CFG Nanodegree Project Report

Meteo-Mapper: The Weather and Map Tracker

CFG Nanodegree Summer 2021 Cohort

Jessica Li, Hannah Boyd, Samanta Norbury-Webster

December, 2021

Contents

List of Figures

1. Introduction

This project consists of creating a web-application that utilises Flask, which has been named "Meteo-Mapper". It will be used for accessing weather and pollution data in a visual way, so that users can stay informed about a variety of cities, based on personal choice. The architecture of this application is a three-tier client server model. The data returned will be based on the user's input of a city (name, postcode) and the relevant data will be presented. This is useful for a variety of users, as explored later in this report, but primarily for research, news, and data-analysis.

2. Background

The application, based on Flask, has three main API features that need consideration. The basic functionality of this weather tracking app, "Meteo-Mapper", uses a simple process flow, as the user inputs the city of interest, which calls the three APIs and returns the relevant data. This is to be able to visualise all three data visualisation aspects- the weather app, pollution data and the map. In order to, add more user engagement to this app, there is an option for the user to sign-up and login to their own site, which will store the calls they have made (ready for visualisation) and have a comments section so that the user is able to make notes of their choice. There is also an engaging CSS/ HTML styling for users to stay engaged. Security wise, the login details (username, password and email) are checked against the databased, so that only necessary data is maintained. For a future update, there may be a newsletter sign-up letter, so that users can have the information sent to their email on a daily or weekly basis (for relevant cities etc). We would also like to add a security feature for checking the user is a human, rather than a bot.

3. Specifications and Design

I. Requirements

In this section, the system requirements will be broken down. These were discussed and analysed by the team, in order to determine the key functional requirements and optional requirements for the app to run efficiently.

i. Functional Requirements:

- The system must have a user input function for cities to be searched.
- The system must create and send a request for weather data from the OpenWeatherApp API.
- The system must recognise the difference between cities of the same name.
- The system must present the weather data in a visually appealing way on the screen (via a table).
- The system must create and send a request for pollution data from the Map Tile API.
- The system must present the pollution data in a visually appealing way on the screen as a widget.
- The system must create and send a request for weather data from the OpenStreetMap API.
- The system must present a visually interactive map for the relevant city.
- The system will allow users to sign up and login to the site.
- The system will allow users to save their cities of interest in their own page alongside adding comments (login).

ii. Optional Requirements:

- *Documentation*
  - Well documented code with relevant comment in the python files for developers.
  - Ensure the documentation of the planning process (scrums).

- *Testability*
  - Unit testing
  - Handle any errors on the site (e.g., 404 or … error messages)

- *Security*
  - User's data will be securely stored in the SQL database, and only the necessary data.

- *Portability*
  - The system must be accessible and adjusted for different devices (widths and operating systems- Safari, Chrome etc)

II. Architecture

The three-tier client-server model was chosen for this project, as it best suited the system we wanted to create and the brief set. This came about after looking into the key functionality and requirements documents that were set for this project.

The system represents a web application that allows users and clients to access information through the internet server, via sending requests to the relevant APIs and displaying this, whilst also being able to access a database, which suited the three-tier client-server model.
There are three main components to this, which are namely, the client side (front-end web browser) via interaction with the Flask package and the HTML and CSS templates; secondly, the requests made to open-source APIs (detailed later) and manging these calls to be displayed. Lastly, the database component communicates with SQL to securely store client's required data (e.g., login details). This idea is based on figure 1, which has been adapted, as evident in figure 2.

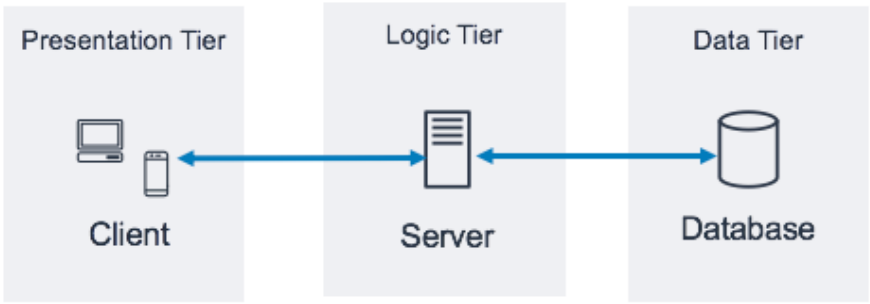


Figure 1: Three tier client-server model design (AWS, 2021)

