1. Introduction

City biking has been traditional across Europe. Expanding cities like Copenhagen, Paris, Berlin, Vienna are often referred as bike friendly capitals. Bicycles are a healthy, environmental friendly way of commute. It needs simple infrastructure of a bicycle lane and bikes station network to operate. A motor road attempts to occupy garbage truck, bus, medium sized trucks, cars and motor bikes at a same time. Unlike this scenario, bicycle tracks carry single sized, single speed traffic providing congestion free commute.

Dublin bikes , which is operational since late 2006 has been a efficient commute option for Dublin citizens. It would be put on test more and more as Dublin is expanding. City has seen population growth of 79,600 merely in last 5 years since record GDP growth of 26% in 2015.

Dublin Bike web application provides a real time availability of bikes and bike stands at Dublin bike stations. Present weather condition is also available given volatile weather experienced in Dublin. Similarly, predicted weather and bike station availabilities for following 5 hours is displayed for source and destination stations.

* 1. Aim

Dublin Bike Web application is made to facilitate existing and new users with near real time bike station information. With forecasted weather conditions and availability of bikes and stands at bike stations, it helps users and subscribers in planning their commute. This application is made for use on desktops as well as on mobile phone browsers. Appropriate visualizations and easy navigation are incorporated to improve user experience.

1. Project Management

This project is an instrument to learn end to end software development life cycle using agile methodology. Entire project was divided into 4 major sprints:

* Planning and Database setup
* Flask and Web UI design
* Data Analytics
* Integration Testing , UX and Improvements

We had team of 3 engineers. A team member was appointed as scrum master and given individual expertise, a team member each lead first 3 sprints.

We used following tools for project management:

* Trello is Digital Kanban board developed by Atlassian is efficient way to plan track project progress. Sprints are broken into lists. And individual stories of sprints are represented by cards. Cards can be assigned labels, checklists and due dates. Each card can be assigned to team members and attachments can be sent.
* GitHub is industry standard version control software tool. Since parallel development cycles and CI/CD are main features of agile development, GitHub enabled us to avoid cumbersome mail chains.
  1. Stand-up meetings and progression

Scope of project was limited to only 4 sprints hence we avoided separate communication channel tool like slack. Instead we used a dedicated card on Trello to maintain stand up meeting logs. Trello provides an efficient cell phone application facilitating continuous communication.

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 \_nalised, team members are Francesco Ensoli,

Fabio Magarelli and Se\_an Keyes. The group is number 22 and the supervi-

sor/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.

6

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.

7

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 \_les in use. We discussed the packages used in the virtual en-

vironment 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

8

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

9

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 ask to push into

10

\_ 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 \_xed.

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 mem-

bers 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 \_guring out the code for it

11

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 \_guring out the code for it

12

6 Sprint 4

Figure 5: Trello board for sprint 4

6.1 Meeting 2/4/19

\_ discussed ask 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 \_guring out the code for it

13

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 \_rst 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 \_rst sprint, everyone adopted the agile

system con\_dently, 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 de\_nitely has taught

something important to the group, about the importance of simple but constant

communication and the essentiality of the scrum master \_gure.

14

Figure 6: Caption

15

1. Development Operations (Dev-ops)

Scope of project was to deploy a working public accessible website hosted on Amazon EC2 cloud and developed using python web framework. This section describe the Development operations decisions taken through the entire project development.

* 1. Local Environment

Project consisted 4 major languages to work with:

* MySQL for Database
* Python for web framework and Data analytics
* JavaScript and HTML for front end

To avoid conflicts with environment variables, all team members worked with anaconda

virtual environment. This enabled unit testing efficient as environments could be shared

and updated as yaml files

* 1. GitHub repository

Two private git repositories were created for this project : each for static development and dynamic development. Team members decided to opensource both repo post academic evaluation.

* Static development consisted of data scraper, Server and database related files, test Time series data analytics algorithms and documentation.
* dynamic development contained driver programs using flask web framework, website front end, and data analytics backend with deployed Machine learning models.

We appointed a team member as master for each repo and others worked on separate branches. This avoided merge conflicts and codes were unit tested regularly.

