

Dublin Bikes

Website: <http://34.254.231.183:5000/>

Code: <https://github.com/marib731/Web-app-stuff>

- **Enxi Cui, 33.3% - prediction modelling and weather scrapping**
- **Marian Barry 33.3% - flask, displaying station data and reporting/meeting notation**
- **Jessica Butler 33.3% - flask, front end, and AWS/server handling**

Upon forming a team we took some time to discuss what was required of each team member and perform some necessary introductory tasks. Among the discussion topics was what to do if any team member encountered a problem, how to deal with disagreements and where and when the team would meet. While it can be viewed as over simplistic to do so, past work experience from a group member suggested that a page with group rules and agreements genuinely does help in making sure every member of a team is beginning with the same expectations and goals. Keeping in contact was important so we exchanged phone numbers and made sure that we had each others email address. In this time we also created a Trello, a Whatsapp group and a Google docs shared drive. Later on in the project when we had finished some of the initial code, we created a Github branches and a shared Slack group.

The initial team meetings after receiving the project specification were spent preparing a more specific project brief. This new project brief specified in great detail what the project should achieve. Before these new features were added to the project brief we undertook research to make sure that each feature was possible in the given time and with our given skills.

What should app do?

Find station near you

View station details (full, partial, empty)

Add favourite station

Follow Dublin bikes main feed for any outages/roadworks (alert system if in your favourites?)

Space to place advertising/in app purchases (possible client asks as later on consideration) diff languages?

Personalize which stations show/don't show at startup

Customize occupancy indicators

Display weather

Predict use later in day – peak times and weather as factors

Route planner between stations where most/least traffic/time estimates/start from, walk to nearest station-cycle, nearest drop station and walk rest

How regular update information?

Check account to see if trip will cost money/how much?/how much will we owe/outstanding debt/where pay?/alert if about to hit charge with suggestion of nearest free space (can turn off though)

Optional tutorial for new users? How to set up account/process of renting and returning a bike

Mobile version of site – app that just links to mobile version of site?

Python scripts often build up and tear down large data structures in memory, up to the limits of available RAM. Because MySQL often deals with data sets that are many times larger than available memory, techniques that optimize storage space and disk I/O are especially important. For example, in MySQL tables, you typically use numeric IDs rather than string-based dictionary keys, so that the key values are compact and have a predictable length. This is especially important for columns that make up the [primary key](#) for an InnoDB table, because those column values are duplicated within each [secondary index](#). (mysql official site)

[fetchone\(\)](#) retrieves a single item, when you know the result set contains a single row. [fetchall\(\)](#) retrieves all the items, when you know the result set contains a limited number of rows that can fit comfortably into memory. [fetchmany\(\)](#) is the general-purpose method when you cannot predict the size of the result

- **Too large to all fit in memory at one time.** You use `SELECT` statements to query only the precise items you need, and [aggregate functions](#) to perform calculations across multiple items. You configure the `innodb_buffer_pool_size` option within the MySQL server to dedicate a certain amount of RAM for caching query results.
- **Too complex to be represented by a single data structure.** You divide the data between different SQL tables. You can recombine data from multiple tables by using a [join](#) query. You

make sure that related data is kept in sync between different tables by setting up [foreign key](#) relationships.

- ***Updated frequently, perhaps by multiple users simultaneously.*** The updates might only affect a small portion of the data, making it wasteful to write the whole structure each time. You use the SQL [INSERT](#), [UPDATE](#), and [DELETE](#) statements to update different items concurrently, writing only the changed values to disk. You use [InnoDB](#) tables and [transactions](#) to keep write operations from conflicting with each other, and to return consistent query results even as the underlying data is being updated.

The updated project brief included the following sections:

Data collection and storage

API Keys

Each team member would acquire an API key for the google maps, open weather and JCDcaux API's. While we have only used one API key for each of the services to collect the data, the team did receive advice from demonstrators warning us of the overuse of one API key. Having extra accessible API keys provided the team with a back up plan in case of over use. These extra API keys were posted on Trello and were accessible to each team member.

Web Scraping

In order to collect the required data for the prediction model, we needed to collect large amounts of data. This would be implemented using a python web scraper.

Database Storage

The scraped data would be stored using a MySQL database. MySQL was chosen as each team member has experience with this database program.

AWS

Amazon Web Services (or AWS) would be used to host the finished flask web application and store the database information.

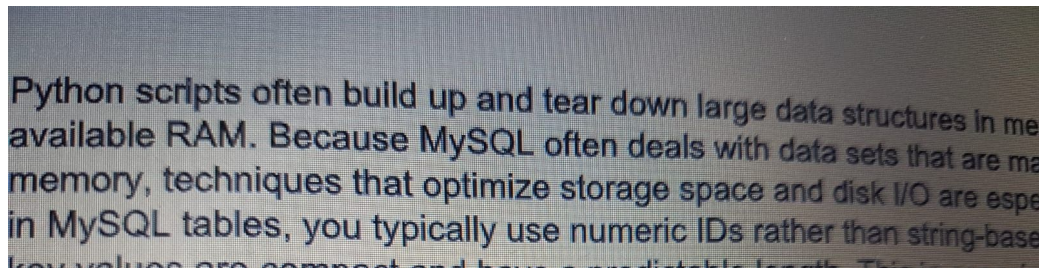
Flask Web Application

*Features - (Extra features will be annotated with a * symbol)*

- Google map (this should also include markers detailing Dublin bike stations)
- Weather data for the current time and location
- Google map which has a toggle button to show bike lanes and traffic.*
- Google route planner. * (abandoned due to time constraints)
- Weather data which can be filtered for another time and location *
- Station data which can be filtered for a particular station
- Prediction model that forecasts the available bikes for a certain station, at a certain time depending on the weather.
- User page which would enable a user to log in * (removed due to time constraints to build additional security - it didn't do much anyway)
- User logged in page. This would enable users to save favorite stations. * (removed due to time constraints to build additional security)
- An alert box which takes information regarding station closures and traffic updates from the Dublin Bikes twitter page.(removed dublin bikes as rarely updated and inconsistently formatted which made it difficult to display nicely) *
- Contact us page.
- FAQ page.
- Unit testing and connection failure catching (abandoned due to time constraints but should always be implemented before deployment)