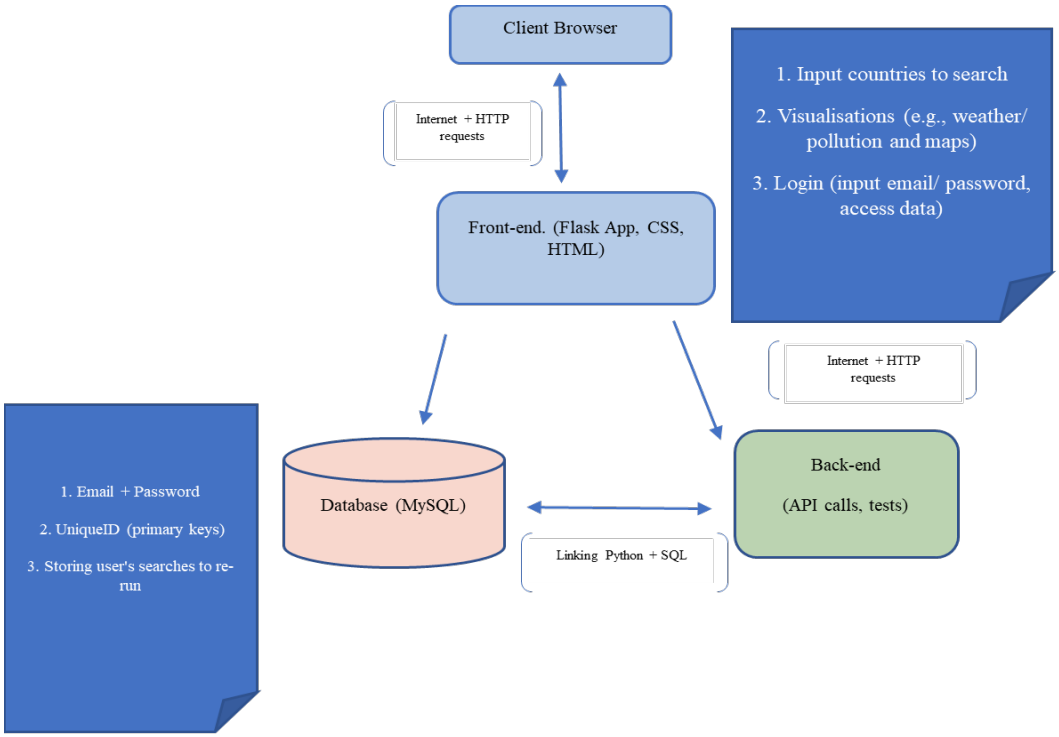


Figure 2: Adapted three tier client-server model design

III. User Stories

User stories were written as part of the software development cycle. This is to understand use cases and propel us to think from the users' point of view. This would help us to narrow down the tasks for building this project. The user stories can be seen in the appendix (fig 3-5).

These user stories were grouped into three main categories, for each developer to work on. The categories were: a) the weather and pollution tracker, b) the database connection to store searches and c) the map and clean design of the web application. These three categories fitted in nicely with the strengths of the team and each developer was assigned a category and based on the user stories, also assigned the relevant tasks. This can be reflected in the Trello board, which showed tasks assigned to each developer, and the backlog of tasks remaining (fig. 6).

IV. Project Structure

The project structure is within two main folders- the tests and the main 'climate' file. Within the 'climate' file, there is a main 'index.py' file that contains the main run method for the app. This contains the environment while that holds the virtual environment. In the website folder, this holds the design and the functionalities of the app. In the website folder, there are the templates and static folders. The static folder contains the media and css folder that holds the photos displayed on the site and the styling of the site. The templates folder holds the HTML files and allows for the Flask app to render the templates. It is structured this way based on online sources. The database includes the data manager and further detail is below in section III.V.

The structure of the project can be seen in figure 7.

V. Database Design

The database was designed based on the discussed need of the app. The MySQL file contains the queries for creating the **user_login_details** database, the **user_account** table and the **saved_searches** table (user searches are stored). The file also contains the queries to create stored procedures for inserting new entries into the **saved_seaches** table, and for creating and updating tables for each client to log and store weather API calls.

Explanation of each of the tables:

1. User_account

The user_account table contains columns for the id, name, email address and hashed passwords of users. The varchar() limitation on the password column is set to 60 characters as an error was generated previously when a hashed password was too long to be inserted into the table. There is a primary key on id so that user information can be linked, using this to other tables in the database - using the concept of normalisation - so all information is stored in specifically designated tables but still linked together. The primary key is set to auto_increment and NOT NULL so that a new id will be generated each time a new user is created. This ID is

also used for the flask 'session' in the main app so that the user has a unique identifier to follow them as they navigate the website.

