COMP30830

Dublin Bikes Report

Group Number: 22

EC2 Address: http://16.170.230.66/

Github:

https://github.com/Glenmic/SWE-Project
https://github.com/BCushen/SWE-Project_ZIP

Team Members and Contributions:

1. Brian Cushen (33%)
2. Michael Glennon (33%)
3. Joel Sajan (33%)

A web application to display occupancy and weather information for Dublin Bikes

COMP 30830

Brian Cushen, Michael Glennon and Joel Sajan

A Report detailing the process of developing a web application for the Software Engineering (Conv.) module (Prof. Aonghus Lawlor)

School of Computer Science

University College Dublin

19 April 2024

Contents:

Section 1: Statement of project objectives and application requirements, team structure

Outlined in the requirements document that was received on starting the project, the finished application was to contain the following main elements:

1. Data collection through JCDecaux API
2. Data collection through OpenWeather API
3. Data management/storage through RDS DB on AWS
4. Displaying of bike station on map
5. Flask web application
6. Occupancy information
7. Weather information
8. Interactivity (click, API request, handle response)
9. ML model for predicting occupancy based on weather patterns, trained on collected data
10. Project served on a name host over the web on an EC2 instance
11. Project must be developed with and stored in a github repository

The overall goal of the project was to implement a web application that displays a map of Dublinbikes stations and which uses a machine learning model to analyse a repository of data acquired from JCDeceaux and OpenWeather to predict bike and bike stand availability based on historical JCDeceaux bike availability data and historical weather data. The project required a record of project management, including planning notes, mockups, discussions, documentations of meetings, sprint retrospectives and planning, sprint backlogs and burndown charts.

For the purposes of task delegation and overall project management, the requirements were roughly split between members - one member was to work mostly on frontend, two members were to work mostly on backend, with some variability that depended on what outcomes we wanted to achieve during each sprint, and what level of interaction was required between different elements of the frontend and backend software.

With respect to the backend, the generation and population of the RDS was handled by one member, and the scripts that requested data from APIs and conducted data analysis/ML model generation were handled by a second member.

Communication of progress between members was achieved via daily scrum meetings, as required by the project description, and further communication was facilitated via a Slack channel/Whatsapp, for specific communication of ongoing issues regarding the codebase, mostly related to those issues which needed more than one individual's involvement in bridging the backend and the frontend elements.

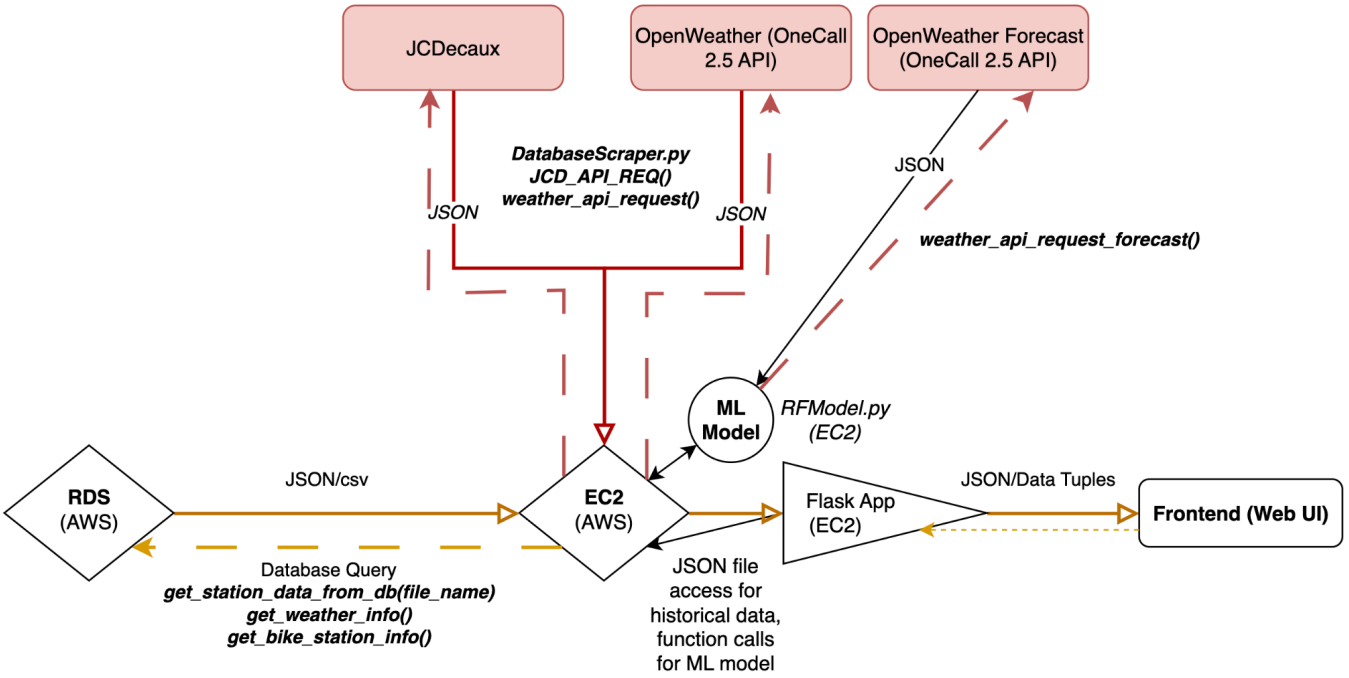
Section 2: Overview of Structure and User-flow Diagram

System Structure and Data Flow:

The general overview of the application architecture is depicted in Figure 1. The application is hosted on an AWS EC2 instance. This instance contains a Python script titled *DatabaseScraper.py* which is executed periodically at intervals of five minutes using Cron. This script retrieves Dublinbikes availability and weather data from JCDeceaux and OpenWeather respectively and stores the information in a database hosted on an AWS RDS instance.

Upon loading the website, the latest bike availability data is requested from the front end via a Flask application named *app.py* and a library of database access functions hosted in a python module named *DatabaseAccess.py*. This information is used to display bike stations and their associated bike and stand availability on the front end. The latest weather is also queried from the database to allow the user to see this information on the website. When planning a future trip, the specific date, time, start station and destination station information is sent to Flask from the front end. In conjunction with an OpenWeather API request for the weather forecast, the planned trip information is used as input to machine learning functions hosted in a module named RFModel.py and a machine learning model trained on historical (collected) JCDeceaux and OpenWeather data is queried to return the predicted availability of bikes and stands at the start and end points of the trip.

Figure 1. General overview of application architecture.

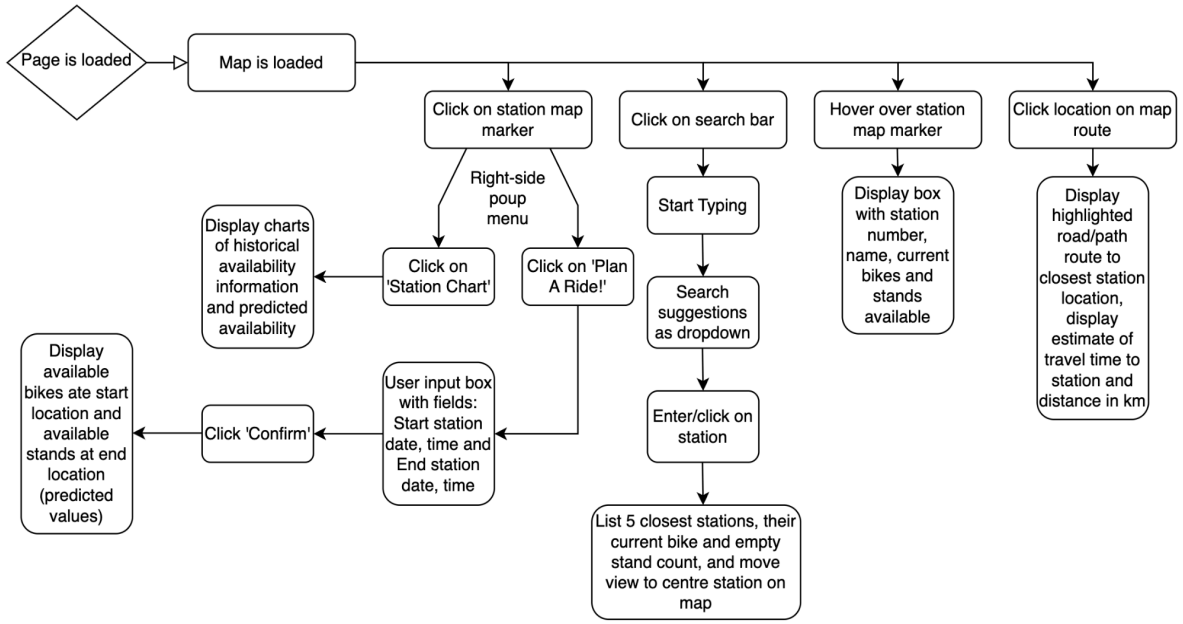


User-flow Diagram:

Upon loading the site, user interaction follows the flow described in Figure 2. Initially the map is loaded after which the user may interact with the site as follows.

- Clicking on a station map marker gives the user information about the current bike and stand availability. It also allows the user the option to click the Station Chart button which displays charts of historical and predicted availability for the station. The user may also select the Plan A Ride button which will then present the user with a popup to allow input of planned trip details and return predicted availability of bikes at the start point and availability of stands at the end point.
- The user may enter an Eircode (e.g. D02 PN40 for Trinity College) in the search bar which will return the nearest five bike stations along with bike and stand availability data.
- Hovering over a map marker displays availability for the selected stand.
- Clicking on any location on the map is not a station marker displays the route to the closest station to that point along with distance and travel time information to that point.

Figure 2. User Flow Diagram



Section 3: Architecture - Feature Implementation Description and Discussion

The purpose of this section is to provide a detailed discussion of the project implementation previously described in Section 2 - Overview of Structure and User-flow Diagram. This discussion will be divided into the stack sections developed to implement the project (i.e. front-end, back-end (Flask), machine learning, database implementation) in addition to the platforms utilised to host the web application and associated database.

The following libraries, resources and languages were used in development of the application.

Back-End

- Python
- Python Libraries: Pandas, Pickle, JSON, Flask, datetime, SQLAlchemy, requests, time, os, Matplotlib, Scipy, Numpy, SKLearn
- SQL (mySQL)
- Gunicorn
- Nginx
- Amazon Web Services EC2 and RDS

Front-End

- JavaScript
- JavaScript Libraries: maps.googleapis.com/maps/api, cdn.jsdelivr.net/npm/chart.js
- HTML
- CSS

Note on Code Cleanup and Code In Use

Note that during execution of the project code cleanup was performed. However, due to GitHub commit reversions, this work was undone and was not performed again prior to application deployment in error. Note that the following code is not used in the final deployed application.

- All files in the “API Testing” folder.
- All files in the “Database Code” folder.
- The following files in the “DB” folder: “database_access.py”, “database_creation.py”, “databasequerytest.py, and “Integrated Scraper.py.” Note lowercase naming.

The specific files used for API calls, database creation, database population and database querying are located in the “DB” folder and are named “DatabaseScraper.py,” “DatabaseCreation.py” and “DatabaseAccess.py.” These functions are discussed later in detail.

