UNIVERSITY OF LEEDS

Website Resource for Flooding Data and Information in West Yorkshire

Michael Stephen Stead

Submitted in accordance with the requirements for the degree of
BSc Computer Science

2015/2016

The candidate confirms that the following have been submitted:

Items	**Format**	**Recipient(s) and Date**
Report (Hard Copy)	Report	SSO (11/05/16)
Report (Digital)	Report	VLE (11/05/16)
Software Code	Repository URL	Supervisor, assessor (10/05/16)
Demonstration	In Person Demo	Supervisor, assessor (10//05/16)

Type of Project: Exploratory Software (ESw)

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student)_____

Summary

This report details the investigation and development of a website resource which uses various types of public data, sourced from social media and government data-sets to map, analyse, and provide live updates, of the extent of the flooding effecting West Yorkshire. These data sources are visualised on the website using various different tools and methods.

Acknowledgements

I would first like to thank my project supervisor Nick Efford for the support and feedback he provided during our weekly meetings. I would also like to thank my assessor Brandon Bennett, for the valuable feedback he provided at the progress meeting. Also, thanks to the Open Data Institute and the people present at the Flood Hack meetings, for providing the initial idea and the extra data-sets that was used in this project. Lastly, I would like to thank my family for their support and encouragement throughout this project and my time at university.

Contents

Summary....iii
Acknowledgements....iv
Chapter 1: Introduction....1
1.1 Project Aim....1
1.2 Problem Statement....1
1.3 Minimum Requirements....2
1.4 Methodology....2
1.5 Schedule....3
1.6 Relevance to Degree....3
Chapter 2: Background Research....4
2.1 Peta Jakarta....4
2.2 Data Sources....5
2.2.1 Flickr....5
2.2.2 Twitter....7
2.2.3 Instagram....8
2.2.4 Government Environmental Data....9
2.2.5 Leeds ODI Flood Hack....9
2.3 Cloud Vision APIs....10
Chapter 3: Development Tools....11
3.1 Django Web Framework....11
3.1.1 Django Models....11
3.1.2 Django Views....12
3.1.3 Django Templates....12
3.2 Bootstrap....12
3.3 Version Control....13
3.4 Programming Languages....13
Chapter 4: Design and Implementation....14
4.1 Website Setup And Initial Flickr Script Development....14
4.1.1 Basic Django Website and Server Setup....14
4.1.2 Initial Flickr Script....14
4.1.2.1 Request Builder Function....15
4.1.2.2 XML Parser Function....15
4.1.2.3 Photo URL Builder Function....15
4.1.2.4 Get Location Function....16
4.1.2.5 Date Taken Function....16
4.1.2.6 Get Locality Function....16
4.1.2.7 Photo Builder Function....16
4.1.2.8 Testing of FlickrInitial Script....17
4.1.3 Basic Homepage and Bootstrap Implementation....18
4.1.3.1 Photo Model and Database Population....19
4.1.3.2 Displaying Photo Data On Homepage....19
4.2 Map Page, DB Class, and Flickr Daily Script Development....20
4.2.1 Map Page....20
4.2.1.1 Map Page Development....21
4.2.1.2 Map Page Evaluation....21
4.2.2 Flickr Daily Script....22
4.2.2.1 Req Builder Function Change....22
4.2.2.2 Photo Builder Function Change....22

4.2.3 DB Class Development....22
4.2.4 Flickr Daily and DB Class Tests....22
4.2.5 Updated Project Schedule....23
4.3 Timeline Page, Instagram and Flood Area Script Development....23
4.3.1 Timeline Page....23
4.3.1.1 Timeline Page Development....24
4.3.1.2 Implementation of Navigation Bar....26
4.3.2 Instagram Script....26
4.3.2.1 Search Location Function....26
4.3.2.2 Updated Photo Model and Flickr Scripts....27
4.3.2.3 Instagram Script Test....27
4.3.2.4 Adding Instagram Data....28
4.3.3 Flood Areas Script....28
4.3.3.1 Search Area Function....28
4.3.3.2 Get Flood Location Function....29
4.3.3.3 Run Instagram Function....29
4.3.3.4 Testing of Flood Areas Script....29
4.4 Twitter Data, Photo Ranking and Map Clustering....30
4.4.1 Twitter Data Implementation....30
4.4.1.1 Tweets Model and DB Class Update....30
4.4.1.2 Twitter Script....31
4.4.1.3 Updating Views.py....32
4.4.1.4 Updating the Templates....33
4.4.2 Photo Ranking....33
4.4.2.1 Updating the Photo Model and Scripts....34
4.4.2.2 Adding Buttons and the Score Field....34
4.4.2.3 JavaScript Button Functions....35
4.4.2.4 Up and Down, View Functions....35
4.4.3 Map Clustering and Unique Icons....35
4.5 Implementing River Level Data and Updating the Home Page....37
4.5.1 River Level Data....37
4.5.1.1 River Level Model and DB Class Update....37
4.5.1.2 River Level Script....38
4.5.1.3 Updating Views.py for River Level Data....39
4.5.1.4 Timeline Template River Level Display....40
4.5.1.5 Rain Level Data....41
4.5.2 Home Page Update....41
Chapter 5: Evaluation....43
5.1 User Evaluation....43
5.1.1 Method and Procedure....43
5.1.2 Results....43
5.1.3 User Evaluation Conclusion....44
5.2 Self Evaluation....45
5.2.1 Website Performance Evaluation....45
5.2.1.1 Home Page Performance....45
5.2.1.2 Map Page Performance....46
5.2.1.3 Timeline Page Performance....46
5.2.2 Website Display Evaluation....47
5.2.2.1 Home Page Display....47
5.2.2.2 Map Page Display....47

5.2.2.3 Timeline Page Display....47
5.2.3 Meeting The Requirements....48
5.2.4 Possible Extensions....49
5.3 Conclusion and Reflection....49
References....50
Appendix A: External Materials....52
Appendix B: Ethical Issues....53
Appendix C: Online Survey....54

Chapter 1: Introduction

Due to the increased flooding risk throughout West Yorkshire in recent years the Leeds node of the ODI (Open Data Institute) decided to hold a "flood hack" to bring together professionals from various industries and backgrounds to see if there were any possible ways in which technology and open data could help with this crisis. The idea for this project was inspired by a web-based resource called Peta Jakarta[1] which uses geo-tagged Twitter posts from the citizens of Jakarta alongside local weather data to map and provide live updates of the floods throughout Jakarta.

1.1 Project Aim

The aim of this project is to create a web-based resource that visualises various types of publicly available data related to flooding, to provide a new understanding and overview of the flooding experienced throughout West Yorkshire. One example of this could be geo-tagged photographs plotted to a map to provide an accurate view of the extent and state of the effects flooding has had throughout West Yorkshire. Another example is historic photographs of any flooding in West Yorkshire sourced from social media plotted to a timeline, which could be used to see the severity of floods in previous years. This website will also aim to be semi-autonomous and have the ability to update with any new flood related data as it becomes available. This would provide local citizens with up-to-date information about any current floods, such as recent flood warnings and any photographs related to the flooding; so the local citizens know which areas are most affected by any flooding and can stay informed. Another potential use of this website is using the historic data gathered to see which areas of West Yorkshire suffer the most from the floods.

1.2 Problem Statement

This project poses a range of different challenges. Research into the types of useful public data available will have to be completed; for example, investigating different types of photographic data, their availability and their sources. Once some promising data sources have been found, any useful data will need to be extracted and processed. Various programming scripts will have to be created to complete the task. Another problem is to decide how the various types of data will be visualised or represented on the website. One preliminary idea is to map geo-tagged photographs sourced from social media. Lastly, the development of the website will be a large part of the project, such as deciding which frameworks and programming languages should be used to create the website. Since the website will also ideally require minimal maintenance an easy solution to removing irrelevant

photographs will also have to be devised.

1.3 Minimum Requirements

- Investigate publicly available photographic, social media and environmental data.
- Investigate similar existing solutions to the problem.
- Program scripts to extract and process relevant flooding data from the various data sources.
- Implement different data visualisation tools and methods.
- Investigate and if possible, implement a cloud-based vision API.
- Design and develop a website to display and organise the visualised flooding data.