2. **Saved_Searches**
   The **saved_searches** table contains columns for the ID of users and stores the cities that they have searched. This table is then called and displayed by a function in the main app when the user logs in. The id column is designated as a foreign key so that searches can be linked together by user and can be linked to the id of the user in the **user_account.**

The stored procedures:

1. **InsertCity (Checkid INT, InsertCity Varchar(25))**

This is a stored procedure which takes an id(integer) and city(string) as its parameters first checkers whether a city has already been saved by a particular user using a SELECT statement with the parameters passed in as ID and City respectively, counts any entries and assigns this to a variable. If the variable is 0 then a record with these two values doesn't exist and is inserted into the database with the use of an INSERT statement. A stored procedure makes it easier to replicate this process exactly and minimises the amount of SQL that needs to be written in the function in the .py file as it can be called, and parameters passed in from a python function into the SQL query written inside this function.

2. **CreateWeatherLog(Checkid INT)**

This is a stored procedure which takes an id number as an argument. It then uses this to create named weather_ + the id number. This means that when somebody wants to create a log for the data from a particular city, their id number can be passed into this stored procedure and a unique table generated which can then be called by a function in a .py document in the project file and displayed for the user when they go to their log page. The table has a column for the id of the user which is then assigned as a Foreign Key. There are also columns to record the weather information received by the API in the main file including country code, city, temperature, pressure, and humidity. There is also a column to store the date and time of the entry and an index for the entry in the table to make it possible to identify the record when the delete function is called in the .py file. The way that MySQL works - the id number can't be passed as the name of the table unless the query is broken up into individual parts and then concatenated. Otherwise, the table is literally named 'checkid' after the variable. Therefore the 'create table' query is written in individual parts, then concatenated, then prepared, then executed within the procedure.

3. **InsertWeatherLog(Checkid INT, CountryCode Varchar(4), CityName VarChar(25), Temperature Decimal, Press INT, Humidity INT)**

The InsertWeatherLog procedure takes a number of different parameters which correspond to the information taken when a call is made to the weather API - namely the country code, name of the city, temperature, pressure and humidity and also the id number of the user to be inserted into this column. These are then passed into an insert statement which adds them to the table which corresponds to the user. The table is selected using the id number which is passed in as the first variable to the stored procedure and then into the part of the insert table procedure which specifies the table. Again, the id number can't be passed as the name of the table unless the query is broken up into individual parts, therefore the 'create table' query is written in individual parts then, concatenated, prepared, and executed within the procedure. Within the procedure the now() function is called automatically in the datetime column and the 'index' column automatically increments, in order to log the date and time

of the insertion of the information into the table and create a unique value for that particular row of data.

To better understand how the tables all work together, an entity relationship diagram (ERD) has been provided (figure 8).

