Software Development Report

Francesco, Fabio, Seán

February 2019

Contents

1 Introduction

The popularity of City Bikes has skyrocketed over the past two decades. Cycling is a healthy and environmentally friendly way to communting to and from work, and allows you to bypass traffic and avoid the annoyance of searching for a parking spot.

This Web Application, provides both real-time and predictive data of bike and space availability for all stations in the Dublin area adjusted for weather conditions on a user friendly and helpful map interface. This application would be helpful to anyone who uses Dublin Bikes on a regular basis, new users who are not familiar with station locations or day-trippers who need information to plan their trip.

This report aims to give the reader a rundown of the work and process that went into the development of this application.

1.1 Aim

The aim of the project is to create a website for the users of Dublin bikes that would like to find an available bike or stands to return the bikes rented. It will also provide weather-based predictions for the current day and also for the upcoming days, to let the user plan in advance if required. The Interface should be easy to navigate and simple to understand, with all the information displayed and visible.

Figure 1: Team FFS-22

2 Project Managment

This is a report documenting all the meetings the Software Development group FFS-22 have had until the date specified. The meetings will be broken down into 4 main Sprint, including various dates, with information on the work done, things decided, issues fixed and changes made to the project and it's design.

3 Sprint 1

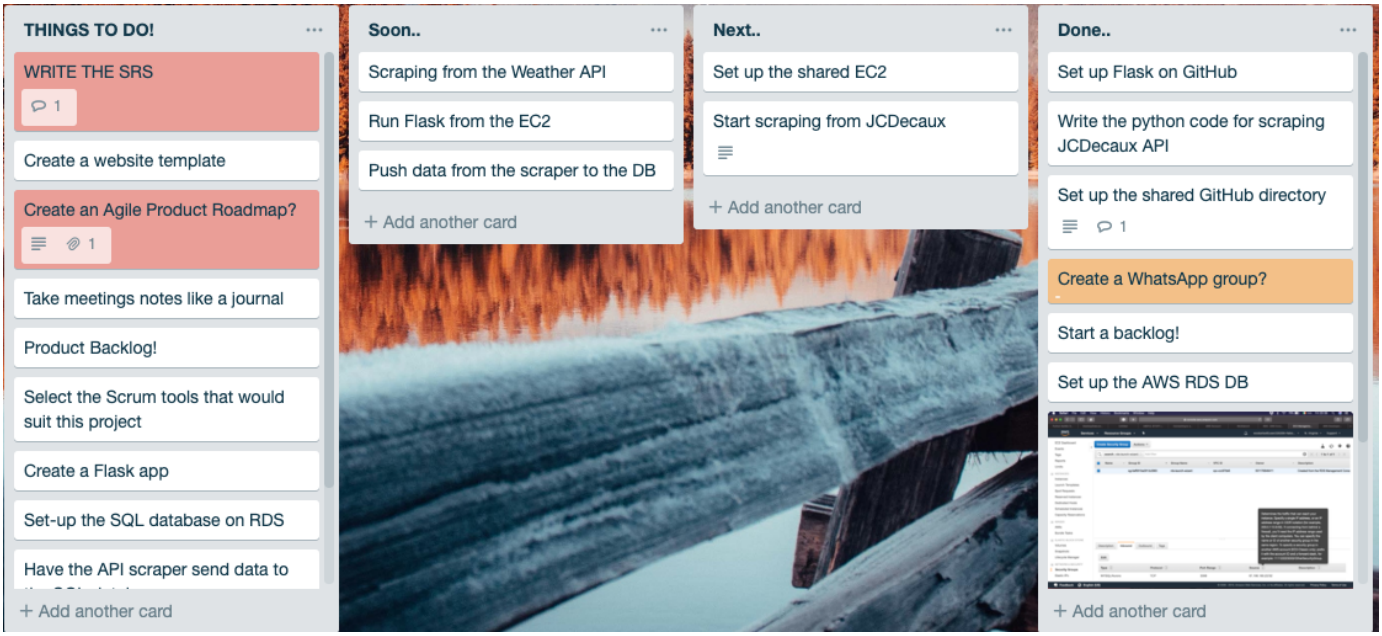


Figure 2: Trello board for sprint 1

3.1 Group formation 12/02/19 9.00-11.00

3.2 Membership

Membership of the team was finalised, team members are Francesco Ensoli, Fabio Magarelli and Seán Keyes. The group is number 22 and the supervisor/customer is Karl Roe (teaching assistant).

3.3 Discussion

We discussed several concepts and examined the brief. We also examined some existing websites and apps that perform a similar function.

3.4 Meeting 1; 14/02/19 9.00-11.00

3.4.1 Discussion

We discussed whether or not to write an SRS, examined existing applications in more detail and discussed how to do the scraping as well as making some mock diagrams of the user interface. Discussed what software we required RE organization and coding.

3.4.2 Work done

- SCRUM environment set up

3.4.3 Work to-do

- Scraping software
- RDS

3.5 Meeting 2; 15/ 02/19 12.00-18.00

3.5.1 Discussion

This was our long Friday meeting. We discussed the workload and our approach towards working as a group.

3.5.2 Work done

- Early version of the scraper
- Set up GitHub
- Set up virtual environments
- Created WhatsApp group for easier communication

3.5.3 Work to-do

- Fix scraper version 1
- Look into setting up RDS

3.6 Meting 3; 18/02/19 17.00-19.00

3.6.1 Discussion

We discussed the static and dynamic data and what needed to be in both. We examined the new scraper version and discussed how it was to work. We discussed the structure of the database and the tables in it.

3.6.2 Decisions made

We decided to add the banking variable to the static data. We also decided that one table with data for all stops was superior to having the data divided into one table per stop.

3.6.3 Work done

- Started the scraper
- Scraper started uploading data to the databases

3.7 Meeting 4; 19/02/19 9.00-11.00

3.7.1 Discussion

We discussed the user interface in greater depth, as well as discussing the use of the weather information with regards to using this information to inform the prediction. We also discussed the errors we were having when scraping the JCDecaux API. We discussed what to do for the next meeting with Karl.

3.7.2 Decisions made

We decided we needed more sample designs for the user interface. We needed to do the project backlog and to write the meetings up in a document for later use. We also decided to create a second error log in order to store the information about when stations don't update.

3.7.3 Work done

We examined our current documentation. We also looked at changes to the GUI.

3.8 Meeting 4 20/02/19 13.00-13.30

3.8.1 Discussion

We discussed the work we did the previous evening. We looked at the Backlog document and discussed the storage and functional uses of error data and the format of the files in use. We discussed the packages used in the virtual environment online and the virtual environments we were each using personally. We discussed the synchronization of the packages within. Documentation of SRS-Like data.
We also discussed chartist.js for visualization of the graph data. We discussed scrum platforms, vivify, jira, (you_plan?)