1.4 Methodology

The first step of the project is to complete background research into any existing solutions, followed by research of the available data sources and their APIs. After this background research has been completed, the development of the website and the scripts to extract and process any relevant data from the various sources would commence.

In the initial plan I had settled on using a single person scrum method for the development of the software for this project. After completing some initial research, however, I decided to switch to an iterative waterfall method. This is because the original plan was to break the project up into bi-weekly sprints where the development of the scripts and website would be worked on simultaneously. After completing the background research I ended up settling on an iterative waterfall method, as this way I could improve the website with every iteration and incorporate different data sources in different iterations. This method was chosen over the bi-weekly sprints because the sprints could result in having none of the scripts for the data sources fully functional by the end of the project. The iterative waterfall would at least guarantee that some of the data sources would be incorporated into the website by the end of the project. This also allowed some extra flexibility; when new data sources are discovered they can be implemented as an extra iteration.

1.5 Schedule

This section outlines the project's schedule in the form of Gantt charts as shown below.

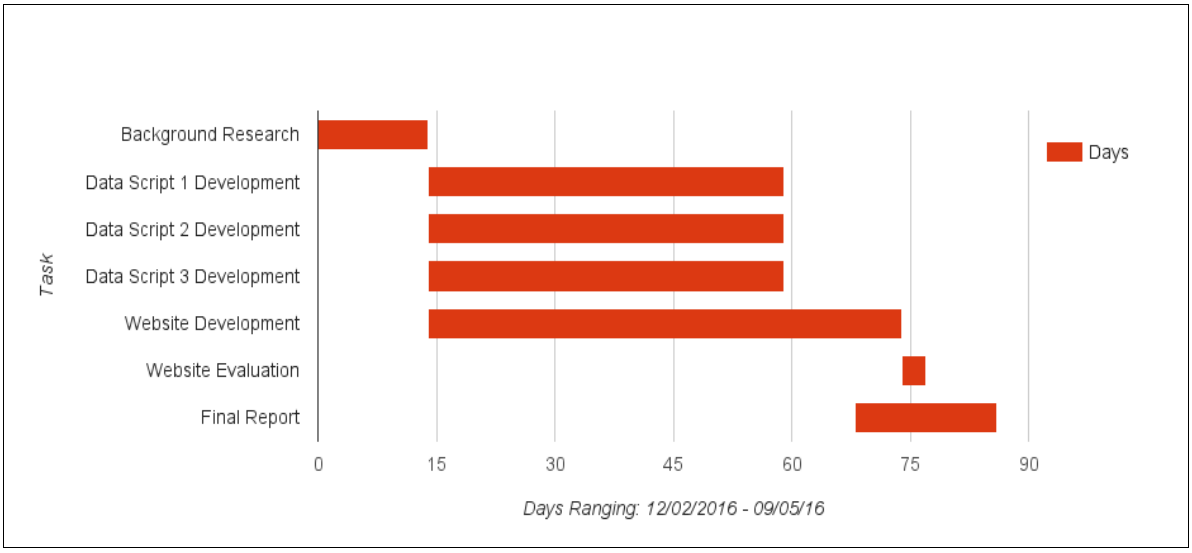


Fig 1. Original Gantt Chart (Bi-weekly Sprints)

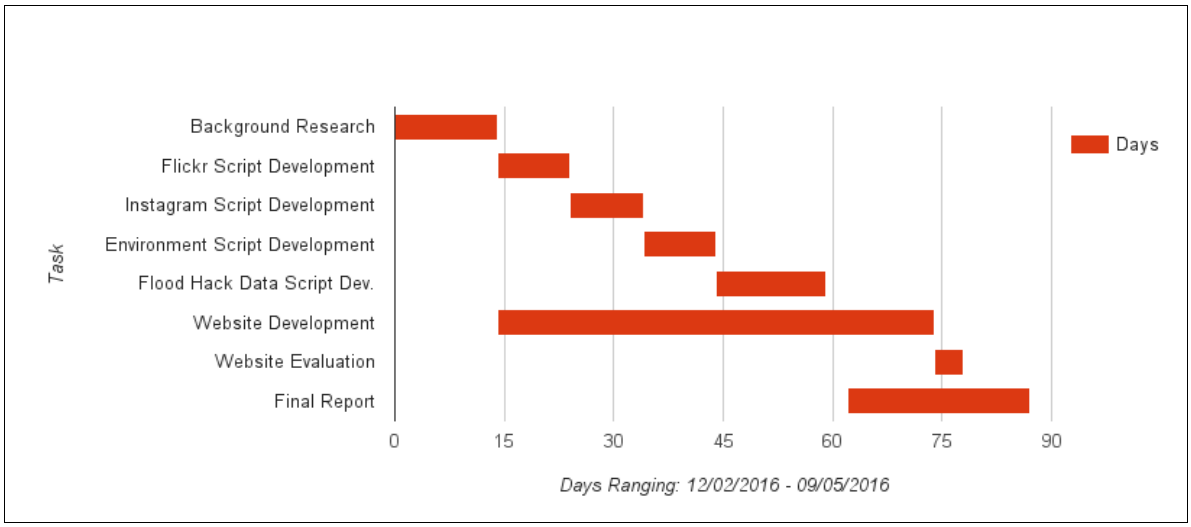


Fig 2. Updated Iterative Waterfall Gantt Chart

1.6 Relevance to Degree

The Python programming skills learnt in the first year Core Programming module will be heavily utilised throughout this project. Also the project management and web development experience gained from the second year Software Development module will be useful when planning the project and developing the website. Lastly, the knowledge of APIs gained from the third year Distributed Systems module will also come into great use when developing the scripts, which will use various APIs to gather data.

Chapter 2: Background Research

This chapter will feature the research done throughout the project, which is all the research completed before starting development and any further research completed during development. First, I will analyse an existing solution to a similar problem, then I will discuss my research of potential data sources and the availability and type of data they can provide. Finally, I will consider possible ways in which the quality of the data gathered could be evaluated.

2.1 Peta Jakarta

Peta Jakarta[1] is a social media and website resource for flooding in Jakarta. It is created by the SMART infrastructure facility at the University of Wollongong, Australia, alongside the Jakarta Emergency Management Agency. Jakarta is an area which is heavily affected by flooding during the monsoon season. This project aimed to aide the citizens of Jakarta by presenting any recent warnings and data on flooding throughout the area, and to support the creation of information for flood assessment, response, and management in real-time.[2] This was accomplished by providing an open source flood map, APIs for their data resources, and live updates through their twitter account on any flooding and flood risks throughout Jakarta.

According to the research gathered by the SMART team, Jakarta has one of the highest concentrations of Twitter users in the world.[2] It is for this reason that they decided to use Twitter over other social media platforms to collect a lot of their data. The Twitter data collection was completed by monitoring all geo-tagged tweets posted in Jakarta during a flood event containing the word “banjir” meaning flood, they then proceeded to add these tweets to a database and plot them to the map. However, due to the sheer volume of tweets collected it was an impossible task for a small team to review and extract only relevant tweets. They managed to solve this by crowd-sourcing the problem. They first listed all of the tweets saved in the database as “unconfirmed tweets”. After this they sent “invitations” to the owners of these tweets; these “invitations” asked the owners to verify that their tweet was in fact related to the floods by tweeting “banjir” to the Peta Jakarta Twitter account. These “confirmed” tweets were then cross-referenced with data provided by Jakarta's Emergency Management Agency. A full example of this process can be seen in Fig 3.

