



POLITECNICO
MILANO 1863

Design Document and Test Plan

**PRESENTATION OF AIR POLLUTION DATA USING AN INTERACTIVE
WEB MAP**

Authors

Ahmed Abdalgader Ahmed Eisaa

Alba Lunner

Evalyn Horemans

Leonard Hökby

Mostafa Mahmoud

Deliverable: SDD
Title: Design Document and Test Plan
Authors: Ahmed Abdalgader Ahmed Eisaa, Alba Lunner, Evalyn Horemans, Leonard Hökby, Mostafa Mahmoud
Version: 2.0
Date: Date: June 7, 2022
Copyright: Copyright © 2022, A.A.E.L.M – All rights reserved

Revision history

Version	Date	Change
version 1.0	26 th of May, 2022	First submitted version
version 2.0	7 th of June, 2022	Second submitted version

Table of Contents

1. Introduction	5
1.1. Purpose	5
1.2. Scope	5
1.3. Definitions	5
2. Architectural design	6
2.1. Overview	6
2.2. Argument for Nuxt – Flask split	6
2.2.1. Ready made resources	6
2.2.2. Business logic	6
2.2.3. Workload management	7
2.2.4. Flask strengths	7
2.3. High level architecture	7
2.4. Flask App	9
2.4.1. Flask-Nuxt Interface	10
2.4.2. Process	11
2.4.3. Queries	12
2.4.4. User functionality	12
2.4.5. Dependencies	13
2.5. Database	14
2.6. Nuxt App	15
2.6.1. Map page	16
2.6.2. Dashboard page	16
2.6.3. Store	17
3. User interface design	17
3.1. Overview	17
3.2 Design	17
3.3. Content Tables	19
3.4. Views	20
4. Implementation, integration and test plan	25
4.1.Implementation plan	25
4.1.1. Project scope and task creation	25
4.1.2. Assign team to the implementation process	25
4.1.3. Testing new features in a testing environment	25
4.1.4. Onboarding other team members	26
4.1.5. Installation and integration of the software parts	26
4.1.6. Feedback from the team	26

4.2. Integration and testing	26
4.2.1. Unit Testing of flask	26
4.2.2. Unit Testing of Nuxt	29
4.2.3. Integration tests	30
4.2.4. System Testing	32
5. References	34

1. Introduction

An important step in software development is the design of software architecture and testing of the functionality of implemented software.

Design schemes are presented to display an overview of the different technologies and the connection between them. A scheme of the Backend describes the relationship between OpenAQ API, queries, data processing and the flask interface. The scheme of the app architecture demonstrates the relationship between the frontend technology Nuxt, the Flask app and OpenAQ API.

Testing of the software's functionality is executed to verify that the program is built correctly and to find out if the software produces the expected result.

2.5. Purpose

In a time of information saturation and global climate crisis, there is a need for distributed accessible high-quality information. This combined with the increasing propensity for the individual to find their own specific information sources tailored to their own needs means that tools are needed which allow individuals to cross-reference the information they receive using reliable and objective sources. Such tools would allow them to substantiate concerns in negotiations with governing bodies and other stakeholders as agents of democracy.

1.2. Scope

The application to be developed, therefore, aims to be a personal tool that provides easy access to air pollution data in order to empower them both in their personal lives and as members of a democratic community. The application is not a source in and of itself, but rather a visualization tool for existing and established data sources. The first section takes a closer look at who the intended users are, followed by scenarios in which a user might turn to the application. Going into more depth, the phenomena around the application and its use cases are explained along with hard requirements to which the app will be developed.

1.3. Definitions

“The web app” or “app” – The application that is being developed

“Use case” – An example of user interaction with the system

“User” – A person using the website that satisfies the description under the “user characteristics” section.

“Bookmark” – Saving a snapshot of the information presented at a location on the map

JSON – JavaScript Object Notation

JS – javascript

df - Dataframe, a pandas dataframe object

gdf – geodataframe, a geopandas dataframe object

RASD – Requirements Analysis and Specification Document

easy later access.

“OpenAQ” – An organisation providing open air quality data through API (OpenAQ, n.d)

“API” – Application Programming Interface

“OSM” – Open Street Map. A database with open access geographical data.

2. Architectural design

2.1. Overview

The web app is built on a flask app at the bottom which collects data from the API:s needed, runs needed calculations, formats output data and serves it to the web. This is then accessed by a Nuxt app that contains the actual application and business logic.

2.2. Argument for Nuxt – Flask split

One key of software engineering is to use the right tool for the job. Flask is an excellent lightweight backend framework that works well for managing requests in a WSGI server very easily. However, its functionality for frontend coding is not as advanced and has two limitations: Few ready-made resources and a cumbersome way of integrating business logic though JavaScript code.

2.2.1. Ready made resources

A big advantage of Nuxt over Flask is that Nuxt is a well-established frontend framework with a lot of available resources. This means that by replacing Flask’s templating functionality (jinja) with Nuxt we are able to use templates available under a public licence (more on the template chosen under 2.5. Nuxt App). With these templates, we eliminate the huge task of frontend design while still offering a product that looks good. We also get code that is well structured and implemented, allowing us to easily find and edit the parts we need to change while not worrying about the other ones.

2.2.2. Business logic

It was clear to have full flexibility around map functionality we needed to use a JS library for mapping. Integrating this into a flask app is possible, but an early proof of concept showed that it would mean writing javascript code as strings in python and then passing to the templating engine, or writing JavaScript code from scratch the same way as if building an application with only js, html and css. While possible, it is definitely harder than using a js framework, since they are generally more developer friendly. Nuxt was then appropriate as it is designed to be developer friendly and considering a few members of the team had experience with Nuxt and the languages needed to use it. This allows for the development of an informal frontend team.

An example of this is the implemented functionality of panning the map based on interactions in another page. This would be very cumbersome without an actual javascript map and implementing the necessary Single Source of Truth without a framework would be as hard of a task. More on the exact implementation of this functionality later.

2.2.3. Workload management

The third bonus comes from better workload management. The split between the two technologies meant that it is easier to divide tasks on different team members as the frontend functionality and visualisation and the backend preparing of the data can be split. This gives the group flexibility in who does what and when.

2.2.4. Flask strengths

Python is a very powerful language for data management, with good and straightforward ways of retrieving data and processing it (preprocessing for visualization in our case). This means that flask was clearly a great choice for exposing the processed data to Nuxt, by working with JSON formats both for retrieving data from world API:s and for exposing the data though the API implemented in flask.