* + 1. Repository Structure
  1. 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 \_rst 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 \_rst

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 e\_ective 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

{ ask 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

* 1. 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 devel-

opers 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 test-

ing. 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 \_nal Database Schema:

* + 1. Static Dublin Bike Data

Dublin\_bike\_staticdata contained one time information about the stations. This information has long shelf life and ideally an entry must not change. New entry are added only when a new Dublin bike station is introduced. This table logs following key features:

Id\_entry(PK): introduced primary key

number(FK): station Id

name: station name

position\_lat: station location latitude

position\_lng: station location longitude

bonus: station bonus

banking: status of e-payment service

data\_entry\_timestamp : Time at which JCDecaux API was queried and station entry was made in Dublin\_bike\_staticdata. Irrespective of last\_update; for consumer of API, time of data entry into database remains ultimate status of station.

* + 1. Dynamic Dublin Bike Data

Table Dublin\_bike\_dynamicdata contain dynamic information regarding all the stations in dublin city. This data is accessed for making ML predictions and generating station statistics of weekly and hourly bike availability.

Id\_entry: introduced primary key

Number: station id

Status: station status open/ closed

bike\_stands: Total capacity of station

available\_bike\_stands: number of empty stands available to park

available\_bikes: Number of bikes available for pickup

data\_entry\_timestamp : Time at which JCDecaux API was queried and station entry was made in Dublin\_bike\_dynamicdata. Irrespective of last\_update; for consumer of API, time of data entry into database remains ultimate status of station.

* + 1. Weather Data

Open\_weather\_dynamicdata stores over 37 parameters of Dublin weather. Only 9 of those are used for actual ML predictions. But we have chosen to preserve this data for future use, where ML model for weather prediction takes more weather features into consideration. Key features of database used are:

Id\_entry: Introduced primary key

main\_temp: Temperature

main\_feels\_like: human perception of temperature

main\_pressure: Atmospheric pressure of Dublin sea level in hPa

main\_humidity: percentage humidity

main\_temp\_min: Minimum temperature at the moment

main\_temp\_max: Maximum temperature at the moment.

wind\_speed: Wind speed in meter/seconds

wind\_deg: Wind direction in meteorological degree

weather\_main ; Group of weather parameters (Rain, Snow, Extreme etc.)

data\_entry\_timestamp : Time at which openweather API was queried and station entry was made in Dublin\_bike\_dynamicdata. Irrespective of last\_update; for consumer of API, time of data entry into database remains ultimate status of station.

* 1. 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

1. Back-end development
   1. API Scrapers

Dublin Bike station data is made public through an webAPI by operating firm JCDecaux. Static and dynamic data is saved into database. Similarly, weather data for Dublin published by openweathermap.com data is made public through an webAPI. API web scraper written in python is used to log bike data and weather conditions into database. Error checking and data processing is done for mentioned factors:

API json response rendered false : log into logfile.txt with datetime

KeyError for json response: When Feature[column] is not returned by API call , append None entry and complete the data frame for data entry

Database connection in error : log into logfile.txt with datetime

Temperature unit conversion: Temeprature parameters published by openweathermap.com are in degree Kelvin. They are converted into Celsius scale before logging.

* 1. Crawl Program
* Periodically call scraper to update database
* Threading – avoid infinite loop
* Automation
* Error check
  1. Flask Application

Flask is a library free web framework written in Python. For a small scale rationed server application like this project, flask is a suitable infrastructure since it is lightweight than counterparts like nodeJS. Flask has simple task of rendering homepage of website on launch. Further, flask application serves requests from webpage, through defined functions on call.

* + 1. get\_data

Argument: None Return: jsonified Dublin Bike Static data

Operation: Fetches all entries in Dublin Bike Static data. The function returns bike station geolocation, banking, bonus and status indicating if station is operational or closed.

* + 1. get\_station\_info

Argument: Return:

Operation:

* + 1. station\_data

Argument: None Return: jsonified latest entry for a station in Dublin Bike Dynamic data

Operation: Data from JCDecaux is routinely scraped and logged into database. Hence, latest availability at the station is found by operating on latest entry for the station.

* + 1. get\_station\_history

Argument: None Return: Jsonified station availability in terms of weekdays and of 24 hours

Operation: Availability of bikes and bike stands at station varies depending on the time and day of week. It is imperative that more bikes would be available in late night and on weekends as office and university hours are over. Hence, User is provided feature to observe daily and hourly availabilities at a station.