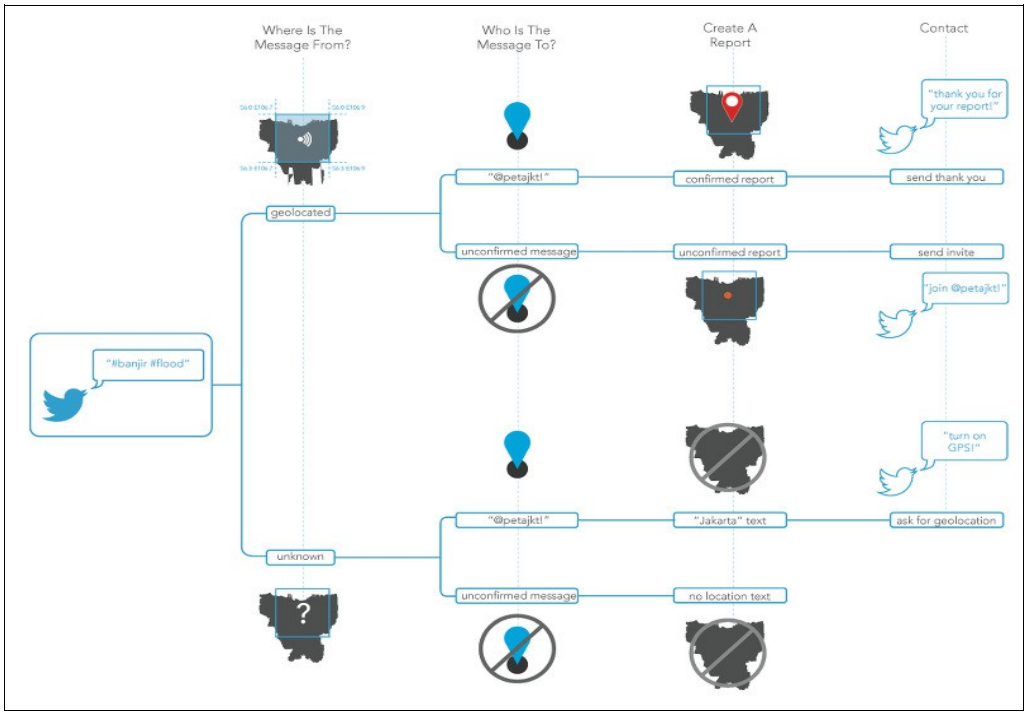


Fig. 3 Peta Jakarta data collection process

2.2 Data Sources

Open data is any data source which is free to use and share[3]. In this section I will provide a broad description of the different publicly available data resources I used and researched throughout the project. I will also provide an analysis of the availability and type of data these resources provided. The full details on how the data was used and processed can be found in Chapter 4 entitled 'Design and Implementation'.

2.2.1 Flickr

Flickr is an online application and community that allows users to upload and share their photographs. Its content ranges from professionally images to amateur photographs taken with a camera phone. Due to their exceptionally large user-base (112 million people)[4] and the staggering amount of photographs hosted on the site (over 10 billion)[5], Flickr proved to be a useful social media resource for gathering photographic data on any flooding throughout West Yorkshire.

Flickr can be considered a partial open data source because it provides the option for users to set their photos as either public or private. If an image is made public by its owner on Flickr then the photo is free to use, so long as proper credit is given to the owner. This can be done via a link back to the owner's page.[6]

Flickr also provides numerous APIs ranging from photo management to extensive search APIs.

Some of these APIs require authentication, however, since this project will be using only publicly available images authentication will not be needed. The project did however need to be registered with Flickr to obtain an API key. This is because Flickr sets a limit of 3600 queries per hour for a single application[7] and the API key is used to track these queries. However, due to the narrow focus of this project it would be highly unlikely that this limit would ever be reached.

For an initial evaluation of the quality and quantity of photographic data Flickr could provide I created a search query which uses Flickr's search API[8]. The search API returns an XML response of all images related to the parameters specified in the search query, for example after setting parameters of latitude: 53.7996, longitude: -1.5491, radius: 20, tags: flood, and has_geo: 1, this query returned 428 photographs which were within 20km of Leeds and were geo-tagged, whose tags include the word “flood”, proving Flickr to be a viable option as a source of photographic data.

```
<photo id="23695333204" owner="128195793@N04" secret="846f963d7c" server="1458" farm="2" title="Flooding near Royal Armouries, Leeds" ispublic="1
isfriend="0" isfamily="0"/>
<photo id="24169707592" owner="128195793@N04" secret="b647eb6065" server="1481" farm="2" title="Flooding near Thwaite Mills, Leeds" ispublic="1"
isfriend="0" isfamily="0"/>
```

Fig. 4 Example XML response from flickr search API

From the data returned by the search API, other Flickr APIs can then be used to obtain more information about the image, such as the getLocation[9] API to retrieve the latitude and longitude of where the photo was taken, or the getInfo[10] API to find out the day on which it was taken. Both of these APIs require the “id” and/or “secret” values returned by the search API seen in Fig 4. After this, in order to check the photographs returned by the search API were of good quality, image URLs can be compiled for a few of the photographs as per Flickr's guidelines[11].

Fig. 5 Photo related to “Flooding near Royal Armouries, Leeds”

2.2.2 Twitter

After seeing the success Peta Jakarta[1] had with using Twitter as a data resource some investigation was needed to determine whether it could prove to be just as useful of a data source for West Yorkshire.

To get an initial idea of the data that could be provided, Twitter's advanced search-tool[12] was used to find any tweets relating to the keywords “flood” or “flooding” in Leeds and its surrounding areas between a range of dates. These preliminary search queries returned a variety of tweets including but not limited to, flood-alerts from official sources, tweets from local citizens about flooded areas, news articles related to the flooding, and photographs or links to Instagram photographs of the floods. However, not all of these tweets were geo-tagged and most of the photographic data was provided via links to Instagram. This prompted research into Instagram as a potential data source, since the main scope of this project was to obtain geo-tagged photographic data. Further research into the APIs provided by Twitter was completed because although it was not a great source of photographs the other data provided could be useful in the future of the project.

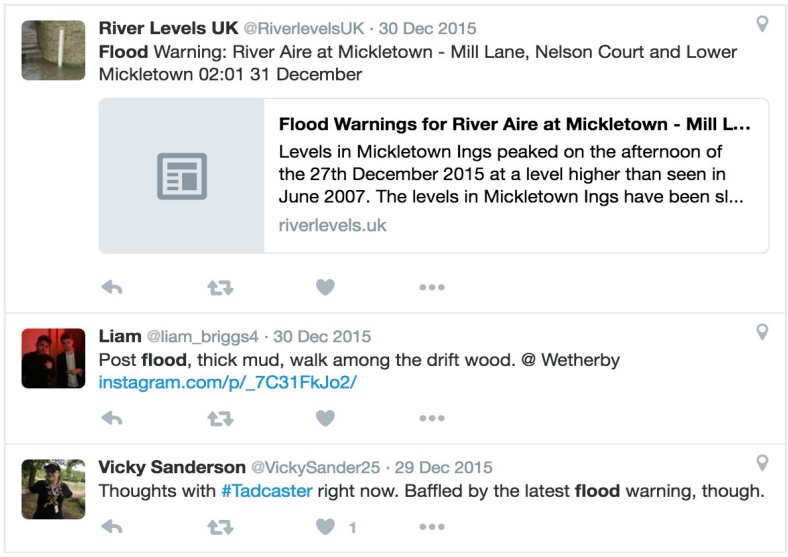


Fig. 6 Twitter advanced search “flooding” result

There is a large range of APIs provided by Twitter but for the scope of this project only research into the Twitter search APIs was completed. Every one of Twitter's APIs requires oAuth authentication to use; even the APIs that only use public data require an API key. There are also rate limits imposed on each of the APIs[13]. These limits vary depending on whether the user or the application is authorised; authorised applications are typically able to make more requests in the 15 minute rate limit window Twitter has set. The rate limit for the main search API is 450 requests/15 minutes for an application.

The main search API[14] provided by Twitter allows the user to build a search query using specified parameters and returns a JSON response; all parameters except the query parameter are optional. For this project the only parameters that would be needed would be “q”, “geocode”, “until” and “result_type”. The “q” parameter specifies the query, for example “flood” and “geocode” specifies the latitude, longitude and radius of the area that is to be searched. The “until” parameter specifies the maximum date of the tweets and searches for any tweets in the past seven days before this date. Lastly, “result_type” is used to specify whether to search for either recent tweets, popular tweets or a mix of both. These parameters can be used to obtain either historical tweets relating to flooding, or, by excluding the “until” parameter, very recent tweets.

2.2.3 Instagram

In the original plan of the project Instagram was overlooked as a potential data source. However the Twitter research revealed that most of the photographs present on Twitter were hosted by Instagram, and this is what prompted the research into the Instagram APIs (or 'endpoints', as Instagram describes them), and the data they could provide.