3.8.2 To-do

- Finish current work; GUI, Documentation and Backlog

3.8.3 Wishlist additions

- Symchronisation of VIs with EC2 instance
- Station update analysis data
- Weather component of the analysis
- Ask Karl about the SRS-and other data.

3.9 Meeting 5 21/01/19 9.00-11.00

3.9.1 Discussion

we discussed the data and its potential uses eg. we could use the data to help JCDecaux decide where they should install additional bikes and stands. It would also help us see if the stands with banking access were more popular then it could indicate that it would be good to add that functionality to other stands. We discussed whether or not we should do a prediction over every 15 or 30 minutes instead of every 5, which would reduce the number of scrapes we need, but make the the information less immediately useful.

3.9.2 Decisions made

Colours for icons, Scrape every 5 min to avoid inconsistency.

3.9.3 Work done

- Error reports in report not website

4 Sprint 2

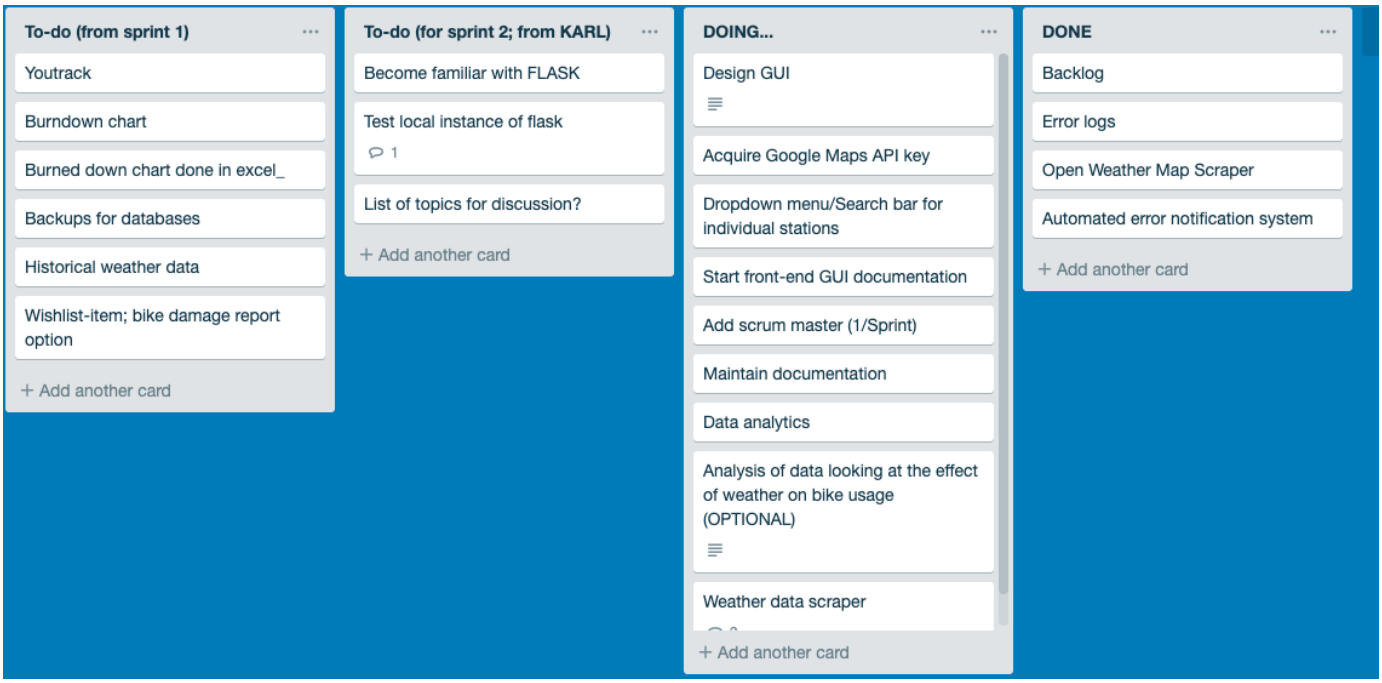


Figure 3: Trello board for sprint 2

4.1 Meeting 6 26/02/19 9.00-11.00

4.1.1 Discussion

We discussed the main issues we faced in sprint 1 and the issues we would have to tackle in sprint 2.

- The need to add a scrum master
- The burndown chart
- The backups for the database
- Error logs
- Youtrack
- Backlog
- Python code on flask to push into

- Weather data; do we scrape twice a day? Do we use historical weather data for prediction or real time scraped data
- Wishlist idea; a section for a user to report a broken/faulty bike so that the issue can be fixed.

4.1.2 Work done

- Put all the things on Trello

4.2 Latest report

- Crash reports; AWS went down on the 28th of february
- error handling
- error messages from the programs if there is an issue is sent to all 3 members of the team
- basic designs of the GUI
- Google maps API
- Dropdown menu search bar for individual stations
- Open Weather Map scraper
- backlog
- Error logs
- backup collected data
- look at historical vs actual open weather data.

4.3 Meeting 7/3/19

- We started trying to move the data on bike locations to the front end using JQuery
- GUI work
- data analysis plan made, started working on figuring out the code for it

5 Sprint 3

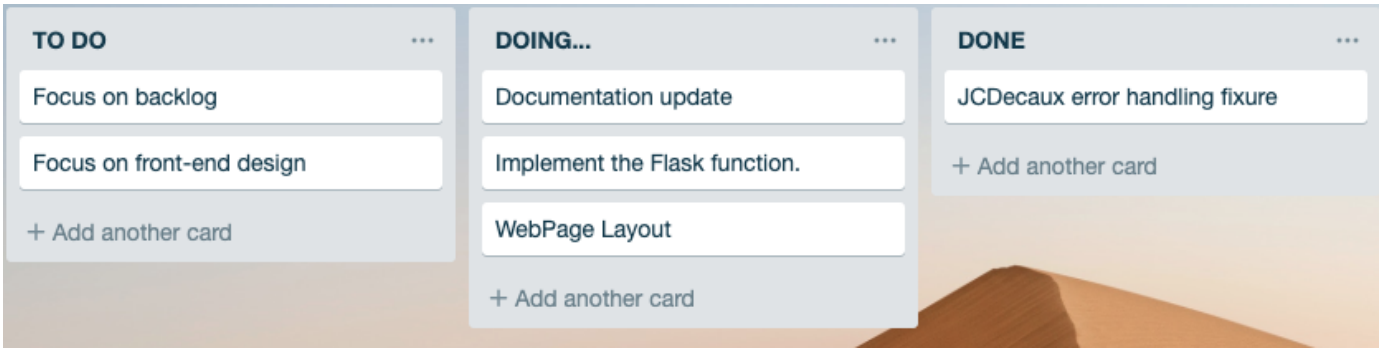


Figure 4: Trello board for sprint 3

5.1 Meeting 16/3/19

- ——We started trying to move the data on bike locations to the front end using JQuery
- GUI work
- data analysis plan made, started working on figuring out the code for it