Front End Implementation

While understanding the project requirement, we were discussing how we need to handle various components of the project. We needed to break down the wireframe to accommodate how the UI from the front end looked. Starting with we designed our UI on paper first and discussed it with the team. One of us was in charge of the design so that others can handle and discuss other sections like DB management and Web scraping. Once designs on paper are done in our daily scrum meetings we discuss the possibilities of how to make the design better.

After brainstorming for a week we finalised the design and started creating wireframe on mockplus to give life to the paper drawings. Our designs were later take to a series of discussions to finally reach the design we are having now.

Figure 3. Initial sketch of website design.

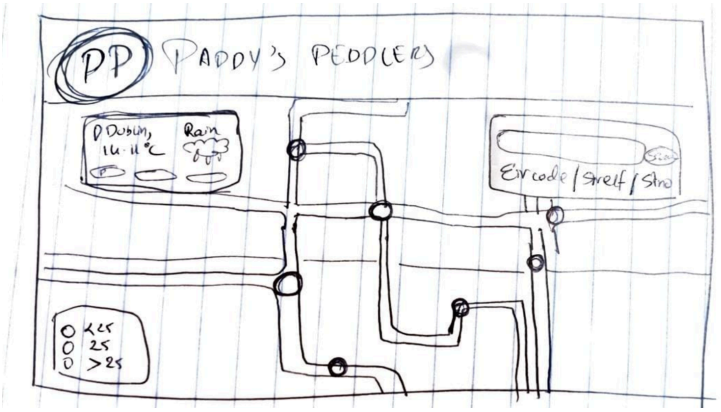
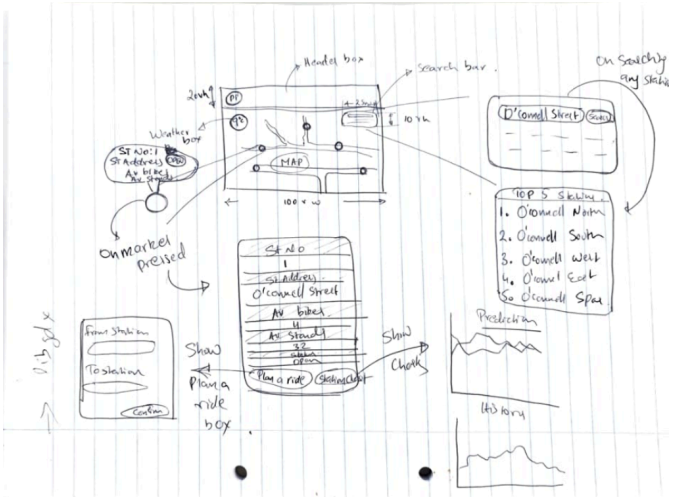


Figure 4. Basic wireframe showcasing the UI Path of the features.



Wireframe in Mockplus

We were using the Mockplus tool to design the initial page structure to specify how the UI looked. We were not fixed on the colours yet, so created a wireframe with adjusting the padding to look how the website will view in mobile and laptop mode. Later we implemented the UI with the inspiration from the designs made.

Figure 5. Wireframe mockup for website load page.

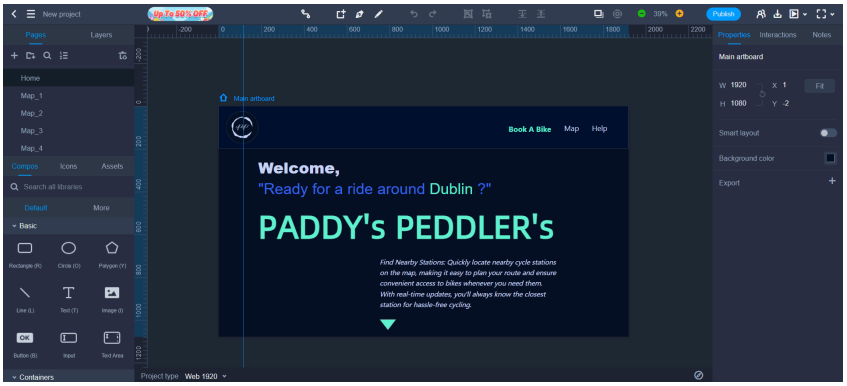


Figure 6. Wireframe mockup for station display on map.

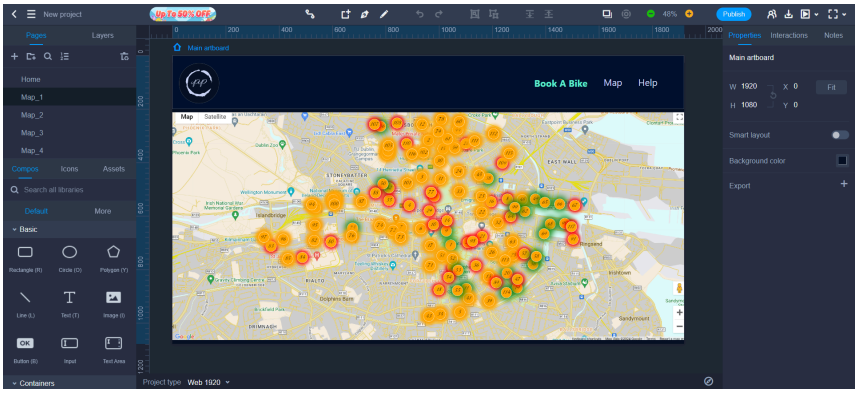
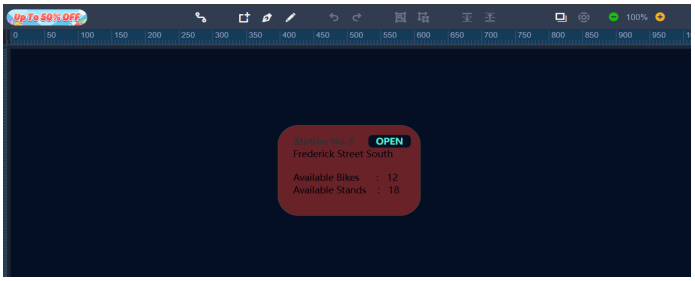


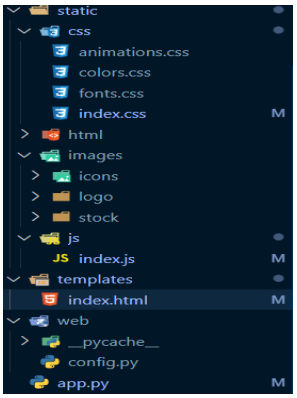
Figure 7. Wireframe mockup for bike station information pop-up.



Creating a Basic Project Structure

A good file skeleton makes the work forward easier, we have a structure made with 3 main folders, Static, Templates and Web.

Figure 8. Overview of front-end directory structure.



Static:

- Css - animations/colors/fonts/index
- Html - header
- Images - icons and logo
- Js - index.js

Templates:

- index.js

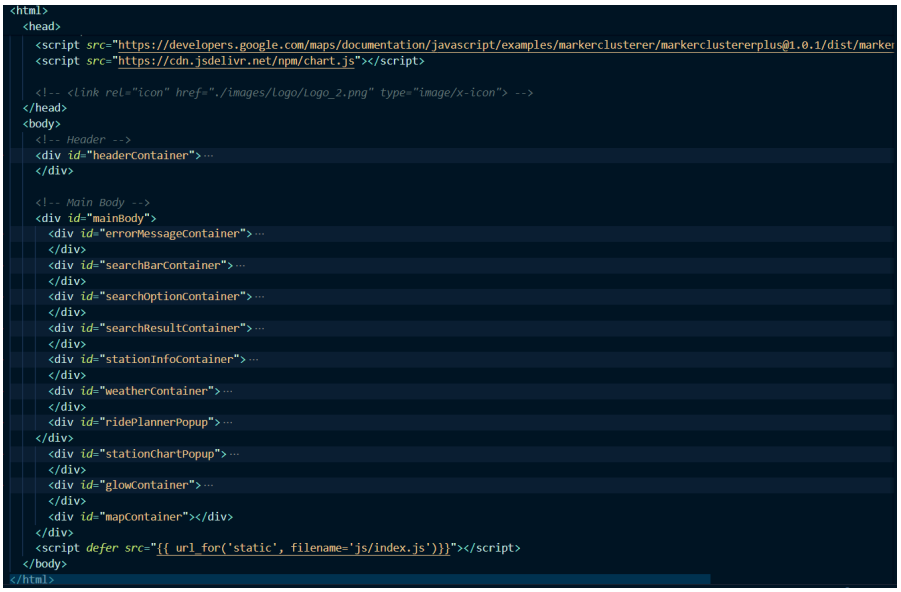
Web:

- app.py

HTML (index.html)

We have added containers for each section of the html file, while each section is made simple for ease of understanding of the other teammates and the invigilator. Html file is written in blocks of code and each of them align properly on one another. Adequate comments are added for ease of understanding code.

Figure 9. Outline of the index.html file.

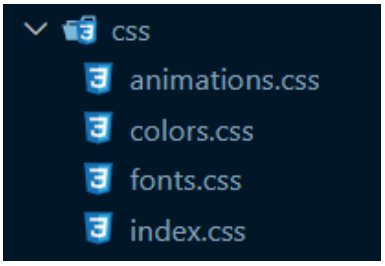


CSS (Cascading Style Sheets)

CSS has been segregated into 3 parts.

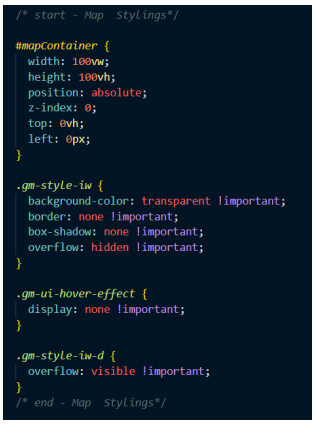
The main CSS file, which is the index.css and another CSS for Fonts, Colors and Animations. The reason why we have segregated the CSS in this way is because it is easier to track and find the files and align more to the SOLID principles.

Figure 10. Directory structure and files for CSS stylesheets.



Each section is segregated by a start and an ent comment line for ease of understanding and keeping the code clean. We have used # for id and _ for class tags.

Figure 11. Examples of CSS stylesheets.



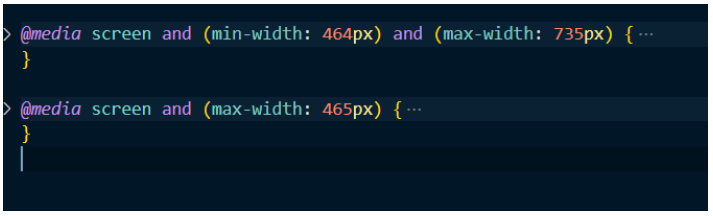
```
/* start - Map  Stylings*/

#mapContainer {
  width: 100vw;
  height: 100vh;
  position: absolute;
  z-index: 0;
  top: 0vh;
  left: 0px;
}

.gm-style-iw {
  background-color: transparent !important;
  border: none !important;
  box-shadow: none !important;
  overflow: hidden !important;
}

.gm-ui-hover-effect {
  display: none !important;
}

.gm-style-iw-d {
  overflow: visible !important;
}
/* end - Map  Stylings*/
```



```
@media screen and (min-width: 464px) and (max-width: 735px) {
}

@media screen and (max-width: 465px) {
}
```

We have added a media query for accommodating different types of views, so the webpage is adequate for any screen size. We have defined the div based on VH and VW thus making sure the UI adapts to the screen sizes making it work in any screens.