There was also the concept of data structures and implementation considered, such as how the data was stored and called, and making the flask application do the heavy computation versus the client's system with Javascript. It was decided to bruteforce at first, and then once the code was

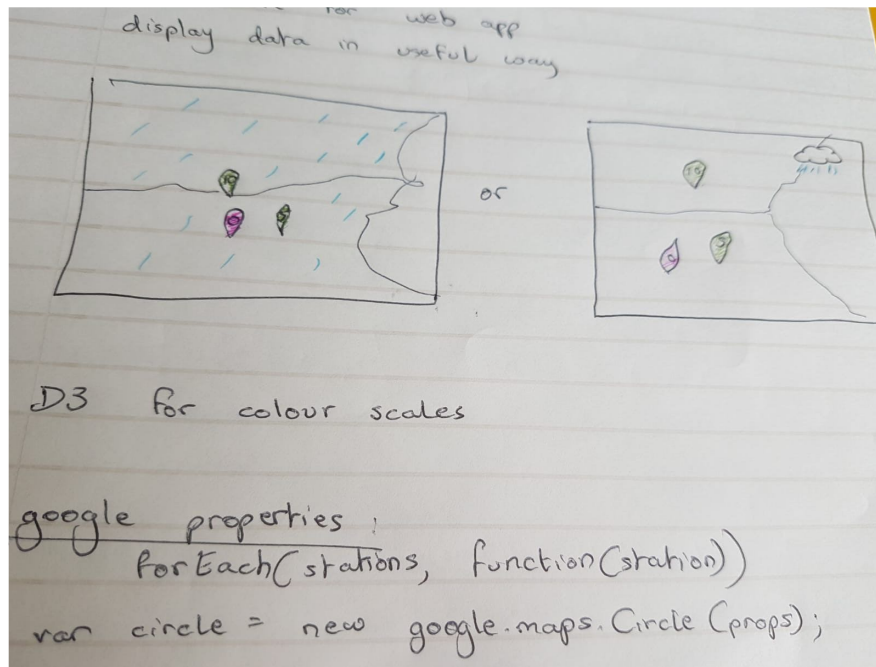
working this concept would be returned to. However, due to time constraints, this issue was never returned to and while the website runs, it is not as optimized as it could have potentially been. If more time had been available and/or a better understanding of Flask, much more of the processing would have been done through python scripts as if the web application was loaded by a tablet or similar device with low processing power, certain features will take significantly longer through the current javascript-implemented methods used. One optimization that did make it into the final code due to this situation was that the stations API calls directly to JC Decaux to return a JSON object, rather than calling all data in the RDS database, jsonifying all of the data and then sending all of the data to the front-end for javascript to process. However, overall many of the necessary optimizations are still lacking due to time constraints, and for this reason the scalability of the application is not what it could be.



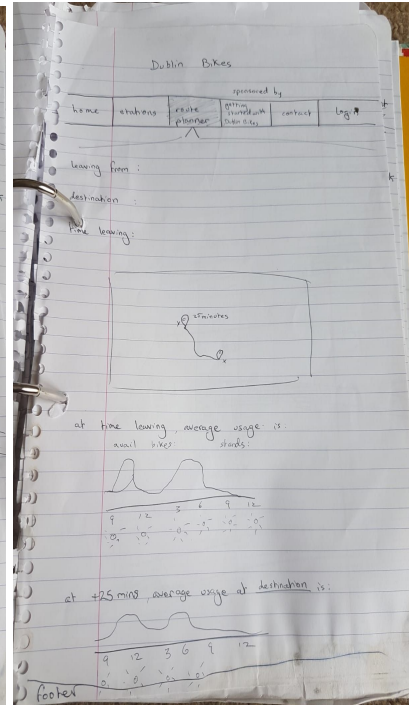
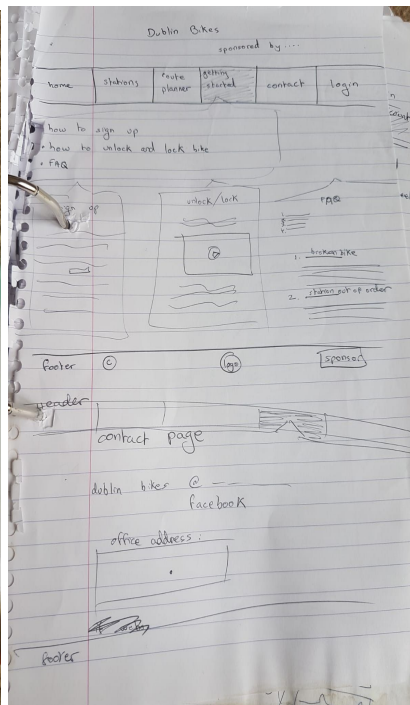
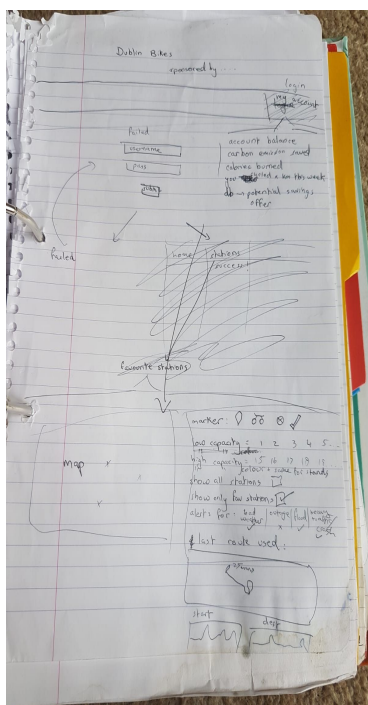
Mockups

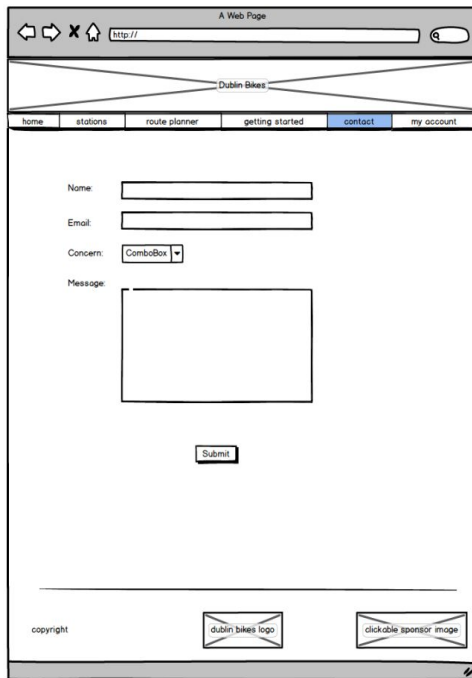
Before coding the website we created a number of mockups. This allowed the team to share the same perspective on the direction in which the assignment was heading. It also provided the team with a basic framework to aim for. The mockups and subsequential wireframes are available below.