In comparison to Flickr, Instagram appears to be a much larger resource for photographic data. It has over 400 million active users, 40 billion photographs shared, and an average of 80 million images uploaded per day[15]. This is significantly larger than Flickr's 112 million active users and 10 billion hosted images[4]. Unfortunately, due to changes in Instagram's API policy in November 2015[16] you can no longer search for historic media. This meant that the bulk of historic flooding photographs would have to be sourced from Flickr. However, Instagram is demonstrably an excellent source of photographic data that could be used for future flood events.

Instagram's APIs have reasonable rate limits, where an application in “sandbox” mode has a rate limit of 500 requests per hour but a “live” application has a rate limit of 5000 requests per hour[17]. Some particular endpoints or APIs have stricter rate limits but these APIs are not needed for this project. Before the Instagram APIs can be used the application needs to be registered, but even when registered the APIs available for use are limited. In order to use all of the APIs, the application needs to apply for full access, which is accepted or denied by Instagram depending on the application. This project's application was approved by Instagram for full access.

The API that would be the most applicable for this project would be the media search endpoint[18] which returns data in a JSON format. This API provides the ability to search for all recent media in a certain location. This is possible by setting the latitude, longitude and distance parameters where the maximum distance is 5 km. This means Instagram's search distance is a lot more localised when compared to the maximum of 20 miles that Flickr allows. This particular endpoint also does not

provide the ability to search by tag or query, meaning that the Instagram posts will have to be processed to extract any posts relating to flooding.

2.2.4 Government Environmental Data

The UK government provides a lot of flood-related APIs, all of which could be excellent sources of data. There is the Flood-Warnings API[19] which allows the user to search for any flood warnings within a certain distance of a location, based on search parameters of the location's latitude and longitude, and the radius of the search area. This API has a JSON response and provides useful information such as a list of flood warnings and their severity, the time the warning was issued, the location that the flood warning applies to, and a description about the flood or flood warning.
Another potentially useful API is the Readings API[20]. This API provides recent readings of water flow or river levels for a particular station; all that is required is the station's id. Although this API only provides recent data it is possible to retrieve historical station data. Every station's readings are archived and can be accessed as csv files, however these csv files are categorised by date and are not station specific meaning that the data will have to be processed to find the relevant data relating to particular stations.

2.2.5 Leeds ODI Flood Hack

During the project the Leeds node of the Open Data Institute held a Flood Hack event. This event was created as a response to the recent flooding experienced in the West Yorkshire area. Many local council services, businesses and data experts came together to share data and explore ways in which this data could be used to help the citizens of West Yorkshire.
One potentially useful data source was a Twitter data set of all tweets made throughout Yorkshire starting at the December 2015 flood event, this data set was provided by Andy Burgin. The data-set consists of over 16000 geo-tagged tweets, 130000 city/place level tweets and 250000 tweets relating to Leeds. Due to the size and variance in the tweets, some processing will be required to extract tweets related to the floods. These tweets could provide essential information into how the citizens reacted to the floods as well as seeing which areas were heavily affected using the geo-tagged tweets. Another possibly useful data set was the rain gauge data from the Leeds Pottery Fields sensor dating all the way back to early 2013. This data set was provided by the Leeds City Council and made available through Leeds Datamill.

2.3 Cloud Vision APIs

An evaluation of possible vision APIs was completed to see if they could be used to help process the photographs used on the website. The primary goal was to see if the vision APIs could be used to detect images of flooding or rather detect any images which are not of flooding. If possible this could be implemented as an automated way of removing any irrelevant or inappropriate photographs from the website, removing the need for manual evaluation of photographs.

Microsoft provides a computer vision API via there cognitive services. A small evaluation of their Analyse Image API[21] was carried out on various images both related and unrelated to flooding sourced from Flickr search queries for flooding photos. The Analyse Image API has 86 different categories[22] a photo can be classified as, such as, “outdoors” or “person”; the aim was to see how various flooding photos would be classified.

Google's Cloud Vision API[23] was also investigated. This API also provides a powerful image analysis tool that detects what objects are present in the image, such as a flower or an animal. Google does not list all of the possible “labels” an image can have but claims that there are thousands of possible objects that can be detected, which suggests it has broader categorisation compared to Microsoft's vision API.

After some initial testing and manual inspection of Microsoft's API and the flooding photographs sourced from Flickr, it became clear that the content of the flooding images varied to greatly. This high variance between photographs makes it extremely difficult to find an over-arching categorisation for photographs of floods. However, although these cloud vision APIs could not be used to remove spam photographs they could be used to remove any malicious or illegal photographs if the ability to upload photos is added to the website. This will not be needed for any photographs sourced from social media as they will have already been screened by their respective sources.

Chapter 3: Development Tools

This section of the report outlines the development tools used throughout the project, which includes the frameworks and tools that were used to develop the website and the scripts that were used to populate the website with the flood data.

3.1 Django Web Framework

As described by the Django website, "Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design."[24] This is one of the main reasons Django was chosen as the web framework with which to create the project's website. Due to its high-level nature it allowed for rapid website development, so more time could be dedicated to the development of the Python scripts that would be used to populate the website with data. Django is also a widely used web framework with a wealth of documentation and tutorials available online. This also sped up the development of the website, as the process of learning the framework could be streamlined. The three core fundamentals of Django is its models, views and templates. It is these three core technologies that define it as a high-level framework. A model of how these technologies interact can be seen in Fig 8. Django also provides a test server and database, this means that any development changes can be checked before adding the changes to the production server. Lastly, Django websites provide an Admin page that lists the data held in the database, and the database tables. This provides an easy way to manually add and remove images, to and from the database.

3.1.1 Django Models

A Django model[25] is a Python class used to describe a data object. A model is created with specific fields and behaviours which defines a particular set of data. This model is then used to create a database table to store this data; each field in the model relates to a column in the database table. An example of a model and its corresponding database table can be seen in Fig 7.

```
class RiverLevel(models.Model):
    date_taken = models.DateTimeField('date_taken')
    river_level = models.TextField()
    place = models.TextField()
    lat = models.TextField()
    lng = models.TextField()
```

id	date_taken	river_level	place	lat	lng
224643	2015-11-13 19:00:00-05	0.942	Leeds	53.794197	-1.535433
224644	2015-11-13 19:00:00-05	1.870	castleford	53.731002	-1.358177
224645	2015-11-13 19:00:00-05	1.919	Kirkstall	53.819613	-1.605025
224646	2015-11-13 20:00:00-05	0.943	Leeds	53.794197	-1.535433

Fig. 7 River Level Model and Corresponding Database Table

These models remove the need to query the database with SQL. Django QuerySet functions can be used to retrieve data from a table, usually with just one line of Python code. There are also similar

methods available for adding new data to the database without the use of SQL. Each function handles the execution of different SQL commands, which provides a level of abstraction from the database. One disadvantage of using a model over regular SQL is that the user is limited to the QuerySets and functions provided by Django. If a particularly complex or unique database command is needed it will have to be constructed in SQL.

3.1.2 Django Views

A Django view is a Python function that handles web requests and returns a web response[26]. Views are typically used to handle web-page requests. For example, the index view will handle the index page request and then return and render the index page template in the browser window as a response. Any database data required by a template is usually queried from within a view function. The requested QuerySet can then be rendered alongside the template. Each view has a URL mapped to it that is used for requests.

3.1.3 Django Templates

Django templates[27] are HTML files which are used to generate the dynamic web pages displayed on a website. Templates are similar to regular HTML files used for web pages; the difference is how dynamic content is generated. The Django template language is used to handle any dynamic content, such as processing model objects and sets that are to be displayed on a web page. This template language is similar to a programming language; any template language present in the template is executed by Django before passing the template to the web browser.

