Data Visualization and User Graphical Interface Prototype

Version 2

Ronaldo Tsela **(September 2023)**

Contents

Table of Figures

Purpose

The purpose of this report is to present the preliminary design stage for the development of a ground-based weather station network and its associated Application Programming Interface (API) and User Interface (UI). This design and development stage consists mainly of creating a user interface application for data access and visualization.

Because of certain circumstances that make difficult the work on the lab, the fourth design stage which refer to the physical development of the station entities as presented in *Report 1: Preliminary Analysis – Requirements, Design Options and Proposed Methodology* has not been implemented yet. Instead, it has been replaced by the development stage 5 which focuses on the design process of the front-end user interface tailored for the specific weather station network under study. Therefore, this document will encompass an overview of the user interface, the design and implementation choices made for creating it and the accompanying API.

Introduction

In the current era, we find ourselves immersed in an abundance of data produced from diverse sensory devices, generated by algorithms, and from various information sources. However, comprehending and making sense of this data poses a challenge for many. This project highlights the intricate nature of multivariate weather and meteorological data, which is complex, and the task of analyzing it is intense and difficult. To address this challenge, a variety of data visualization techniques, including graphs and charts, as well as representation methodologies such as statistical analysis, are employed through specific user interfaces. These interfaces play a crucial role in interpreting and harnessing data to derive valuable insights for a wide range of end users.

Before delving into the implementation of the user interface that will facilitate access to the UOA Weather Station Data System, it is essential to establish a clear definition of the term "*web application*". In this context, a web application is defined as provided by [1]. Accordingly, we define a web application as an interactive program built using web technologies. It stores and manipulates data and is utilized by individuals or teams to perform tasks over the internet. A web application comprises several layers: the physical layer, backend, persisting layer, middleware, and frontend. These layers interact with varying levels of abstraction. The physical layer pertains to the actual hardware needed to host the web application. The backend encompasses the underlying programs that operate on the physical layer to execute basic operations. The persisting layer, a subset of the backend system, addresses static entities like database schemas and data paths. Middleware refers to a collection of high-level functions linked to the specific application, often forming an Application Programming Interface (API). The frontend represents one of the multiple views that utilize these APIs to manage user requests related to the application.

Application Programming Interfaces (APIs) in general offer effortless access to low-level functionalities. Essentially, an API comprises a suite of software functions tailored to specific tasks, streamlining interaction with end applications by providing a higher level of abstraction during the application development process.

The development of web applications differs in several aspects from traditional software applications. In a web application, all operations are managed on the server-side instead of standalone end-user computers. Therefore, logic designed for specific tasks is intended to run on a server device employing web technologies, leading to distinct design and development strategies. Additionally, user requests are made from browser applications via the internet. The application must accommodate multiple concurrent users, giving rise to challenges in access and authentication. The design and implementation approach also diverge significantly, from traditional slow-code methodologies involving manual creation to low-code methodologies, where automation substantially reduces the necessary code for constructing fully functional applications.

This document's scope encompasses the presentation of the Frontend User Interface Application for the UOA Weather Station Network, in conjunction with the UOA Weather Station Network API. Together, these components contribute to the broader UOA WSN System.

Objectives

User friendly interface: The main objective of this design stage is to create an intuitive and user-friendly interface that facilitates easy data access, visualization, and control for both end users and administrators. This involves incorporating data visualization tools such as graphs, tables, and charts, complemented by insightful statistical data pertaining to the datasets.

Real-time monitoring: Another pivotal goal revolves around real-time data monitoring both the prevailing weather conditions and the operational status of the stations. It is important that each new update is promptly reflected and accessible on the user interface.

Controlled access: A further objective is to control access via token-based authentication, granting entry solely to designated and registered users who are authorized to interact with the system.

Maintenance and scalability: While the application is specifically tailored to support the UOA WSN System, its design and implementation should possess a certain degree of autonomy from that specific weather station system. This strategic decision enhances scalability, with the ultimate aim of this application serving as the foundational structure for the integration of new features and future extensions. Hence, importance is given to the application's maintainability and readability.

API: In addition to the user interface, this project intends to create an API to control network data flow and station operations. Consequently, the final objective within this design phase is the development of an API that aligns with the UOA WSN System, supporting a wide range of possible applications associated with the domain of weather monitoring and forecasting.