The entire website is made with just these many colours which is added to a common colours file, Advantage is that the theme of the Webpage can be controlled by changing these 4 variables.

Figure 12. CSS colour properties.

```
/*
 ┌─┐
 Colors
 └─┘
 */

:root {
  --primary_purple: #5e1675;
  --small_glow: #c750ef;
  --medium_glow: #8a3ca4;
  --big_glow: #3b084d;
}
```

JS (JavaScript)

Javascript is done by segregating each section of the code, In the long run the code was simplified a lot because of this. The names follow camelCasing and continues throughout the project

Figure 13. Example of index.js file.

Logo Design

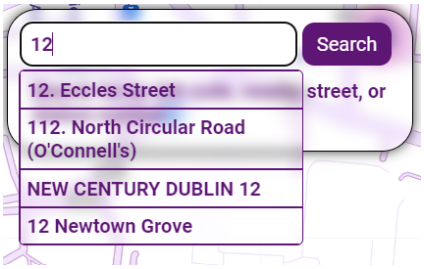
Figure 13.5. Logo finalised for the project..

Webpage UI Implementation

1. **Custom Search Bar**

The custom search bar is responsible for searching either the station number, your location name or a specific street. It populates a list with the station number or place name. Here we have integrated the station list with the Autocomplete feature of google maps.

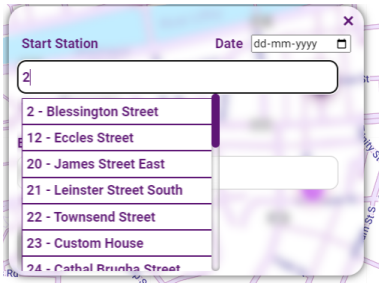
Figure 14. Location search bar.



2. **Scroll**

Scroll bar is designed based on the theme of the website, As the entire website is predominantly made with darkish purple, we made sure the scroll looks the same as well

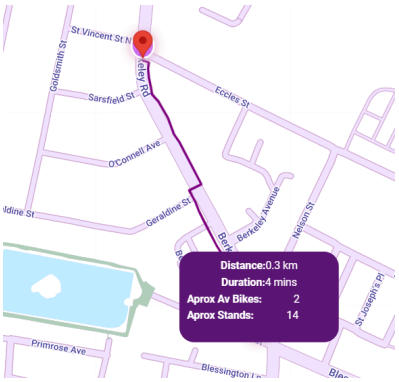
Figure 15. Implementation of scroll bar.



3. **Route /Distance Calculator**

When you select to plan a ride and add you from and to station details, with the date and times this feature calculates the distance and duration between the 2 stations and shows an Infowindow with the details. We have implemented this as part on the plan a ride feature by the google maps api.

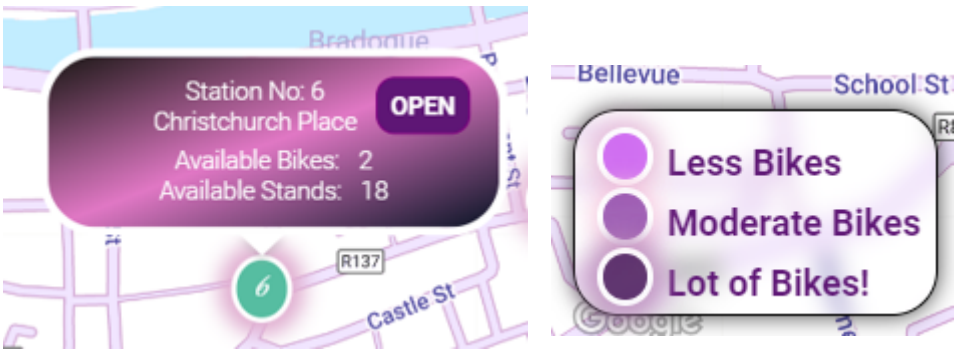
Figure 16. Route distance calculator.



4. **Advanced markers and infowindow**

Google maps recommended to use Advanced markers where we have added custom made markers whose blinking intensity and colour determine the number of bikes available. The morebikes, the more intense the colour. A custom made info window has also been added when you hover over the markers. Each markers displays its respective station number.

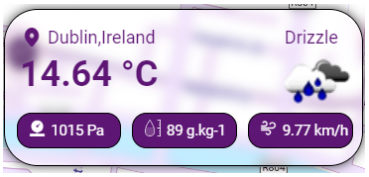
Figure 17. Advanced markers and info window.



5. **Weather box with pressure/humidity/wind speed info**

The weather box displays along with today's climate, the pressure, humidity and wind speed as well.

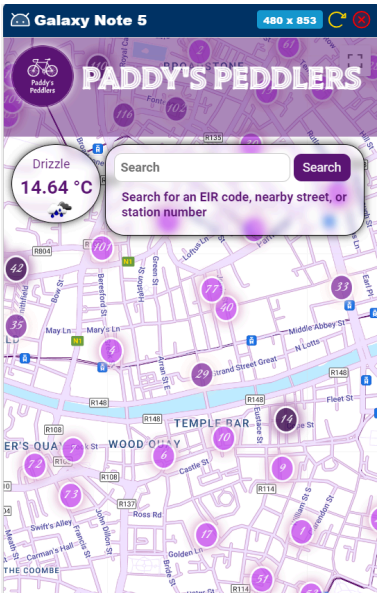
Figure 18. Weather information box.



6. **Mobile view implementations**

Along our design, we implemented the features by checking how it looked in the mobile view as well and making sure none of the content breaks for device of any size and shape

Figure 19. Mobile implementation view.



7. **Animations**

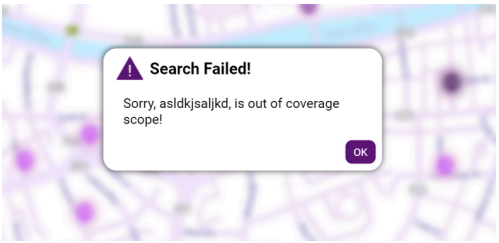
There are 7 Types of animations added to the UI

- A main animation for the logo
- An animation for popup fade in and fade out
- An animation for popup slide in and slide out from both op and left

8. **Custom Error Message**

We have created a custom error message for letting the user know in case if the server is down or if there exist any issues with the webpage.

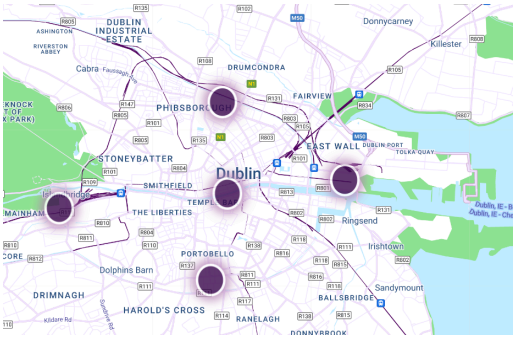
Figure 20. Custom error message.



9. **Marker Clustering**

   Advanced markers have no implementation of marker clusters unlike the Markers in google maps. We have made a custom made marker cluster where we are plotting the North South East and West on the map. On tapped on the marker clusterer, it will zoom into that location.

Figure 21. Marker clustering.

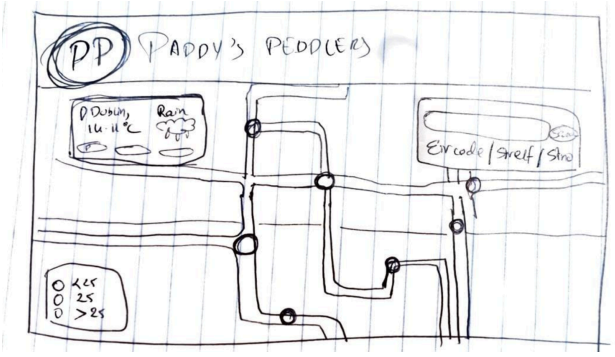


10. **Error Handling:**

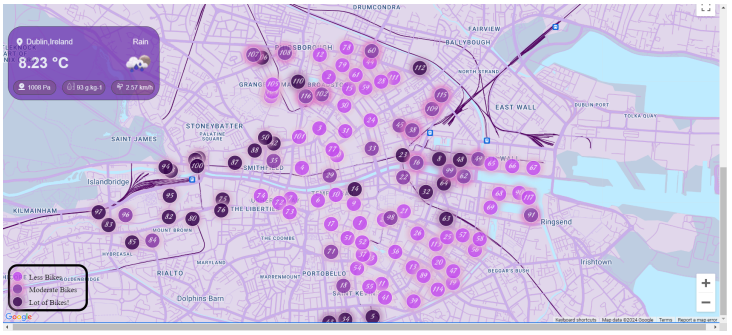
    Error is thrown to the user to alert in the form of an error message in all the stages of the website. Also the error is logged onto an error log file and will be displayed on the console also for debugging purposes.

Figure 21. Comparison of front end design across sprints.

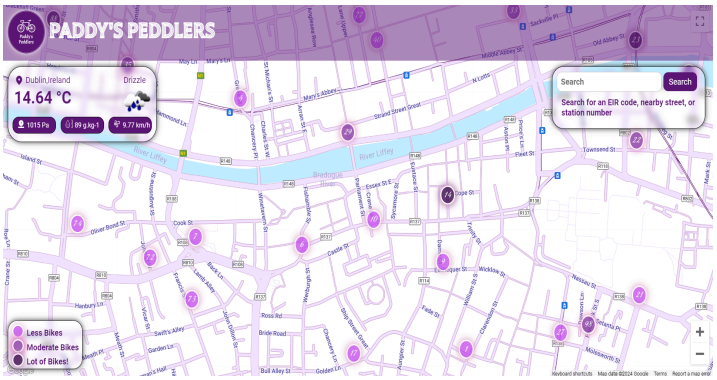
Before (On Paper) (Sprint 1-2)



Minimum Viable Product (Sprint 2-3)



Final Product (Sprint 3-4)



Back-end (Flask and API Scraping)

We have integrated flask and we are calling the flask to the Javascript Using fetch. Here we have made mainly 5 methods in the flask

GET:

1. **get_station(/stations)** : This is used to get station list from the json file. We are holding onto the station data on a json file and calling them
2. **get_weather(/weather)**: This is used to get the weather info. This call is responsible for plotting the weather details like pressure , humidity, windspeed, temperature, and so on
3. **get_image_url(/get_images)**: A simple method to get the complete list of images

POST:

1. **get_station(/getStationInfo)**: A post method where the station number selected by the user is passed onto the method and this will return the x and y coordinates in a json format to plot the graphs on the screen for that specific station.
2. **upload(/getBikesAndStandsInfo)**: A post method used when you select a start and destination station on a specific date and time and this will predict the number of bikes and stands available on that day.