* + 1. get\_prediction

Argument: None Return:

Operation:

* + 1. current\_weather:

Argument: Return:

Operation:

* + 1. toCSV

Argument: Return:

Operation:

* 1. Data Analysis
  2. Accuracy check

By design, crawl program is supposed to continuously scrape data. But given COVID – 29 pandemic, stay at home phase was implied by government of Ireland in last week of March 2020. Hence, data forth 24 March is omitted for model training and models are trained on approximately 24 days of data. The individual model performance can be observed in following result files:

* result\_available\_bike\_stands.csv
* result\_available\_bikes.csv
* resultWeather.csv

Average performance of the models seem satisfactory for a 100 layer random forest with unnormalized features. Metrics of predicted parameters are tabulated bellow:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **PARAMETER** | **AVERAGE MAE** | **AVERAGE MSE** | **AVERAGE RMSE** | **AVERAGE R2** |
| **Bike Station** | | | | |
| Available Bike Stands | 0.754 | 2.285 | 1.447 | 0.966 |
| Available Bikes | 0.752 | 2.272 | 1.442 | 0.966 |
| **Weather** | | | | |
| main\_temp | 0.256538908 | 0.173240186 | 0.416221318 | 0.982769756 |
| main\_feels\_like | 0.425911731 | 0.579625668 | 0.76133151 | 0.93579148 |
| main\_pressure | 0.17960511 | 0.106446225 | 0.326260977 | 0.999453893 |
| main\_humidity | 2.555911731 | 14.00315912 | 3.742079518 | 0.873247242 |
| main\_temp\_min | 0.278315912 | 0.205021719 | 0.452793241 | 0.980531098 |
| main\_temp\_max | 0.307409988 | 0.241941463 | 0.491875455 | 0.97548472 |
| wind\_speed | 0.451118118 | 0.665969461 | 0.81606952 | 0.891950984 |
| wind\_deg | 7.286504065 | 446.1720049 | 21.12278402 | 0.901932654 |

9.3 Flask Application

Our web application use Flask to compute all the main computations and con-

nect to the various APIs and the database. In particular, the the following are

the functions implemented in the ask 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 num-

ber 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 : f available bikes : value , available stands : value ,

banking : value g

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 = f'temp' : value, 'description' : value, 'wind speed' :

value, 'icon link' : valueg

26

9.3.4 get weather inuence

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 di\_erence between the 2

predictions.

return : inuence = fbikes: value, stands: valueg { 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 \_le with the prediction of the average hourly

availability of bikes/stands along the day

argument(s) : station number, week day

return : h pred = ftime : [values], bikes : [values], stands : [values]g

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 = fday : [values], bikes : [values], stands : [values]g

9.4 Data Analysis

When implemented the Flask functions: request weekly prediction,

request hourly prediction and get weather influence we used the aver-

age availability of bikes and stands for given input(s) to make a \_rst data anal-

ysis 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

27

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 di\_erent \_les:

\_ 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 \_le 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.

\_ ask error.txt : contain all the eventual logs of errors due to a failure

connecting the web app to the data analysis python scripts through ask.

This is important to detect and correct eventual bugs. Note: since a

proper model for data analysis is not been implemented, this \_le 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.

28

29

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 \_les 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 \_le is inside the main folder, meanwhile the css \_les and

the images used are respectively inside the CSS and images folders.

30

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 \_les 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

31

relation to modularity and display arrangement has been created and modi\_ed

within the `grid.css' \_le. For instance the header width spacing has been done

using ex grid, which makes the header responsive and keeps the ratio between

all the elements included. The second \_le 'style.css' is the one where all the

style options have been done: from the size, to the color of di\_erent 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

\_le. We have used some vanilla code to implement some functions, and some

Jquery code to do the pull requests to the server. Javascript o\_ers many frame-

works. 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

speci\_c data to \_ll di\_erent sections of the website.

Speci\_cally:

request static data: this function returns all the information about the stands,

like name, number, status, location and if it has banking available. These infor-

mation 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 lon-

gitude, 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 speci\_c 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

32

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 inuence: function returns the predictions calculated based on

the weather for the day selected. These predictions are returned on the initial

splash screen.

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 fur-

ther manipulate it.

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 inuence on bike and stands availability.

33

11.2 Performance optimisations

To improve performance when retrieving data from the database, we imple-

mented on the Python built-in LRU-cache library. Due to the deployment oper-

ational 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 certi\_cate to

operate. Being unable to retrieve a certi\_cate 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 fu-

ture update of the app. These features are: Data analysis models (e.g Linear

regression), proper deployment methods (e.g usng NGINX), Geolocation and

SSL certi\_cates, security improvments of the server and the database, proper

analisys of station that systematically fail to update using station error.csv

\_le, proper accuracy analysis of the predictions based on accuracy.csv \_le.

Other considerations that have been made during the development phase, were:

to provide the user with a registration \_eld, 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/