Requirements

Hosting Consideration: The frontend design should be tailored to be hosted on the server provided by the *TopHost* hosting service, aligning with the rest of the application's hosting environment. Therefore the traditional implementation of the user interface with the web development tools HTML/JavaScript/PHP/CSS is prerequisite.

Differentiated user views: Users must be categorized into two distinct groups: end users and administrators. End users should have access limited to data visualization and representation. In contrast, administrators should possess access and control over the entire system, extending from users to individual stations. Access to administrator viewpoints must remain restricted from all other end users.

Controlled access: Access must be done via a token-based login approach. Unregistered users lacking the specific password from the administrator should be barred from entry. Only the administrator should possess the authority to manage user access, including the addition and removal of users.

Real-time monitoring: The application must promptly display daily data gathered from weather stations across a designated region in real time. Additionally, the application should provide also station telemetry data, providing an overview of the current station operation conditions. Geographical information about the stations is important. Depicting the various stations on a map frame is considered a good approach. Lastly, daily data must be available to end users selectively for all the possible operative stations.

Data visualization and analysis: The platform should facilitate the visualization of daily data recorded from weather stations using a range of graphical elements including graphs, tables, and charts. Furthermore, the interface must support data analysis and presentation of relevant statistics such as the maximum daily value, the minimum daily value, the average daily value for temperature, humidity, pressure, wind speed and in addition the total rainfall for the time period under study.

Seamless integration: The user interface must seamlessly integrate with both the low-level backend stack and the set of high-level views tailored to diverse tasks. Utilizing the same development approach and interfaces is important to achieve seamless integration between components.

Maintenance: The interface design should incorporate maintenance plans and features, guaranteeing long-term functionality and support. Readable source code and well-organized files and directories are crucial for future maintenance and updates.

Development Standards: Adherence to established development standards and protocols is crucial when developing full-stack applications. Given that the platform is web-based, adopting the server-client approach is advised. While the utilization of specific web-development frameworks such as Django and Flask (Python based) can expedite delivery times, traditional web-application development approaches (HTML/CSS/JS) should be employed as a supplementary measure for enhanced security and verified functionality.

Application architecture: For this initial version of the frontend, compatibility with the hosting platform (Top.Host) is extremely important. Therefore, the frontend is being developed from scratch using the HTML/JS/CSS approach: the layout will be structured with HTML templates, CSS will handle visual design, and JavaScript along with PHP will be utilized to implement the core application logic and manage the backend queries.

Scalability: The platform should accommodate the seamless addition of new software features and views as additional components. Modifications in the backend and physical stack should have a minimal impact on the overall application stack, affecting only specific segments. This application consists solely of the backbone on which future applications will be based on. Solid design and implementation of every part of the UI considering this scalability feature is important.

Incorporated API: The platform must include a robust API accessible to external applications. This API should empower automation and facilitate the seamless integration of various applications. The API is divided into two

parts: the first one must consist of low-level functions interfacing directly with the backend system and the second one must consist of high-level functions interfacing with the previous low-level functions providing a more abstract representation. For the implementation of the high-level API Python is considered the best choice.

Methodology

Frontend design overview

The frontend design encompasses both the user graphical interface and a high-level API. As illustrated in the Figure 1, the API serves as a gateway for accessing data and controlling stations from a programmable perspective. Specifically, the API empowers end users to perform various actions, including generating custom datasets, retrieving daily statistics, receiving station telemetry, transmitting telecommands, and engaging in real-time monitoring of atmospheric conditions within regions where stations are deployed. These functionalities are facilitated through a Python script featuring a series of specialized member functions that achieve a notably high level of abstraction.

The utilization of a high-level API in application creation is very handful. In addition to this API however, the UOA WSN System extends accessibility to non-programming end users through a graphical user interface, as demonstrated in the Figure 1. Upon accessing the base URL, an authentication process is initiated. Users requesting access are categorized as either end users or administrators. Accepted end users are directed to a user-oriented view, while administrators gain access to an admin-specific view.

Within the user view, all accessible data are presented within a graphical interface on the user's browser. Specifically, the user-oriented view features:

- A map pertaining to stations as markers on geographical positions and providing real-time weather information.
- A table with daily statistics.
- A table with station telemetry.
- Charts visualizing all the weather variables.

End users have also the capability to download datasets on daily bases for the selected region of interest. By leveraging the power of the API, users can not only download datasets with daily data, but also create custom time-series datasets.