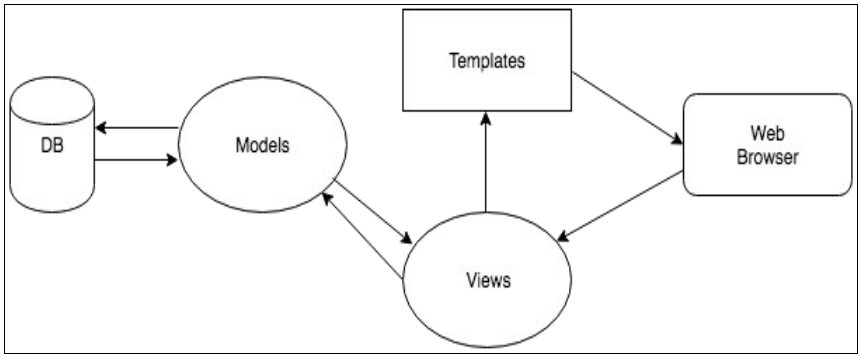


Fig. 8 Django web page request and response

3.2 Bootstrap

Bootstrap is a popular HTML, CSS, and JavaScript based front-end web framework supported by most desktop and mobile browsers[28]. One of the main reasons Bootstrap was chosen is because it is

a “mobile first” framework[29]. This means websites built using this framework are optimised for mobile device displays. This is an important feature as according to Ofcom research “smartphones are now the most important device for internet access”. The report goes on to state that 33% of internet users use a smartphone to access the web compared to 30% for laptops[30].

Another reason Bootstrap was chosen to create the front-end of the website is because of the number of free bootstrap templates available online. This greatly sped up the development of the website, as less time would need to be spent on designing web pages with an acceptable user interface. For example, the timeline page on the website was created by refactoring a bootstrap template rather than building the timeline from scratch. Other frameworks were investigated, such as, Yahoo's Pure framework, but these frameworks were less popular than Bootstrap, which meant that they had less documentation and templates available.

3.3 Version Control

Git was used to handle the version control of the project. A private GitHub repository was created to host the project, and this repository was then connected to the machine hosting the website. This meant any local changes made to the project could be sent to the GitHub repository and then any updates could be pulled by the host to update the website. The GitHub repository provides back ups of the project, along with historical records of the project's code. Any changes made to the project could therefore be reverted, if the changes caused any problems.

3.4 Programming Languages

Python was chosen to develop the scripts used to populate the website. This is because Django is Python based, so using Python for the scripts would allow for easier integration with the Django project. Python is also a simplistic, but powerful, programming language. It also has a variety of libraries (such as the requests library) that would be useful in the development of the scripts.

Chapter 4: Design and Implementation

This section outlines how the website and the scripts used to populate the website with data were designed and implemented. It consists of the five iterations to complete the project and details what was completed in each iteration.

4.1 Website Setup And Initial Flickr Script Development

The main focus of this first iteration was the initial website creation and setup using the Django framework and a digital ocean server. This website continued to be developed over the project's timeline. The second goal of this iteration was to develop an initial Flickr Script which would be used to gather the initial photo data used on the website. The final goal after this was to add Bootstrap to the website and create a basic homepage that would display the gathered Flickr data.

4.1.1 Basic Django Website and Server Setup

For the initial website, Digital Ocean's "one click setup" was used to create an initial Django website and its server. This "one click setup" creates an extremely basic Django website which runs on an Nginx and Gunicorn server with a PostgreSQL database, all hosted on a Ubuntu machine[31]. An example of this initial website can be seen in fig 9. The machine hosting the website was then connected to github; the initial Django project folders and files where then pushed to the repository. This allowed for the development of the website to be completed on a local machine connected to the repository rather than on the machine hosting the website. This was done by pushing any changes to the project on the local machine to the repository and then pulling the changes on to the machine hosting the website.

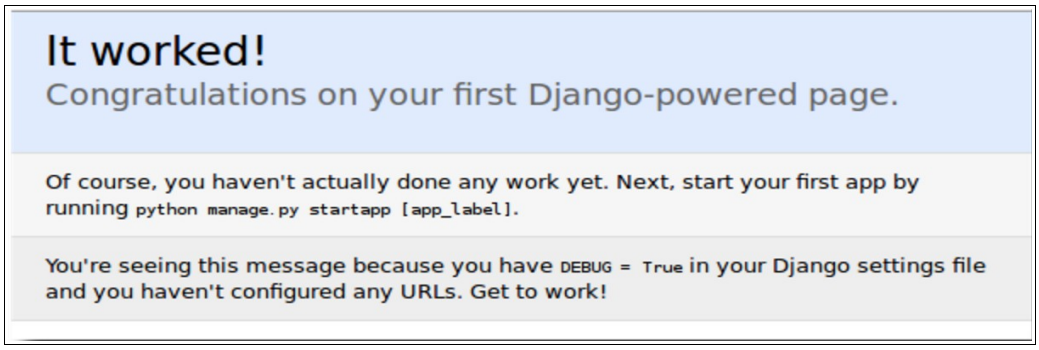


Fig 9. Example of Basic Django Home Page

4.1.2 Initial Flickr Script

This initial Flickr Script (FlickrInitial.py) was used to gather any photographs of flooding within a

20km radius of Leeds. It made use of various APIs provided by Flickr to request and gather the photographic data and then save this data to a text file later used to add the data to the website. A Scripts folder was also created in this process which would house all of the scripts created and used. The first execution of this script returned over 400 photos of flooding. For an example of how the below functions interact see Fig. 11.

4.1.2.1 Request Builder Function

The *reqBuilder* function takes 4 parameters that are used to construct a URL as follows:

- Tag – Search for photographs based on this tag, for example the tag "flood".
- Lat – Set the latitude of the search location, for Leeds this is "53.7996".
- Lon – Set the longitude of the search location, for Leeds this is "-1.5491".
- Rad – Set the radius of the search area, maximum value possible is 20km.

This function then creates a URL based on the format specified by the flickr.photos.search API using these parameters and the projects API key. The "has_geo" parameter of the URL is also set to "1", this is so the only photographs returned are the ones which have geographic data. The parameters specified above are just a few of the parameters the flickr.photos.search API accepts. A full list of the possible parameters the flickr.photos.search API accepts can be found online[8]. The Python requests library[32] is then used to request the URL and get the page(s) of XML data related to this request. These XML responses are then saved to a list and are then returned to be processed by the *xmlParser* function.

4.1.2.2 XML Parser Function

This function takes the "requestArray" produced by the *reqBuilder* function as its parameter. It then cycles through the requests in the list and uses the ElementTree library to process the XML pages to retrieve each individual XML photo object and converts each into a photo dictionary object. An example of these XML photo objects can be seen in Chapter 2, Fig 4. These photo dictionaries are then added to a list and returned to be processed individually by the *photoBuilder* function.

4.1.2.3 Photo URL Builder Function

The *photoUrlBuilder* function takes a single photo dictionary object as its parameter. It then retrieves the values associated with the "id", "server", "farm" and "secret" keys in the dictionary object. These values are then used to construct a URL, which, when opened, directs to a Flickr image. The URL is constructed using Flickr's specifications[11].

4.1.2.4 Get Location Function

This function uses the flickr.photos.geo.getLocation API[9] to get the latitude and longitude related to a particular photo. It takes the photograph's id as its parameter and constructs a URL to access the API based on this parameter. Then, similar to the *reqBuilder* function, it uses the requests library to get the XML response. The response is then processed by the ElementTree library to parse the XML data to retrieve the latitude and longitude of the photo. The latitude and longitude are then returned as strings in a list.

4.1.2.5 Date Taken Function

This method is extremely similar to the *getLocation* function. It also takes a particular photograph's id as its parameter and then uses the flickr.photos.getInfo API[10] and the requests library to request and retrieve the XML data. The date on which the image was taken is then extracted from this data using the ElementTree library. This function then returns a string of the data taken.

4.1.2.6 Get Locality Function

The *getLocality* function returns a locality based on the latitude and longitude of a photo retrieved by the *getLocation* function. For example, the latitude and longitude " 53.7996" and "-1.5491" returns the locality "Leeds". A URL is built with the latitude and longitude parameters which the request library then uses to call Google's geocode API[33]. This API returns locality data in an XML response based on the latitude and longitude. From this XML response the "postal town" relating to the image's location is retrieved using the ElementTree library and the function then returns this as a string. If there is no "postal town" data available, the locality "Yorkshire" is returned by the function instead.