6 Sprint 4

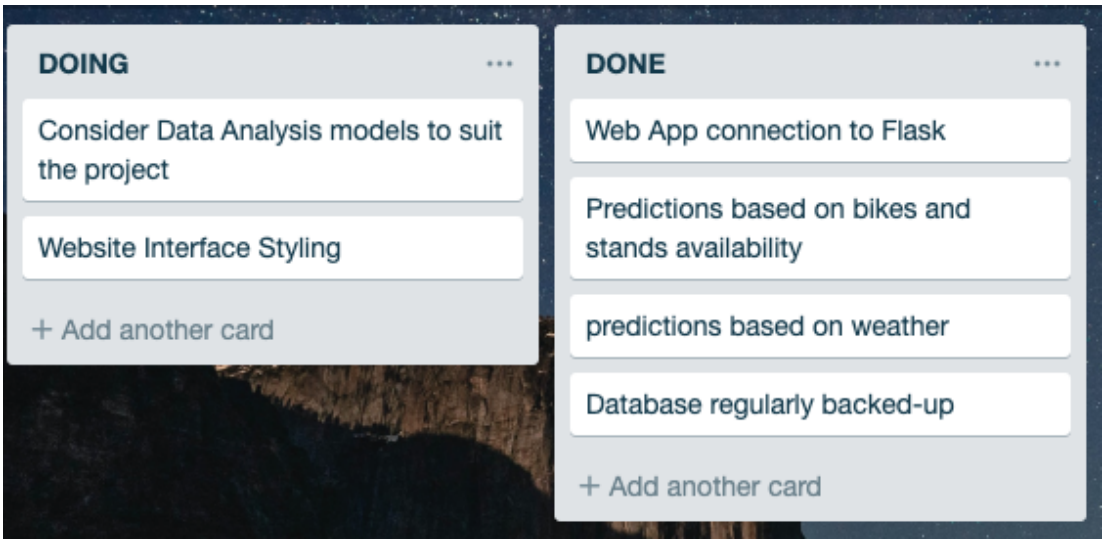


Figure 5: Trello board for sprint 4

6.1 Meeting 2/4/19

- discussed flask functionality and backlog items

6.2 Meeting 12/4/19

- We started trying to move the data on bike locations to the front end using JQuery
- GUI work
- data analysis plan made, started working on figuring out the code for it

7 Burndown Chart

The Burn-down Chart for the four sprint gives a rough estimate of the time spent on the most important sections of the project. In respect to the progresses made, the following surmise has been deducted. By looking at the chart, it is clear to notice that the first sprint in which the group got together, gave us the chance to discuss on weaknesses and strengths of each team member, to organise and decide which planned feature each person would have started to look into, hence mostly planning took place. After the first sprint, everyone adopted the agile system confidently, and got busy working on their respective features, whilst occasionally getting together during the allocated weekly practical times. By the start of the third sprint, due to unforeseen circumstances, one of the team members gradually disengaged himself, not committing to attending meeting and provide updates on his part. This member was the one who's taken the task of being the scrum master but also to work on the segment for the data analytic for the predictions. It was by the start of the forth sprint that the other members realised what was happening and started working on implementing those features that were missing or not completed. It definitely has taught something important to the group, about the importance of simple but constant communication and the essentiality of the scrum master figure.

Task	start hours	sprint 4	sprint 3	sprint 2	sprint 1	total hours
GitHub repository	2	0	0	0	2	4
EC2 instance	2	0	0	0	2	4
Flask app	12	4	4	4	1	25
Dublin bikes scraper	4	0	0	0	4	8
RDS Database	2	0	0	0	2	4
Meeting reports	8	6	2	0	0	16
Front-end design	12	4	4	4	0	24
Interaction front-back end	8	4	4	0	0	16
Data analysis	8	4	0	0	0	12
Database backup	4	2	0	0	0	6
Actual remainin time	62	38	24	16	5	
Estimated remaining time	62	46.5	31	15.5	0	

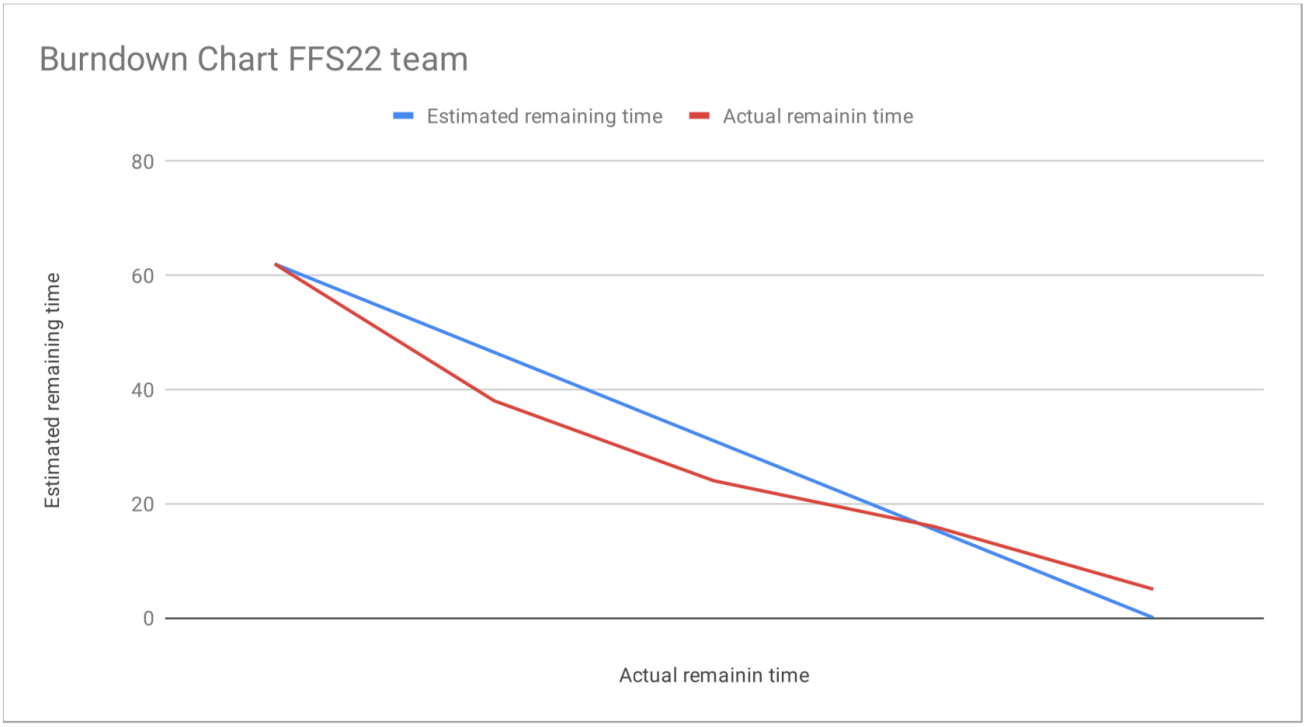


Figure 6: Caption

8 Development Operations (Dev-ops)

This section describe the Development operations decisions taken through the entire project development.