2.3. High level architecture

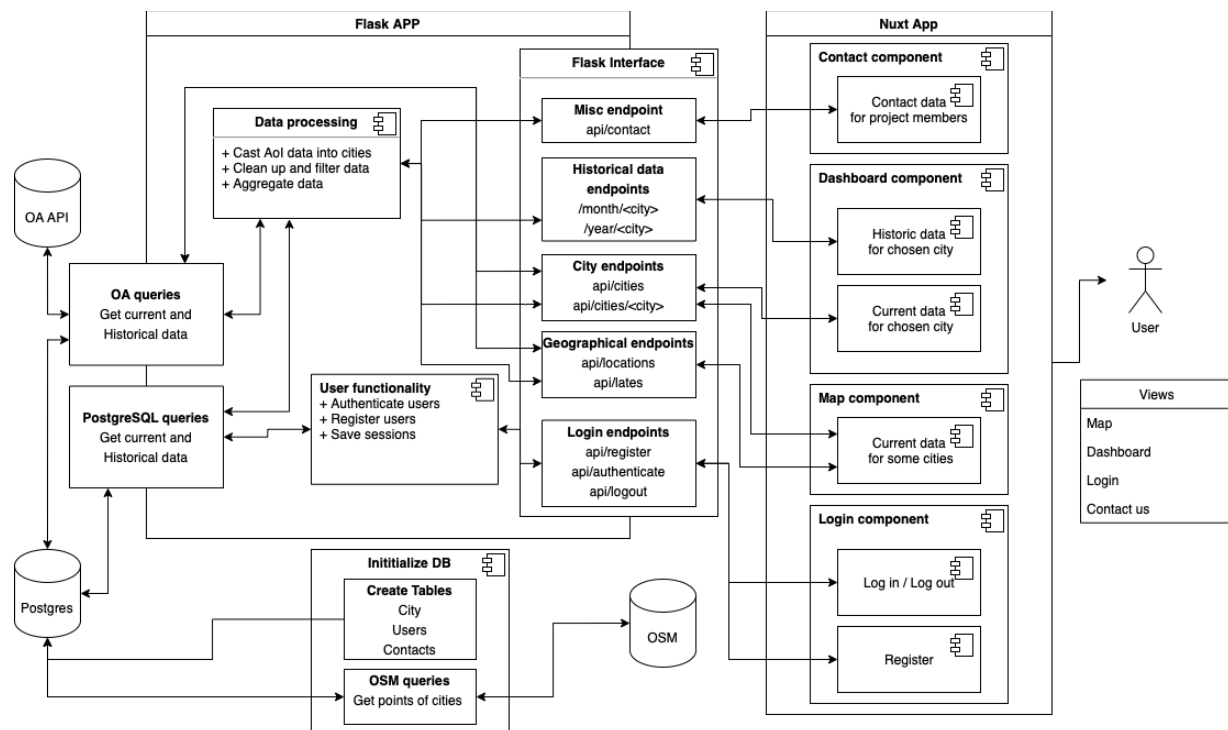


Figure 1: High level architecture of the web app

As figure one shows, the flask App and the Nuxt app are wholly separate. They are run on different servers and are located in different folders. This means that the only way for them to

communicate with each other is by the API exposed by the flask app. This API is implemented as the flask app run on an WSGI web server, and has five different types of endpoints, aimed at five different types of functionality in the Nuxt app. These are further described below.

On the other hand of the flask app is the interactions with world API:s , most notably the Open AQ API and with the database. Open AQ is responsible for providing data and is mostly queried on demand (a very limited cache has been added, more on that later).

Between these two ends of the flask app lies the processing logic, which will run calculations on incoming data to be served through the interface towards the Nuxt app.

The Nuxt app in turn shows two main views with data and two other views, login and contacts. Each data view and the contacts view is implemented as a “page” meaning that they are loaded separately by the browser. Each page will, once it is requested to load, send a request to the flask app for the data it needs and then render the data according to the logic of the page. More on these pages below.

The login function works through dialog windows called “modals” which prompt the user to take the action needed for successful login. In the current version, logging in means your latest choice of city to show in the dashboard will be saved and retrieved again when you log in next time.

There is also a third part on top of the Flask and Nuxt apps which is the database and its initialization code. The database is a PostgreSQL database running locally on the computer. It is initialized to have three tables, city, users and contacts (the contacts table currently envisioned for future functionality). These are created when the database is initialized and the city table is filled with data from OSM. This data could be queried on demand, but is chosen to be “cached” in the database since the OSM API “overpass” is usually quite slow. From now on “DB” will refer to the PostgreSQL database.

2.4. Flask App

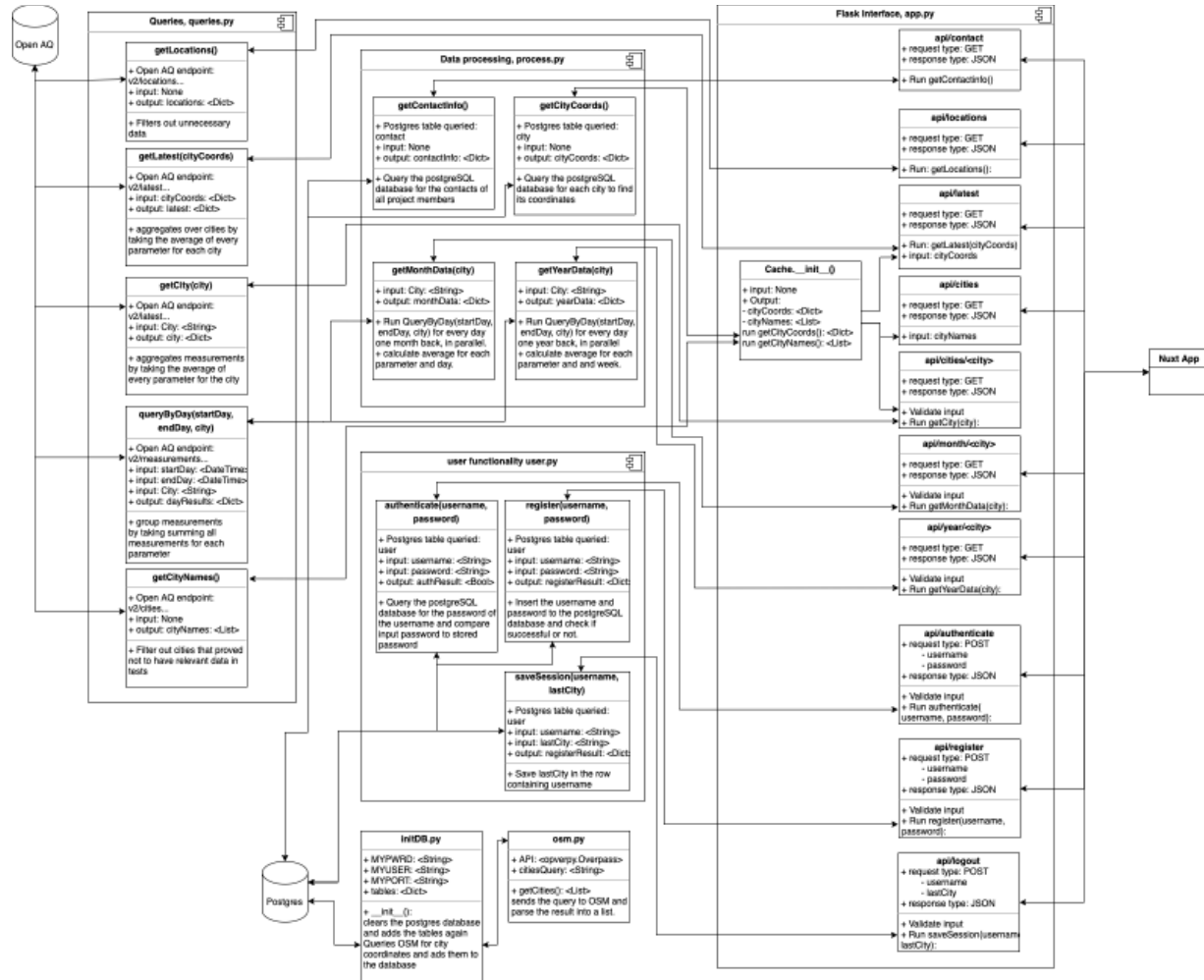


Figure 2: Flask App function diagram explains how every function in the backend works and how they are related.

The flask app and the DB are together the backend of the project. The flask app consists of the querying part, the data processing part, the user management part and the flask interface part. The DB code is the actual database and its initialization programs. These are separate from the flask app since they only have to run once before launching the flask app for the first time.

When a request comes in to any of the endpoints supplied by flask, they are handled by the WSGI server. Then, depending on which flask endpoint received the requests, the request triggers a query to an outside database (Open AQ or the DB), triggers a data processing task or triggers something to be served from the cache. If a data processing task is triggered, this task will in turn query the data that is needed for that specific task.

No matter what is triggered, the endpoint receiving the request will run the appropriate functions (as described in figure 2) and receive a python dictionary. It will then “jsonify” the dictionary and send back as a result to the querier.

2.4.1. Flask-Nuxt Interface

The interface presented by the flask app consists of five types of endpoints: Historical data, Miscellaneous data (Misc), Geographical data, City data and User Management. Table 1 shows the different endpoints for each type of data currently implemented. As is shown in figure 2, all of these endpoints will trigger a call to an appropriate function to handle any request that is sent to them.