4.1.2.7 Photo Builder Function

The *photoBuilder* function takes a list of photo dictionaries returned by the *xmlParser* function as its parameter. It then cycles through the list using a for loop and extracts the photograph's id, owner, and title, key-values from each dictionary object. The *photoUrlBuilder*, *dateTaken*, *getLocation,* and *getLocality* functions are then used to gather the relevant information about the image. This data is then written to a comma separated text file where each line in the file represents a photo object and its relating data. An example of this can be seen in Fig 10.

```
66434265@N00,St Robert's Cave interior,2016-03-15 17:00:55,https://farm2.staticflickr.com/1469/26283973802_579151ab0b.jpg,54.004930,-1.436414,Knaresborough,
45770770@N07,Hebden Bridge Floods 2015,2015-12-30 16:09:42,https://farm2.staticflickr.com/1563/24244642692_e2e00c5003.jpg,53.740633,-2.011845,Hebden Bridge,
128195793@N04,Flooding near Royal Armouries Leeds,2015-12-26 16:07:10,https://farm2.staticflickr.com/1458/23695333204_846f963d7c.jpg,53.793663,-1.534762,Leeds,
46833951@N04,Floods at Hebden Bridge Boxing Day 2015,2015-12-26 15:23:55,https://farm2.staticflickr.com/1530/24251766636_1d766f01d1.jpg,53.742423,-2.014004,Hebden Bridge,
128195793@N04,Flooding near Thwaite Mills,2015-12-27 16:14:10,https://farm2.staticflickr.com/1697/23790460159_5a21d566c0.jpg,53.776130,-1.512456,Leeds,
```

Fig. 10 Photo data saved to text file for location: Leeds, tag: Flood

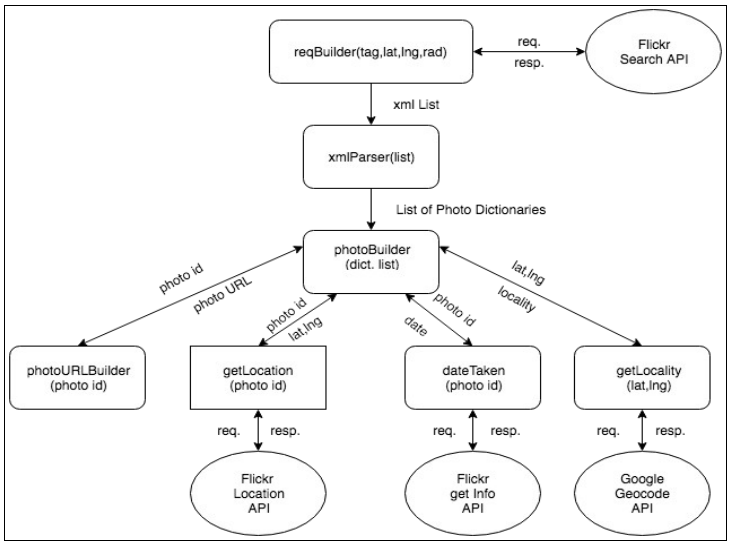


Fig. 11 Model of flickr initial function interaction

4.1.2.8 Testing of FlickrInitial Script

The initial Flickr script was tested by running it on multiple locations throughout the UK which are prone to flooding, and also running it for different tags such as "skateboarding". The comma-separated text files produced by these executions where then saved and reviewed to check the responses and images returned. The program also has multiple print statements that keep the user updated. For example, if no photos are found when searching for a particular tag or location the program outputs "No Photos available".

If there are photographs available, the program outputs the number of images to the console display which decreases as each image is added to the text file. It also outputs the data that is being written to the text file. An example of these outputs produced by running the program on the latitude and longitude of Oxford can be seen in Fig 12. It can be seen that there is at least a total of 1102 photographs with the tag "flood" within 20km of Oxford. Due to the time it takes to run each of these tests (around 20 minutes to process 500 photos), I only ran each test for 15 minutes. Although some tests required less time; the "skateboarding" tag only returned 33 photos for Leeds.

```
1102
79562587@N00,Cherwell flood,2016-03-30 18:29:43,https://farm2.staticflickr.com/1
561/26072072761_fd94e966d5.jpg,51.776935,-1.247978,Oxford,

1101
48031854@N00,The Thames path,2016-03-13 12:36:37,https://farm2.staticflickr.com/
1615/25224994314_e2261f6486.jpg,51.634544,-1.149903,Wallingford,

1100
9964400@N07,River Cherwell..,2016-03-10 15:33:16,https://farm2.staticflickr.com/
1471/25045890224_e1d471b99f.jpg,51.999847,-1.275370,Banbury,

1099
9964400@N07,River Cherwell in flood..,2016-03-10 15:33:27,https://farm2.staticfl
ickr.com/1576/25583757731_6e537029a6.jpg,51.999236,-1.281303,Banbury,
```

Fig. 12 FlickrInitial.py output for location: Oxford, tag: flood

4.1.3 Basic Homepage and Bootstrap Implementation

A basic homepage implementing Bootstrap was created which displayed links to the images gathered using the Flickr Script. A more appropriate home page would be developed later on in the project. This initial homepage was used to test the photo model, and to check that the Flickr data had been correctly uploaded to the database. This homepage would also check that bootstrap and the photographic data was being correctly used by the website. This basic homepage can be seen in Fig 13.

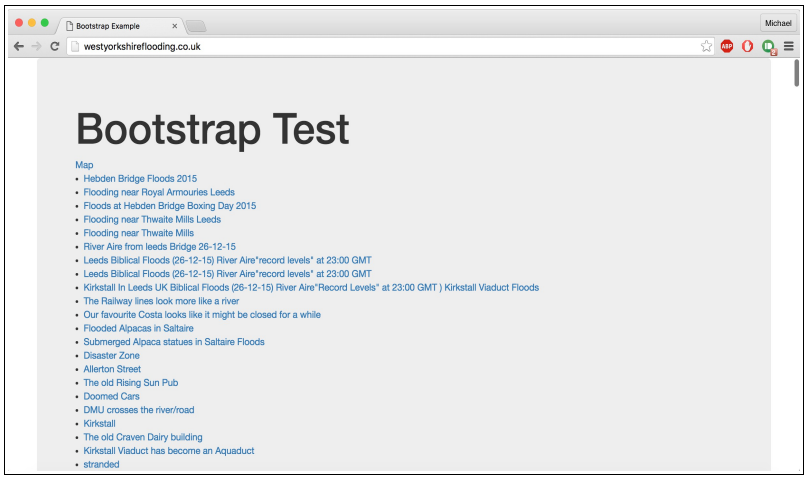


Fig. 13 Basic homepage.

The development of this basic homepage was completed as follows:

- Addition of a photo model to models.py
- Creation of database script to extract the data from a text file and add it to the database.
- Edit the index view in views.py to retrieve the photo objects from the database and pass it to the index template.
- Edit the index template to use Bootstrap and display the links to the Flickr photographs.

4.1.3.1 Photo Model and Database Population

The photo model is what Django uses to construct a database table for photographs. The photo model at this stage creates a database table with the following columns:

- owner – This is the owner id of the Flickr image, e.g. “45770770N07”.
- title_text – This is the title of the Flickr image, e.g. “Hebden Bridge Floods”.
- date_taken – The datetime value of the image, e.g. “2015-12-30 16:09:42”.
- url – This is the Flickr URL which directs to the image hosted by Flickr.
- Lat – The latitude of the image, e.g. “53.740633”.
- Lon – The longitude of the image, e.g. “-2.011845”.
- Locality – The postal town in which the image was taken, e.g. “Hebden Bridge”.

Django's manage.py can then use its “migrations” function to create a table from the model.
After this, a script (Dbconn.py) was created to populate the table with the Flickr data. This script was used to open a text file produced by the FlickrInitial Script, it then runs on each line in the file and extracts the comma-separated data which is then added to the photo table in the database. The DB script uses the “psycopg” postgreSQL database connector to populate the database, this is because the Django Models library was not currently working within the Scripts folder.

4.1.3.2 Displaying Photo Data On Homepage

Passing the photo data from the database to the home page is rather straightforward due to Django's high-level nature. All that was required was to import the photo model into views.py and then use “photo.objects.all()” command in the index view to get a Django QuerySet of all photo objects in the database. The index template is then rendered with this QuerySet by the index view.
To actually display the photo data a for loop created with Django's template language was used to run through the QuerySet. Each cycle of the for loop extracted the relevant data from the photo object and displayed it as a HTML list item with the format seen in Fig 14. It was also at this stage that Bootstrap was added to the index template. An example of this implementation can be seen in Fig 13.

```
{% if photo_list %}
    {% for photo in photo_list %}
        <li><a href="{{ photo.url }}">{{ photo.title_text }}</a></li>
    {% endfor %}
{% else %}
    <p>No photos available</p>
{% endif %}
```

Fig. 14 Django template language for loop

4.2 Map Page, DB Class, and Flickr Daily Script Development

The second iteration was mainly dedicated to the development of the map page of the website. However, some time was also spent on creating a new Flickr Script, which was an adaption of the Flickr Script in first iteration. This new script can be executed daily when a flood occurs and automatically updates the database with the data through use of a Database class (DB.py).