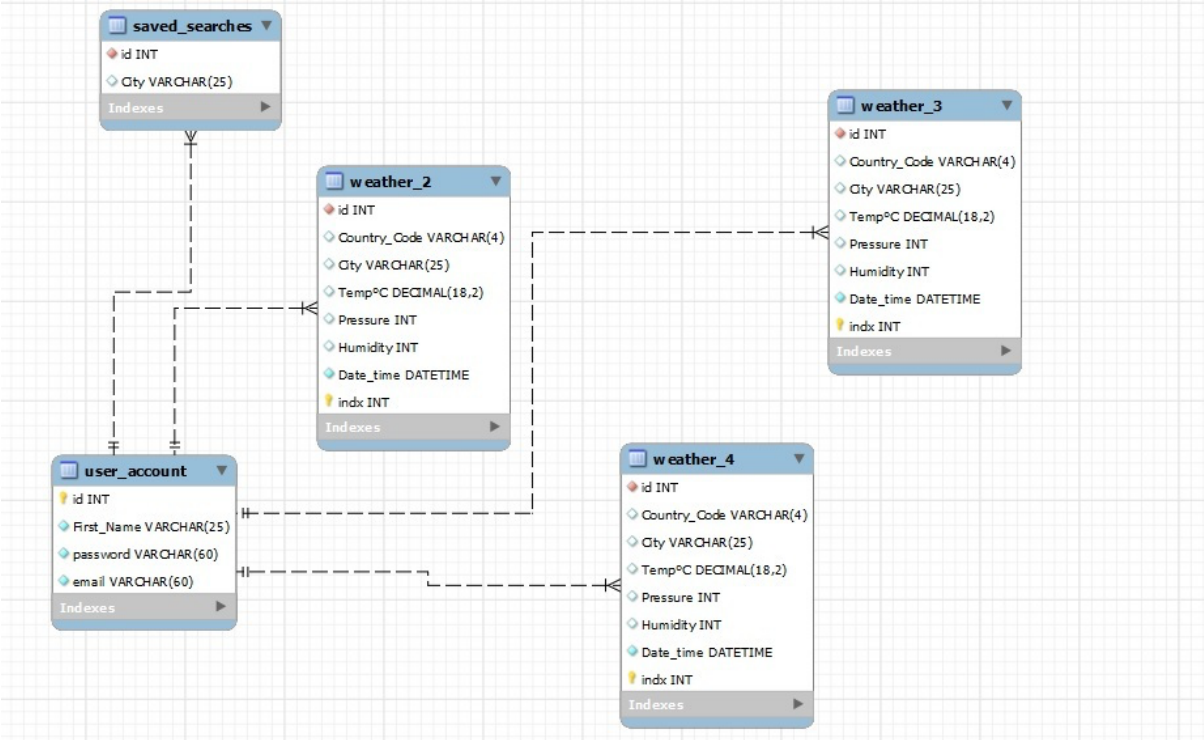


Figure 8. Entity Relationship Diagram of our database structure.

4. Implementation and Execution

I. Development Approach

This project took an agile approach. User stories were created to identify the features required, which were based on the user's perspectives. This improved the team's understanding of the project's requirements and narrowed the tasks down for each developer to work on. The tasks were compiled on Trello and divided based on the main-focus assigned to each developer. Trello helped us to keep track of the tasks- completed, challenges and questions, and to-dos. Our prior diagram (figure 8) also helped us to envision our front-end design.

i. OpenWeather API + OpenStreetMap API

To interact with the OpenWeather and OpenStreetMap API's, requests needed to be installed and API keys needed to be generated. This allows users to query and search relevant cities (under the OpenWeather API) to find data – weather temperature (kelvin and Celsius), pressure and humidity. The OpenStreetMap is viewed via the OpenWeather API and displayed through the leaflet package.

ii. Database

The database is created to store the login details of the user. The user is able to input the details into the front end of the 'sign-up' page, which then connects via the sqlconnector to the backend, where the data is encrypted using the package bcrypt before being stored. In order to keep users and their information face passwords have to undergo hashing, whereby a hash algorithm converts passwords into a string of characters to be stored by the site database preventing access to the site admin, and any unauthorised hackers that might access the database during an attack.
There are various different types of hashing algorithms with their own limitations, a limitation of traditional password hasting is that repeat words used by different users will be output by the hash algorithm as the same each time, meaning that those passwords will show up as having a common element (set of characters, likely a common word). One of the most common ways to 'crack' passwords is by using a dictionary of frequently used words and passwords automatically try possible password combinations.
For this project we have used bcrypt which adds a 128-bit 'salt', a salt is the addition of 'random' characters unknown to the user that added and run through the hashing function before being concatenated with the results of the user password. This bypasses several risks associated with precomputational dictionary attacks. It is important however that the full 128-bit salt is implement.
The data can then be recalled on the login page.
The user can store their searches and the results on their site for later access.

If the user has not created a login on the sign-up page or if the details are not recognised, an error is raised for the user to redirect themselves to the sign-up page and create a login, or for them to try again.

iii. Flask and HTML

The flask framework is used to deploy this web application and CSS/HTML templates were created to display the web application. This worked alongside the Flask framework by passing variables and other data between the app and the front-end.

Flask is used to create an initial create-app function that contains all the necessary @app.route functions and each route loads to a separate html page, and variables are passed into this html by adding the variable to the route path, which displays the output in the front end. The front end is connected to the relevant styling CSS documents too. Flask is used to handle errors too.
The leaflet package is used to display the code for the map (after the requests from the API's were made). The Wekzeug library was used to redirect between routes in the flask app. The Flask_Login and SQLAlchemy is used to store the database, which is connected using 'mysql_connector_repackaged'. The Bcrypt library was used to encrypt the login details of passwords and to keep them secure in the database.

iv. Team Member Roles

The duties were divides using the Trello boards. Jessica worked on the HTML and CSS coding, as well as partly integrating the API. Samanta focused on making the API calls and getting the backend to work, so that it can connect and integrate with the front-end via Flask. Hannah worked on the database and connecting this to the front-end. We set two weeks to finish these duties. Then for the remaining three to four days, we focused on unit testing. Finally, Jessica wrote the report, created the instructions in the readme file and focused on making the site responsive. Hannah focused on the unit testing for the API and flask. During the process, we reviewed each other's code and refactored it according to the data structure we needed it to be in and for it to be easily readable. For the report, there were inputs from all team members for better clarification on the code.