Table 1: Endpoints of flask-Nuxt interface

Endpoint type	Endpoint URL	Data supplied	input Accepted	JSON scheme
City Data (GET)	/api/cities	List of all cities available	None	{"cities": [<String>, ...]}
City Data (GET)	/api/cities/<city>	Values of all parameters for that city	City name: String	{"city": {"cityName": <String>, "lastUpdated": <String>, "Measurements": [{"parameter": <String>, "unit":<String>, "value": <Int>}, ...]}}
Misc Data (GET)	/api/contact	Contact information of all project members	None	{"contactus": [{ "description": <String>, "email": <String>, "first_name": <String>, "last_name": <String>, "nationality":<String>}, ...]}
Geographical Data (GET)	/api/locations, /api/latest ¹	/locations sends latest data for each station, /latest sends for each city.	None	{locations: [{"cityName": <String>, "Coordinates": [lng: <Float>, lat: <Float>], "particles": [{"parameter": <String>,"value": <Float>,"unit": <String>,

¹ Two endpoints in the same row means that their output JSON structure is the same.

				"lastUpdated": <String>},...]], ...]]}
Historical data (GET)	/api/year/<city> /api/month/<city>	Year data is aggregate per week, month data is aggregated per day.	City name: String	{"timeMonth":"timeYear":{"parameterName<String>: [<Float>, ...], ...}}
User Management (POST)	/api/authenticate	Sends username and password, returns success or not.	Payload JSON: { "username": <String>, "password": <String>}	{ "user": { "username": <String>, "userID": <Int>, }, "access": <Bool>, "lastSearch": <String>}
User Management (POST)	/api/register	Sends username and password, returns success or not.	Payload JSON: { "username": <String>, "password": <String>}	{ "user": { "username": <String>, "userID": <Int>, }, "register": <Bool>, }
User Management (POST)	/api/logout	Sends username and last city visited, returns success or not.	Payload JSON: { "username": <String>, "lastCity": <String>}	{ "user": { "username": <String>, "userID": <Int>, }, "saved": <Bool>}

The code implementing these endpoints can be found in the flaskVueBackend/app.py file.

2.4.2. Process

Calculations are necessary mainly for the historical endpoints since each of these have to send many requests. In order to limit the size on each request to a level that Open AQ could accommodate² only one day is queried at each time. For this to not take too long a time when there are 365 days for the /year endpoint, all requests are running in different threads at the same time. This multithreading is done in the getMonthData and getYearData functions of the calculations module, using a thread pool executor from the concurrent futures module of python. Each request aggregates into one output for each day, which is in turn aggregated into the entire response as the different threads are finished. When all threads are finished, the result is either output directly in case of /month endpoint, or further aggregated into weeks in the case of the /year endpoint. All aggregations are made as averages.

The other part of the process module is the gathering of coordinates for cities and the contact information from the DB. This is later used to enrich the /latest endpoint with these coordinates,

² We avoided using the Open AQ pagination function as much as possible, since it did not work for many pages.

as the cities in Open AQ does not contain coordinate information and to fill out the contacts page.

The process functions can be found in the flaskVueBackend/process.py file

2.4.3. Queries

Queries is the module that actually sends queries to OpenAQ. It queries 4 different endpoints in Open AQ found in table 2

Table 2: Endpoints of Open AQ used

Open AQ endpoint	Description	Querying function	Use in flask endpoint
v2/locations	Provides a list of all locations	getLocations()	/api/locations
v2/latest	Provides the latest measurements from all locations	getLatest() getCity(city)	/api/latest /api/cities/<city>
v2/measurements	Provides a list of measurements	queryByDay(startDay, endDay, city)	/api/month/<city> /api/year/<city>
v2/cities	Provides a list of cities within the platform	getCityNames()	/api/cities/ ³

All functions in queries do some filtering. On top of that, both the /latest and the cities/<city> endpoints are serving data based on cities, but since Open AQ does not have only one datapoint per city but rather gives us all sensor data related to a city, the different measurements have to be aggregated to the city level to allow the end user to browse this data. The aggregation is done in the functions directly and the aggregation policy is the mean of all measurements related to the city. A sensor is considered related to a city if it is returned by Open AQ when that specific city is queried.

The query functions can be found in the flaskVueBackend/queries.py file

2.4.4. User functionality

For the user functionality it is necessary to interact with the DB where the user information is stored. In the user functionality module, there are functions for handling authentication, registration and logout. All these take the username as input and the first two also take the

³ The data from getCityNames is used to check the input for all endpoints that accept city as input, so in fact, it is used also for those three endpoints.

password, while the last one takes the last city queried by the logged in user. The login state is not tracked by the backend but only by the frontend. All passwords are also hashed before being sent to the backend and thus it is not possible to reverse engineer someone's password from a stolen database.

When an authentication request comes in, the authenticate function will check if there is a row with the incoming username and password in the database and output will be returned according to table 1. For a register request, the register function will try to register the new user. Since the username column in the DB is set to unique, this will cause an exception if the username already exists which will mean registration has failed. The appropriate response is sent back according to table 1. For the logout request, the logout function will write the received last city visited and return a message according to table 1.

The user functionality functions can be found In the flaskVueBackend/user.py file

2.4.5. Dependencies

The flask app has a number of dependencies. These are python libraries that need to be installed for the app to work properly. These are found in table 3.

Table 3: Python dependencies for backend

Library	Use
overpy	Used to query OSM
Flask	Set up and run a WSGI server through the flask framework
jsonify (from flask)	Format JSON responses correctly
request (from flask)	Enables parsing of POST-requests
requests	Send HTTP requests to world API:s
psycopg2	Database manager for postgresSQL
flask-cors	Enable Cross-Origin access so the Nuxt app can access the flask app.
Concurrent.futures	Enable the use of multithreading.
Datetime	Handles working with dates for the historical data endpoints
json	Helps working with JSON files
sys	Helps adapt to different operating systems.

Of these, only Flask, requests, psycopg2, flask-cors and overpy have to be installed on top of python 3.9.

2.5. Database

The database is a PostgreSQL database with a unique installation for every group member. The different installations are kept in sync by initializing them with the Pg class of the initDB.py file. This class will create empty tables (table 4 describes these tables) and fill them with the appropriate data to start with.

Table 4: Specification of tables in DB

Table	Usage	Columns	Preloaded Data
users	Keep track of all users.	User_id, (autogenerated, identity) user_name (unique and not null) user_password (not null) last_search	None
city	Stores coordinates for cities	city_id (autogenerated, identity) city_name (not null) longitude, latitude	Coordinates of cities from OSM
contacts	Stores information about us for the contacts page	contact_id (autogenerated, identity) first_name (not null) last_name (not null) description nationality email	Data on each group member.