deciding how to display weather on the map:



Initial concepts:





A Web Page

http://

Dublin Bikes

home stations route planner getting started **contact** my account

Name:

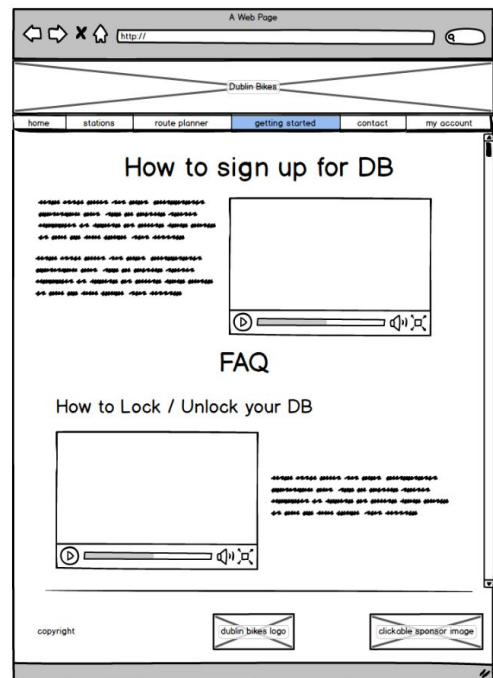
Email:

Concern:

Message:

copyright dublin bikes logo clickable sponsor image

Contact us page



A Web Page

http://

Dublin Bikes

home stations route planner **getting started** contact my account

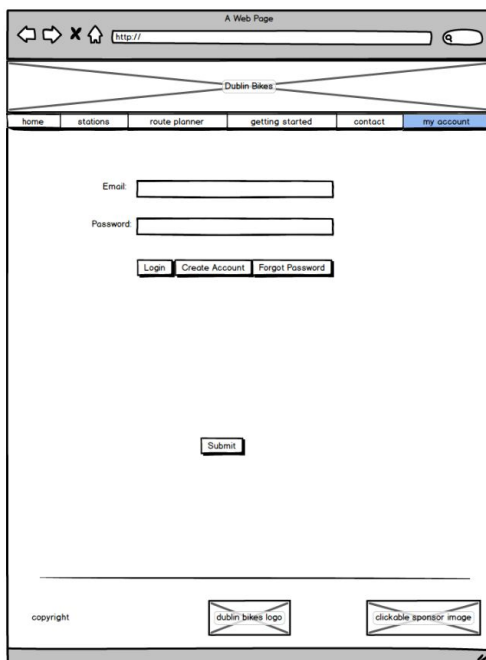
How to sign up for DB

FAQ

How to Lock / Unlock your DB

copyright dublin bikes logo clickable sponsor image

FAQ page



A Web Page

http://

Dublin Bikes

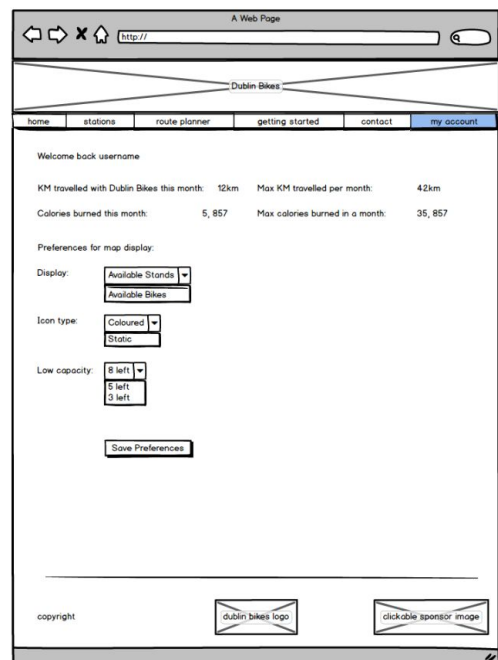
home stations route planner getting started contact **my account**

Email:

Password:

copyright dublin bikes logo clickable sponsor image

User login page



A Web Page

http://

Dublin Bikes

home stations route planner getting started contact **my account**

Welcome back username

| | | | |
|--|-------|---------------------------------|--------|
| KM travelled with Dublin Bikes this month: | 12km | Max KM travelled per month: | 42km |
| Calories burned this month: | 5,857 | Max calories burned in a month: | 35,857 |

Preferences for map display:

Display:

Icon type:

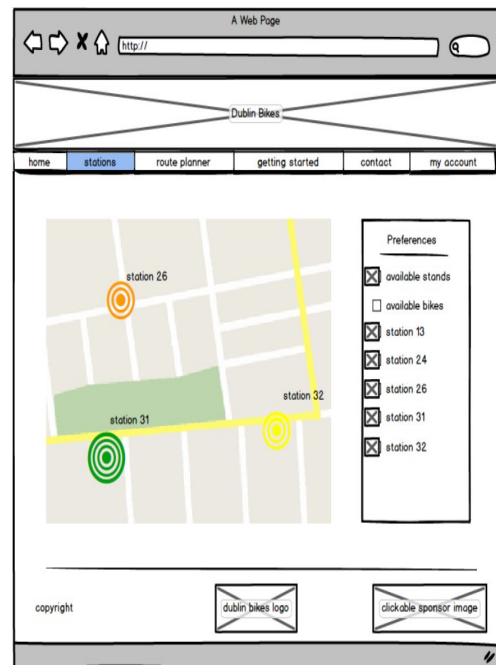
Low capacity:

copyright dublin bikes logo clickable sponsor image

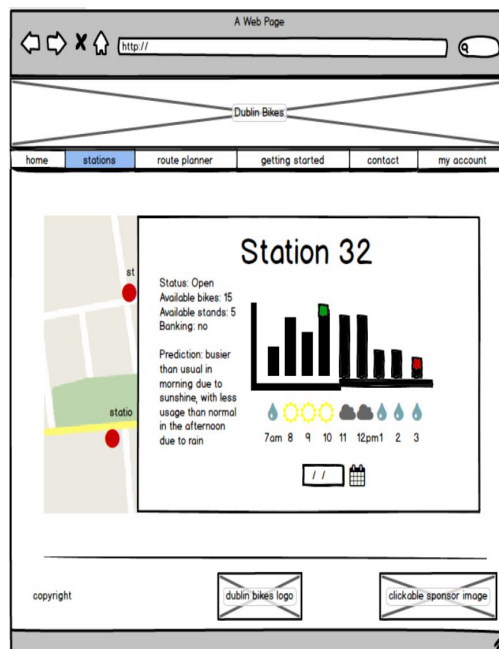
User logged in page



Google route planner page



Station page



Prediction model on station page

Project Management:

While constant contact was maintained throughout the completion of the assignment, the team followed a laissez-faire approach to the scrum process due to the difficulty in deciding timelines when no one was familiar with how many of the processes worked, let alone estimating how long it would take to make them once an understanding was achieved.

The daily scrums took place every week day with a few exceptions due to other assignment deliverables. On a monday a longer scrum meeting took place detailing the progress made, or problems encountered by team members during the weekend. From our research we have noted that a scrum master controls the daily scrum. Initially we had a rotating scrum master position where one person would control the meeting and ask the three questions: What did you do yesterday? What will you do tomorrow? Did you encounter any obstacles? The role of the scrum master became blurred as the project went on and we adopted an open floor session. While this open floor version of the daily scrum was just as useful as the traditional daily scrum, without the scrum master it lacked accountability. We found it more difficult to keep concise and accurate notes as there was no designated note taker. The quick concise meetings we had originally eventually morphed into longer team group sessions as we discussed best approaches and solved problems together.

Sprint planning took place after the daily scrums and we also considered any on going issues or problems other team members had. After the mock up and enhanced project brief stage we started to use the sprint planning process more often. While typical sprints are quite short, due to the easy going nature of our team we broke the project into larger, longer sprints. Our first sprints were as follows, for one person to create the web scrapers, one to do front end and another to do the back end flask work. While the general idea of the sprint planning process is to create small, easy to handle tasks our

initial long sprint was intended to create a base level skeleton of the project. This base consisted of the python web scraper and a basic flask web app framework. While this approach did originally work, when issues became apparent, the sprints were abandoned in favour of correcting errors. After the issues had been corrected there was a more casual approach to the splitting up of work, namely a divide and conquer process. Instead of specific tasks we would each take a feature from the enhanced project brief and try and implement it. While this did work for the project it presented a number of issues, namely there was no time scale for these divide and conquer elements. This led to team members running into issues and instead of raising them and tackling the issues together, the person working on the task would continue to work on the task until the problem was solved. If we had have used sprint planning correctly, issues would have been raised earlier and this would have allowed other team members with different expertise to help.

When new features had been completed we undertook a team review. This involved each team member making sure that the code would run on their computer. After this we would discuss optimising the code and the features. In some cases this involved improving the user experience. This involved for example changing layouts or in some cases dropping features that could not be implemented in the time we had left. The sprint review process took place with our client. The sessions with the client were especially useful. It gave us an opportunity to showcase the completed code so far and get advice on our implementation. In one case we were advised to remove a search box as this would leave us open to an SQL injection attack. It also allowed us the chance to prioritise certain required features.

When the project was completed the team conducted a sprint retrospective. The main topic of the discussion was the scrum process. We felt as a team that the application would have been more successful and also less work in the long run if we had followed the scrum process. Not only would we have accomplished more in a shorter period of time, the scrum process would

have allowed us to identify issues more quickly. The daily meetings morphed from a scrum meeting to a group work session which left us with very little documentation to reflect upon. If used correctly the meetings could have been a time to provide a common direction to work towards and given us an opportunity for discussion. Dividing the larger tasks into smaller tasks would not only have given each team member a chance to work with each part of the project but also would have led to more features being developed. Ultimately, the team suffered from not using the scrum process. It led to unnecessary pressure being put on the team and the project suffered as a result. We had intended to have a user log in section, a user home page and a route planner, but due to time constraints these were abandoned in favor of returning the required deliverables.

Prediction:

Data preparation:

After collecting the Dublin bike and weather data from the databases, over 700,000 rows of bike data and 1,000 rows of weather data were available for analysis. These were saved to a CSV format, to perform data analysis on without altering the database itself. Features were merged by the commonality of hour. Duplicate and null data was removed by dropping rather than imputation to reduce any potential human error that could alter the result patterns in the data. Mathematical analysis such as correlation between features were implemented for the purpose of visualising patterns in the data. A number of continuous and categorical features were highlighted as influencing bike usage, including temperature, weather, and time of day. As the weather data had a significant number of categorical features, which are not as is usable in data models, these features were converted to 'dummies' to represent them as numerical data. After data analysis and cleaning, approximately 280,000 rows remained.

Modelling the data

From our data analytics class we learned random forest was generally more accurate for large data as it combines multiple other methods. We were also made aware that random forests has a very large time complexity, and thus the decision was made to use random forests with the constraint that the model would not be retrained as little as possible. The data was eventually split 70/30 for training and testing purposes, as initial model was overfitting and producing a 98% accuracy as it had simply memorized the data rather than predicting it. The process for the decision trees and random forest modelling was as follows: if we consider Y the data we want to predict (available bikes), and x is the continuous data that we want use to predict the number of available bikes. The goal of separating the sites by the ID of the different sites is, first, to predict the bike of each site by separating the sites, since the availability of the available bike is also related to location Can make predictions more accurate. The second reason is because the amount of data over 280,000 is very large, and it can take a long time to fit them all at the same time, so we chose to write with ID separate station.

First, we train a decision tree classifier model, Instantiate sklearn estimator, fit with training set. Train a classification tree with `max_depth=10` on all dat, use a for loop to fit each station. Split them, and get the each station's prediction and its accuracy, mean absolute error, and mean squared error.

A for loop was implemented to store each station's prediction as a pickle file. As the pickle files can grow very swiftly in size, splitting them by station allowed for better resource and storage management.

