Access To Recreational Quality (ARQI) Website Documentation

In this documentation, decisions around the design, the functionality of the webpages/web map and further bespoke resources will be discussed and provided. The document will be split as follows:

1. Flask
2. Homepage
3. Interactive Web Map
4. Query and Download
5. About Us

Flask

The website was developed using Flask, a Python-based micro-web framework. This was used due to its capability to provide structure in our webapp. The code is organised into a main folder and sub-directories:

```
CGS
├── main.py
├── recreation
│   ├── __init__.py
│   ├── query.py
│   ├── views.py
│   └── webmap.py
├── static
│   ├── .css
│   ├── .jpg/.png
│   ├── .js
│   ├── .geojson
│   ├── .csv
│   ├── .ico
│   └── .pdf
├── templates
│   ├── base.html
│   ├── homepage_ARQI2.html
│   ├── query_download.html
│   ├── about_us.html
│   └── webmap.html
├── README.md
└── Group_6_Documentation.pdf
```

The **`recreation`** acts as the main folder storing all the Python files used to create the website.

`main.py`, located outside the **`recreation`** folder, imports all the app routes and instantiates the webapp. This can be run from the terminal when we want to start the website. In the development server, this returns a URL in development mode. When deploying the website, we used *Gunicorn* – a Python HTTP server for WSGI applications – to serve the webapp.

`__init__.py` makes the directory **`recreation`** a Python module package and stores a function that registers all extensions to the webapp via *Blueprint*.

`views.py` stores all of the main routes or the URL end points for the front-end aspects of the website, where users can navigate between pages e.g. The Homepage and *About Us.* The *Blueprints* of these are then imported and registered in the `__init__.py` file.

`query.py` stores the code used for the query and download application page.

`webmap.py` stores the code created for the interactive map.

The **`static`** folder contains all 'decorative' files such as CSS, JS and PNG/JPG files as well as datasets used for the web map.

In **`templates`** are files that contain all HTML files as well as the base template to which all elements contained within extend to the other HTML templates. Flask uses the Jinja2 template library to render templates dynamically. This ensures every webpage in the application will have the same basic layout. Special delimiters are used to differentiate between Jinja2 syntax and the static data in the template. Code between `{{ … }}` calls the variable in the Python files to be rendered in the HTML files. Meanwhile, `{% … %}` indicates a control flow statement. Where a call to a specific function in the python files or CSS files are required, this is achieved using `{{ url_for }}`.

Key Resources:

https://flask.palletsprojects.com/en/2.0.x/

https://docs.gunicorn.org/en/stable/

Homepage

Both CSS and JavaScript were used to control web design and its interactive elements. The CSS files used were a mixture of self-generated elements as well as inheritance from the W3Schools CSS template. Below, I highlight further resources that point to bespoke plugins used:

Slick Carousel:

This was created using Slick developed by Ken Wheeler which uses a jQuery plugin to power the responsive carousel. Further JavaScript was added to control the cycle speed in line with the client's browsing pace.

http://kenwheeler.github.io/slick/

AnyChart Word Cloud:

To allow for a more succinct and visually pleasing way of displaying survey data, the AnyChart API was utilised to provide cross-browser HTML5/JavaScript charting. The data was first transformed into JSON format and hard-coded into the HTML file before JavaScript was employed to create and customise the chart.

https://docs.anychart.com/Quick_Start/Quick_Start

Interactive Web Map

The interactive web map uses a Python wrapper library called `Folium` to visualize spatial data on a Leaflet map.

We used four built-in tile sets from OpenStreetMap, Stamen and CartoDB, giving the user a range of choices for a base map.

Several CSV and GEOJSON files are used as static data for the map layers. These are imported using the `os` module and read in with `pandas` library; the latter also provided means of manipulating the data. Custom legends are created with `branca.colormap`, a utility module for dealing with colour maps.

In order to reduce client efforts e.g. through excessive clicking to retrieve information, the map layers are coded to enable a "hovering" effect, thereby showing key information once the cursor overlays a polygon. The user has several choices for visualizing the data including the ability to add, combine and minimise map layers depending on their needs.

To enhance user interactivity, the web map uses additional plugins to deliver the distance/area measure widget using `MeasureControl`; zoomable density maps using `MarkerCluster`; and finally, a heatmap using `Heatmap`.

Collectively, the web map comprises of the following layers:

1. ARQI choropleth map – Edinburgh layer displays the ARQI for each Datazone.
2. Greenspaces choropleth map – Edinburgh greenspaces layer displays only those used in our study.
3. Marker Cluster map – Edinburgh Greenspaces layer shows access points for each greenspace in the study.
4. Heatmap – Same layer as the access points, only displays the clustered access points.
5. SIMD – Scottish Index of Multiple Deprivation layer allows to compare results with our ARQI.

Key Resources:

https://python-visualization.github.io/folium/

Interactive maps (autogis-site.readthedocs.io)

Query and Download

Python scripts were written to extend query functionality from the server-side to the HTML. In `query.py`, two functions were defined. The first establishes a connection to the DB using *cx_Oracle*, and a SQL query written to assist with the retrieval of all (recorded) greenspace names in alphabetical order. This therefore allowed the drop-down menu items rendered in the `query_download.html` file to be populated dynamically using Jinja2 templating. The second drop-down menu containing all three services were generated manually as these were not recorded in the DB and did not require a great deal of effort to hard code in the HTML file.

The HTML forms (drop-down menus) used the POST method to send data to the server side, and the `action` attribute was defined to display retrieved data in another URL (*/submitted*). When inputs are submitted, the first element (in this case, first word of the greenspace name) is submitted (e.g. Inverleith, and not Inverleith Park). This triggers the second function in the Python script which uses an SQL bind variable to retrieve the correct data from the DB tables. Whilst the adage of "never trust the user's input" is applicable here, the use of "sanitised" data in the HTML forms avoids SQL injection attacks in general use-cases.

Finally, table headers are rendered onto HTML alongside the retrieved rows of data (in tuple format) using *flask.MarkUp*. Note: this only appears to work on Safari and Firefox.

Additionally, data download is offered as well to increase functionality for a range of users. The overall webpage is designed using the same CSS files as the other webpages to maintain consistency.

Key resources:

https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world

https://cx-oracle.readthedocs.io/en/latest/user_guide/bind.html

https://flask.palletsprojects.com/en/2.0.x/templating/

https://tedboy.github.io/flask/generated/generated/flask.Markup.html

About Us

CSS and JavaScript are used to design web pages.

CSS achieves the layout effect we want by controlling the width, height, float, text size, font, background and other styles of the HTML tag object. JavaScript is used to add dynamic functions and interactive behaviours to HTML web pages. In the About Us page, it is mainly reflected in the pop-up window of the button on the top and the send button in the contact us section.