2.6. Nuxt App

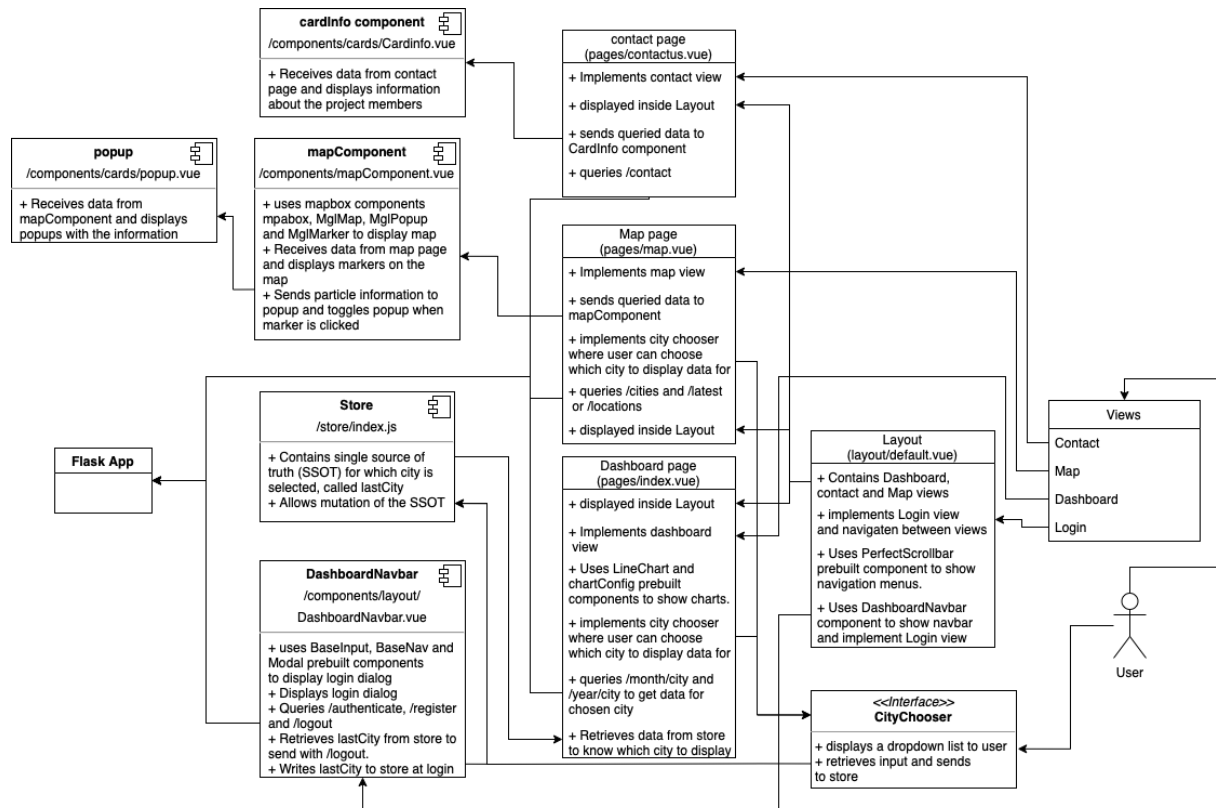


Figure 3: The architecture of the Nuxt app parts that are partly or wholly built by us and their relation to each other and other components.

Nuxt is a library for javascript (JS) that is used for development of client, server and static site rendered web pages. It is an open source framework making web development simple and powerful.⁴ Our Nuxt app is server side rendered meaning that the entire web page will be created on a web server before being shown in the client. The only exception of this is the map itself which is client side rendered.

The Nuxt app is built on a template created by Creative-Tim⁵ and licensed under the MIT licence⁶. This has made it possible to achieve a well designed and good looking webapp without the associated effort. The template contains 10 different pages which have their own URL, and a number of components. The Nuxt app uses 2 of the predesigned pages tailored to our needs. Of the components used in the Nuxt app most are from the template, but significantly the

⁴ <https://nuxtjs.org>

⁵ <https://www.creative-tim.com/product/nuxt-black-dashboard>

⁶ Link to the license statement: <https://github.com/timcreative/freebies/blob/master/LICENSE.md>

component of the actual map, “mapComponent”, the popup component, the contacts page and the store components are custom built. The rest of the components, pages and layouts described in figure 3 are heavily edited by us.

2.6.1. Map page

The map page is one of three pages in the Nuxt app, containing the map that shows air pollution data for different cities. It contains a headline and a map component. The map component is custom built by us, giving us the freedom to choose technology and challenge ourselves to implement the most important part of the Nuxt app ourselves. It is implemented using Mapbox GLJS⁷ and Vue-mapbox. The former is a powerful mapping library to build interactive web maps and the latter is a wrapper for easier integration into Vue JS and Nuxt JS.

The background map used is from OpenStreetMap⁸ and is not retrieved through the flask app, but requested directly from the browser.

On each marker there is a popup implemented by us. It shows the different values for the city and the date from which the values are taken.

It also has a dropdown list for choosing cities. When a city is chosen from this list it is registered by the store. The map component then listens for changes in the stores and updates the map by zooming to the city selected.

2.6.2. Dashboard page

The dashboard page consists of two different graphs describing our data (more on this below). The graph components are designed with the vue-chart.js⁹ library. The graphs show historical data from the last year with one datapoint every week, and historical data for the last month with one datapoint every day.

When the user enters the dashboard page and chooses a city for which they want data to be displayed, a query is sent to the flask app to retrieve the correct data. This is then stored in the “store” component. The graphs listen to and detect any change in this store component and update themselves when the stored value changes.

⁷ <https://www.mapbox.com/mapbox-gljs>

⁸ <https://www.openstreetmap.org/#map=15/45.4690/9.1694>

⁹ <https://vue-chartjs.org/guide/>

2.6.3. Store

The store component is a bit special. It is an implemented vuex¹⁰ store which means it is especially built to be a Single Source of Truth, SSOT. This means that it has the same relation to every component, and is not a child or parent to any component. Its state cannot either be manipulated directly by a component but must be manipulated through an specially implemented mutation which in turn means that its updates can be reacted to by other components.

These characteristics make the vuex store perfect for storing user states, such as the last city chosen. When the user chooses a city in the dashboard this causes a mutation to the store, whose state becomes the new city. Then, the graphs in the dashboard reacts to this change and retrieves the state, queries the new city from the backend and displays it. The same happens if an old user with a last_city stored in the database logs in. The login function mutates the store with the city retrieved from the flask app and causes the graphs to update. When the user logs out, the logout function retrieves the state of the store and sends it to the flask app who stores it in the database as the last_city.

The store is shared between the map page and the dashboard page, meaning that the same process is working for the map page. In the map page, no new data is loaded when the state changes, but the map is zoomed to the city selected. The sharing of the store also means that if you choose a city on the map page, this city will also be chosen in the dashboard and vice versa.

3. User interface design

3.1. Overview

Use of the Creative-Tim template enabled a cohesive design across user interface views, which include 3 main layouts: a map view, a dashboard view, and Contact Us, an informative page about the development team. Likewise, components such as the OpenStreetMap background were chosen with the goal of adhering to platform conventions, thus making the user experience more intuitive. Included here are some of the design considerations, a brief summary of the content in each page, and representations of each view the user can navigate to.

3.2 Design

Visual features of the web app were chosen considering usability and accessibility principles in addition to functional needs. One method used as a guide was heuristic examination to improve our level of compliance to certain recognized usability principles. Nielsen's heuristics are the most

¹⁰ <https://nuxtjs.org/docs/directory-structure/store/>

well-established of these principles and were considered in the development of our user interface. A full list of these heuristics can be found on the Nielsen Norman Group's webpage, but a few examples pertinent to our design follow:

- **Consistency and standards – follow platform and industry conventions.** Consistency was maintained across pages and configurations of the web app through the use of a well structured template. Platform conventions were also respected by choosing recognizable icons (e.g., the user account icon), familiar terminology (e.g., Login/Logout). The cognitive load was also reduced by implementing a familiar base map (OpenStreetMap) that could be navigated without the user learning new actions.
- **Error Prevention.** Certain design features were chosen to avoid high-cost errors. For example, a dropdown menu was inserted into the map and dashboard views to eliminate error prone conditions where a user might misspell an Italian city name, input the Italian version of the name where the English one is required, or input a city not included in the database. While this helped to remove the memory burden on the user when recalling city names, it also avoids a user experiencing a long load time followed by no result.
- **Help users recognize, diagnose, and recover from errors.** Error messages presented in plain language (e.g., “Username needed”), as well as visual treatments like a colour change to orange help our users recover from invalid input in the login or registration use cases.
- **Recognition rather than recall.** Where possible, actions and options were made readily visible by use of bright, contrasting colours and choices rather than empty input fields. For example, the bright pink toggle feature for selecting parameters shown on graphs, or the green dropdown menus providing a list of available cities. Likewise, help is offered in context and only when necessary instead of elsewhere in a tutorial or documentation.

Accessibility was also taken into consideration in overarching features of the web app. The decision to use ‘dark mode’ is a step towards accessibility for users with low vision, and can benefit all users, as inverted text reduces eye strain and fatigue. () Use of a sans serif font with high contrast is helpful for users with low vision, but is also recommended for users with other reading difficulties such as dyslexia.

Lastly, as a team we chose an aesthetic we feel represents our mission to provide a modern product. The colour scheme also reflects our collective mentality and the focus of the project, supporting small scale environmental initiatives.

3.3. Content Tables

Included in these tables are an overview of the content available on each of the main pages of the user interface.

Table 5: content for map view

Main Headline	Title
User Account Icon	Icon expands when clicked and contains links allowing a user to login or register.
Sidebar	Landmarks leading to other pages of the web app
Map	Interactive map with dynamic markers
Map dropdown	List of cities which modifies the map view
Map Popup	Expands when a marker is clicked displaying summary data pertinent to the marker's location (test, floats, dates).