Figure 22. Structure of app.py Flask file.

```
from flask import Flask, render_template, url_for , jsonify, request
import json
from ML.RFModel import generate_model_and_coords, generate_model_and_coords_stands, trip_predict, t


app = Flask(__name__)


@app.route("/")
def index():···

@app.route("/stations")
def get_stations():···

@app.route("/weather")
def get_weather():···

@app.route('/get_images')
def get_image_url():···

@app.route('/getStationInfo', methods=['POST'])
def get_station():···


@app.route('/getBikeAndStandInfo', methods=['POST'])
def upload():···


if __name__ == "__main__":
    app.run(debug=True);
```

Frontend And Flask References:

Map Design Inspiration:

- Dribbble: Map UI

UI Design Inspiration:

- Dark Mode Design
- LandingFolio
- App Shots Design
- Land Book
- Page Collective
- Refero Design
- Lapa Ninja

Python Flask Tutorial:

- YouTube - Flask Tutorial

Map API Integration Tutorial:

- YouTube - Map API Integration

Other Useful Tools:

- CSS Tricks - A Guide to Flexbox
- Pattern Monster
- CSS Border Radius Generator
- Google Fonts - Protest Riot

The website demo link:

https://www.youtube.com/watch?v=lYO35chaPpc&ab_channel=JOELUKRANSAJAN

AWS EC2 Instance Creation

In accordance with project instructions an AWS EC2 instance was created to host the web application. A miniconda3 virtual environment was also created to hold all dependencies required for executing the python scripts required for the application to function. The platform selected for the EC2 instance was Linux/Unix (Ubuntu). This was chosen as recommended in the module materials for COMP30830. For specific configuration details and properties for the EC2 instance refer to Table 1.

Table 1. BC_COMP30830 EC2 instance properties.

Property Name	Value
Instance Type	t3.micro
Platform	Ubuntu
Instance Name	BC_COMP30830
Platform Details	Linux/UNIX
Public IPv4 Address	16.171.42.129
Public IPv4 DNS	ec2-16-171-42-129.eu-north-1.compute.amazonaws.com

Database Hosting

In accordance with recommendations provided as part of the content of this module, an Amazon Web Services (AWS) Relational Database Service (RDS) instance was chosen to host the database component of the stack. AWS RDS provides a variety of database engines including MariaDB, PostgreSQL, MySQL and more. (https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html)

MySQL was selected as the engine for this project as it is a free, open-source solution. It is recognised as being one of the fastest available databases which is a desirable property when developing a web application. Furthermore, as there is a large community of users there is a wealth of online information and support available for MySQL which is also advantageous. (https://blueclawdb.com/mysql/advantages-disadvantages-mysql/)

Finally, as MySQL (and MySQL Client software) were introduced in the COMP20240 Relation Databases and Information Systems module, each group member had existing knowledge of their usage making it the optimum choice of relational database management system for the project. Refer to Table 2 for detailed configuration and property information regarding the RDS instance created.

Table 2. Configuration of the AWS RDS MySQL database instance.

RDS Instance Property	Value
DBS Instance ID	dublinbikesdb
Database Name	DBProject_InitName
Engine	MySQL Community
Engine Version	8.0.36
Class	db.t3.micro
Region	EU-north-1a
Endpoint	dublinbikesdb.c1aeqwowc1uf.eu-north-1.rds.amazonaws.com

Database Design and Structure

The database implemented in this project was designed to facilitate storage of Dublin Bikes information scraped from JCDeceaux and weather information from OpenWeather. To accommodate the information retrieved using the JCDeceaux API, the schema structure depicted in Tables 3, 4 and 5 was implemented. The database consists of three tables, Station, Availability and Weather.

The Station table serves as a table which contains descriptive information regarding each table in the Dublinbikes network. This information is required in order to display stations and associated information via the front end. The “number” attribute of this table serves as the primary key for the table and is not nullable. It is an integer value that uniquely identifies each bike station. All other attributes were created with their default properties. This table is considered static as the information contained within is not expected to change over the timescale of the project. However, the table is updated upon each call of DatabaseScraper.py when populating the database to account for situations where any of the content stored within changes (e.g. station is assigned a new number.)

The Availability table is intended to serve as a record of historical bike availability data for each station for use in creating a machine learning model for predicting bike availability for future planned trips. It may also be queried to return the latest bike availability information retrieved from JCDeceaux. The “number” and “last_update” attributes serve as a composite primary key and “number” serves as the foreign key of Availability on Station. This table also includes a non-nullable “scrape_time” attribute. This “scrape_time” value is assigned on each execution of the DatabaseScraper.py script. This script fetches data from both JCDeveaux and OpenWeather each time the script runs and the “scrape_time” value is added to both the Availability and Weather tables to provide a mechanism for identifying Availability and Weather table entries fetched during the same execution cycle of DatabaseScraper.py.
The Weather table serves as a record of weather data for use in development of a machine learning model that takes weather data into account when predicting bike availability for future planned trips. The “station_id” and “time” attributes serve as a composite primary key. The “scrape_time” attribute serves the same purpose as described for the Availability table. Note that “station_id” in this context refers to the weather station for which data was acquired and not bike stations.

For a complete description of the database schema refer to Tables 3, 4, and 5. Refer to Figure 23 for the entity relationship diagram associated with this database.

Table 3. Schema for Station table.

Attribute Name	Description	Data Type	Properties
number	Station number	int	Primary key, not nullable
address	Station address	varchar(128)	Default
banking	Availability of digital payment	int	Default
bikestands	Number of bike stands	int	Default
name	Station name	varchar(128)	Default
positionlat	Station latitude	double	Default
positionlong	Station longtitude	double	Default

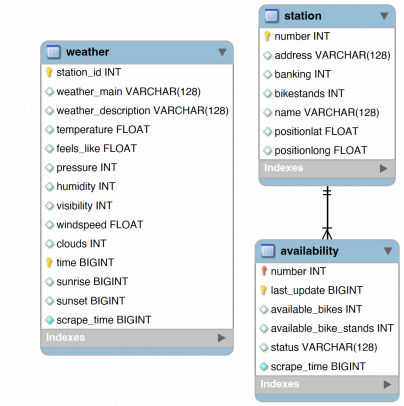
Table 4. Schema for Availability table.

Attribute Name	Description	Data Type	Properties
number	Station number	int	Primary key, foreign key, not nullable
last_update	Time of last update in UNIX epoch format	bigint	Primary key, not nullable
available_bikes	Number of available bikes	int	Default
available_bike_stands	Number of available bike stands	int	Default
status	Open or Closed status	varchar(128)	Default
scrape_time	Time of scrape in UNIX epoch format	bigint	Not nullable

Table 5. Schema for Weather table.

Attribute Name	Description	Data Type	Properties
station_id	Weather station ID	int	Primary key, not nullable
weather_main	Overall weather description	varchar(128)	Default
weather_description	Additional weather description	varchar(128)	Default
temperature	Temperature	float	Default
feels_like	“Feels like” temperature	float	Default
pressure	Air pressure	int	Default
humidity	Humidity	int	Default
visibility	Visibility value	int	Default
windspeed	Windspeed	float	Default
clouds	Presence/Absence of clouds	int	Default
time	Data measurement time in UNIX epoch format	bigint	Primary key, not nullable
sunrise	Sunrise in UNIX epoch format	bigint	Default
sunset	Sunset in UNIX epoch format	bigint	Default
scrape_time	OpenWeather scrape time in UNIX epoch value	bigint	Not nullable

Figure 23. Entity relationship diagram for the database.



Database Creation

Creation of the database was handled using SQLAlchemy. SQLAlchemy is a popular Python SQL toolkit which allows developers to interact with databases using Python objects rather than direct SQL queries. SQLAlchemy is an ideal choice for creating databases using Python as it's Pythonic interface provides high-level abstraction and readability of code while eliminating the need to directly write SQL code to create and populate the database. It also provides protection against SQL injection vulnerabilities and facilitates cross-platform compatibility.

SQLAlchemy was implemented in the DatabaseCreation.py file in order to create the database on the AWS RDS instance. Furthermore, SQLAlchemy was also used in the DatabaseAccess.py file which was used to retrieve data from JCDeceaux and OpenWeather and commit this data to the database.

However, SQLAlchemy is an Object-Relational Mapping (ORM) library and as such introduces performance overhead in comparison with raw SQL queries. Furthermore, there is a considerable learning curve when using an ORM library for SQL queries which was not considered to be an efficient use of time over the project timescale. Considering the time constraints of the project, it was decided to restrict SQLAlchemy usage to database creation and population only. Finally, in the development of SQL queries to return data from the database, flexibility is advantageous when performing troubleshooting or query testing tasks. This flexibility is better provided by Pandas where a raw SQL query is stored within a Python file as a string which is easily readable, understandable and modifiable. It was therefore decided to use Pandas as the method to return data from the database. For a complete list of functions used in database creation refer to Table 6.

Database Population

Database population methods were implemented using SQLAlchemy. The Station table is populated using the "update_stations_to_db" method, the Availability table is populated using the "update_availability_to_db" method and the Weather table is populated using the "update_weather to database" method. These functions are executed on each call of the DatabaseScraper.py script and are defined in DatabaseAccess.py.

The DatabaseScraper.py script was uploaded to the previously discussed AWS EC2 (i.e. BC_COMP30830) instance upon completion and confirmation of satisfactory performance of the script. Cron was selected as the mechanism to facilitate periodic execution of the script at intervals of five minutes. Execution was achieved using a shell script which activates a virtual miniconda3 environment, runs the DatabaseScraper.py file and finally deactivates the miniconda3 environment. This script was titled "run_scraper.sh" and the contents of the script are as follows:

```
---------- Start of run_scraper.sh ----------
#!/bin/bash

# Activate the virtual environment
source /home/ubuntu/miniconda3/bin/activate comp30830_flask

# Run the specific Python file
python /home/ubuntu/git/SWE-Project/DB/DatabaseScraper.py

# Deactivate the virtual environment (optional)
conda deactivate
---------- End of run_scraper.sh ----------
```

Confirmation of the ongoing population of the database was performed using a connection to the database created in MySQL Workbench. Database Client, a lightweight VSCode extension, was also used for periodic monitoring of database population to ensure data acquisition continued.

Database Queries

Database queries were implemented in the DatabaseAccess.py file using Pandas to facilitate database access. The full code of these functions is not provided in this report. However, the SQL queries contained within will be explicitly discussed. For a complete list of functions implemented in this file refer to Table 6.

Discussion of the SQL queries will begin with the query required to return static station data and the latest bike availability data for use in the front end. This query is contained in the "get_station_data_from_DB" function and the query implemented is the most complex of the SQL queries used. The SQL statement is provided below and SQ1 and SQ2 are shorthand for Subquery 1 and Subquery 2 and are indicated in green text and blue text respectively.

Subquery 1 selects "number" (i.e. station number) and "last_update" (i.e. UNIX epoch time value for the latest update as "latest_update") and groups them by number to provide a "last_update" value for each bike station in the Dublinbikes network.

Subquery 2 then joins the result of Subquery 1 back on the Availability table based on the equality of the "number" and "last_update" attributes and returns all attributes of the Availability table. The result of this query is the latest entry for each station in the availability table.