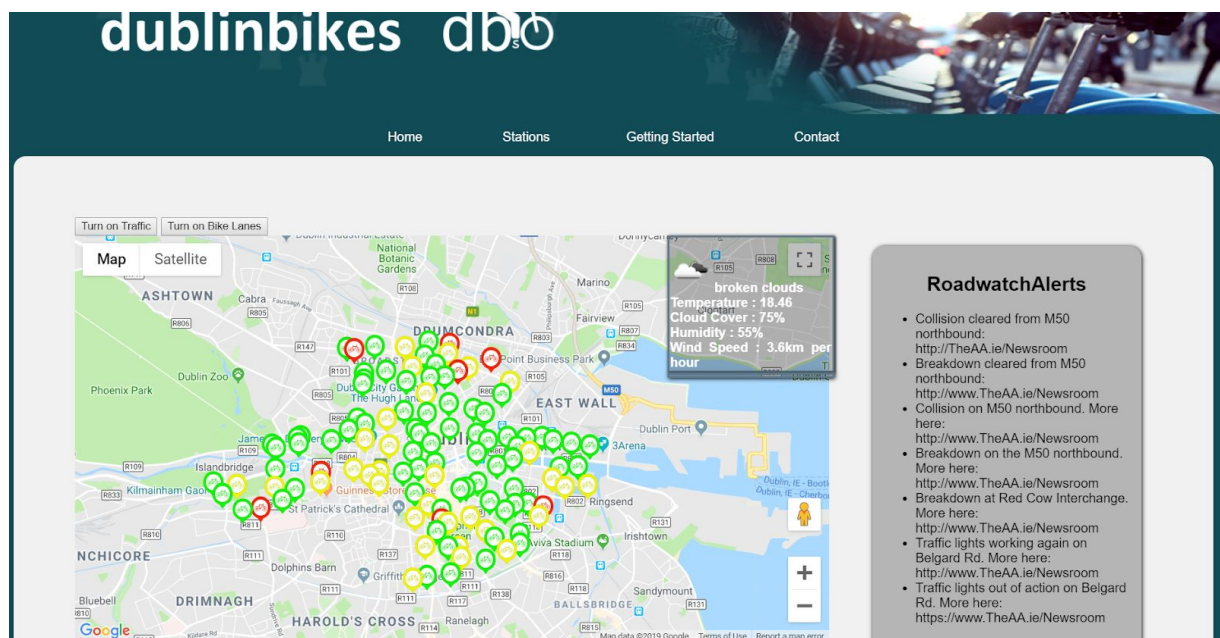
Final features of the finished application

The application consists of a standard webpage. We have used standard features such as navigation bars, buttons, drop downs and search boxes to

allow users to interact with the application more easily. The web page also resizes dynamically and the google map is zoomable.

In case of issues or questions, there is an easily accessible FAQ section accessible on all pages by clicking the 'getting started' section of the navigation bar. If users need further clarifications or help using the web page, they can access the contact information and email for help.

Home page - featuring google map with station markers - this map has buttons to turn on bike lanes and traffic information. An alert system also features on this page, it takes information from AA Roadwatch's twitter page. It originally also took data from the official website on updates for station closures, but the times of update were very infrequent, with no consistency for how the news article was written so it was ultimately removed as it kind of just looked messy. This advises users of any station closures or issues with dublin bikes. As a precursor, it also covers some overlap with the stations page, where clicking on the station brings up the available bikes and stands at that station. It is not intended to be thorough, but instead cover the logical choice that some users may make by clicking on an individual station to view more current information about it.



Station page - featuring real time weather information for the user's location, weather data for a user specified time and location, prediction model which predicts station capacity for a certain time depending on the weather, station information which is filtered by the station number. By having this split, current and prediction data is potentially 1-2 clicks for any user of the web application, which provides clarity and ease of use for anyone wanting to know this information.

Choose a date(in 5 days) and
time to see the weather:

dd/mm/yyyy

06:00:00

Show forecast

06:00:00

SMITHFIELD NORTH

Show Bikes Prediction

Input a
station ID
2

Look up

Clear

Station ID: 2

Address: Blessington Street
Total Bike Stands: 20
Available Bike Stands: 19
Available Bikes: 1

Station ID: 3

Address: Bolton Street
Total Bike Stands: 20
Available Bike Stands: 17
Available Bikes: 3

Station ID: 4

Address: Greek Street
Total Bike Stands: 20
Available Bike Stands: 14
Available Bikes: 6

Station ID: 5

Address: Charlemont Street
Total Bike Stands: 40
Available Bike Stands: 28
Available Bikes: 12