II. Implementation Process

i. Issues and Solutions

There were few issues that we came about during this project. This happened in the process of development, implementation and writing up the code for the software. The first hurdle was determining which APIs to use, as not all requests would be free to make and thus we stuck to the weather and map features. These were included within our API keys. Another issue we encountered was connecting the API to the front-end of the html files with the right css styling. This was resolved by watching some tutorials on this aspect, which taught us to refactor the code in the correct way and adding in some specific code that linked the correct css templates to the htmls. The variables are passed into the HTML files through rendering templates, and the routes are used to pass variables between each other on the web application for the front-end to work.

On the API front, there were issues regarding the incorporation of the weather data alongside the pollution data to be displayed on the same HTML file. The pollution data proved tricky to request, as the website did not specify whether it would be available in a .json, .html or .xml format. Jessica emailed the company OpenWeather to ask for input on this front, whom clarified it was .json, and Hannah resolved it by editing the code structure to be more specific.

Database wise, there were some initial issues for connecting the database to the front-end, but this was resolved by Hannah through watching the sessions back and utilising her problem-solving skills. There was another issue regarding the saving of the data to the correct page and as a txt file, which was resolved after watching tutorial regarding this.

ii. Changes

Initially, we wanted the weather data to be shown on the map, however this functionality provided trickier to implement then we initially thought, thus we separated this data into two html sections.

We also wanted to have a reactive timeline on the history page, which would have pop-downs to show more text. However, despite the code being written, the 'expand' functionality would not work. Therefore, this had be adjusted for.

iii. Successes

- Rendering the HTML pages, so that they interacted with the flask app by passing variables through them.
- Connecting the CSS and html together with Flask, so that the front-end experience was responsive and interactive.
- Learning how OpenWeather API's worked, including the map functionality.
- Learning to connect the SQL database to the front-end and flask, so that inputted data can be collected, validated, and checked for.
- Learning to store the data in an encrypted way so that the data is safe.
- Learning to undertake specific unit-testing

iv. Future goals

- Utilising the pandas library to save the text files as csv's.
- Undertake specific Flask unit testing.
- Implement a label on the map page.
- Allow users to be able to save the map data.

5. Testing and Evaluation

I. Testing Strategy:

Our testing focuses on two areas of testing— unit testing and UI testing. Our process utilises a waterfall approach, as it was undertaken after the bulk of our coding. We utilised unit tests to check our API functions worked, ensured that these functions would catch out exceptions where expected and it would catch areas where the inputs weren't as expected.
We used the standard unit test module to execute our test cases, to increase our test coverage and to test the speed of our test execution.

i. Unit Test Areas:

The status codes: we wrote tests cases to check for a 200-response code on each of our web app pages (e.g., test landing page and login page).
We have a test case to check the login details are correct and stored, and if so, a Welcome message would come up.
We have a tests case for when the login details are incorrect, and a Try again message would come up if it was wrong. We have another test for being logged in and logged out.
There is a test for incorrect sign up, such as if the data already existed in the database.

ii. User Acceptance Testing (UAT):

Once the application was developed and tested, the web application was tested for being responsive as an end-user, and to ensure that everything was working as our specification required. To fully take this into account, we tested some of our key user stories that had been created during the planning stage.