Finally, the main portion of the query joins the result of Subquery 2 with the Station table based on equality of the "number" attribute. The final output of this query is the latest availability data for each station from the Availability table combined with the static data describing each station from the Station table. The result of this query is saved in a JSON format file, "stations.json", for use in the front end.

```
SELECT
      s.number,
      s.address,
      s.banking,
      s.bikestands,
      s.name,
      s.positionlat,
      s.positionlong,
      SQ2.last_update,
      SQ2.available_bikes,
      SQ2.available_bike_stands,
      SQ2.status

   FROM station s
   JOIN (
      SELECT a.number, a.last_update, a.available_bikes, a.available_bike_stands, a.status
      FROM availability a
      JOIN (
         SELECT number, MAX(last_update) as latest_update
         FROM availability
         GROUP BY number) SQ1
         ON a.number = SQ1.number AND a.last_update = SQ1.latest_update) SQ2
      ON s.number = SQ2.number;
```

The next query which will be discussed is contained in the “latest_weather” function. The purpose of this query is to return the latest weather data from the Weather table of the database. The query is as follows:

```
SELECT * FROM weather
WHERE scrape_time = (SELECT MAX(scrape_time) FROM weather);
```

This query selects the Weather table entry with the maximum UNIX epoch time value for the “scrape_time” attributes. This portion of the query is highlighted in blue and serves as a subquery which allows the main query to select the row (including all associated attributes) which correspond to this scrape time. The results of this query are stored in a JSON format file, “weather.json”, for use in the front end.

The final SQL query used in this project is contained in the “availability_weather_join” function. The purpose of this query is to associate Availability table entries and Weather table entries fetched from JCDevceaux and OpenWeather during the same execution cycle of the DatabaseScraper.py script. The machine learning component of this project requires association of bike availability data with weather data in order to predict the number of bikes available for future planned trips under given forecast weather conditions. As previously described in the Database Design and Structure section, this is achieved by recording a scrape time value on each execution of DatabaseScraper.py and storing this value in both the Availability and Weather tables. The query joins the Availability and Weather tables based on the equality of the “scrape_time” attribute and the SQL statement is as follows:

```
SELECT * FROM availability A
   JOIN weather W
   ON A.scrape_time = W.scrape_time;
```

As the results of this query are used as an input to the machine learning component of this project, the output of the “availability_weather_join” function is a Pandas dataframe.

Note that, while the SQL queries discussed above and their containing functions are located in the DatabaseAccess.py file, this file is not directly executable and serves as a repository of functions that support database access.. Refer to Table 6 for the locations where these functions are called.

Table 6. Database access functions and their associated definition and call locations.

Function Name	Definition Location	Call Location
update_stations_to_DB	DatabaseAccess.py	DatabaseScraper.py
update_availability_to_DB	DatabaseAccess.py	DatabaseScraper.py
update_weather_to_DB	DatabaseAccess.py	DatabaseScraper.py
get_station_data_from_DB	DatabaseAccess.py	app.py
latest_weather	DatabaseAccess.py	app.py
availability_weather_join	DatabaseAccess.py	app.py

Reference Material used for Database Implementation

Proclus Academy, Pandas: How to Read and Write Data to a SQL Database - https://proclusacademy.com/blog/practical/pandas-read-write-sql-database/

SaturnCloud, Converting Pandas DataFrame to JSON Object Column: A Guide - https://saturncloud.io/blog/converting-pandas-dataframe-to-json-object-column-a-comprehensive-guide/

StackOverflow Convert Pandas DataFrame to JSON format - https://stackoverflow.com/questions/39257147/convert-pandas-dataframe-to-json-format

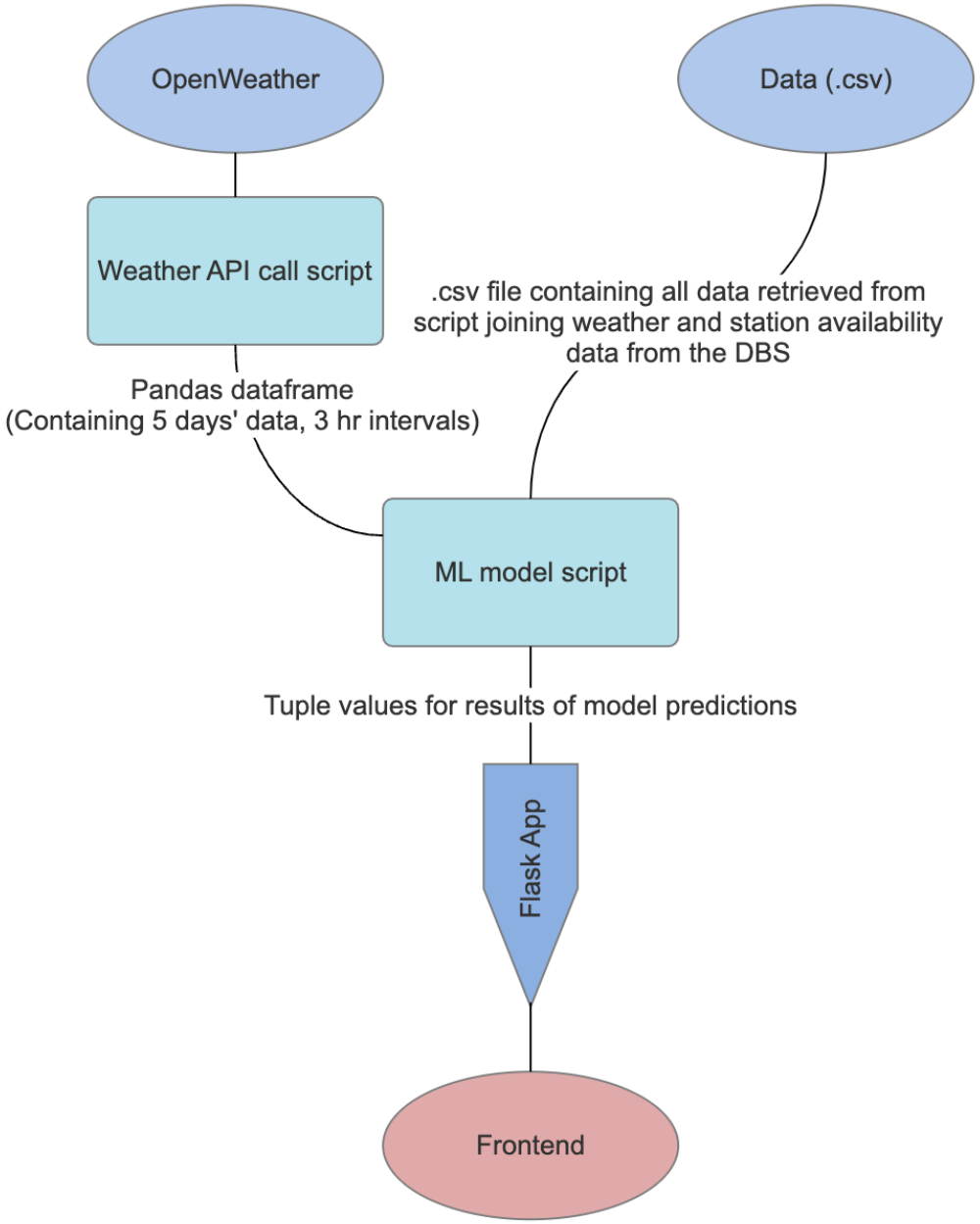
Pandas Documentation, - pandas.DataFrame.to_json
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_json.html

SQL Alchemy vs. Pandas for Database Interaction:
https://python.plainenglish.io/sqlalchemy-vs-raw-sql-queries-in-python-a-comparative-example-b7a838ebef82

Machine Learning

The machine learning model portion of the project is contained within two scripts, one which contains a function to collect forecast information using the OpenWeather API, and another to take an input from the scraper and database scripts and generate, train and utilise models to predict bike availability and stand availability using input entered during function calls from the frontend. The models are created and used with the scikit-learn library.

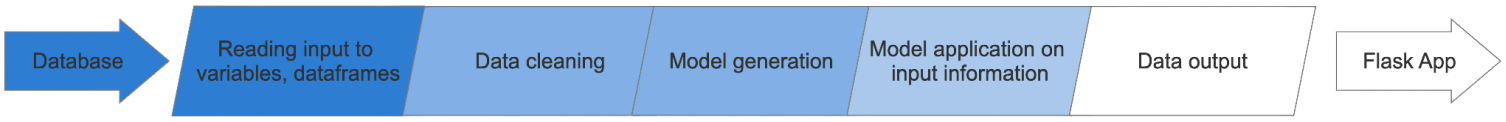
Figure 24. Overview of the machine learning model.



The script to generate the regressor models followed the below series of steps, with an input of the weather forecast information, and a csv file which was returned by the function to join the weather and bike/stand availability tables from the frontend. The .csv file is read directly, while the weather

data is requested using a function imported from another script, returning a pandas dataframe of the next 5 days of forecast data, in 3-hour intervals.

Figure 25. Steps involved in regressor model script.



The data cleaning process involves taking the .csv input and selecting columns to be used in the data analysis process, converting the epoch time into constituent month/day/hour values, and selecting columns from the dataframe to be used for training the machine learning models.

With each row in the input data there is associated a ***'scrape_time'*** value - this value was taken to be the time that could be used to associate the weather and bike/stand availability tables in the database, and is shared between the associated rows. In the data analysis, this value is taken as the 'time' input of the models, for training and prediction purposes. The time value was determined to be best suited for analysis when separated into the ***'day_of_week'*** and ***'hour'*** values, with the day of the week taking the value of an integer between 0 and 6, and the hour value having a decimal (float) value between 0 and 24, which made input into the models easier from the frontend. Month was indicated by the number of the month.

After data cleaning the data frame it contains the following columns: ***'number'***, ***'month'***, ***'hour'***, ***'day_of_week'***, ***'temperature'***, ***'humidity'***, ***'visibility'***, ***'windspeed'***, ***'available_bikes'***, ***'available_bike_stands'***.

A 'for' statement iterates through the station count (obtained through the ***.unique()*** pandas method on the dataframe), applying a filtering condition that selects rows from the selected_columns dataframe that have their 'number' column entry matching that station number. These are separated into entries of a dictionary that contains keys representing the station numbers, and values of dataframes containing the associated rows of the input.

The functions ***create_models()*** and ***create_models_stands()*** are then used to generate and train the models, using each separated dataframe for each station, in the previously-generated dictionary.
A 'for' loop iterates across the length of the dictionary and takes the values associated with each station, separates it into two dataframes (input features and target features), and trains a random forest regressor model for each station data input.
Input features: ***'month'***, ***'hour'***, ***'day_of_week'***, ***'windspeed'***, ***'temperature'***, ***'humidity'***, ***'visibility'***
Target feature:
create models() - 'available_bikes'
create_models_stands() - 'bike_bike_stands'

The models from this loop are then added to a new dictionary which organises the models by a key of the station number.

Models are serialised and saved (py pickle).
Model generation occurs once, and then other functions are used to predict using them.