Table 6: Content for Dashboard View

Main Headline	Title
User Account Icon	Icon expands when clicked and contains links allowing a user to login or register.
Sidebar	Landmarks leading to other pages of the web app
Interactive Charts	Interactive charts showing monthly and annual time series for multiple datasets.
Chart dropdown	List of cities which modifies the chart view
Chart toggler	Buttons allow the user to toggle between parameters.

Table 7: Contact Us Page View

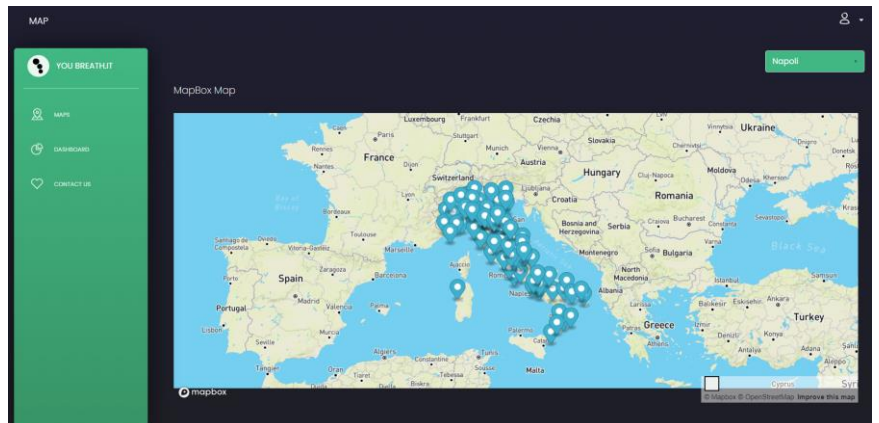
Main Headline	Title
User Account Icon	Icon expands when clicked and contains links allowing a user to login or register.
Sidebar	Landmarks leading to other pages of the web app
Interactive Charts	Interactive charts showing monthly and annual time series for multiple datasets.
Team Cards	Elements containing information on each team member (Name, nationality, brief description).
Email icon	Interactive icon allowing the user to automatically generate an empty email to each team member.

3.4. Views

Each page of the user interface has a number of configurations available to the user. Below is a brief description of the content available in each configuration, or view. Each view is accompanied by an image representing typical behavior of its elements.

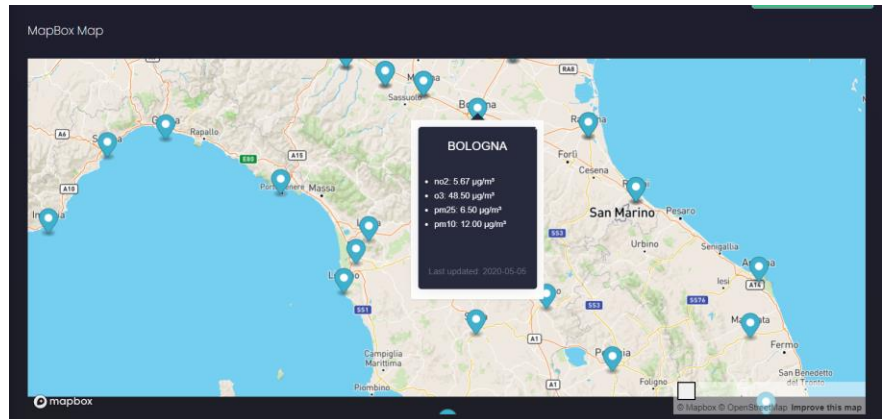
Table 8: Views of the user interface.

View	Interaction Summary
Map	Map displays with markers



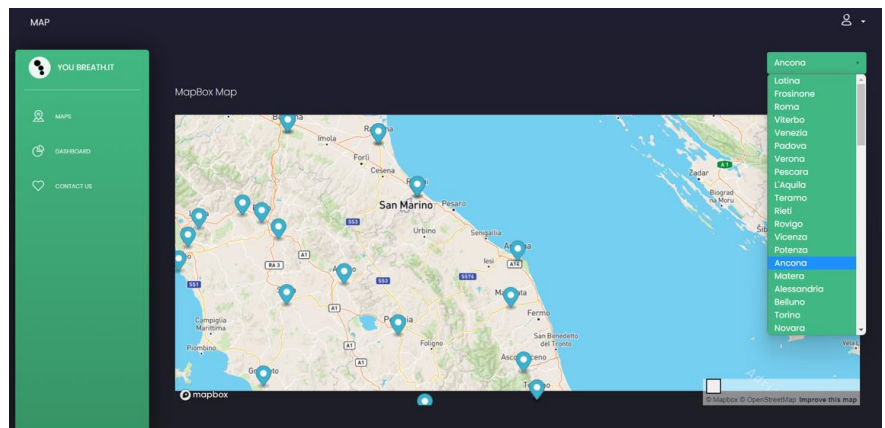
Map Pop-up

Following the selection of a marker (click), a pop-up with the marker's location (city name) and latest measurements is displayed, including the time of the most recent update.



Map dropdown list

The dropdown is displayed in collapsed form when the map is rendered. Clicking the dropdown expands the list and selecting a city (by click) triggers the map to zoom to the marker for that city.



Dashboard

Dashboard displays with two graphs. Graphs change when a parameter is chosen using the buttons of each map.



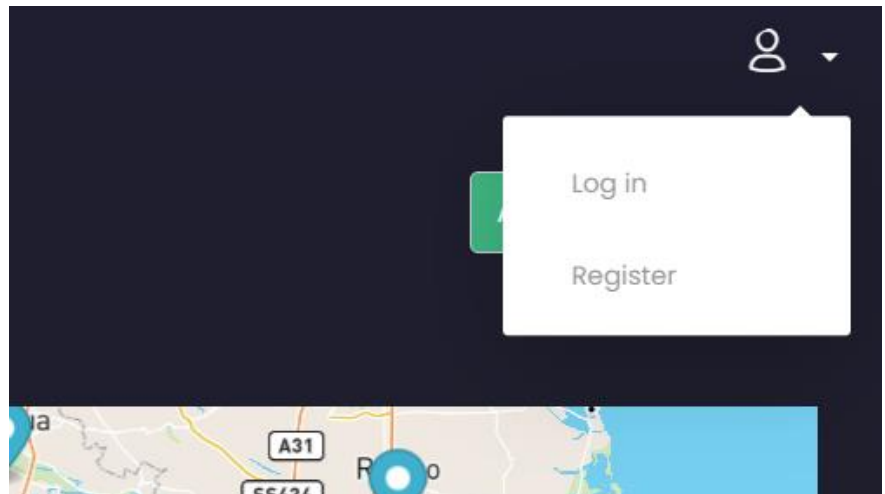
Dashboard
dropdown list

The dropdown is displayed in collapsed form when the dashboard is rendered. Clicking the dropdown expands the list and selecting a city (by click) triggers the charts to display data for the chosen city.



Login/Logout/Reg
ister Buttons

The user account icon expands, allowing the user to choose either login or logout and register.

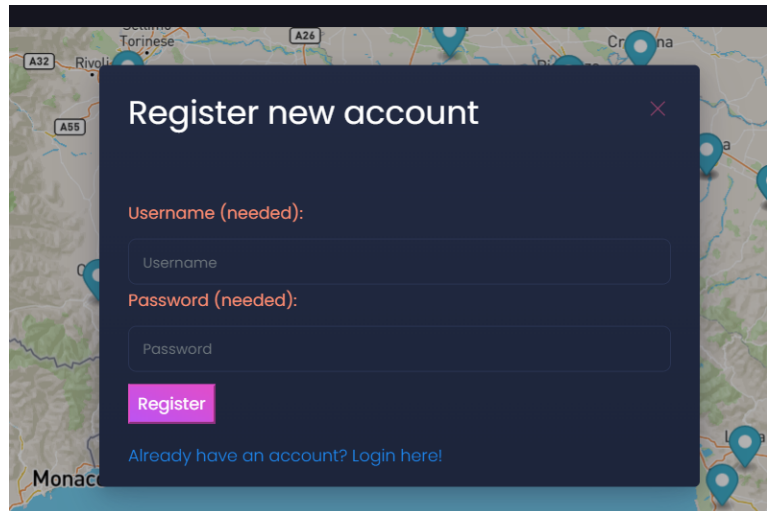


Register (Empty)

User credentials are accepted in the correct format. The user can switch to Login if necessary.