8.1 Technologies

- (local) virtual environment
- GitHub repository
- Amazon AWS server
- Amazon RDS MySQL Database
- Digital Ocean Backup Server
- Deployment on EC2 Instance

8.2 virtual Environment

Each developer have an anaconda virtual environment on his/her local machine in which to install eventual additional packages. in order to develop and test new components and features of the product. This one won't be uploaded on the EC2 instance: concerns regarding the packages update can be ignored considering the short time of this project. Furthermore, eventual packages updates can be performed directly on the EC2 instance if necessary.

8.3 GitHub Repository

A private GitHub Repository has been created to store all the data relative to the project. All team members, the lecturer and our demonstrator Karl Roe, have been invited as contributors. We agreed to use a single master branch and commit all our changes on the same branch. Each member would have worked on different files and performed a `git pull origin` every time before starting to work to reduce the risk of version conflicts. Some minor conflicts have been resolved hard pulling or pushing with caution. The team also agreed that the GitHub repository would have contained all the files relative to the project such as html files, python scripts, documentation, images etc.

Link: https://github.com/fabiom91/bikes_dublin_FFSTeam22

8.3.1 Repository Structure

```
bikes_dublin_FFSTeam22 :
  scrapdynamo.py
  README.md
  main.py
  index.html
  open_weather_map_scraper.py

  static:
    CSS:
      grid.css
      style.css
    images:
      ...

  accuracy_log:
    accuracy.csv
    accuracy.py

  logs :
    error_logs.txt
    station_error.csv
    flask_error.txt

  documentation :
    GUI_definition
    Backlog
    software_development_reports.pdf
    docs_img :
      (documentation images...)
```

Software_development_group_report.pdf

8.4 Amazon AWS Server

As main cloud server we've used Amazon AWS EC2 free tier ubuntu instance. While in the development phase has been set to accept connections only via SSH, this has been changed to accept all connections on deployment. The EC2 Server runs in multiplexing (using tmux) the Dublin Bikes API scraper, the Open Weather Map API scraper, the Accuracy evaluation script and the Flask server application. In the last months of development we got 2 server crashes which were most likely due to AWS: the first one occurred on the 28th of February at 16.00 but speaking with other teams, this crash occurred to everyone else. The second crash has not been tracked correctly. After the first

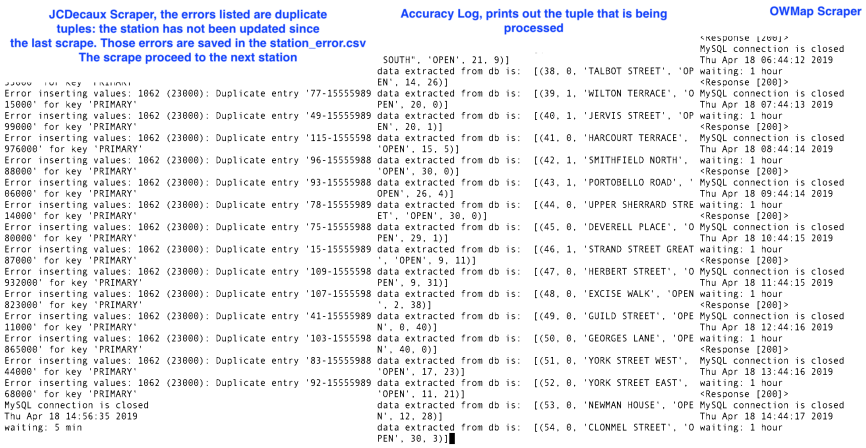
crash we agreed to implement a function that would have sent an email to each one of the team members when a crash occurs. The actual function works well to catch crashes due to API Scrapers malfunctions and Database connection errors but is not effective to catch server crashes. Maybe a further implementation on the backup server would have been able to catch them but this feature has not been implemented due to time restrictions.

8.4.1 Custom Packages installed

Python 3 and Conda shall be installed on the EC2 Instance.

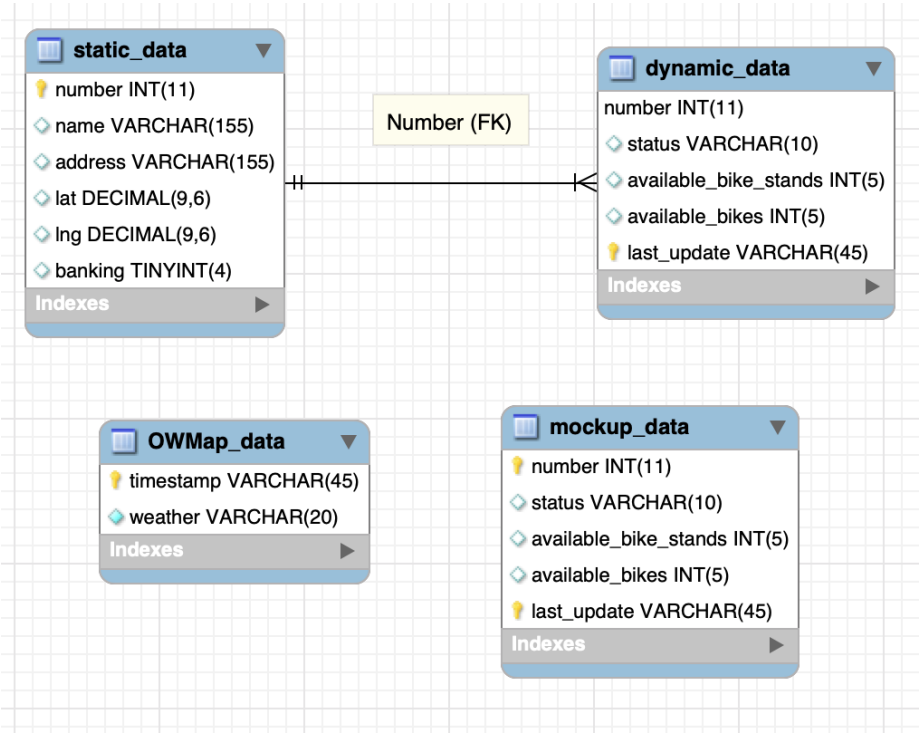
- **pip list**
  - flask 1.0.2
  - matplotlib 3.0.2
  - mysql-connector-python 8.0.15
  - numpy 1.15.4
  - pandas 0.24.0
  - pip 18.1
  - pypyodbc 1.3.4
  - requests 2.21.0
- **brew list**
  - wget
  - mysql
  - tmux

```
Fabios-MacBook-Pro:~ fabiomagarelli$ cd /anaconda3/envs/comp30830
Fabios-MacBook-Pro:comp30830 fabiomagarelli$ ssh -i "key_comp30830.pem"
ubuntu@ec2-34-238-40-161.compute-1.amazonaws.com
Welcome to Ubuntu 18.04.1 LTS (GNU/Linux 4.15.0-1031-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Thu Apr 18 14:55:52 UTC 2019

  System load:  0.0               Processes:           113
  Usage of /:   86.9% of 7.69GB   Users logged in:     1
  Memory usage: 30%               IP address for eth0: 172.31.93.120
  Swap usage:   0%

  => / is using 86.9% of 7.69GB

 * MicroK8s is Kubernetes in a snap. Made by devs for devs.
   One quick install on a workstation, VM, or appliance.

    - https://bit.ly/microk8s

 * Full K8s GPU support is now available!

    - https://blog.ubuntu.com/2018/12/10/using-gpgpus-with-kubernetes


  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

 * Canonical Livepatch is available for installation.
   - Reduce system reboots and improve kernel security. Activate at:
     https://ubuntu.com/livepatch

137 packages can be updated.
0 updates are security updates.


*** System restart required ***
Last login: Thu Apr 18 11:02:33 2019 from 193.1.165.202
ubuntu@ip-172-31-93-120:~$
```