4.2.1 Map Page

The map page was created using MapBox[34] and Leaflet.js[35]. This page features a map centred on West Yorkshire, geo-tagged photographs are then plotted on the map using each photo's latitude and longitude. An example of this map page can be seen in Fig 15. Each point on the map relates to a flooding photograph and, when clicked, displays a popup of the photograph. This can be seen in Fig 16.

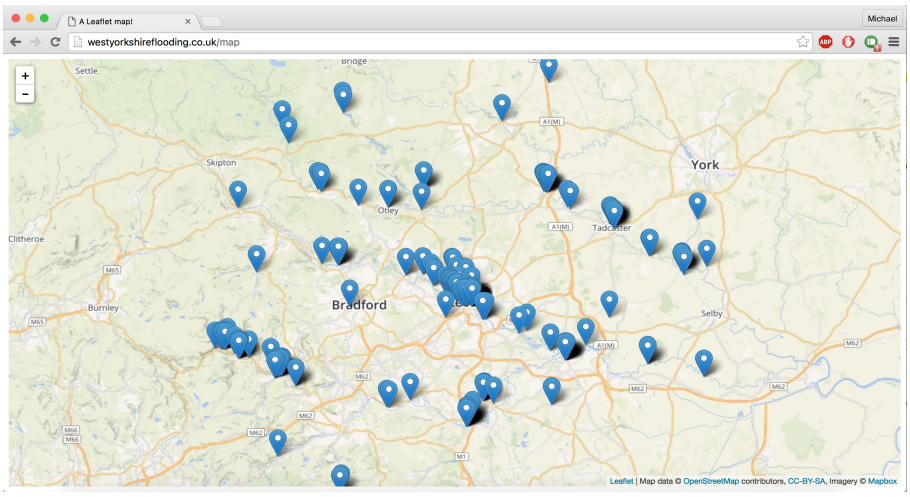


Fig. 15 Map Page

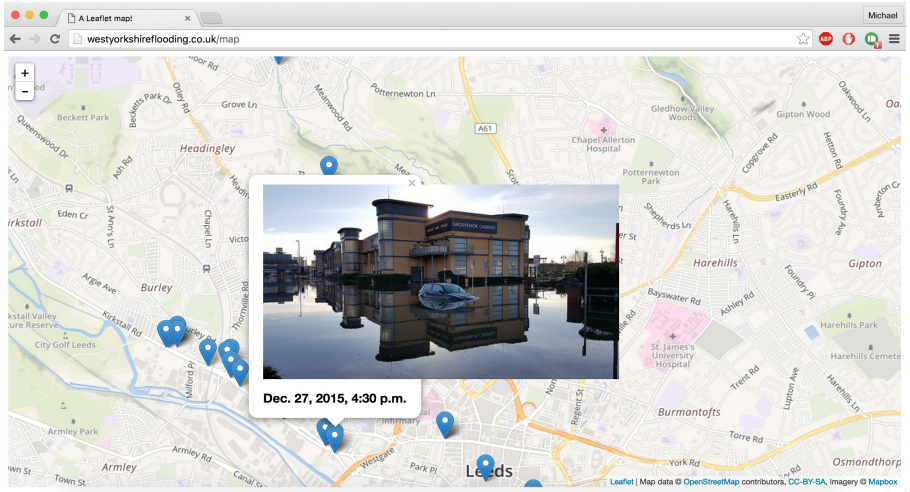


Fig. 16 Map Page Popup

4.2.1.1 Map Page Development

The stages of the map page development are as follows:

- Create a Map Page template.
- Create a Map view in views.py to render the Map page template and pass the photo data to the template.
- Add the Map Page to urls.py so it can be accessed via "/map.html".
- Create a map using MapBox.
- Display icons and photo data using Leaflet and the MapBox map on the map page.

The development of the map page was fairly straightforward as it was similar to the development of the index page. The map view was the exact same as the index view except it rendered the map template instead of the index template. In the map template, a MapBox map was embedded in the template. A for loop was then used to cycle through the photo QuerySet, the same as in the index template. The only difference is that each loop iteration now created leaflet markers based on the photo object's latitude and longitude; these markers where then plotted to the map using the *addTo* function provided by the leaflet library. To get the images to display in the popups, leaflets *bindPopup* function was used on each marker to add HTML, which uses the photo object's URL as its image source. An example of this can be seen in Fig 17.

```
var latitude = "{{ photo.lat }}";
var longitude = "{{ photo.lng }}";
var url1 = "{{ photo.url }}";
var date = "{{ photo.date_taken }}"
marker = new L.marker([latitude,longitude]).bindPopup('<img src='+ url1 +' width="100%"/><h2>'+ date +'</h2>');
marker.addTo(markers)
```

Fig. 17 Adding Markers to a Map with Leaflet

4.2.1.2 Map Page Evaluation

The map page was demonstrated to the project supervisor at one of the regular weekly meetings. At this meeting he suggested that the markers on the map page looked cluttered and did not visualise the data optimally. This made it difficult to select and view the popup images associated with each marker. It was also difficult to see which areas on the map had the most markers and had therefore been heavily affected by the floods. A possible solution to this problem is using Leaflets marker clusters plugin[36] or adding filters, such as filtering markers by date, which could be applied to the map.

4.2.2 Flickr Daily Script

The Flickr Daily script is an update of the Flickr Initial script documented in the first iteration. This updated version only searches for Flickr photos produced on the previous day, rather than searching for all historical data. It also differs by no longer producing a text file of comma-separated data; instead it automatically updates the database with any Flickr photo data that is discovered.
Only two of Flickr Daily Script's methods differ to the Flickr Initial script:

4.2.2.1 Req Builder Function Change

This function now retrieves the previous day's date using Python's date-time library. This date is then included as an extra parameter in the flickr.photos.search API[8] URL so only photos taken on that particular date are returned.

4.2.2.2 Photo Builder Function Change

This method now returns a list of photo dictionary objects, rather than producing a text file of the photo objects. An instance of the DB class is then used to process this list and add the photo objects to the photo table in the database.

4.2.3 DB Class Development

A database class was created to make it easier to add data to the database. The DB class has different functions based on the data that needs to be added to the database. In this first implementation the only function present was the *addPhotoToDatabase* function. Each script can now create an instance of the class to add data to the database, rather than having to repeat the database connection code.
The *addPhotoToDatabase* function accepts a list of photo dictionaries as its parameter, such as the list produced by the Flickr Daily script. It then attempts to connect to the database, and if successful it will begin adding each photo dictionary object to the photo table in the database. This DB class also uses the “psycopg” connector to connect to the database; if time is available at the end of the project, this class will be modified to use Django's models.

4.2.4 Flickr Daily and DB Class Tests

The Flickr Daily could not be tested using the “flood” tag as there was no flooding in the UK at the time of its completion. Instead the script was ran using tags such as “Leeds” to perform the search. Any photographs found for that day where then added to the database by the DB class; so this also

tested the functionality of this DB class. The Django Admin page was then used to check the photographs had been correctly uploaded to the website. The photographs where then deleted from the website through the Admin page.

4.2.5 Updated Project Schedule

This first and second iteration required more time than originally anticipated. This was due to inexperience with Django, and inexperience with the requests and ElementTree libraries during the script development. However, the skills learnt in these iterations will significantly speed up the development process in later iterations. So, the overall schedule should not be too greatly affected. The updated schedule can be seen in fig 18.