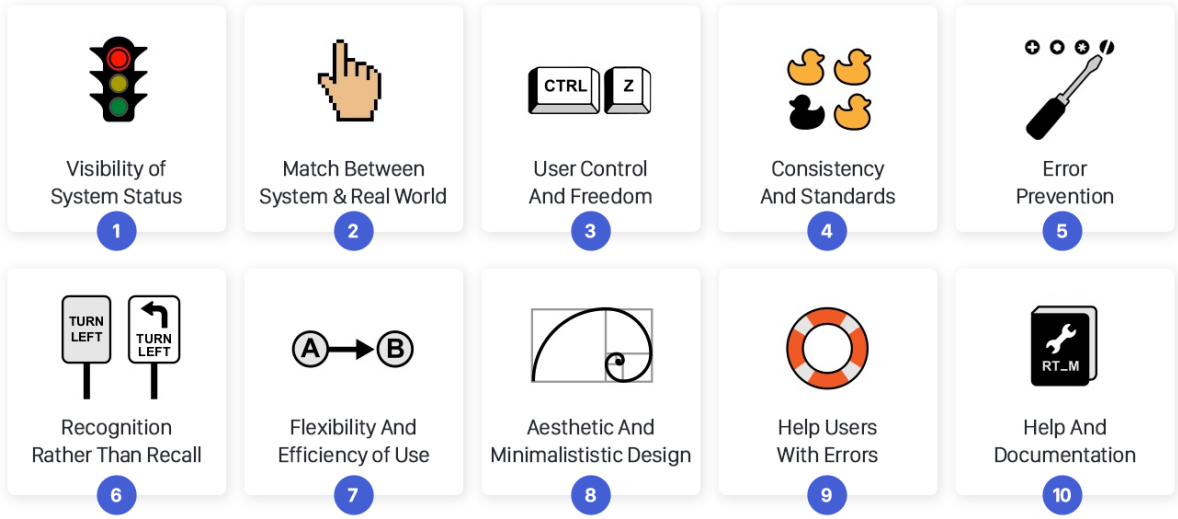
iii. Evaluation

We evaluated our flask application UI using the Nielsen heuristic model, as shown below in figure 9.

Figure 9:

Criteria	Pass/Fail	Comments
Visibility of System Status	Pass	The user is able to interact with the system and receive feedback on interaction, for example, when they fail to login they are prompted to try again or sign up.
Match between system and real world	Pass	Recognisable symbols are used (such as a magnifying glass for searching).
User control and Freedom	Pass	Navigation buttons are present and visible across all the pages, and users are able to backtrack and use the navbar at all times. The emergency exit is the home button that takes them back to the welcome page.
Consistency and Standards	Pass	The web application app has a comprehensible system that is not confusing. It is consistent with the colour scheme and user interface.
Error Prevention	Pass	Users are prompted to try again or to sign up when their login account details are not recognised. This prevents users from entering wrong details over and over again.
Recognition Rather than Recall	Pass	There is a recognisable navbar on all the html sites, which makes it easy for users to navigate the pages. The home button is consistently located in the same area.
Flexibility and Efficiency of Use	Pass	The application tailors to both one-time, novice users, and frequent users. There is a save to profile button, which allows common searches and the information to be stored and accessed at a later date.
Aesthetic and Minimalistic Design	Pass	Only relevant information is displayed on each page. The web application has an aesthetic design that matches the intuition behind the idea. It is minimalistic with few

		icons and a succinct nav bar and footer on each page.
Help Users with Errors	Pass	Users are able to diagnose and recover from simple errors. There is a plain error message that comes up if the site doesn't recognise the location, user details or sign-up details. Therefore, the user is able to correct these aspects and try again. It is easily rectified.
Help and Documentation	Pass	A detailed readme file is included on how to install, run and deploy the application. A short video demo is included that aids users to use this application. Recommendation: Further help buttons or chatbox function to gain help on utilising the app.



(UX Design, 2019)

6. Conclusion

This project resulted in the successful development of a web application, utilising flask, which can act as a weather and pollution tracker. It is able to give real time data on the cities of choice by the user, which can then be saved on a profile. This allows users to collate a database for certain cities, or to simply, as novice users, to explore data of different cities. The web application has an OpenWeatherMap embedded, which allows users to navigate around the globe to various cities and see the layered details of the cities (e.g., precipitation, cloud cover, temperature and air pressure). The application utilises the OpenWeather API, the OpenStreetMap API (embedded into the OpenWeather Map) and the pollution aspect of the OpenWeather API. The application also has a database that is created by linking SQL to Python, and finally, it has HTML templates, CSS files and a javascript file that link together to complement and build the frontend of the application. This project has allowed our group to create an application that is ideal for various users (see user stories) and especially for researchers interested in climate change. It allows further projects to be undertaken, as it can help researchers in data mining and web-scrapping. It is also fun to navigate as an everyday user.

Appendix

- **Figure 3. User Story 1**

John Doe

- A 51-year-old undertaking research on a conservation consultant project, based in Cornwall, and is quiet and reserved.
- Interested in pollution, as he worked in London for 30+ years and recently moved away due to pollution related health problems.
- Likes:
  - To be able to discover pollution data for different cities in UK.
  - To be able to analyse the pollution data, based on longitude and latitude data that allows him to compile a csv later and utilise the data in R Studio to create different maps with shapefile data.
  - Understanding different cities geographies and planning trips.
- Dislikes:
  - Loud noises
  - Being seen as technologically inept, and websites that do not have guidance on logging in.
  - Cars and pollution.

- **Figure 4. User Story 2**

Jane Doe

- A 14-year-old worried about the sustainability and climate change issue, inspired by the XR movement and Greta Thunberg, wants to be better informed of climate concerns.
- Likes:
  - Climbing trees and playing with her pet hedgehog
  - Hanging out with her friends and talking about sustainable fashion
  - Exploring websites to learn more about climate issues and changes.
- Dislikes:
  - Long websites with lots of jargon
  - Not understanding terminology, and being looked down on for being young
  - CEOs of polluting companies, like Shell.