Transitioning to the administrator view, the browser-based interface showcases graphical representations of system management. Administrators have the authority to create, update, or remove both users and stations. Furthermore, administrators are equipped with the ability to send direct commands to individual stations controlling different station actions and monitor the stations operation variables.

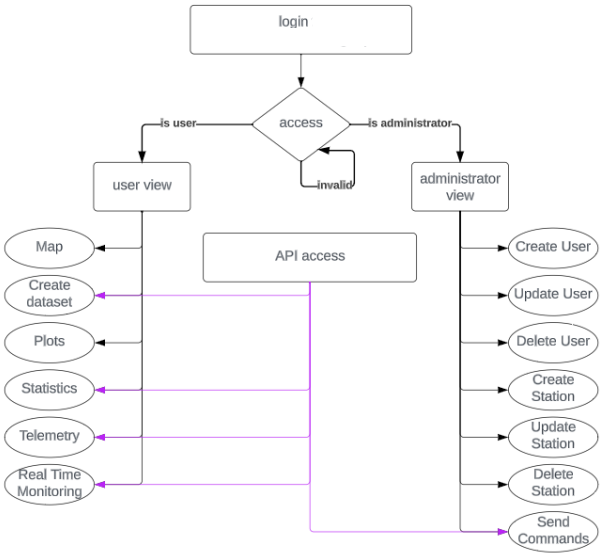

Figure 1 *– The frontend flow diagram*

The high-level API

The high-level API was designed to function atop the low-level backend API, which was developed for tasks such as reading weather data, telemetry data, and transmitting telecommands to control the weather stations. This API, referred to as the *UOA Weather Station Controller*, comprises a range of functions, integrated as members within the *Weather_Station_Controller* class developed in Python. Each of these functions require the parameter *station_id*, serving as the distinct identifier for a station within the network. This API moreover serves as a versatile tool capable of supporting various weather data-driven applications, designed specifically for the development of such metrological data-driven applications. Below, we outline the member functions of the *Weather_Station_Controller* class. More information about the operation of the API can be found on the respective *colab* file that issues the utilization of the API.

1. *get_data(station_id)*: This function returns a Pandas Data Frame containing the daily recorded data. The included features consist of Date, Time, Temperature, Humidity, Pressure, Wind Speed, Wind Direction, and Rainfall.
2. *get_stats(station_id)*: This function yields three separate Pandas Data Frames, each containing the minimum, maximum, and average values for temperature, humidity, pressure, wind speed, and rainfall. Additionally, this function provides the cumulative rainfall value.
3. *get_telemetry(station_id)*: This function produces a Pandas Data Frame encompassing details such as internal temperature, bus voltage level, bus current intensity, solar panel voltage level, heartbeats, and mode providing insights on the station's operation state.
4. *get_dataset(station_id, from_day, from_month, from_year, to_day, to_month, to_year)*: Enabling customization, this function returns a dataset that spans from the specified *from* date (day, month, year) to the specified *to* date (day, month, year).
5. *set_command(station_id, command_id, argument)*: This function issues control commands to the weather stations. In addition to the *station_id*, this function necessitates the inclusion of the *command_id* and the corresponding *command argument* that outlines the intended action to be transmitted and performed by the stations.

Authentication

Token-Based Authentication: Authentication operates on a token-based mechanism. Users are required to have a unique username-password combination issued by the administrator. Each registered user is mandated to possess a distinct set of these credentials. Notably, the current version has the administrator's username and password hardcoded as constants, rendering them unmodifiable. It's essential to recognize that users in this prototype version refer to access points rather than physical individuals. Consequently, a single application end user can be shared among a group of individuals, eliminating the need for a dedicated user account for each person.

Access and Authentication Process: Accessing the base URL redirects users to the authentication login page. At this point, users are prompted to input a username and password within an HTML form (*index.html)*. An event listener function awaits the submission of the designated button, subsequently retrieving the provided username and password from the respective form fields. Blank entries are preemptively denied. The collected username and password are then transmitted to the *login.php* script via an HTTP POST request.