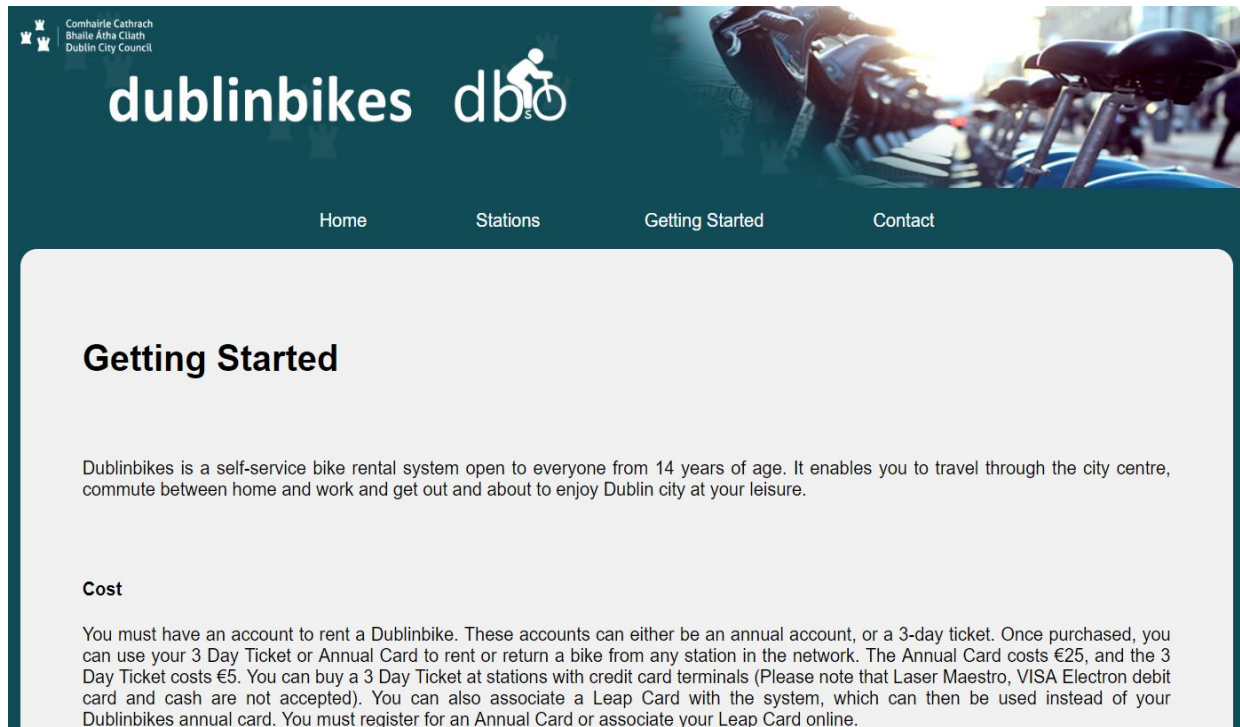
Station ID: 6

Address: Christchurch Place
Total Bike Stands: 20
Available Bike Stands: 20
Available Bikes: 0

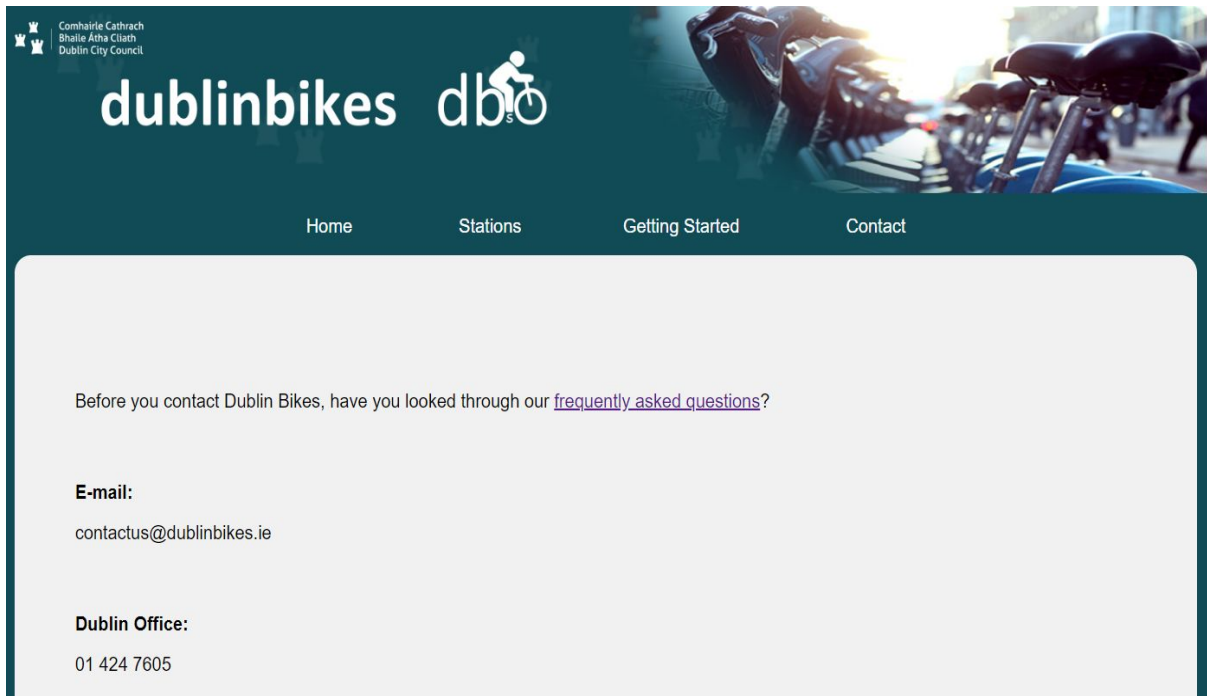
Station ID: 7

Address: High Street
Total Bike Stands: 29
Available Bike Stands: 29
Available Bikes: 0

Getting started page - This page outlines how to start using Dublin bikes , as well as frequently asked questions. The users were decided to largely comprise of people who would use or want to use the DublinBikes service, and so typical questions of how to use the bikes and deal with the system were presented here for users.



Contact page - page provides easily visible contact information for those having an immediate issue who may be too panicked or in an unsuitable emotional state to scroll through pages and pages on the official page to find out who to talk to about any particular issue such as bike theft, broken station, accidental charges and more.



Issues:

Web scraping

It had been agreed to scrape data from the API's once every 10 minutes. This would allow us to build an accurate predictive model. The web scrapers were built and data was feeding into the database. It was decided to keep the 24 hour time frame as bikes were unable to be rented for only 4.5 hours, and during this time were still allowed to be returned. Weather could also have a 24 impact, with heavy overnight rain still impacting usage in the mornings. It was discovered in the timeframe of sprint 3 that the Dublin bikes database had been overriding it's data each time, despite having unique identifiers. This issue was resolved, but in order to correctly train the model a more complete dataset from a different web scraper was used. We also eventually discovered that weather information had stopped feeding into the database. Upon reviewing the python web scraping file for the weather information, we quickly realized that instead of scraping the data every 10 minutes, the python file was scraping every 10th of a second. As we had

extra API keys available for our use, it was no problem to update the python file and the issue was quickly resolved.

Database connection

There were initial issues of team members not being able to access the shared MySQL database. This was due to security issues being restrictive with which IP addresses could access the database. This was resolved by opening connections for all incoming and outgoing traffic to any IP.

AWS Data usage:

Early in the project we started to receive notifications from AWS explaining that we had reached the free tier usage limit. This problem arose because of the security group settings we had changed. The security group acts as a filter that specifies who is allowed access the teams AWS instance. In order for all team members to be able to access the instance the settings were changed to allow access to any IP address. Requests made from non-team IP addresses were causing the high data usage. In order to solve this problem, we set up a custom rule to allow our IP addresses to access the instance. We also each signed up for the Github student developer package which gave each team member AWS credit of \$150.