- **Figure 5. User Story 3**

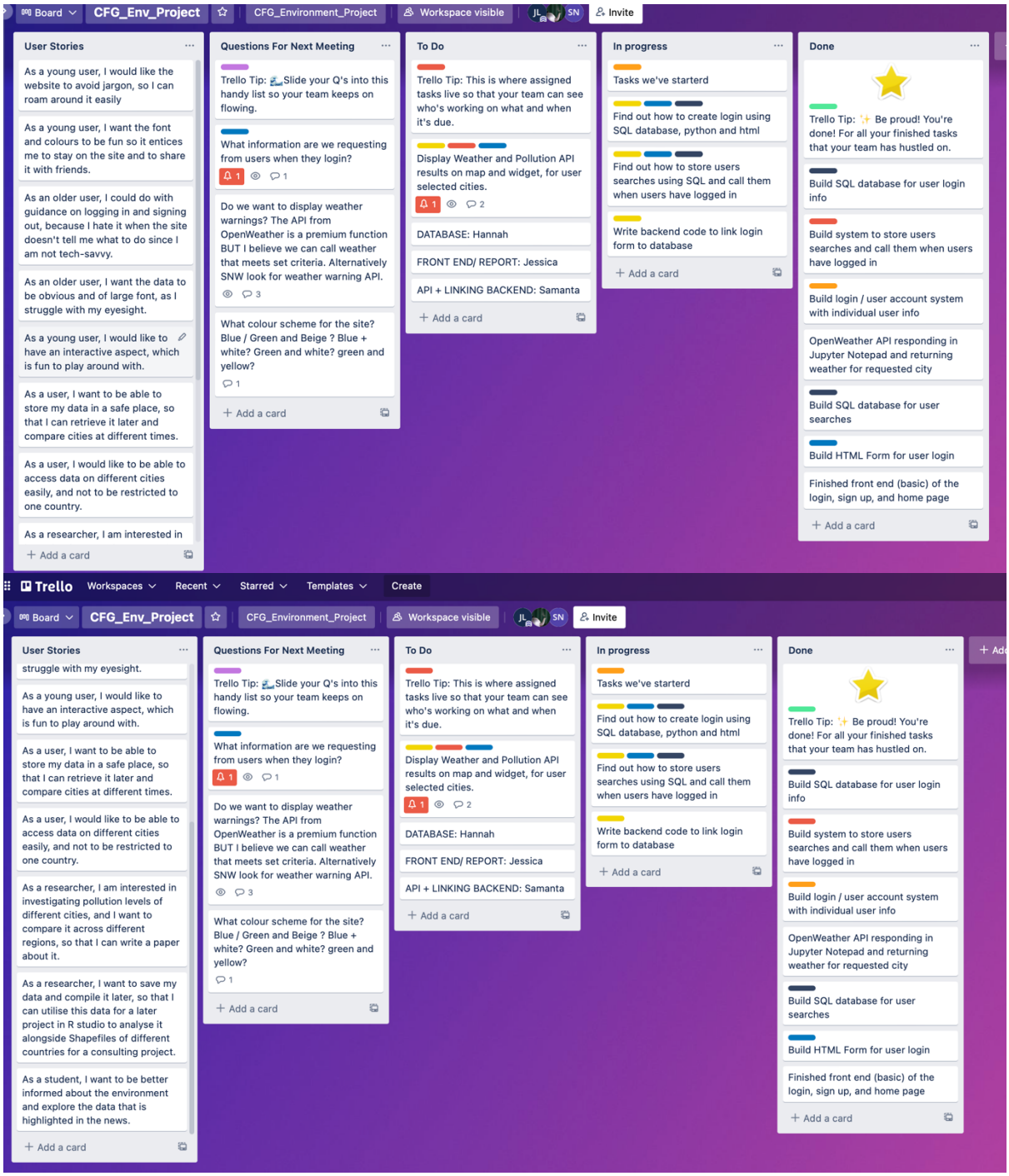
Jamie Doe

- A 21-year-old gen Z undertaking an environmental studies masters, based in central London, loud introvert.
- No time due to commuting, and busy deadlines.
- Likes:
  - Quick way to discover weather data on developing vs developed countries, so she can compile a report on climates of different cities for a coursework.
  - Wants to discover data other than weather data to elaborate on her report.