Authentication Logic: Within the *login.php* script, the logic establishes a connection to the UOA WSN database (through the *connect.php* script) and performs a query to ascertain the existence of a registered user matching the provided username and password. If no corresponding user is found on the database as a valid registered user, the authentication process is halted, and the user is redirected again to the authentication login page. However, in the event of a match, a session is initiated, either as an end user or administrator, based on the user type. In the current prototype, the administrator's credentials are hardcoded as [redacted] for the username and [redacted] for the password. In case of an end user authentication the target URL is [redacted]

[redacted] and in case of the administrator the target URL is [redacted]

User view

Upon gaining authorized access as an end-user, the system directs the user to the page *user_view/main.php*, which serves as the main graphical user interface. This interface is divided into three main sections: the map, user actions, and data visualization. The underlying logic is implemented using PHP. The user view focuses primarily on data visualization and representation, with proper logic to perform SQL queries on specific tables within the UOA WSN database for retrieving weather data. The PHP logic is categorized into five distinct operations: statistics computation, weather data retrieval and dataset creation and download. These functionalities are all accessible through the same *main.php* file.

Map and geographical data representation

A significant portion of the graphical user interface is dedicated to generating a map that combines the produced data with geographical information. To achieve this, we employ Leaflet, an open-source JavaScript library for interactive map creation. Further details about Leaflet can be found at [2]. This approach is crucial due to the nature of the data, which is characterized of their low coverage and their high spatial resolution. Consequently, each monitoring station is depicted as a marker on the map, positioned at its corresponding geographic location. The number of markers on the map corresponds to the number of stations, and relevant data such as station ID, coordinates, and station state are extracted from the respective *station* table. The station state can be active or inactive, where active stations contain recent data. For active stations, the *weather_data* table is accessed to retrieve the latest readings. This information is encapsulated within a popup window that emerges upon marker selection, facilitating real-time monitoring while combining finally the data with spatial information.

Statistics and plots

The remaining segment of the graphical interface is allocated to presenting and visualizing station data. This encompasses two key components: statistics and real-time daily plots. By inputting a specific station ID, users can access daily statistics presented in tabular format.

Graphical representation of daily data from network stations can be achieved through charts. Upon selecting a specific station, individual charts are generated for each recorded variable. This is facilitated using the Chart.js library, which employs JavaScript to create numerical data visualizations. Additional information on Chart.js can be found at [3]. The chart creation process necessitates three components: a canvas HTML element with unique identifier, a JavaScript event listener to update that Document Object Model (DOM) content, and a JavaScript function responsible for chart generation. Bellow is demonstrated this logic.

$$< style >$$

$$\#chart\{$$

$$/* \ chart\ style \ */$$

$$\}$$

$$</style >$$

$$< canvas\ id = "canvas_i" ></canvas >$$

$$< script >$$

$$document.addEventListener('DOMContentLoaded', function()\{$$

$$createChart(data_i, "label_i", "canvas_i");$$

$$\}$$

$$</script>$$

The *data* parameter must be in JSON format, *label* refers to the chart name, and *canvas* specifies the canvas ID where the chart will be displayed. It's essential to provide a style to visualize the plot, even if the values are empty. Labels linked to specific variable values are crucial; they correspond to the time of day when measurements were taken, given the daily presentation of data. Linear representation enhances aesthetics, even though the discrete nature of data points is evident.

Dataset creation and download

Users can access daily bases weather datasets by defining a region of interest and subsequently downloading this data in CSV format. After selecting a station ID an action is triggered on the backend to request data from the database based on the chosen parameters and a file is generated. If relevant data exists, an internal server-side file named *dataset.txt* is created, enabling users to download the dataset. This is facilitated through an HTML link and button as follows:

$$< a\ href = "./dataset.txt"\ download = "dataset.txt"\ style = "text - decoration: none;\ "\ >$$

$$< button > Download\ Dataset </button >$$

$$$$

The *href* attribute contains the link to the dataset, while the *download* attribute specifies the download name-currently "*dataset.txt*". The applied style ensures that only the button graphics are visible, effectively masking the URL for esthetic purposes. Detailed implementation can be referenced in the respective code that implements the prototype frontend application.

Admin view

When access is granted as an administrator, the system directs the user to the page *admin_view/admin.php*, which serves as the graphical control interface for the system. Similar to the user view, the admin view utilizes PHP scripts for the underlying logic due to its compatibility with the overall architecture of the server provider.