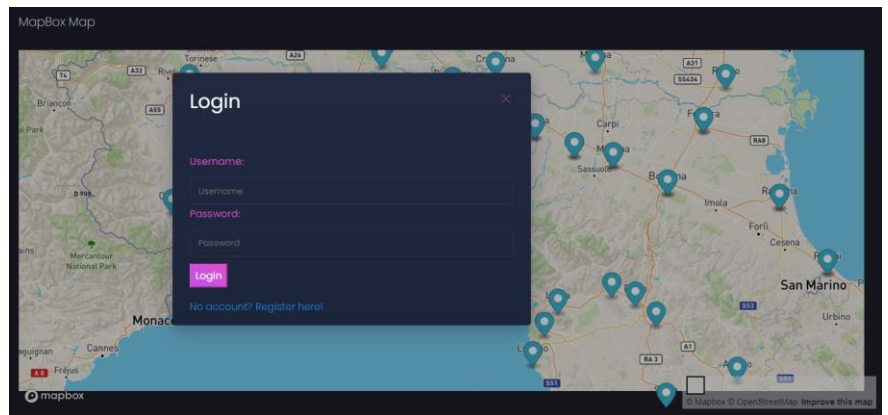
Register (Invalid input)

Suggestions are provided if the user submits invalid user credentials.

A dark blue modal form titled "Register new account" with a close button in the top right. It contains two input fields: "Username (needed):" and "Password (needed):", both with placeholder text "Username" and "Password" respectively. Below the fields is a pink "Register" button. At the bottom, there is a link: "Already have an account? Login here!". The form is overlaid on a map background showing locations like Monaco and Torino.

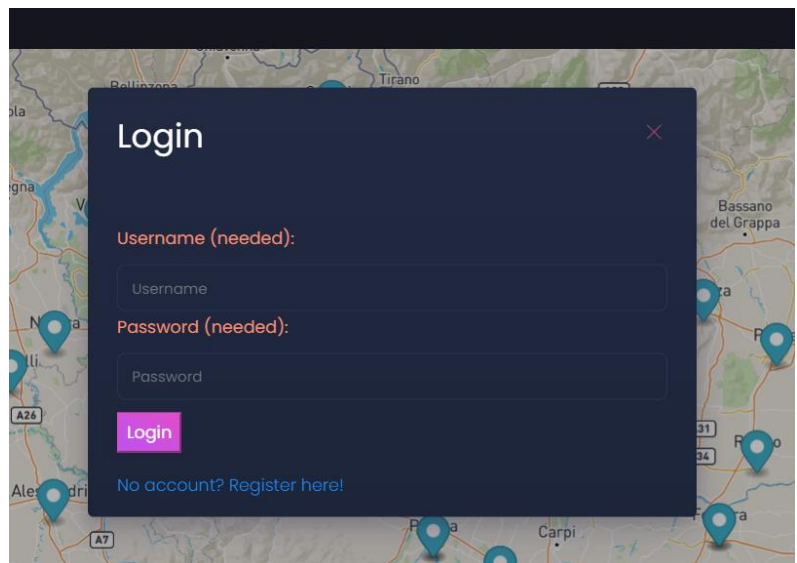
Login

User credentials are accepted in the correct format. The user can switch to Register if necessary.

A dark blue modal form titled "Login" with a close button in the top right. It contains two input fields: "Username:" and "Password:", both with placeholder text "Username" and "Password" respectively. Below the fields is a pink "Login" button. At the bottom, there is a link: "No account? Register here!". The form is overlaid on a map background showing locations like Monaco and San Marino.

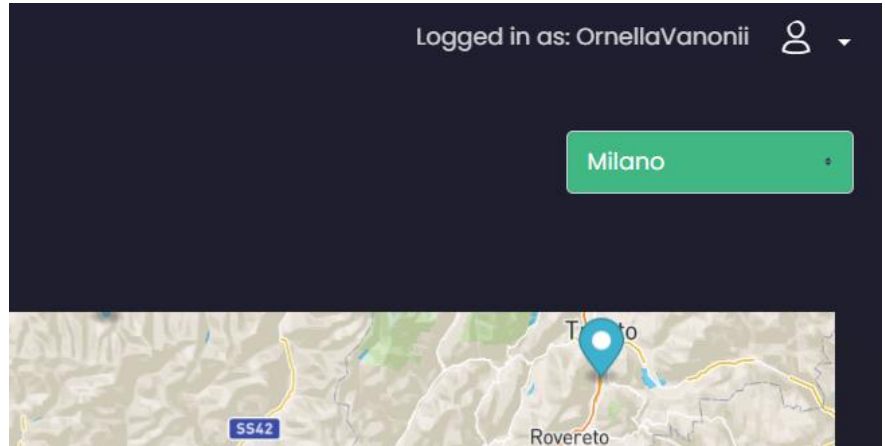
Login (Invalid input)

Suggestions are provided if the user submits invalid user credentials.

A dark blue modal form titled "Login" with a close button in the top right. It contains two input fields: "Username (needed):" and "Password (needed):", both with placeholder text "Username" and "Password" respectively. Below the fields is a pink "Login" button. At the bottom, there is a link: "No account? Register here!". The form is overlaid on a map background showing locations like Tirano and Bassano del Grappa.

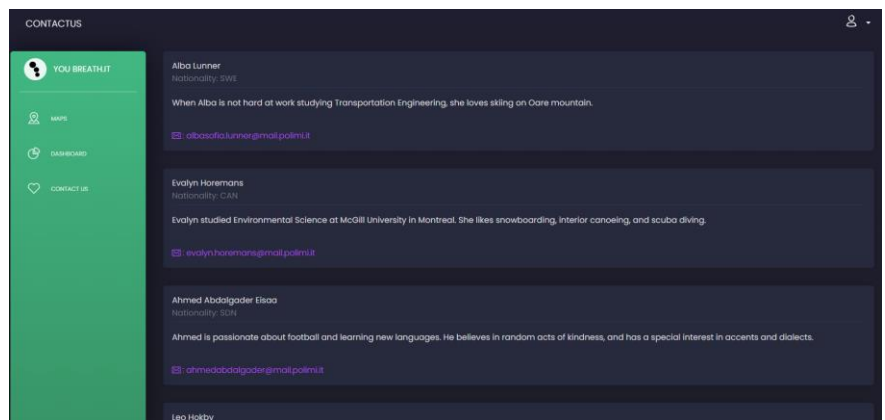
Logged In As

User account icon is modified to reflect the user session.



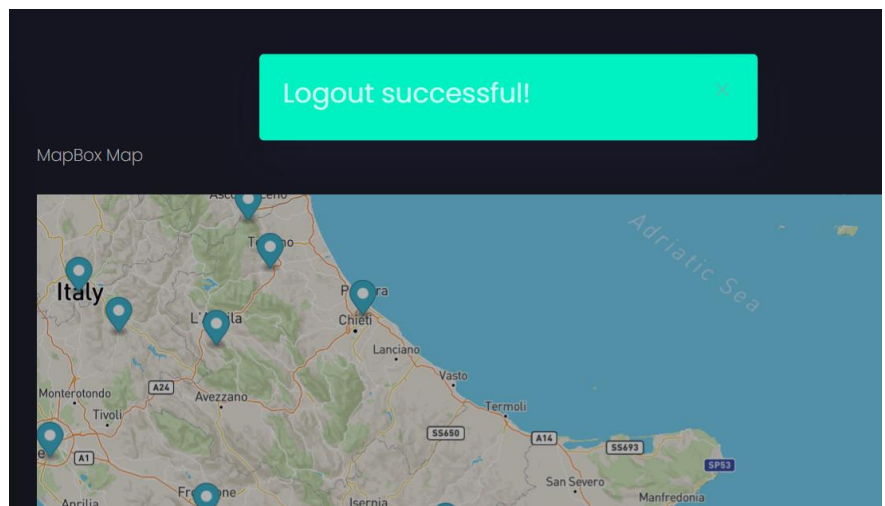
Contact Us

Static details for each team member are displayed.