A secondary issue arose with AWS, where a group member installed a clashing edition of Anaconda3 directly to the EC2, resulting in consultation with experts on how to resolve the issue. It was eventually discovered this was the cause of the server issues and remedied with much assistance.

Finally, unusual storage patterns emerged with the AWS instance, with space issues declaring 7.7GB of storage used, despite the entire code being under 1GB. The reasoning was never discovered, but the issue was resolved by hand removing old instances of the kernel. This did create additional issues

with have to reinstall environments, but was quickly resolved. Clearing the cache did not solve this problem alone, but did make some space. In order to reduce this issue from recurring we created a bash script cache cleaner that implemented the crontab to run at 2am and midnight each day as a pseudo-garbage collector. It was later accidentally deleted when removing old instances of the kernel and resolving environment and package issues.

Database issues

In trying to create a prediction model, we encountered an issue with splitting. When creating the prediction model, the original database which held the dublin bikes station information was split into 4 extra databases. We could not rectify this issue and instead asked another team for a copy of their historic database data. We continued scraping data from the API keys and adding to the database for the remainder of the project.

Github issue:

When the prediction model was completed, the pickle files were uploaded to Github to enable all team members to work with the files. Shortly after this, each team member started to experience issues with Github. This was mainly in the form of push and pull times. In some cases it took over an hour to retrieve a simple HTML file. After getting advice from demonstrators we realised that the issue was the uploaded pickle file. We decided as a team to remove the file from the team Github and instead upload it to Slack. We chose this approach as Slack has the capability to handle larger file sizes. While this issue was ongoing we simply emailed each other completed code. When the issue was dealt with the code was uploaded to Github.

A secondary issue which arose through GitHub was that the group was very unfamiliar with the process, meaning many human-error based issues arose such as committing to different forks, committing to the wrong GitHub

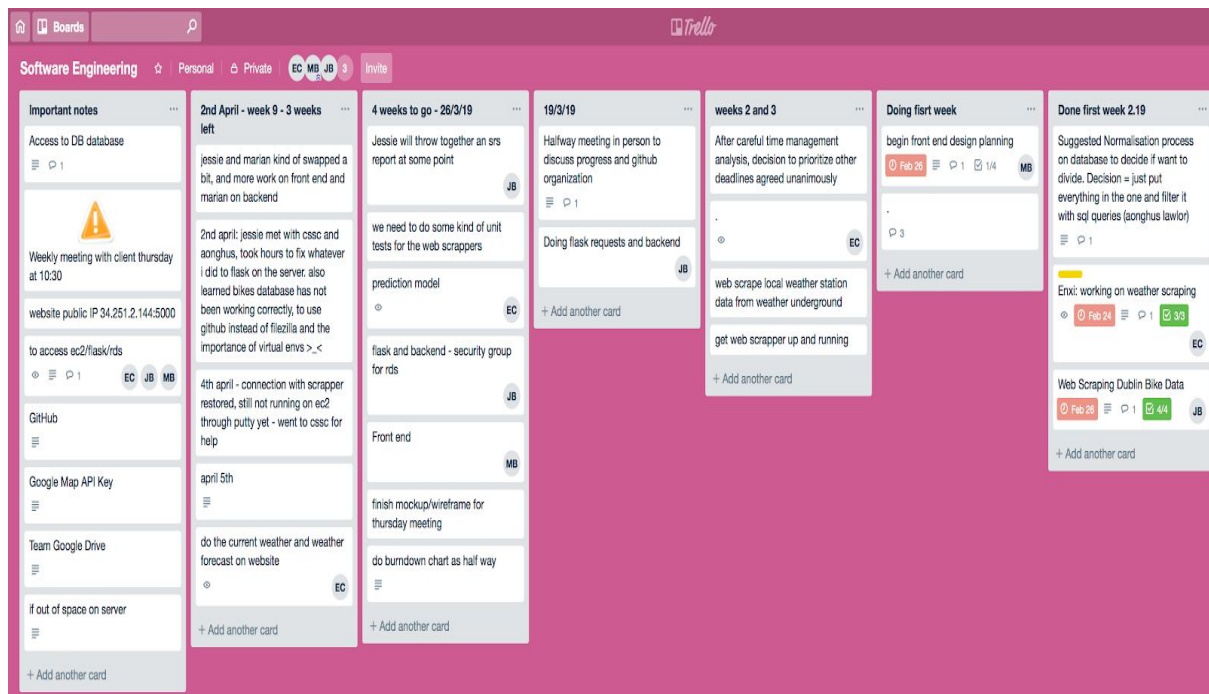
account, and merge conflicts on both local and server sides. These were all eventually resolved, and much was learned in the process.

AWS Instance issue

After uploading all flask and pickle files to the teams AWS instance we began testing the webpage. An issue was noted with the prediction model returning only '0 bikes available' for each station, at each time of day. The final working, locally running flask code was checked against the uploaded code and we found no differences. Upon further investigation we found that the issue was due to us reaching the storage limits on our AWS instance. Due to time constraints we decided that the best course of action was to change from an AWS t2.micro instance to a t2.large instance. The increased storage capacity allowed the remaining pickle files to be uploaded and the issue was resolved.

Sprint organization:

As discussed in the project submission the team followed a chaotic scrum process. As we did not have access to accurate documentation, below is an approximation of the teams process laid out in the scrum organizational style. Followed by a screenshot of the Trello from which the sprint estimates were summarized.



Scrum timeline:

| W4 | W5 | W6 | W7 | Study Break | | W8 | W9 | W10 | W11 |
|----------|----|----------|----|-------------|--|----------|----|----------|-----|
| | | | | | | | | | |
| Sprint 1 | | | | | | | | | |
| | | Sprint 2 | | | | | | | |
| | | | | | | Sprint 3 | | | |
| | | | | | | | | Sprint 4 | |
| | | | | | | | | | |

Sprint 1 Burndown

Time for each sprint in hours:

Write enhanced project brief - 1

Web scraper for weather data - 5

Web scraper for bike data - 5

Set up MySQL database - 2

Develop basic flask framework - 5

Set up google doc - 0.5

Set up Trello - 0.5

Set up Whatsapp group - 0.5

Set up Github - 0.5

Total time for sprint = 20 hours

Sprint 1 Burn down chart



Please note, the dip in the chart relates to issues with the web scraper.