8.5 Amazon RDS MySQL Database

The RDS Database have been developed mostly using MySQL Workbench. All the relations of the single database have been created manually by the developers and populated dynamically through python scripts and functions. The initial Database schema was composed by three tables: one for the static data from the Dublin Bikes API, one for the dynamic ones and the last one for testing. While the testing table has been left into the database till the end of the deployment, one other table have been added to store info regarding the Open Weather Map API scraper. We discussed regarding the opportunity to save data with a certain frequency and which data were meaningful for our application. This is the final Database Schema:



8.5.1 static_data

Contain static informations regarding all the stations in dublin city. Since those informations does not change in the short time, they can be updated as needed (even just once).

`number (PK), name, address, lat, lng, banking`

8.5.2 dynamic_data

Contain dynamic informations regarding all the stations in dublin city. Those data are updated every 5 minutes by the scraper running on the EC2 Instance. Since number and last update are primary keys, the relation does not accept duplicates tuples in which both station number and time already exist. Furthermore, number is a foreign key of the static_data relation "number" such as the relation accept data only if they matches a known station.

`number (PK) (FK), status, available_bike_stands,`
`available_bikes, last_update (PK)`

8.5.3 mockup_data

Contain Dynamic data for practice. It's been used when implementing new features or components in order to preserve the actual data from unexpected changes. This table is been populated with the dynamic data of 24h with a scraping interval of 10 minutes.

`number (PK), status, available_bike_stands,`
`available_bikes, last_update (PK).`

8.5.4 OWMap_data

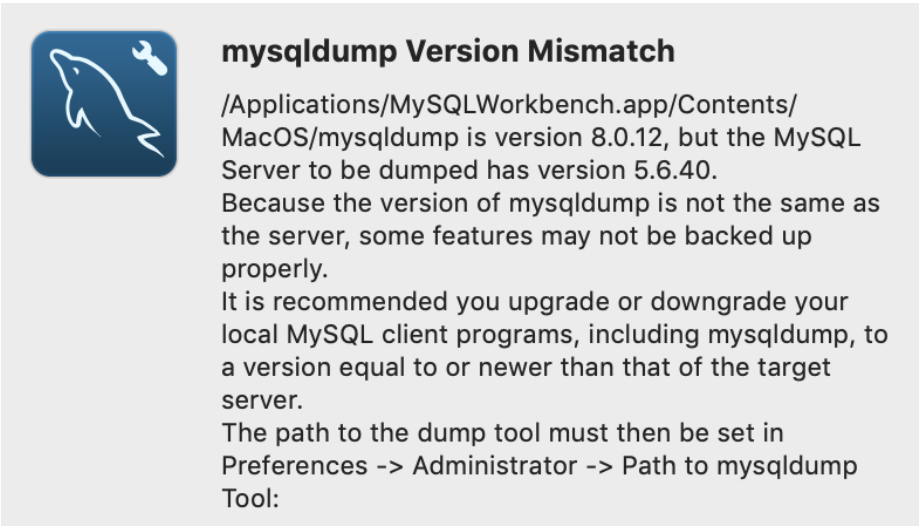
Contain a timestamp as primary key and a weather condition such as rainy, sunny etc. It is populated with the main value of the weather scraped from the Open Weather Map API with an interval of 1h.

`timestamp (PK), weather (NN).`

8.6 Digital Ocean Backup Server

Database backups are made on csv files. The database is checked for new entries every hour on a separate server and every 1000 new records regarding dynamic_data table are saved into a backup file while a single file contain the OWMap data. Due to space limitation, a dedicated server has been created on Digital Ocean in order to store the backups. At this stage there is no other strategy regarding the deletion of old data. All data are stored physically on the backup server as csv files.

Different solutions have been attempt in order to make regular and exeptional database backups during the development: At first we tried exporting the database with MySQL Workbench but we encounter a compatibility issue with mysql version and the mysqldump one which we weren't able to solve in a practical way so we discard this option.



After this first attempt, we implement a python function running on a local machine which query the database and store the data received into 3 csv files: one for each table. We had 2 problems with this solutions: the query of the dynamic_data table took more than an entire night to compute and the csv file was in the order of the 10Gb which wasn't acceptable for any of the local machine of the team members.

The third solution, the one currently working, is not the best solution nor a definitive one but for now it suits the team needs: A new server instance has been created on Digital Ocean and on this instance, a python script has been uploaded and run in a mutliplexer which query the dynamic_data table of the database for the last 1000 entries and save them in a separate csv file. The Server solution takes advantage of the computation power of the cloud server and its big hard drive to store the data until a definitive solution will be found. (such as backing up only the last n number of rows etc.)