For prediction using these models there were two objectives for frontend features - a plot showing the predicted bike and stand availability over time (5 days), and specific information about availability for stations chosen between specific times, in selecting a journey.

For the plot, a set of values (a tuple of two lists) is returned from a function in the ML script. There is one function,
data_clean_for_predict(stationNumber, weatherForecast),
which creates a new dataframe that is input to the models for prediction. The function first applies a new column, ***'number'***, to the weather forecast dataframe returned from the ***weather_api_request_forecast()*** function, adding the station number to each row (as is required by the regressor model).
It then takes every feature required for the regressor model from the forecast and adds them to the dataframe.
Epoch time is converted to ***'month'***, ***'hour'*** and ***'day_of_week'***.
There are two functions that can be used to predict with a single model. The function that is actually used to return data to the frontend is the ***predict_with_single_model_high_hours_res(model, stationNumber)*** function, which makes additional modifications to the dataframe before it is input to the models.
The forecast data is in intervals of 3 hours, but the model must be able to predict data between two times reasonably expected to be within journey times of the bicycle users.
A 'for' loop iterates across the rows in the dataframe and adds 11 rows, below each row. This makes it so that each forecast update has an associated 12 rows in the dataframe, each row corresponding to a 15-minute time interval.
To make the time data in the rows reflect these intervals, the ***(rows' indexes)%12*** is taken for the length of the dataframe. This finds the distance that the row is from the last weather update (distance from the last 3-hour mark), and adds that result of this modulus operation, multiplied by 0.25 (15-minutes as a decimal fraction of an hour) to the ***'hour'*** value of each row.
This function allows the return of a list of values so that frontend implementations can make use of a minimum of 15-minute resolution.

The ***generate_model_and_coords()*** generates the predictions with the ***predict_with_single_model_high_hours_res(model, stationNumber)*** function, and then adds to a tuple the result from that predict (as the y-values) and a list of corresponding hour values (as the x-values).
There are two other functions for the journey planner - one to predict the number of bikes available at the start bike station and a second function to predict the number of stands available at the end station.
The ***trip_predict(JSON)*** function takes as an input a JSON file. The input JSON is flattened into a dictionary, containing specific keys related to start and end times, dates, station numbers, and unix epoch timestamps.

The function ***weather_api_request_forecast()*** is called, and the returned dataframe is assigned to a variable. The function then calculates, for each epoch timestamp in the JSON input, the closest timestamp values in the rows of the forecast data, by index, and retrieves the rows corresponding to these closest values.

Two empty dataframes are created, and the input features for the regressor model are added as columns and values to the dataframes from the dictionary and from the relevant weather information from the rows previously taken from the weather forecast dataframe. All features are to be input into the same models generated for the plotting of the graphs, and so all data input into the models from these dataframes are identical, in terms of the datatypes, time formats, and types of weather indicators. The function loads two of the regressor models corresponding to the start and end station numbers (using pickle.load()) - the statement to load finds the names from the input station number in the JSON input file. They are stored in variables, and applied to the dataframes constructed earlier in the function. The results of the model predictions are returned as a tuple (start,end).

Machine Learning Model - Testing and Performance

The following are plots showing typical prediction performance of the models. They are tests run with randomised training data and testing data, taken from collected data, in a ratio of 80:20.

Figure 26. Machine learning model actual vs. predicted values.

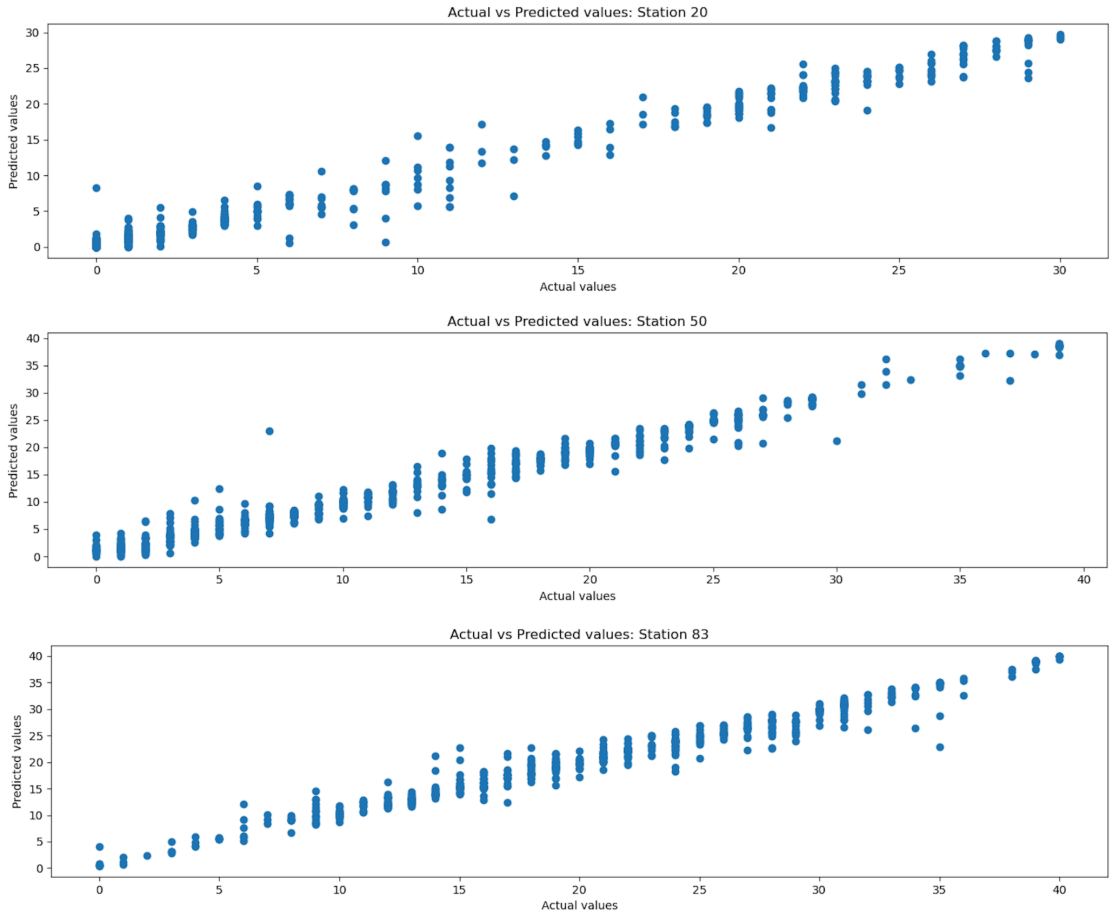
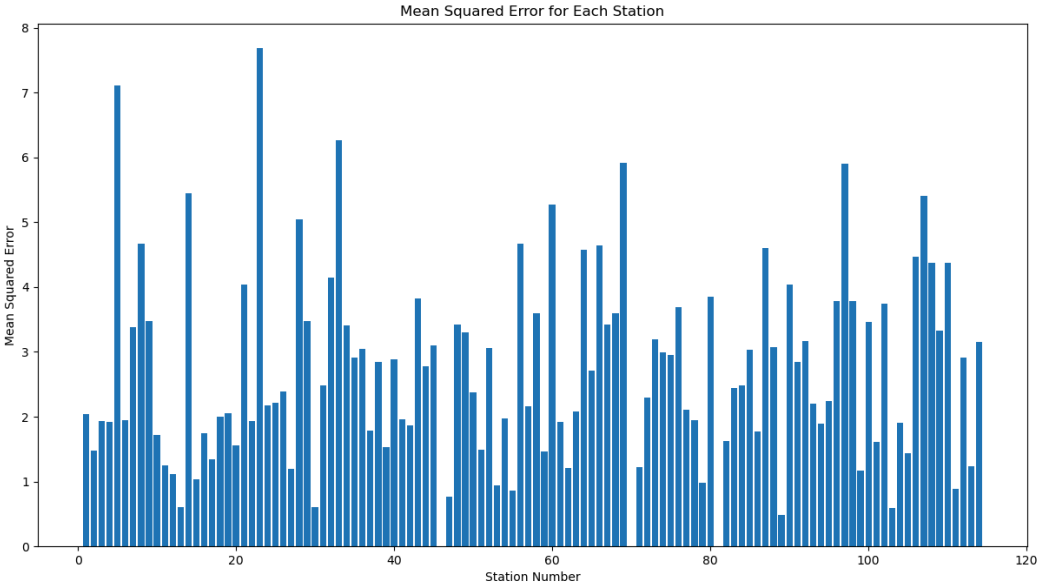


Figure 27. Mean Squared Error vs. Station



Above is a chart showing the mean squared error for one run of the models' prediction across all stations

There was some limit to the accuracy of the models, as the data collected to train them was limited to a certain extent, and data was collected for the final models over a period of 6-7 weeks. The models were therefore trained with very little variation on the 'month' feature.

Resources for ML development:

Several introductory youtube videos (Scikit-learn tutorials, random forest regression explainer videos)

DA Module experience with pandas library, dataframes, etc.

Stackoverflow for JSON module information and Time module.

Documentation of libraries.

Application Deployment

Gunicorn and nginx were downloaded to the hosting EC2 instance (i.e. “COMP30830_Backup_Instance”) and the application was deployed as per instructions provided in Lecture 11 “Deployment on EC2.”
During deployment a number of issues were encountered which were troubleshooted collaboratively by the team resulting in a successful deployment of the application.
Prior to deployment it was observed that the EC2 previously created to host the application (i.e BC_COMP30830) was inaccessible via SSH with the Public IPv4 address. However, it was possible to SSH into the instance using the Public IPv4 DNS. Lack of connectivity via the Public IPv4 address was anticipated to cause issues in accessing the application upon deployment. The issue was troubleshooted without resolution. A new EC2 instance named “COMP30830_Backup_Instance” was created which permitted successful access via SSH. See Table 7 for details of the new EC2 instance. It was decided to deploy the application to this new instance while leaving the DatabaseScraper.py script running on the original BC_COMP30830 instance to ensure uninterrupted population of the database. This was required as the scraper and database functionality involves fetching JCDeceaux and OpenWeather data from their respective sources and populating the database. The most up to date information is then retrieved from the database on demand when the web application is loaded. Furthermore, DatabaseScraper.py was not functional on the new EC2 instance due to code and directory structure changes that occurred during project progression. Ideally, functionality of DatabaseScraper.py should be moved to the new EC2 instance, however, given the time constraints of the project this task was deprioritised.

Another issue which contributed to deployment difficulties was related to cloning the SWE-Project repository from GitHub. An error occurred whereby all files required for the application functionality were not present on the EC2 instance after cloning. This may have been caused by the extensive branch structure of the repository that occurred during application development. In an attempt to troubleshoot this issue, the most up-to-date and functional branch of the SWE-Project repository was duplicated and committed to GitHub as a new repository named SWE-Project_ZIP. This repository was cloned to the new EC2 instance. Additional errors, specifically HTTP 404 errors, occurred due to issues with path names in the index.html file, however, these were resolved and the application was deployed to Public IPv4 16.170.230.66 successfully.

Table 7. BC_COMP30830 EC2 instance properties.