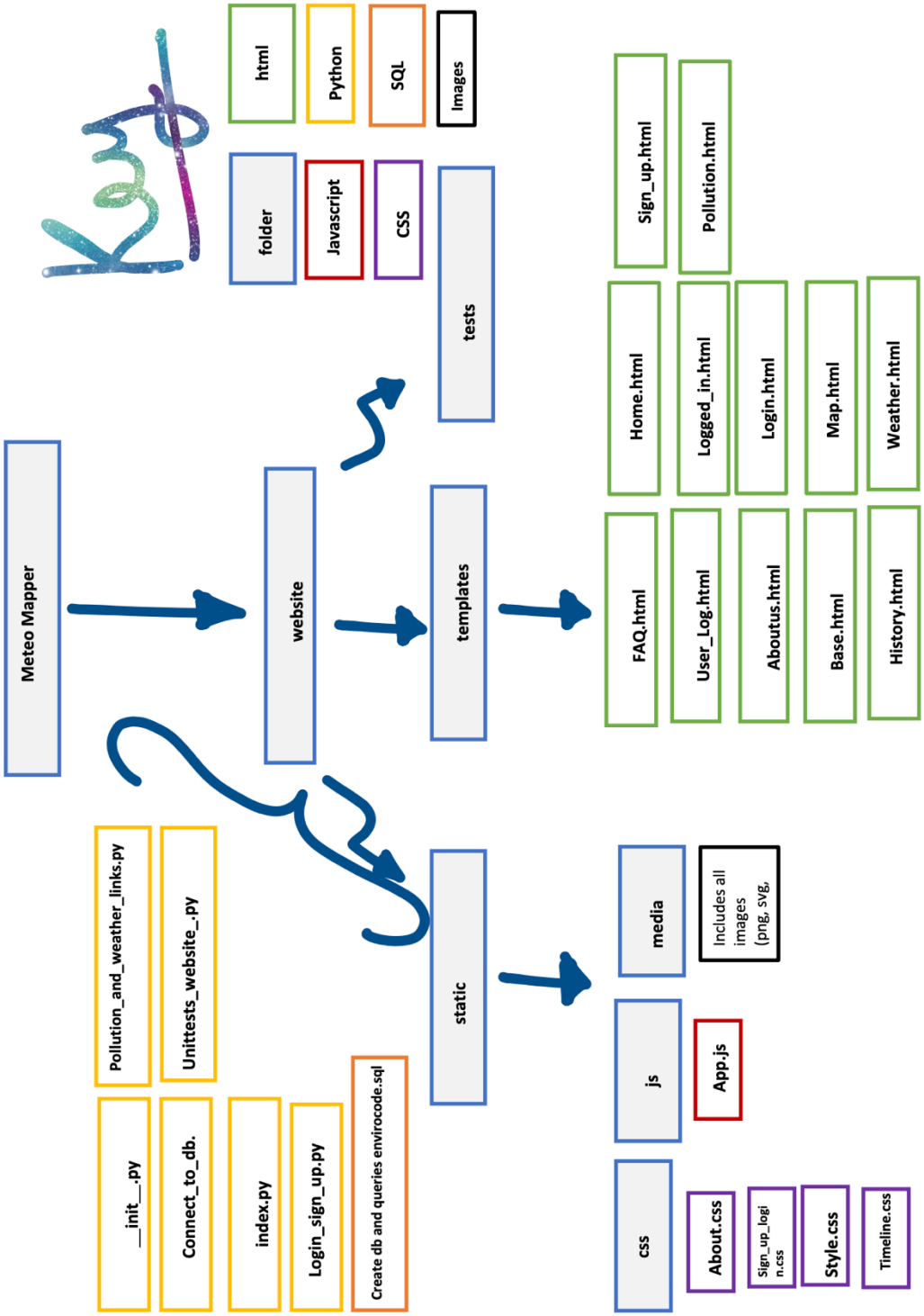
- To be able to store her data, as she has a short attention span.

- Dislikes:
  - Hates forgetting where she put her keys when she leaves home, as she is forgetful and suffers from a short attention span due to social media.

- **Figure 6. User Stories, Backlog, and Tasks on a Trello Board**



- **Figure 7. Structure of Project**



Bibliography

Amazon Web Services Whitepaper (2021). "AWS Serverless Multi-Tier-Architectures with Amazon API Gateway and AWS Lambda". Available at: https://docs.aws.amazon.com/whitepapers/latest/serverless-multi-tier-architectures-api-gateway-lambda/serverless-multi-tier-architectures-api-gateway-lambda.pdf#three-tier-architecture-overview, accessed 15th December 2021.

UX Design (2019). "10 Usability Heuristics Every Designer Should Know". Available at: https://uxdesign.cc/10-usability-heuristics-every-designer-should-know-129b9779ac53, accessed 17th December 2021.