML, CSS, and JS library in the world. [Online]. Available: https://getbootstrap.com/ [10-11-2022]

[‡29] Semantic UI - User Interface is the language of the web. [Online]. Available: https://semantic-ui.com/ [10-11-2022]

[‡30] Foundation - The most advanced responsive front-end framework in the world. [Online]. Available: https://foundation.zurb.com/ [10-11-2022]

[‡31] Bulma: a modern CSS framework based on Flexbox. [Online]. Available: https://bulma.io/ [10-11-2022]

[‡32] Requirements engineering - Wikipedia [Online] Available in: https://en.wikipedia.org/wiki/Requirements_engineering [15-11-2022]

[‡33] Obtención de requerimientos. Técnicas y estrategias. [Online] Available: https://sg.com.mx/revista/17/obtencion-requerimientos-tecnicas-y-estrategia [15-11-2022]

[‡34] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., ... & Kern, J. (2001). Manifesto for Agile Software Development. [Online]. Available: http://agilemanifesto.org/ [15-11-2022]

[‡35] Trello, Inc. (2023). Trello. [Online] Available: https://www.trello.com [15-11-2022]

[‡36] Complete Vue Developer 2023: Zero to Mastery (Pinia, Vitest). [Online] Available in: https://zerotomastery.io/courses/learn-vue-js/ [10-10-2022]

[‡37] Mockplus. Design + Collaboration. [Online]. Available in: https://zerotomastery.io/courses/learn-vue-js/ [25-09-2022]

Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

ANNEXES

TFG TITLE: Design, development and testing of full-stack web service for a trajectory computation algorithm.

DEGREE: Double Bachelor's Degree in Aerospace Systems Engineering and Telematics Engineering

AUTHOR: Maria Cáliz González

DIRECTOR: Xavier Prats Menéndez, David De La Torre Sangrà

DATE: July 6th, 2023

ANNEX A

SOFTWARE REQUIREMENT SPECIFICATION (SRS)

DYNAMO WEB SERVICES

INTRODUCTION

1. General description

The software, named "Dynamo Web Services," aims to provide an extensive and user-friendly web interface for interacting with a complex trajectory computation software known as Dynamo. This web service simplifies the intricacies of Dynamo by offering an intuitive interface that incorporates a variety of functionalities ranging from user authentication and simulation configuration to data visualization and error handling.

2. Product scope

Dynamo Web Services has a two-fold objective: to augment the user interaction with Dynamo, and to democratize the usage of Dynamo to a wider user base by simplifying its complex functionalities. The software empowers users to configure, execute, and monitor simulations in Dynamo without delving into the complexities of its XML configuration process.

3. Product value

Dynamo Web Services brings substantial value to its users. The software serves as a bridge, making the powerful computation capabilities of Dynamo accessible to users without the need to comprehend its intricacies. This opens doors for users to focus more on the results of the simulations, enhancing productivity and fostering innovation.

4. Intended audience

Dynamo Web Services is designed for a broad spectrum of users. This ranges from academic researchers and students in the aviation and data science fields to professionals in the aviation industry. It is particularly beneficial for those who are keen on utilizing the computational prowess of Dynamo but are daunted by its complex configuration process.

5. <u>Intended use</u>

Users interact with Dynamo Web Services through a web interface. They are able to create an account, log in, configure and run simulations, monitor simulation status, and view the results. Furthermore, users can manage their simulations, download result files, and have a streamlined, easy-to-use experience when running complex simulations.

FUNCTIONAL REQUIREMENTS

1. **User Authentication**: The system should support user registration, login, and profile management. It should enforce appropriate access control measures for different types of users (e.g., standard users and advanced users).
2. **Simulation Configuration**: Users should be able to configure their simulations by filling out forms or uploading pre-configured XML files. Form inputs should be validated to ensure correctness and completeness.
3. **Simulation Execution and Monitoring**: The software should allow users to execute their configured simulations, monitor the status of running simulations, and stop simulations if necessary.
4. **Data Visualization**: The software should present simulation results in a user-friendly manner. This includes tabular displays, charts, and maps, among other visualizations.
5. **File Management**: The system should manage the files associated with each simulation. This includes storing input configuration files, output result files, and allowing users to download these files.
6. **Error Handling**: The software should handle errors gracefully and provide clear and useful error messages to the users.

EXTERNAL INTERFACE REQUIREMENTS

The software leverages Vue.js to create a highly interactive and user-friendly web interface. The backend is powered by Flask and Python, which handle incoming requests, manage simulation executions, and facilitate data exchange with Dynamo. MongoDB is used to persistently store user data and simulation configurations. The software's external interface features well-designed screen layouts, clear navigation menus, and intuitive controls, all built with the Tailwind CSS framework.

NON-FUNCTIONAL REQUIREMENTS

1. **Security**: The system should implement secure authentication protocols, encrypt sensitive data, and enforce strict access control measures.
2. **Usability**: The software should be user-friendly and easy to navigate, even for users without previous experience with Dynamo or trajectory computation software.
3. **Performance**: The system should quickly respond to user requests, execute simulations efficiently, and present data without noticeable delays.
4. **Scalability**: The software should be designed to handle a growing number of users and simulations without degrading performance.
5. **Responsiveness**: The user interface should be responsive, meaning it should provide a seamless experience across different screen sizes and devices.

USE CASES

- An "End User" interacts with the system by registering an account, logging in, creating and configuring a simulation, executing the simulation, monitoring its status, and viewing the results. The user interface guides the user through each step, facilitating a smooth and intuitive interaction with the system.
- An "Advanced User" enjoys the privileges of an End User and also has the ability to upload pre-configured XML files for running simulations.
- An "Admin User" has the authority to manage user accounts, control system-wide settings, and has full access to all simulations and their corresponding data.