Property Name	Value
Instance Type	t3.micro
Platform	Ubuntu
Instance Name	COMP30830_Backup_Instance
Platform Details	Linux/UNIX
Public IPv4 Address	16.170.230.66
Public IPv4 DNS	ec2-16-170-230-66.eu-north-1.compute.amazonaws.com

Section 4: Sprint Information (Sprint Planning Records)

This section summarises the output of Sprint Planning meetings held during the course of the project and results of Sprint Retrospective meetings held at the end of each Sprint. It consists of notes taken during Sprint planning meetings and summarisation of discussions at Sprint Retrospectives.

Process

The workflow process for implementation of the project generally followed the guidelines set out in the COMP30830 Software Engineering lectures. The project was divided into a series of four sprints. Each sprint consisted of the following.

- Sprint planning meetings with a Sprint Planning document as the output deliverable. Sprint planning documents were used as a means to distribute work tasks among team members and ensure that resources were distributed evenly across the workload for project realisation.
- Daily Scrum meetings to document work planned for that day, work completed the previous day and any issues that require troubleshooting or help from the team.
- Sprint Retrospectives documenting “What helped us move forward?”, “What could have gone better?”, “How could we do things differently?”

Additional project management tools used include a Sprint Backlog record and Sprint Burndown charts. Refer to Sections 5 and 6 respectively for examples of the contents of these records.

Sprint 1

Sprint Planning

Date : 09 February 2024

- Starting with Initial understanding of the project 2h (p.p.)
    - Understanding the outcome
    - Deciding the platform or way of approach
    - Working on Resource gathering
    - Role assigning
- Wireframe and Designs
- Understanding AWS RDS
- Database creation in AWS RDS
- Connecting AWS RDS with Python
- API working understanding
- Creating Authentication in JCdecaux and capturing the key
- API tryout with calling JCdecaux data from the api

1. Michael: Figure out how the APIs work for OpenWeather and JCDeceaux. Create the GitHub repository.
2. Joel: Creating a basic Flask directory structure and initial website designs.
3. Brian: Research creation of AWS RDS instance, database initialization and EC2 instance creation.

Sprint 1 Retrospective

A formal retrospective was not performed for Sprint 1 as it mostly consisted of research into how to perform tasks required to implement the application. Team communication at this stage was mostly informal. Future sprints were planned to be more organised and communication on project progression and task planning more structured.

Sprint 2

Sprint Planning

Date: 19th Feb 2024

Backend Tasks:

Week 1:

- Python script integration to fetch and store API data, get a working, timed fetch to load data, including a JSON file that is wiped after each update. Must be able to consistently query data. ~ *2.5hrs*
- Update existing database code to include error handling (try except). ~ *1 hr*
- Review Slides on SQL Queries and start coding. ~ *2 hours*
- Create a JSON backup for all used data. ~ *1.5hrs*

Week 1 Backend Total Time Estimate: ~ 5 hrs

Week 2:

- Include the OpenWeather API data ~ *1hrs*
- Flask App - get representative data to be accessible to frontend. ~ *3hrs*
- Local iteration ~ *2hrs*
- EC2 iteration ~ *1hrs*

Week 2 Backend Total Time Estimate: *~ 4hrs*

Frontend Tasks:

Week 1:

- Initial project setup and adding pages ~ *3hrs*
- Working on basic css styling sheets ~ *1hr*
- *Front page UI designing with stying ~ 2hr*

Week 2:

- *Understanding the working of Map API ~ 2hr*
- Integrating MAP api and creating a sample tryout to make sure you get your doubts clarified ~ 5hr

Documentation:

- Elaborate on existing README file (new flowchart, description of Google API, Weather). ~ *45 mins*

Scrum meeting schedule:

Date	19/2	20/2	21/2	22/2	23/2	24/2	25/2	26/2	27/2	28/2	29/2	1/3	2/3	3/3
Time	15:00	9:00	14:35	9:00	15:00	19:30	10:00	15:00	9:00	14:35	9:00	15:00	19:30	10:30

Sprint 2 Retrospective

What helped us move forward?

- Gained understanding of JCDeceaux scraping.
- Gained understanding of AWS RDS usage.
- Wireframe models worked on.
- Project management tools (i.e. Miro, Trello, Slack) organised.
- Functions created to parse JSON files.
- EC2 and RDS instances created.
- Basic JCDeceaux scraper created.
- Coding to create database and tables.
- GitHub repository set up.

What could have been done better?

- Team Scrum meetings required more organisation to optimise attendance.
- Breaking down tasks into subtasks to prevent vague tasks.
- Document required tasks in a backlog sheet.
- Database creation code should include error handling for troubleshooting.
- Records of discussion with TA not properly documented.
- Push code to GitHub prior to Sprint Planning meetings.

How could we do things differently?

- Pair coding suggested where difficulties are encountered.
- Review accomplished tasks to increase team exposure to all elements of project.
- Clearer assignment of time-frames to tasks.
- Have set default times for scrum meetings.

Sprint 3

Date: 5 May 2024

Backend Tasks:

Week 1:

- Flask ~ *0.75 h*
- Connect RDS to EC2 ~ *0.5 h*
- Add scripts to EC2 ~ *0.5 h*
- Test Cron on EC2 ~ *0.5 h*

Week 2:

- Research ML - Create new tasks relevant to this task as info is found ~ *3 h*
- Create basic ML model ~ *2 h*
- Get latest weather info from the database ~ *1hr*
- Add mechanism for associating weather and bike availability in database.

Frontend Tasks:

Week 1:

- Weather data call from frontend
- Add tap on station functionality to see details
- Search bar for stations
- Find shortest route/best route possible

Week 2:

- Expand on tasks defined for Week 1

Documentation:

- Continue documentation of project as appropriate.

Scrum meeting schedule:

Date	19/2	20/2	21/2	22/2	23/2	24/2	25/2	26/2	27/2	28/2	29/2	1/3	2/3	3/3
Time	15:00	9:00	14:35	9:00	15:00	19:30	10:00	15:00	9:00	14:35	9:00	15:00	19:30	10:30

Sprint 3 Retrospective

What helped us move forward?

- Further work on understanding and implementing database functionality.
- Use of practicals to discuss project progression.
- Use of branches on GitHub to separate work
- Use of more specific tasks and subtasks provided clarity.
- Sprint backlog better defined.
- Defined Scrum schedule provided more structure.
- Improved Scrum documentation helped track progress.
- Documentation and project management skills by responsible team member facilitated more organised work.
- Good progress made on front-end.

What could have been done better?

- Use of VSCode Database Client was useful for monitoring the database but caused issues in query development. Switch to MySQL Workbench would have been useful earlier.
- Prioritisation of tasks required a bit more consideration.
- Inaccurate prediction of task time in some cases.
- Approaches to issues and troubleshooting performed independently in some cases and discussed in hindsight.

How could we do things differently?

- More detailed consideration of task prioritisation would help.
- Work on retrieving only the required data (e.g. some information from OpenWeather is not particularly useful)
- Improve cross-review of each other's work to share learnings.

Sprint 4:

Date: 3 Apr 2024

Backend Tasks:

Week 1:

- Add prediction request block to ML model script. Michael
- Add weather forecast API scrape - 1.5hr
- Code block to generate new pd df to be fed to ML model 1.5hr
- Code cleanup - Brian
- Write function to return latest weather data from database. - Brian

Week 2:

- Upload finished project to EC2 instance - troubleshooting will probably be necessary
- Review assignment brief to ensure requirements are satisfied.

Frontend Tasks:

Week 1:

Continue to refine frontend - missing features

Week 2:

- Deploy application to EC2 instance. Brian

Documentation:

Week 1: No tasks for documentation this week.

Week 2: Clean up all in-semester documentation to prepare for adding to the report. Work on the report and personal learning logs.

Scrum meeting schedule:

Date	19/2	20/2	21/2	22/2	23/2	24/2	25/2	26/2	27/2	28/2	29/2	1/3	2/3	3/3
Time	15:00	9:00	14:35	9:00	15:00	19:30	10:00	15:00	9:00	14:35	9:00	15:00	19:30	10:30

Sprint 4 Retrospective

What helped us move forward?

- Large portion of work on front end optimisation.
- Large portion of work on machine learning performed.

What could have been done better?

- More attention required to daily Scrum recording.
- Some tasks not suitable for reporting at meetings consistently (e.g. ongoing research into how to accomplish tasks).
- Database access function to return latest weather information from RDS not yet implemented.

How could we do things differently?

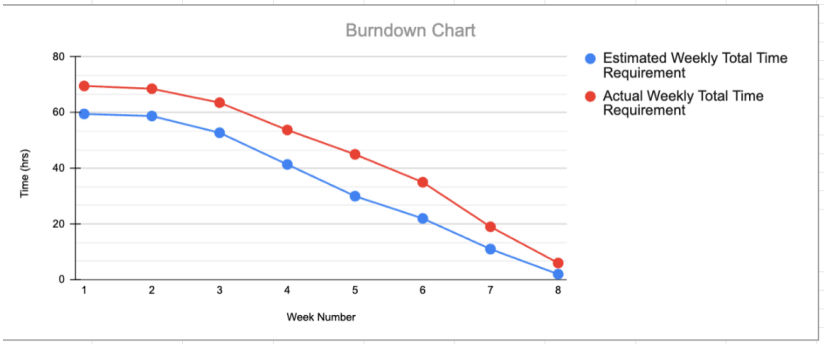
- At this stage the group is working well together and have learned from previous sprints. Problems are mostly task-related and not due to project management issues.