Sprint 2 Burndown (extended seven days due to study break)

Time for each sprint in hours:

Display google maps on flask framework - 5

Return Json data from database to flask app - 8

Display map markers on google map - 5

Display weather info on flask app - 5

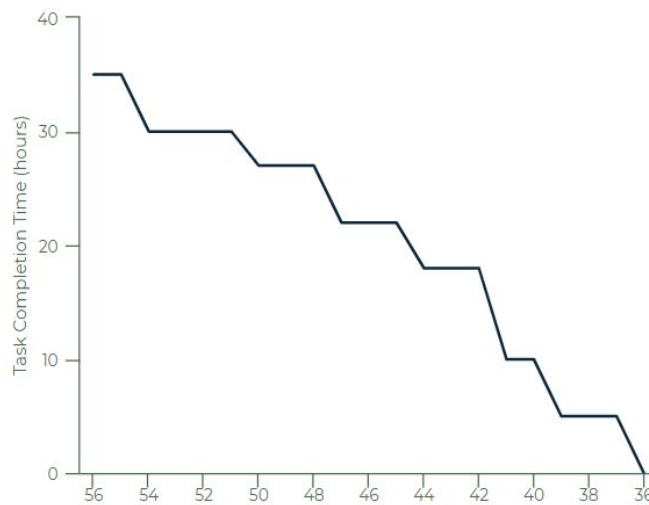
Display station info on flask app - 5

Develop templates for flask web app - 3

Combine each users code - 4

Total time for sprint in hours = 35 hours

Sprint 2 Burn down chart



Sprint 3 Burndown (extended seven days due to study break)

Time for each sprint in hours:

Display bike lanes on google map - 4

Display traffic information on google map - 4

Create route planner - 8 (subsequently abandoned)

Create FAQ page - 3

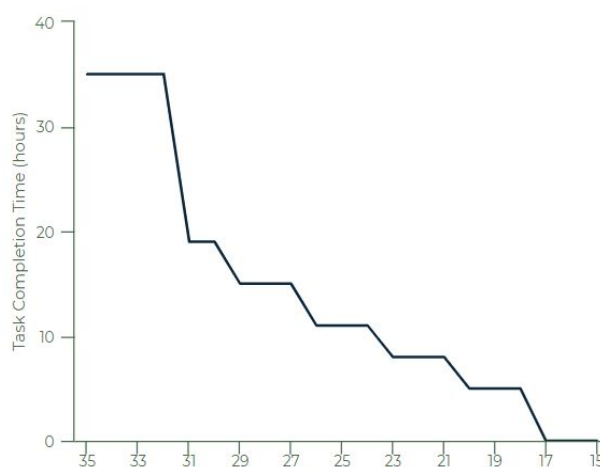
Create alert system -5

Create contact us page - 3

Develop prediction model - 16

Total time for sprint in hours = 35 hours

Sprint 3 Burn down chart



Sprint 4 Burndown

Time for each sprint in hours:

Ensure each item displayed on application is working - 3

Get final application running on AWS - 16

Finalize CSS - 6

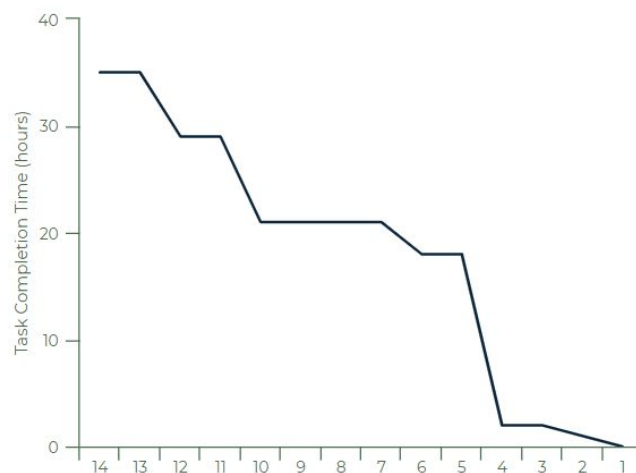
Final submission documentation - 8

Gathering of all information for submission - 1

Final read through and testing -1

Time for each sprint in hours: 35 hours

Sprint 4 Burn down chart



Final thoughts and project reflection

Upon starting the assignment the team was ambitious. We set about planning our ideal version of the web application to not only return the project deliverables, but to also allow an improved user experience over the original dublin bikes website. Given the time available and the scrum planning process, we were confident that the extra features we devised could be implemented.

Having discussed what the team would do differently if we had the opportunity, we settled on the following:

Follow the Scrum process:

At first, the scrum process was successfully implemented. Daily scrums were taking place with a rotating scrum master and tasks were being divided among team members. It wasn't until the team started encountering issues that the scrum process became much looser and eventually fell by the wayside.

The team's organisation suffered from the lack of detailed shared scrum notes and it was harder to split and assign tasks. Eventually it was not up to the team to collectively discuss and assign tasks but rather to the individual to decide what they would like to do. The lack of accountability regarding when tasks should be finished also caused the final project to suffer.

Had the team followed the scrum process, the final product would have not only been more successful but it would have been less stressful.

Focus on the deliverables:

In the beginning of our project we decided to create an enhanced project brief to flesh out ideas and provide us a basic framework. We were ambitious as a team and decided to implement extra features without taking the time to consider if these were possible to implement in the time we had.

If we had an opportunity to redo the project we decided that we would focus on achieving the project deliverables. Only when these deliverables were completed would we then look to add extra features. Our approach caused the application to suffer as we tried to implement everything at once. When we had to prioritise tasks to finish the application, several half

finished features had to be abandoned. The time used to complete this half finished tasks could have been used to create a higher finish on the final project, or indeed to finish some easier to implement features.

Get help on issues before they become a big problem:

During the assignment we encountered a number of issues. Some of these issues were relatively small and easy to fix such as the initial problems we had when connecting to the database. Other issues were major and required help and intervention from the computer science support center, demonstrators and Aongous. By not using the scrum process issues were allowed to carry on for far longer than they should have. We did not have the dialogue that the daily scrum allowed and some issues were not discussed until they became large problems. If attempting this project again we would ensure that all issues were discussed in the daily scrum and that help would be sought if we could not resolve the issues as a team.