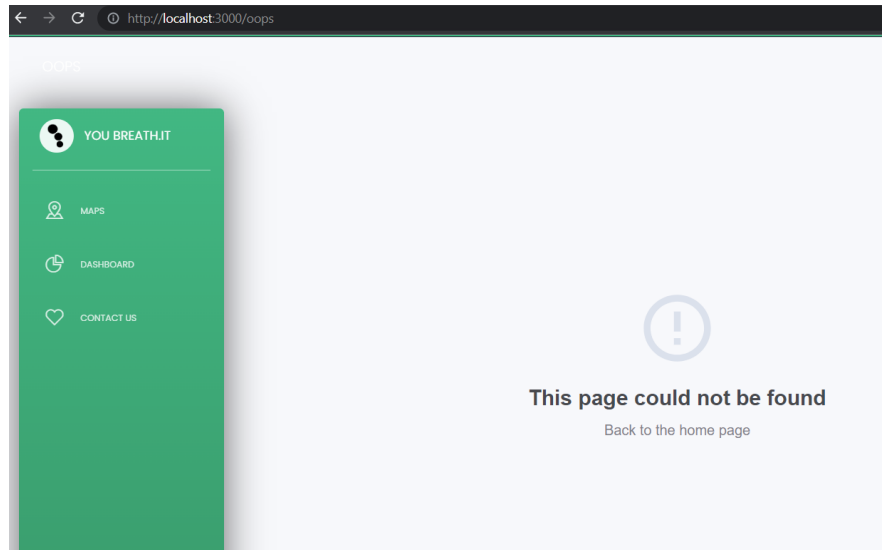
Logout

Username no longer shows in the top right and the user is presented with a "logout successful" message.



Error page

a 404 page not found is shown.



4. Implementation, integration and test plan

4.1.Implementation plan

A decision to keep the implementation method simple was made at the beginning of the project. It was decided to adapt an agile methodology short development cycles and through the duration of the project. The following steps have been repeated numerous times.

4.1.1. Project scope and task creation

The use cases were defined in order to see the issues faced and to define an abstract solution that could be effective. This was then broken down into issues that were added on GitHub. Each issue was described so that the whole team could understand what the issue referred to. Issues was also split so that any feature would be divided on the parts that concerned the backend (flask) and frontend (Nuxt). This meant that developers could accept tasks according to their interests.

4.1.2. Assign team to the implementation process

The produced issues are then self assigned by the developers who have time. The developer creates a new repository where they start building the feature to the specifications of the issue, RASD and discussions with teammates. This provided an agile based development environment; a couple of releases were done before the final one and new issues could pop up and be taken care of dynamically.

4.1.3. Testing new features in a testing environment

Every single part of the software was unit tested during the development process the in order to assure that different functionalities were working properly before being integrated. Each developer had the responsibility to do so for their own features, and the developers would not submit a pull request for their repository until the content had been unit tested.

4.1.4. Onboarding other team members

After a new component is made, it is necessary to make sure that all team members are aware of what it is doing and its effects on all other parts of the project and that the entire team was capable of understanding all parts. This was done at two weekly meetings when each developer had the chance to explain what had been done or what issues they had.

4.1.5. Installation and integration of the software parts

Integrating all parts together is a milestone in the implementation plan and the life cycle of the project, linking the backend to the frontend and linking the backend with the database. This was done through pull requests and merges in GitHub where one team member integrated the new code into all the rest of the project (i.e. the main branch of the GitHub repo). Here, integration tests were also carried out by someone other than the developer who wrote the code. This makes sure that all code is approved by at least two people and that it is checked twice.

4.1.6. Feedback from the team

Every new feature or functionality has to have support from the team members. After performing individual unit tests and successful integration other members' feedback were sought in order to improve the outcome of the work and plan the next development step. At the weekly meetings we also discussed together which new issues we had and went back to step 1 of this process.

4.2. Integration and testing

Testing is divided into three parts, unit testing (on the individual component level), integration testing (testing the interaction of the components) and system testing (testing the complete application).

Integration and testing of the software is considered to be successful only if all tests have been completed successfully. The tests will be executed systematically with specific targets. For the system testing, these are related to the Use cases defined in the RASD (Eisaa et al, 2022).

Black box-testing is used to test the system's ability to provide expected results. Test cases are described under subheading 4.2.3 for all Use cases which include the functional and non-functional requirements from the RASD.

4.2.1. Unit Testing of flask

Unit tests are conducted separately for the flask app and the Nuxt app parts of the project and further divisions of tasks may be made based on functionality. The unit test is done on each component as it is built by the developer who built it .

In the flask app, the unit tests were implemented for each endpoint, checking both the developed functions and the returned data. This was done by sending a request for the endpoint through the browser and comparing the actual result with the expected result, as well as ensuring no exceptions were raised. Tables 9 to 11 describe the tests for each endpoint.

These tests are telling for all parts of the backend since the entire backend exists to handle these requests. If a function is developed and does not produce the required effect in the API or breaks something in the API it is not correct - if it on the other hand produces the expected results from the tables below, it passes the test.

Table 9: Test for endpoints without inputs

Endpoint	Test	Hypothesis on system:	Expected outcome
api/locations	Send request through browser	Flask server turned on and listening to localhost:5000	JSON string matching format in table 1 returned
api/latest	Send request through browser	Flask server turned on and listening to localhost:5000	JSON string matching format in table 1 returned
api/cities	Send request through browser	Flask server turned on and listening to localhost:5000	JSON string matching format in table 1 returned
api/contacts	Send request through browser	Flask server turned on and listening to localhost:5000	JSON string matching format in table 1 returned
api/anythingelse	Send request through browser	Flask server turned on and listening to localhost:5000	404 not found error

Table 10: Tests for endpoint with city as input

Endpoint	Test	Hypothesis on system:	Expected outcome
*/<city> ^{11 12}	Send request through	Flask server turned on	JSON string matching

¹¹ */ means that all endpoints with this ending are tested like this.

¹² A sample of 5 different cities are tested, along with any others desired by the developer. Roma, Firenze, Milano, Salerno and Napoli.

	browser with existing city	and listening to localhost:5000	format in table 1 returned
*/<City>	Send request through browser	Flask server turned on and listening to localhost:5000	JSON string matching format in table 1 returned
*/<CITY>	Send request through browser	Flask server turned on and listening to localhost:5000	JSON string matching format in table 1 returned
*/<notacity>	Send request through browser	Flask server turned on and listening to localhost:5000	400 bad request error, "city notacity doesn't exist in database"

Table 11: Unit tests for POST endpoints with payload

Endpoint	Payload	Test	Hypothesis on system:	Expected outcome
/api/register	{"username": <String>, "password": <String>}	Send request from Nuxt app	Flask server turned on and listening to localhost:5000. The username does not exist in database	JSON string matching format in table 1 returned, new user registered in database
/api/register	{"username": <String>, "password": <String>}	Send request from Nuxt app	Flask server turned on and listening to localhost:5000. The username already exist in database	JSON string matching format in table 1 returned, no change in database
/api/register	{"anything": <Whatever>, "other": <Something>} ¹³	Send request from Nuxt app	Flask server turned on and listening to localhost:5000.	400 bad request error.
/api/authenticate	{"username": <String>, "password": <String>}	Send request from Nuxt app	Flask server turned on and listening to localhost:5000. Username and password is correct	JSON string matching format in table 1 returned

¹³ This signifies any payload that does not have the correct format. The developer will choose the format.

Table 11: Unit tests for POST endpoints with payload

Endpoint	Payload	Test	Hypothesis on system:	Expected outcome
/api/register	{ "username": <String>, "password": <String> }	Send request from Nuxt app	Flask server turned on and listening to localhost:5000. The username does not exist in database	JSON string matching format in table 1 returned, new user registered in database
/api/authenticate	{ "username": <String>, "password": <String> }	Send request from Nuxt app	Flask server turned on and listening to localhost:5000 username or password is incorrect	JSON string matching format in table 1 returned
/api/authenticate	{ "anything": <Whatever>, "other": <Something> }	Send request from Nuxt app	Flask server turned on and listening to localhost:5000.	400 bad request error.
/api/logout	{ "username": <String>, "lastsearch": <String> }	Send request from Nuxt app	Flask server turned on and listening to localhost:5000. Username and lastsearch is correct	JSON string matching format in table 1 returned. "lastsearch" added at user row in database
/api/logout	{ "anything": <Whatever>, "other": <Something> }	Send request from Nuxt app	Flask server turned on and listening to localhost:5000.	400 bad request error.