Section 5: Sprint Backlogs

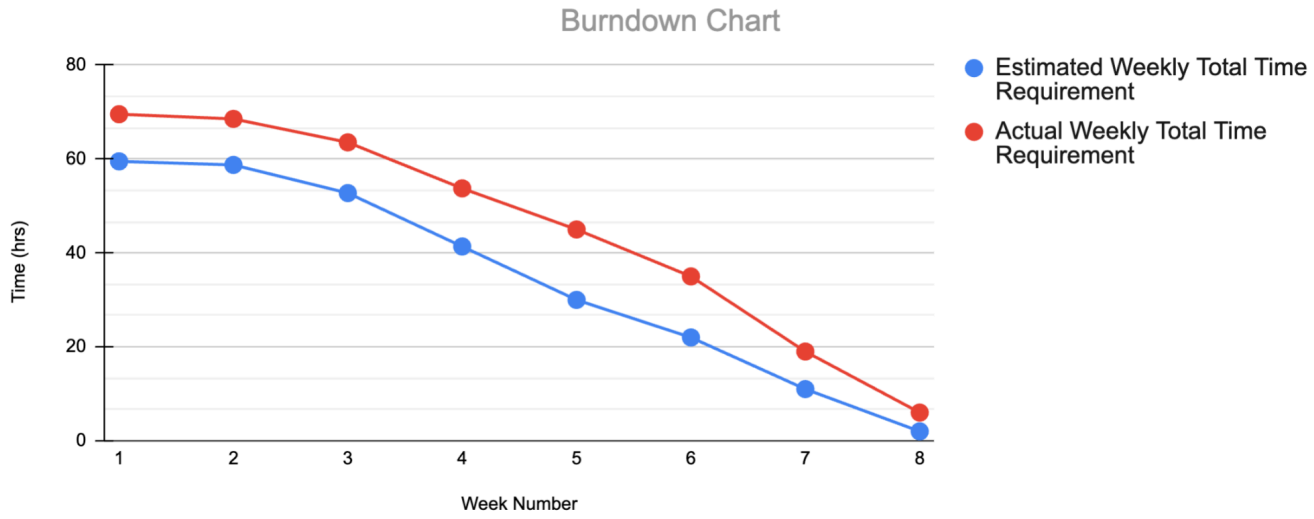
Sprint Number	Task allocation by sprint (sprint backlog)
1	Research API usage and access necessary keys
	Create folders/files for containing the relevant documentation for the project
	Set up github with at least 3 branches - main, dev, frontend
	Set up excel SS for the burndown data and chart - time-based bd chart
	Create basic strucutre of website (wireframe)
	Create a DBS, Provide access to all team members
	Python script for scraping data (basic)
	Database creation, manipulation scripts
	Update DB code to provide error handling for troubleshooting
	Review SQL and provided slides for DB understanding
	Research SQLAlchemy for database interaction
2	Database query script
	Basic flask application to serve data to frontend
	Research crontab
	Map integration and initial UI design
	Designing frontened in html/css
	Create README, basic flowchart
	Python script for scraping data (error handling and integration with database manipulation script (functions)
	Write SQL queries
	Restructure database access functions
	Create database table for weather information
	Integrate new weather DB function into scraper
	Set up mySQL connection to DB, Database Client in SQL causing SQL query development issues
	Test recovery of information from DB
	Update DB code to check for existing entries to prevent SQL duplication errors.
3	Create JSON backup
	Research ML - create new tasks relevant to this task as info is found
	Look into ways to associate weather table entries and bike availability table entries based on UNIX epoch time (multiple options)
	Update DB scraper code to record a scrape time variable and add this to database.
	Wipe DB and rebuild with new scrape time attribute in Weather and Availability tables.
	Connect EC2 and RDS instance
	Research Cron for scrape scheduling
	Clone GH repository to EC2 instance, test functionality of scraper and troubleshoot
	Get Cron up and running with DB scraper
	Write SQL query to join Availability and Weather tables and return in format for use in ML section.
	Generate Basic ML model
	ML Functions for forecast data integration
	ML graph production
	Route selection - Adding the final UI changes to create a plan chart
	Booking bike route - Adding UI changes to plan a ride
	Flask - Creating a connection from Flask to JS
	ML/Flask/Front End - Return planned trip details to Flask, integrate Flask with ML
	Code cleanup
	Error Handling
	Map integration and initial UI design
4	Create ER diagram for DBS tables
	Back end code cleanup
	Create function for returning latest weather from DB, test and push to GH
	Work on JS function to take values from trip planner popup
	Work on resetting values in trip planner popup after it is closed. Create reset function.
	Update app.py to accept POST from front end for trip planning.
	Add technical details to report on EC2, RDS, DB (setup, design, creation, functions, queries etc.)
	Create new EC2 instance due to potential problems with original.
	Recreate conda environments
	Install and configure Gunicorn and nginx
	Research connecting flask and front end trip planning function.
	Test return of predicted trip values from Flask app to front end with dummy variable.
	Work on integration of app.py with ML model functions to return planned trip data to front end
	Finish deploying app to EC2
	Finding HTTP 404 errors, resources not loading, troubleshoot.
	GH files not cloning correctly to EC2 instance, troubleshoot.
	ML - create models as external files
	Refine frontend

Section 6: Burndown Chart and Scrum Meeting Notes Example

Times in hours	SPRINT 1 EST Sprint 1 Task Time Requirements Est.		SPRINT 1 ACT Sprint 1 Task Time Actual Requirements		SPRINT 2 EST Sprint 2 Task Time Requirements Est.		SPRINT 2 ACT Sprint 2 Task Time Actual Requirements		SPRINT 3 EST Sprint 3 Task Time Requirements Est.		SPRINT 3 ACT Sprint 3 Task Time Actual Requirements		SPRINT 4 EST Sprint 4 Task Time Requirements Est.		SPRINT 4 ACT Sprint 4 Task Time Actual Requirements
	Week 1		Week 1		Week 1		Week 1		Week 1		Week 1		Week 1		Week 1
Backend	0	Backend	0	Backend	5	Backend	2.5	Backend	4	Backend	3	Backend	3	Backend	5
Frontend	0	Frontend	0	Frontend	6	Frontend	7	Frontend	4	Frontend	7	Frontend	5	Frontend	6
Documentation	0.75	Documentation	1	Documentation	0.375	Documentation	0.3	Documentation	0	Documentation	0	Documentation	1	Documentation	2
Total	0.75	**Total**	1	**Total**	11.375	**Total**	9.8	**Total**	8	**Total**	10	**Total**	9	**Total**	13
	Week 2		Week 2		Week 2		Week 2		Week 2		Week 2		Week 2		Week 2
Backend	6	Backend	5	Backend	4	Backend	0.25	Backend	6	Backend	7	Backend	1	Backend	1
Frontend		Frontend		Frontend	7	Frontend	8	Frontend	5	Frontend	9	Frontend	1	Frontend	2
Documentation		Documentation		Documentation	0.375	Documentation	0.5	Documentation		Documentation		Documentation	0	Documentation	3
Total	6	**Total**	5	**Total**	11.375	**Total**	8.75	**Total**	11	**Total**	16	**Total**	2	**Total**	6



Week Number	Estimates	Actual
1	0.75	1
2	6	5
3	11.375	9.8
4	11.375	8.75
5	8	10
6	11	16
7	9	13
8	2	6
1	59.5	69.55
2	58.75	68.55
3	52.75	63.55
4	41.375	53.75
5	30	45
6	22	35
7	11	19



Scrum meeting example notes

Sprint 3 (Week 2)	Team Member	Work Done Yesterday	Plans for Today	Help required? Issues?
Day : 25th Mar	**Michael**	No significant progress.	ML Research	N/A
	Brian	DB coding for creating weather table, integrated weather API into Integrated scraper, created function for populating weather data into database. SQL query and associated function completed for pulling most recent availability data from database. Created Data folder to allow Database Scraper to run.	Nothing planned yet.	Need a bit of help with MySQL queries. Issue might be due to using Database Client in VSCode. Will set up connection in MySQL workbench and try practice queries there.
	Joel	Marker Info box added on the screen		
Day : 26th Mar	**Michael**	ML Research	Create functions for more detailed data cleaning for implementing prediction using forecast data	Functions return type clarifications
	Brian	No progress yesterday.	Update DB coding to include a "scrape time" column in Availability and Weather tables to allow for ease of joining those tables for ML. Connect EC2 instance and RDS instance. Clone Dev to EC2 and set scraper running using Crontab.	Had issues with Crontab but resolved eventually.
	Joel	Styles added to Marker Info windows with station details		
Day : 27th Mar	**Michael**	ML functions for prediction	Graph plotting	None.
	Brian	Updated DB coding to include a "scrape time" column in Availability and Weather tables to allow for ease of joining those tables for ML. Connected EC2 instance and RDS instance. Cloned Dev to EC2 and set scraper running using Crontab.	Write DB query and associated python function for joining Availability and Weather table.	No help needed.
	Joel	Added marker click and mouse over implementations		
Day : 28th Mar	**Michael**	Finished function to return tuples for graphing predictions	Test functions and implementations of journey-related predictions	No help required.
	Brian	No progress.	Nothing planned.	No help required.
	Joel	Weather flask created to obtain weather details from the api		
Day : 29th Mar	**Michael**	Test functions and implementations of journey-related predictions	Nothing planned.	
	Brian	No progress.	Nothing planned.	No help required.
	Joel	Created a weather box to display weather details		
Day : 30th Mar	**Michael**	Test functions and implementations of journey-related predictions	Test functions and implementations of journey-related predictions	
	Brian	No progress.	Nothing planned.	No help required.
	Joel	Integrated weather onto the screen with pressure and humidity and wind speed as well		
Day : 31st Mar	**Michael**	Finished required ML functionality	Nothing planned.	N/A
	Brian	No progress.	Nothing planned.	No help required.
	Joel	No progress.		

Section 7: Analysis of Results

There were several features developed during the project that were implemented in a way which prioritised short-term compatibility between different sections of the code and the project.
Despite close co-operation, some features were developed with less-than-ideal levels of interconnection, so that the various sections were not integrated to the maximum extent they could have been. A selection of these are discussed in the following Future Work section.

Future Work
While the implementation of the project can generally be regarded as successful with the primary objectives having been achieved, there are several improvements that can be made to improve the quality of the application.

The following points are some of the further improvements that the group would aim to implement if work on the application was to continue:

- The DatabaseScraper.py function was an integration of code implemented to acquire Dublinbikes and OpenWeather and code implemented in DatabaseAccess.py to populate the database. Integration of this code relied on interim storage of API call code results as JSON files which are then parsed and the data committed to the database. This approach was taken due to team familiarity with working with JSON files from a previous module. Ideally, these functions would be updated to remove the JSON intermediary files with API call results assigned to variables within DatabaseScraper.py before forwarding to the database.
- Similarly, results of the *get_station_data_from_db* and *latest_weather* functions return JSON files which are read by functions on the front end. Ideally, these would also be updated to remove intermediary JSON files where possible.
- As previously described above, due to time constraints and prioritisation of deployment troubleshooting tasks DatabaseScraper.py was not deployed to the new “COMP30830_Backup_Instance” instance and instead continued to run on the original “BC_COMP30830” instance. This is an important update which would be prioritised if work on the project was to continue.
- Database creation and population was handled using SQLAlchemy and database querying relied on Pandas. Further updates to align all database related functionality to one platform would result in a cleaner implementation.
- While all team members undertook tasks related to their own strengths and interests during the project, in a professional software engineering environment it would be important to ensure that all team members can contribute comfortably across the entire stack.
- Code cleanup, while performed prior to project submission, was reverted in error prior to application deployment. Removal of this code should ideally have been performed before deployment and is an important improvement task.

Fulfilment of Requirements and Project Conclusion

The requirements and expectations for this project were defined in the project description and listed in Section 1. Upon completion and deployment of the application the team has determined that all of the requirements of the application have been satisfied. Data collection using the JCDeceaux and OpenWeather was achieved using Python scripts that were also integrated with database population functionality thereby also fulfilling the data management and storage requirements. A Flask application was successfully implemented and deployed. A successful machine learning model was created and provides predicted bike station occupancy information based on a repository of historical bike availability and weather information. A user-friendly front-end implementation was created which provided the requisite interactivity, allowing users to find routes to bike stations, determine current bike occupancy at individual bike stations, determine nearest bike stations to points on the map, and plan trips with predicted bike and stand availability based on the machine learning model. Furthermore, the project utilised AWS EC2 and RDS services and progress in the codebase was recorded using GitHub repositories. Overall the project was reasonably successful as a first at development of a web application. As expected and discussed above in the Future Work section there are definite improvements that could be made, however, the team achieved its objective of developing and deploying the application and learning the particulars of the Scrum Agile software development process over the course of the project.