The layout of the *admin.php* file is designed to be simple and straightforward. Specifically, the HTML form comprises three main containers. The first one contains a map similar to the map utilized for the user's viewpoint. Stations are again represented as pins on the map at the corresponding geographical points and for each station telemetry is provided inside a popup window upon selection. Telemetry data includes the timestamp of the latest reading, bus voltage and current, solar array voltage, internal temperature, heartbeat since the last reset at constant intervals, and station operation mode. This data is sourced from telemetry files unique to each station, accessible at [redacted]. Furthermore, two actions are available: Users and Stations each referring to information contained for both entities.

Upon selecting any of these actions, content is presented in the second container. Specifically, the second container holds a table displaying data about users and stations respectively. This data corresponds to all entries in the database related to these entities for the user data-related and station data-related actions, respectively. Each column is editable, as fields are presented as HTML entries. Therefore, each row can be updated or deleted easily.

The third container is dedicated to adding new entries to the two tables.

Administrator operations

The administrator can perform operations such as adding a new entity, updating an existing entity, or deleting an entity within both user and station data.

Update: Each entry can be updated with new values. However, caution is needed when performing this action. Certain restrictions apply:

- The username cannot be altered for a specific user. Valid editable fields include the password, email, registration date, and the “other” field. The username here serves as a conditional parameter and unique key for the SQL query.
- Similarly, the station ID cannot be changed for a specific station. Valid fields that can be altered encompass latitude, longitude, construction date, station operation state, and station operation mode. The station ID acts as a conditional parameter for the SQL query to take place.
- The current administrator must not be altered, since the username and password are hardcoded and therefore any change on the fields admin-username and admin-password will create connectivity issues to the current administrator on future access attempts.

For the update to be issued, the respective “save” action must be selected. The underlying logic for updating the databases is implemented in external PHP scripts named *update_user.php* and *update_station.php*

Delete: Each row except the “save” action contains also a “delete” action. When the last one is selected then a delete action is performed removing the current entry from the database. The delete action is implemented also in PHP external scripts named *delete_user.php* and *delete_station.php*. Once again, this action should be executed with caution. Once an entry is deleted, it is permanently removed from the database and cannot be restored.

Create: To create a new user or station entry, the “Add New” action should be selected. The necessary fields for the new entries are displayed in the third container. For both user and station entries, all fields are required.

- User: Username, Password, Email, Registration Date (YYYY-MM-DD), Other
- Station: Station ID (stxx), Latitude (deg), Longitude (deg), State (1,0), Mode, Construction Date (YYYY-MM-DD), SIM# (up to 12 digits)

For the entries to be stored in the database the “save” action must be selected. The creation operation is carried out through the PHP scripts *add_user.php* and *add_station.php* respectively.

Telecommands: In this current prototype version, the functionality for telecommand entry has not yet been implemented.

Results

By combining the above design choices, a prototype graphical user interface is generated as illustrated in the following figures. Specifically, Figure 2 depicts the login page, where access is exclusively granted to registered users and differentiated between end users and administrators. Moving forward, Figure 3 showcases the end user view. As evident, the stations are shown as pins on a map at their respective geographical locations. The adjacent container contains user actions and data representation fields. Daily statistics collected for the specified region where that station is installed are presented in tabular form. Furthermore, all weather variables are visually depicted through plotted graphs. Notably, users have the capability to generate datasets for specific regions by selecting target dates, subsequently enabling dataset downloads.

Transitioning to Figure 4, the administrator view is unveiled. Administrators yield the authority to create, update, and delete both users and stations. A map depicts the station location and the telemetry data as the station operation is monitored. In Figure 5, an expanded administrator perspective is showcased, offering an overview of station manipulation actions. It's important to note that all data showcased in these figures serve as test data, in order to validate the application's functionality. Therefore, these figures do not represent real weather variables and telemetry; instead, they are simulated using the UOA WSN Simulator.