Since the size of the backup is quite prohibitive, there is no way to actually show it. If you need a demonstration, ask a Fabio Magarelli to show you the backup server running (before the end of May, then the server will be definitely shut down). This is a screenshot of the backup server:

```
Last login: Thu Apr 18 15:21:43 on ttys000
Fabios-MacBook-Pro:~ fabiomagarelli$ ssh root@178.62.98.249
Welcome to Ubuntu 16.04.6 LTS (GNU/Linux 4.4.0-145-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

11 packages can be updated.
0 updates are security updates.

New release '18.04.2 LTS' available.
Run 'do-release-upgrade' to upgrade to it.


Last login: Thu Apr 18 14:20:47 2019 from 87.198.180.22
root@dublinbikes:~#
```

```
490000.csv   591000.csv  692000.csv  793000.csv  894000.csv  984687.csv
491000.csv   592000.csv  693000.csv  794000.csv  895000.csv  985687.csv
492000.csv   593000.csv  694000.csv  795000.csv  896000.csv  985791.csv
493000.csv   594000.csv  695000.csv  796000.csv  897000.csv  986791.csv
494000.csv   595000.csv  696000.csv  797000.csv  898000.csv  987689.csv
495000.csv   596000.csv  697000.csv  798000.csv  899000.csv  988689.csv
496000.csv   597000.csv  698000.csv  799000.csv  900000.csv  989341.csv
497000.csv   598000.csv  699000.csv  800000.csv  901000.csv  990341.csv
498000.csv   599000.csv  700000.csv  801000.csv  902000.csv  990866.csv
499000.csv   600000.csv  701000.csv  802000.csv  903000.csv  991866.csv
500000.csv   601000.csv  702000.csv  803000.csv  904000.csv  992223.csv
501000.csv   602000.csv  703000.csv  804000.csv  905000.csv  993223.csv
502000.csv   603000.csv  704000.csv  805000.csv  906000.csv  993647.csv
503000.csv   604000.csv  705000.csv  806000.csv  907000.csv  994647.csv
504000.csv   605000.csv  706000.csv  807000.csv  908000.csv  995111.csv
505000.csv   606000.csv  707000.csv  808000.csv  909000.csv  996111.csv
506000.csv   607000.csv  708000.csv  809000.csv  910000.csv  996983.csv
507000.csv   608000.csv  709000.csv  810000.csv  911000.csv  997983.csv
508000.csv   609000.csv  710000.csv  811000.csv  912000.csv  998983.csv
509000.csv   610000.csv  711000.csv  812000.csv  913000.csv  999117.csv
510000.csv   611000.csv  712000.csv  813000.csv  914000.csv  OWMap_data_bk.csv
511000.csv   612000.csv  713000.csv  814000.csv  915000.csv
root@dublinbikes:~/dbBackups/backups#
```

```
root@dublinbikes:~# cd dbBackups/
root@dublinbikes:~/dbBackups# ls
         get-pip.py  temp_backup.py
root@dublinbikes:~/dbBackups# python3 temp_backup.py
1046149     <— Total number of tuples in the table
select * from Dublin_bikes_db.dynamic_data limit 999117, 1000117
backups/999117.csv   <— CSV file created
select * from Dublin_bikes_db.dynamic_data limit 1000117, 1001117
```

8.7 Deployment on EC2 Instance

Important: Due to a technical issue with `pip` on the EC2 server, we had to temporarily deactivate all cache features from the python code. The issue cannot be resolved in time prior the submission of the product. This issue is due to a `pip` package problem since the `functools.lru_cache` packet is now incorporated into the latest versions of python3 and shouldn't require any install through `pip`.

Deployment has been made on the EC2 instance modifying the default ip and port: `127.0.0.1:5000` to `0.0.0.0:80` and setting the security group of the Server to allow any connection. Other solutions such as nginx and Gunicorn have been considered but discarded due to operational reason. Furthermore, we are not much concerned about eventual security breaches at this stage. the link to the running web application is :
`http://ec2-34-238-40-161.compute-1.amazonaws.com`

9 Back-end development

This section describe the Back End Development decisions taken through the entire project development.

9.1 Technologies

- API Scrapers
- Flask Application
- Data Analysis
- Accuracy Check
- Error Logs

9.2 API Scrapers

API Scrapers are essentially python programs to make some API calls to retreive data from the Dublin Bikes API and the Open Weather Map

9.2.1 Dublin Bikes Scraper : scrapdynamo.py

JCDecaux API python scraper algorithm, get the data from JCDecaux and save the relevant ones (dynamic) in the RDS Database. Loop every 5 minutes. It also handle errors such as:

- **station update missing** : write the station number and the timestamp into logs/station_error.csv .

- **station update missing** : write the station number and the timestamp into logs/station_error.csv .
- **API call errors** : write the error log into logs/error_logs.txt and send an email to the team!.
- **DB Connection errors** : write the error log into logs/error_logs.txt and send an email to the team!.

9.2.2 OWMap Scraper : open_weather_map_scraper.py

Open Weather API scraper, get the data re weather conditions in Dublin City: gets only the "main" realtime forecast: rainy, sunny etc. and save it in the RDS Database. Loop every 1 hour. It also handle errors re API calls errors and database connection errors. save those errors into logs/error_logs.txt. and send an email to the team!

9.3 Flask Application

Our web application use Flask to compute all the main computations and connect to the various APIs and the database. In particular, the the following are the functions implemented in the flask app (main.py):

9.3.1 request_static_data

request the static data from the database to populate the google map with the stations.

9.3.2 request_info_box

request the last update of number of bikes and stands for a given station number in order to display them in an info_box on a station marker click on the map.

Get as argument the station number and return the available bikes/stands and banking at the moment (The last available update)
argument(s) = station_number
return = station_number : { available_bikes : value , available_stands : value , banking : value }

9.3.3 request_weather

Request real time weather from OWMap API direct call.

Return the real time (now) weather from OpenWeather Map API to display over the map For the user to know what is the weather now.
return : weather_dict = {'temp' : value, 'description' : value, 'wind_speed' : value, 'icon_link' : value}

9.3.4 get_weather_influence

compare for every station the mean of available_bikes and available_bike_stands when the weather is "clear" and when the weather is the current weather (get this info from OWMap API). return the percentage of difference between the 2 predictions.

return : influence = {bikes: value, stands: value} – *e.g. value : 12% more/less*

9.3.5 request_hourly_prediction

This prediction is made for a selected station returning the average number of available_bikes and stands for a given day, grouped by hour.

Get as argument the number of the station and the day of the week (Monday, tuesday...) and return a JSON file with the prediction of the average hourly availability of bikes/stands along the day
argument(s) : station_number, week_day
return : h_pred = {time : [values], bikes : [values], stands : [values]}

9.3.6 request_weekly_prediction

This prediction return the average availability of bikes and stands for a selected station grouping them by day of the week.

Get as argument the station number of which we want to know the prediction and return the prediction of the average daily availability of bikes/stands along the week
argument(s) : station_number
return : w_pred = {day : [values], bikes : [values], stands : [values]}

9.4 Data Analysis

When implemented the Flask functions: `request_weekly_prediction, request_hourly_prediction and get_weather_influence` we used the average availability of bikes and stands for given input(s) to make a first data analysis to predict the future availability of bikes and stands. This method is not the most accurate one. We considered to use a linear regression model but due to time constraints this option has been discarded.