4.2.2. Unit Testing of Nuxt

Unit tests on the Nuxt app were made by testing each call to the flask app by printing the returned value of the call to the console. Then, the built feature or interface is tested by launching a development version of the website and navigating to the page containing the feature. There, the feature is tested by the developer interacting with it and making sure that the output is as expected.

Steps include:

- Checking the input data listed in the console
- Checking the representation, namely in the specific component; assuring that the right data are being used and presented.
- Testing the dynamicity and the interactivity of each component with the possible different inputs.

4.2.3. Integration tests

Integration tests are carried out to ensure that separately developed pieces of code are able to function together. Integration of code is done when merging branches in the project GitHub. If the merged code is able to run successfully, without any errors, the integration test has been successful. After each merge, the person responsible for the merge runs the website and tries every function that is affected by the recent merge and is supposed to work in that version of the web app. Table 12 reports which tests will be done when their respective functionality is integrated.

Table 12: Integration tests to be carried on functions that are considered finished

Function	Test	Hypothesis on system	Expected outcome
Map	Navigate to the map page, refresh the page.	Set-up complete ¹⁴	Map displays with markers
Pop-up	Navigate to the map page and click a number of markers	Set-up complete	Map displays with markers, when markers are clicked, they produce a pop-up with correct information
Map dropdown list	Navigate to the map page, choose a city from dropdown list	Set-up complete	Map displays, dropdown displays. Clicking a city in the dropdown causes the map to zoom to that city.
Contacts page	Navigate to the contacts page	Set-up complete	User is shown a list of the project members and their information
Dashboard	Navigate to the dashboard page and click on different parameters for the graphs	Set-up complete	Dashboard displays with two graphs. Graphs change when a parameter is clicked.

¹⁴ Flask and Nuxt servers are running, database has been initialized

Dashboard dropdown list	Navigate to the dashboard and choose a city from the dropdown list.	Set-up complete	Dashboard displays as above, and data for the selected city are shown.
Login	Click the “person” icon on the top right corner and open the login dialog. Fill out username and password in the dialog	Set-up complete. User is not logged in Username and password are correct.	User is logged in and the username is displayed in the top right.
Login	Open the login dialog. Fill out username and password in the dialog	Set-up complete. User is not logged in. Username or password are not correct.	User is presented with a “Authentication failed” message.
Login	Open the login dialog. Fill either username or password in the dialog	Set-up complete. User is not logged in	User is presented with a “Both username and password are required” message.
Login to register and back	Open the login dialog. Click the link to register. Click the link to login	Set-up complete. User is not logged in	User is moved to the register dialog and then back to the login dialog.
Register	Click the “person” icon on the top right corner and open the register dialog. Fill out username and password	Set-up complete. User is not logged in. Username is not in the system	User is created in the database and user is presented with a “account creation successful” message
Register	Open the register dialog. Fill out username and password	Set-up complete. User is not logged in. Username is already in the system	User is presented with a “Username is already in use” message
Register	Open the register dialog. Fill either username or password in the dialog	Set-up complete. User is not logged in.	User is presented with a “Both username and password are required” message.
Logout	Click the “person” icon and click “logout”	Set-up complete. User is logged in. User has selected a	Username no longer shows in the top right and the user is presented with a “logout

		city.	successful” message. Last Search is added to database
Logout	Click the “person” icon and click “logout”	Set-up complete. User is logged in. User has not selected a city.	Username no longer shows in the top right and the user is presented with a “logout successful” message. No change to database
Logout login	Login a user, search a city in the dashboard, logout. Reload page and log in again	Set-up complete.	On the second log in, the dashboard defaults to the city chosen before.
Error page	Navigate to a non existing page by typing http://localhost:3000/anything in url bar	no page called “.../anything” has been implemented	a 404 page not found is shown.

4.2.4. System Testing

System testing is done at the end when a product is developed. It will be done by executing a set of predefined test cases. The test cases in table 13 for system testing are directly related to the use cases in the RASD. The results of these tests are reported in the test report document.

Table 13: Systems tests and their relations to use cases.

Function and Use Case	Test	Hypothesis on system	Expected outcome
Map U1 U2	Navigate to the map page, refresh the page.	Set-up complete ¹⁵	Map displays with markers
Pop-up U1 U2 U3	Navigate to the map page and click a number of markers	Set-up complete	Map displays with markers, when markers are clicked, they produce a pop-up with correct information

¹⁵ Flask and Nuxt servers are running, database has been initialized

Map dropdown list U1 U4	Navigate to the map page, choose a city from dropdown list	Set-up complete	Map displays, dropdown displays. Clicking a city in the dropdown causes the map to zoom to that city.
Contacts page	Navigate to the contacts page	Set-up complete	User is shown a list of the project members and their information
Dashboard U1 U3 U4	Navigate to the dashboard page and click on different parameters for the graphs	Set-up complete	Dashboard displays with two graphs. Graphs change when a parameter is clicked.
Dashboard dropdown list U1 U4	Navigate to the dashboard and choose a city from the dropdown list.	Set-up complete	Dashboard displays as above, and data for a selected city are shown.
Login U6	Click the “User” icon on the top right corner and open the login dialog. Fill out username and password in the dialog	Set-up complete. User is not logged in Username and password are correct.	User is logged in and the username is displayed in the top right.
Login U6	Open the login dialog. Fill out username and password in the dialog	Set-up complete. User is not logged in. Username or password are not correct.	User is presented with a “Authentication failed” message.
Login U6	Open the login dialog. Fill either username or password in the dialog	Set-up complete. User is not logged in	User is presented with a “Both username and password are required” message.
Login to register and back U5 U6	Open the login dialog. Click the link to register. Click the link to login	Set-up complete. User is not logged in	User is moved to the register dialog and then back to the login dialog.
Register U5	Click the “person” icon on the top right corner and open the register dialog. Fill out username and password	Set-up complete. User is not logged in. Username is not in the system	User is created in the database and user is presented with a “account creation successful” message
Register U5	Open the register dialog. Fill out username and password	Set-up complete. User is not logged in. Username is already in the system	User is presented with a “Username is already in use” message

Register U5	Open the register dialog. Fill either username or password in the dialog	Set-up complete. User is not logged in.	User is presented with a “Both username and password are required” message.
Logout U7	Click the “person” icon and click “logout”	Set-up complete. User is logged in. User has selected a city.	Username no longer shows in the top right and the user is presented with a “logout successful” message. Last Search is added to the database.
Logout U7	Click the “person” icon and click “logout”	Set-up complete. User is logged in. User has not selected a city.	Username no longer shows in the top right and the user is presented with a “logout successful” message. No change to database.
Logout login U6 U7	Login a user, search a city in the dashboard, logout. Reload page and log in again	Set-up complete.	On the second log in, the dashboard defaults to the city chosen before.
Error page	Navigate to a non existing page by typing http://localhost:3000/anything in url bar	No page called .../anything has been implemented	A 404 page not found is shown.

5. References

Eisaa et al. (2022). *Requirement Analysis and Specification Document - PRESENTATION OF AIR POLLUTION DATA USING AN INTERACTIVE WEB MAP*.

OpenAQ. n.d. Available: <https://openaq.org/#/> [Fetched: 2022-05-26]

Kovacic, D., & Beard, K. (2021, April 29). *Accessibility inspired: Dark mode*. Habanero Consulting Inc. Retrieved June 7, 2022, from <https://www.habaneroconsulting.com/stories/insights/2021/accessibility-inspired-dark-mode>

World Leaders in Research-Based User Experience. (n.d.). *10 usability heuristics for user interface design*. Nielsen Norman Group. Retrieved June 7, 2022, from <https://www.nngroup.com/articles/ten-usability-heuristics/>

Choosing dark mode for low vision. Veroniiiica. (2020, August 25). Retrieved June 7, 2022, from <https://veroniiiica.com/2020/05/15/dark-mode-for-low-vision/>