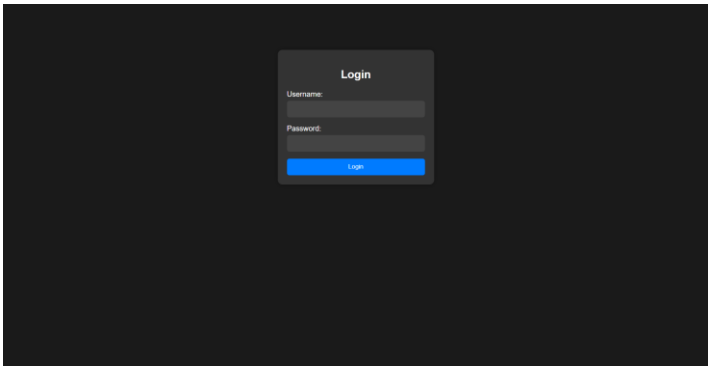

Figure 2 *- Login page*

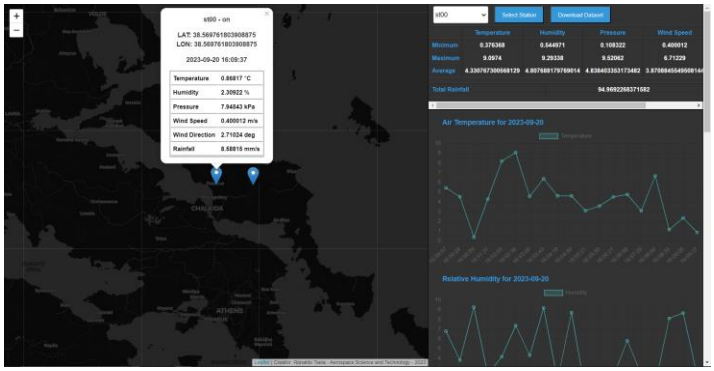

Figure 3 *- End User Frontend View*

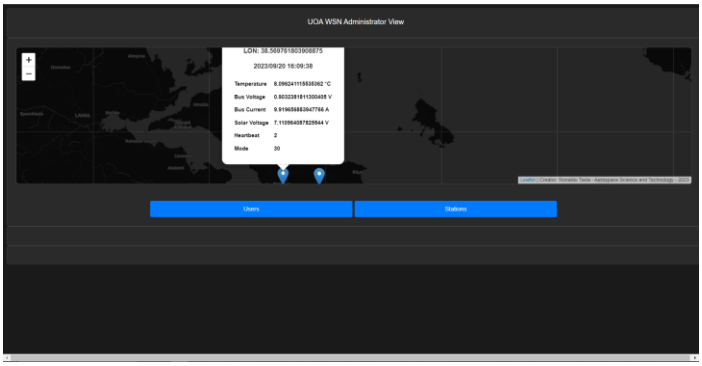

Figure 4 *- Administrator Frontend View*

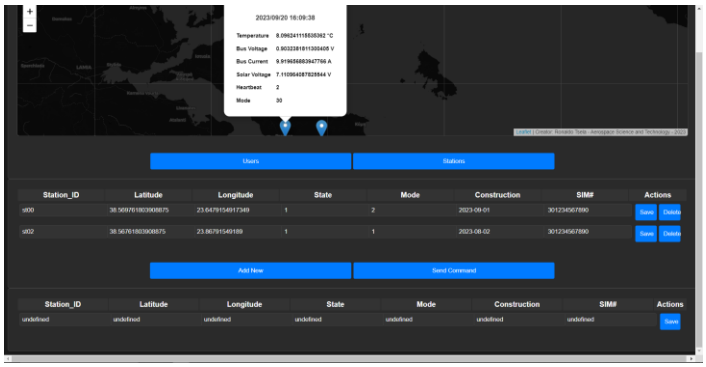

Figure 5 *- Administrator Frontend View, user management*

Further Work

The prototype has effectively addressed all specified objectives and requirements. Notably, the user interface is designed to be user-friendly, for both academic and non-academic end users. The accessibility to data and visualizations is intuitively structured, for both administrators and end users respectively. While real-time monitoring has been achieved, it's worth noting that data updates currently necessitate manual refreshing of the entire application view. This issue stands as a focal point for future iterations to address and resolve.

Access control is facilitated through token-based logins, controlled by an administrator. While changes in the system's backend should be minimized, there may be the need to refine features that enhance security and offer a more adaptable login process. The separation between users and administrators is established; however, there is potential for future development wherein a single end user could potentially have administrative privileges, and vice versa—an administrator could access the user's viewpoint using the same token.

The utilization of the slow-code approach for designing and creating this user interface renders it easily modifiable. As we look forward, future versions might consider leveraging specific frameworks and enhanced design tools, embracing a low-code approach. Promising candidates for this transition include the Node.js React framework and the Python-based Django framework. By adopting such frameworks, the user interface and a substantial portion of the application could be seamlessly transitioned, promising increased efficiency, flexibility, manageability, and scalability.

References

[1] How to Build a Web App: Beginner's Guide (2023) – budibase.com – link

[2] Leaflet.js – link

[3] Chart.js – link