9.5 Accuracy Check

Since we couldn't implement a linear regression model in time for submission, an accuracy study has been performed on our "model" which use the average availability of bikes and stands in the past to predict the future one. The results can be found in this repository in `accuracy_log/accuracy.csv`. The accuracy log currently show an accuracy of the 20%. The accuracy of the prediction

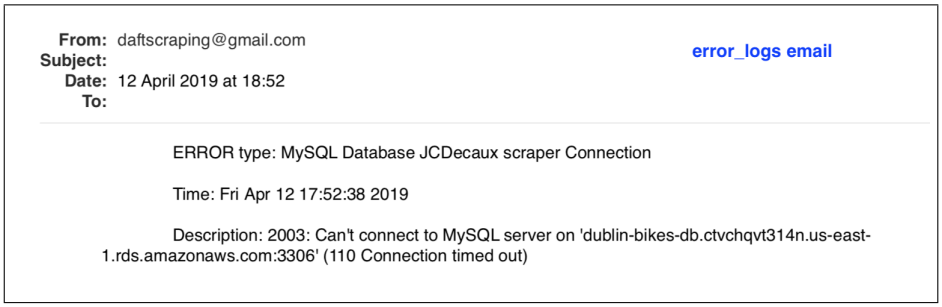
is calculated for the hourly prediction comparing the predicted values for each hour against the actual ones.

9.6 Error Logs

As already mentioned, errors regarding API and MySQL connection and station update into dynamic data are catched by the various functions in through the application back-end. All those error logs are saved in different files:

- **error_logs.txt** : contain all the eventual logs of API Connection and DB Connection errors occurring from the scrapdynamo.py script run. Those error logs are very important because they can cause the scraper to crash which may result in a loss of data.
- **station_error.csv** : this csv file contain informations regarding errors that occur when a station data is not updated: since the number of station and the last update are foreign keys of the relation, the database cannot accept tuples in which both of those values are duplicates of a previous tuple. This kind of errors does not make the scraper crash but the record is skipped. It could be relevant to collect data regarding the un-collected records in order to monitor the functionality of the JCDecaux service.
- **flask_error.txt** : contain all the eventual logs of errors due to a failure connecting the web app to the data analysis python scripts through flask. This is important to detect and correct eventual bugs. **Note:** since a proper model for data analysis is not been implemented, this file is not currently catching any errors.

Furthermore, errors logs due to API and MySQL connection are sent via email to all the members of the team. **Note:** the sender email address for the error logs (daftscraping@gmail.com) is an email already used for other applications since this would be seen only by the team members there was no need to create a new email address for this purpose.



error_logs.txt

```
ERROR type: JCDecaux API
Time: Mon Mar 25 23:20:17 2019
Description: <Response [503]>


ERROR type: OWM (Open Weather Map)
Time: Wed Apr 10 21:18:46 2019
Description: <Response [500]>


    ERROR type: MySQL Database JCDecaux scraper Connection
    Time: Fri Apr 12 06:26:04 2019
    Description: 2003: Can't connect to MySQL server on 'dublin-


    ERROR type: MySQL Database JCDecaux scraper Connection
    Time: Fri Apr 12 06:33:43 2019
    Description: 2003: Can't connect to MySQL server on 'dublin-
```

station_error

TIME	ERROR_DESCRIPTION
Sat Mar 2 18:13:33 2019	1062 (23000): Duplicate entry '30-1551549878000' for key 'PRIMARY'
Sat Mar 2 18:13:33 2019	1062 (23000): Duplicate entry '54-1551549823000' for key 'PRIMARY'
Sat Mar 2 18:13:33 2019	1062 (23000): Duplicate entry '56-1551549946000' for key 'PRIMARY'
Sat Mar 2 18:13:33 2019	1062 (23000): Duplicate entry '18-1551550076000' for key 'PRIMARY'
Sat Mar 2 18:13:33 2019	1062 (23000): Duplicate entry '32-1551549796000' for key 'PRIMARY'
Sat Mar 2 18:13:33 2019	1062 (23000): Duplicate entry '52-1551550064000' for key 'PRIMARY'
Sat Mar 2 18:13:33 2019	1062 (23000): Duplicate entry '23-1551550025000' for key 'PRIMARY'
Sat Mar 2 18:13:33 2019	1062 (23000): Duplicate entry '106-1551550042000' for key 'PRIMARY'

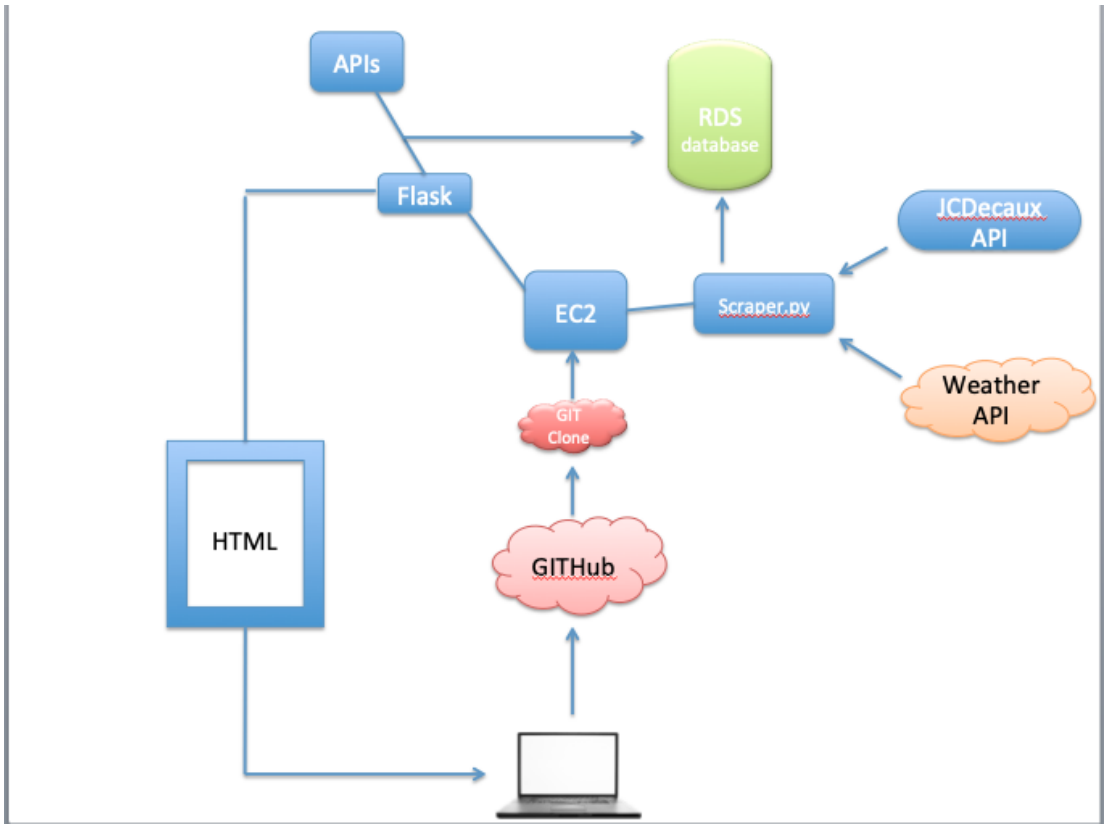


Figure 7: Architecture structure

10 Front-End development

This section explains the technologies used for the front-end development along with the structure of the files deployed. Having used Flask throughout the testing phase, the structure required by Flask to work is as follow:

- bikes_dublin_FFSTeam22
- index.html
  - Static
    - CSS
    - images

The main HTML file is inside the main folder, meanwhile the *css* files and the images used are respectively inside the *CSS* and *images* folders.

10.1 Front-End technologies

- HTML
- CSS
- JavaScipt
- JQuery

10.2 HTML

For the project we have used the HTML5 standard, which has made easier to structure an entire page, relegating most of the styling to *CSS*. Utilizing newer elements like *"header"* to create the uppermost page DIV and *"footer"* for the lowermost one. But also *canvas* used on the charts DIVs and several other elements like *calendar* and *Geolocation*, which can track the user position in real time if required, making the whole process very easy to implement. The whole page is divided into four main sections, namely the header, the map, the information window and the footer.
• Header
• Map
• Information window
• Footer
The header is where we have decided to show the weather predictions coming from the weather API. Below it there is the map which takes up most of the screen estate. The map we have used is from the Google Map API. The map can be customized to ones preference and has very well written manuals to further customize it and implement on all of its functionalities. Following the map there is the section where all the information is displayed. To welcome the user and avoid cluttering the screen with much information at the start, there is a splash screen covering the section. The idea is for the user to only see the general sensible info and later select the extra information required. Upon selecting one of the markers, the splash screen disappears and the actual window replaces it. For the purpose of this application the footer is left empty. In a future update information like the person to contact or the product owner name could be added if wanted.

10.3 CSS

CSS3 was used to stylize in the web page the arrangement of the elements. There are two files for the CSS:
`/.static/css/grid.css`
`/.static/css/style.css`

Throughout the page elements have been displayed using *'Grid'* layout. While inside sections the 'Flex box module layout is also used. Everything in

relation to modularity and display arrangement has been created and modified within the 'grid.css' file. For instance the header width spacing has been done using flex grid, which makes the header responsive and keeps the ratio between all the elements included. The second file 'style.css' is the one where all the style options have been done: from the size, to the color of different elements, including info-windows. The style that was chosen for the web-page vaguely resembles that of an arcade video game, in term of colors and proportions.

10.4 JavaScript

JavaScript is the main engine behind the site, transforming the originally static page into a dynamic one, increasing the client interactions. It is used to make requests to the scraper, and stylize the various DIVs in which the information is contained or sent to. For the project the Javascript code resides in the HTML file. We have used some vanilla code to implement some functions, and some Jquery code to do the pull requests to the server. Javascript offers many frameworks. We have used Jquery extensively (described next) and also ApexCharts to visualize the charts on the site. These charts are already implemented to animate and be dynamic, adapting to the div they are placed in. Furthermore using a JSON style format is it possible to customise them to anyone's liking.

10.5 JQuery

This JS framework is used mostly for the Ajax capabilities of easily sending pull requests to the server and for the way it simplify complex methods into few lines of code. For the site we used two versions one for the standard framework and one for the jQuery-UI theme implementation.

The jQuery has several functions relative to the Flask application, that pull specific data to fill different sections of the website.
Specifically:

request_static_data: this function returns all the information about the stands, like name, number, status, location and if it has banking available. These information is then used to create all the markers on the map, with corresponding variables.
Furthermore by knowing the location of each station using its latitude and longitude, it is also possible to calculate the distance with the user and advice on the nearest stations. Within the request is also possible to create a div to show the information pulled from the server to the user, in this case the nearest stations from the user.

request_info_box: is used to request to the scraper the specific information about a stand. It returns the latest update on the number of bikes and stands available and displays it in an info-window that is created within the function

request_hourly_prediction and *request_weekly_prediction*: functions are both sending the required data to the respective chart, placed at the bottom of the screen. The charts are respectively showing the amount of bikes and stands available throughout the day and on a weekly prediction.

get_weather_influence: function returns the predictions calculated based on the weather for the day selected. These predictions are returned on the initial splash screen.

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
<script src="https://code.jquery.com/ui/1.12.1/jquery-ui.js"></script>
```

In the next example we can see how a request is made and variable is passed. The example shows how the request is made to retrieve the station data, to further manipulate it.

```
// Function to calculate the distance between the user cordinates and the nearest stations
function functdistance() {
  $.getJSON(ROOT + '/request_static_data', null, function(data) {
    var dublinbikes = data; // static data of all stations
```

11 Project Delivered

The web app interface provide sensible information about the current status of the Dublin Bike service, with advanced functionalities to make predictions based on weather forecast.

11.1 Key Features

- Display real time availability of bikes and stands.
- Display real time weather forecast.
- Predict hourly and weekly availability for bikes and stands.
- Display current position of the user (hard coded).
- Display nearest stations based on the user location.
- Display weather influence on bike and stands availability.

11.2 Performance optimisations

To improve performance when retrieving data from the database, we implemented on the Python built-in LRU-cache library. Due to the deployment operational issues (described in the dev-ops section) this feature has been temporary deactivated.

Another issue encountered when deploying the system on the server, has been with the HTML Geo-location feature, which requires a SSL certificate to operate. Being unable to retrieve a certificate before the deadline, we have been advised to hard-code the user location just to show proof the functionality.

11.3 Key Shortfalls and Planned Improvements

Due to unforeseen circumstances with one of the team members, some of the features that were originally planned, have been delay or postponed to a future update of the app. These features are: Data analysis models (e.g Linear regression), proper deployment methods (e.g usng NGINX), Geolocation and SSL certificates, security improvments of the server and the database, proper analisys of station that systematically fail to update using `station_error.csv` file, proper accuracy analysis of the predictions based on `accuracy.csv` file.

Other considerations that have been made during the development phase, were: to provide the user with a registration field, which would grant access to a user area, where is possible to report issues like faulty stations, faulty bikes with a direct line of contact in case of any other issue related to the service.

12 LINKS:

GITHUB repository for the project
https://github.com/fabiom91/bikes_dublin_FFSTeam22

Project Web page
http://ec2-34-238-40-161.compute-1.amazonaws.com/