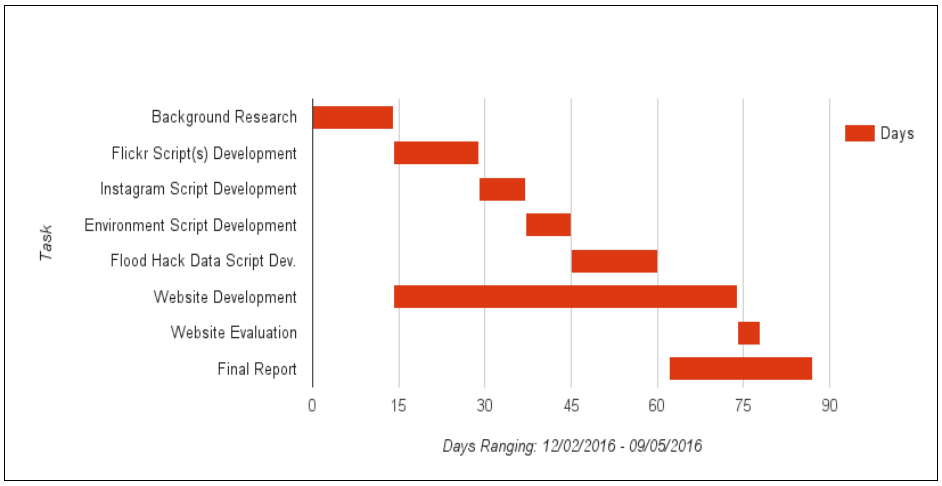


Fig. 18 Updated Schedule

4.3 Timeline Page, Instagram and Flood Area Script Development

In this iteration the first version of the timeline page was created. After this, a script for gathering flooding images sourced from Instagram was developed. This was followed by the development of a Flood Area script, which will be used daily to check for any flood warnings and updates within 20km of Leeds, and to execute the Instagram script. It was also during this stage that a navigation bar was added to all of the template pages.

4.3.1 Timeline Page

The timeline page uses a Bootstrap timeline template to displays photographs in date order. This page also has the ability to filter the photographs based on “flood-events”, where a flood-event is a range of dates, such as 26/12/2015 – 28/12/2015, for which there are consecutive flooding photos available. An example of these components can be seen in Fig 19. The map page was a way to

visualise the photographic data in a geographic sense, and the timeline page attempts to visualise the data based on when the flooding occurred and how the flooding has developed over time. Any new photographs found at later dates by the Flickr Daily script will also be added to this page; so the most recent photographs are displayed first. This will provide users with up-to-date photographs, which will allow them to see how severe the flooding is and when and where the floods are occurring.

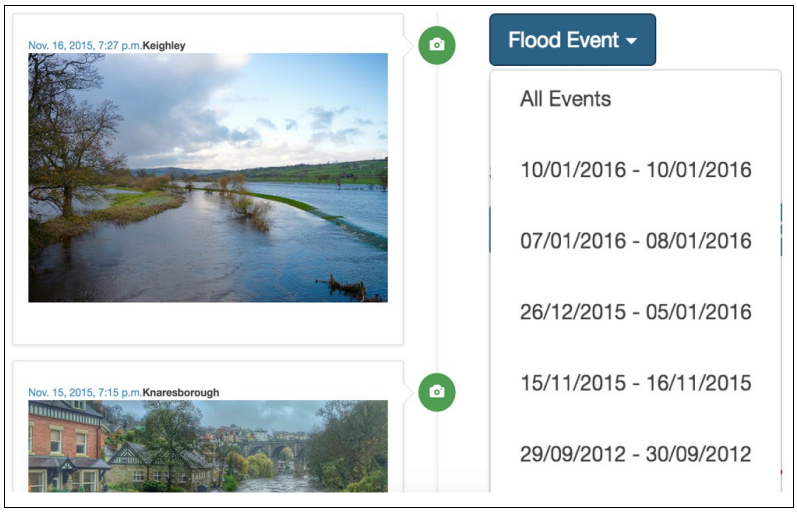


Fig. 19 Timeline display and Flood-Event dropdown

4.3.1.1 Timeline Page Development

The stages of development of the timeline page was as follows:

- Creation of timeline template.
- Creation of timeline view in views.py.
- Add the Timeline Page to urls.py so it can be accessed via "/timeline.html".
- Implementation of bootstrap timeline and photo display.
- Development of Flood Events function in views.py and its implementation in the template.

The creation of the timeline template, view, and addition of the URL to urls.py was the same as outlined in the Map page and Home page development process. The only difference was an addition of the flood_events list (produced by the flood event function), being passed from the timeline view to the template alongside a date ordered photo QuerySet. The implementation of the bootstrap timeline photo display, and how the flood_events method and filtering is developed and applied is explained below.

Bootstrap Timeline Implementation:

The Bootstrap timeline was implemented using code taken from Bootsnip's Bootstrap templates resource;[37] all it took was a little refactoring of the timeline code and bootstrap template to get it to

run on the timeline template. To generate the timeline objects, a Django template language for loop was used to generate a HTML photo object for every photo in the ordered photo QuerySet, that was received from the timeline view. As the photos had already been ordered, the for loop produced the timeline objects in the correct date order, using JavaScript's *Document.write* function to add the objects HTML to the timeline. Each object displays an image related to flooding, it's locality, and the date on which the photo was taken. The date is also a link which directs the user to the owners page.

Flood Events function and its application:

The flood events function accepts an ordered list of dates created from all of the photo objects in the database. This ordered list is then cycled through using a for loop, where the current date in the list is compared to the "start_date", which is initialised as the first date in the list. If the difference between the start date and the current iteration's date is less than one day, the current date is added to a flood_event list. This list holds multiple consecutive dates. If the difference is greater than one day, then the previous iteration's date is added to the flood_event list, and a check is then completed to see if the flood_event list has more than two elements. If there are more than two elements present, then the first and last element of the flood_event list (which corresponds to the start date and end date of the flood_event) is stored as a list. This list is then added to a list called flood_events. The flood_event list is then reset as an empty list and the "start_date" is set to the current iteration's date. The for loop then continues until the ordered date list is complete. When the for loop has finished, the flood_events list is returned by the function. An example of a flood_events list can be seen in Fig 20, while the Python code for the algorithm can be seen in Fig 21.

```
[[26/12/2015,28/12/2015],[01/01/2015,05/01/2015]]
```

Fig. 20 example of flood_events array returned

```
ordered_date_list = list
flood_events = [] #empty list to hold all of the flood events
start_date = ordered_list[0].date() #Initialise start_date
flood_event = [] #list to hold individual flood event dates

for x in range(0, len(ordered_list)): #cycle through the list

    difference = (ordered_list[x].date() - start_date).days #calculate the difference between dates between photos
    if difference <= 1: #check the difference between dates is less than or equal to 1 day
        flood_event.append(ordered_list[x]) #if photos within 1 day of each other add date to flood event

    else: #if difference between dates is greater than 1 day
        flood_event.append(ordered_list[x-1]) # add current objects date to flood event

        if len(flood_event) > 2: #if flood event contains more than 2 dates
            flood_events.append([flood_event[0],flood_event[-1]])
        flood_event = [] #reset flood event to empty

    start_date = ordered_list[x].date() #set new start_date to the next photo date in list.

return flood_events
```

Fig. 21 Python Flood_Events Algorithm

The flood_events list is then passed to the timeline template via the timeline view rendering process. This flood_events list is then used to generate the dates in the flood events dropdown menu

as seen in Fig 19, using JavaScript code. Each of the flood events in the dropdown menu are links to the timeline page but with the dates included as a substring. For example, clicking on the flood event “26/12/2015–05/01/2015” opens a page with the following URL:

- westyorkshireflooding.co.uk?12/26/2015+05/01/2015

JavaScript code in the timeline template can then pull any dates from the URL, and filter the photographs based on these dates using if and else statements. This also allows users to filter by any dates of their choice by entering the dates as a substring in the URL. In the form:

- westyorkshireflooding.co.uk?start_date+end_date

Where the dates have the format, “M/D/Y”.

4.3.1.2 Implementation of Navigation Bar

Following the creation of the timeline page, a navigation bar was added to the website. This was a simple implementation of a bootstrap “navbar” to make it easier to navigate between the pages of the website. This “navbar” has the ability to highlight the current page the user is viewing and each button links to the relevant page.

Fig. 22 Navbar display on Map Page

4.3.2 Instagram Script

The Instagram script uses the media/search API[18] to retrieve all recent Instagram media within 5km radius of a specified latitude and longitude. The media is then filtered by its tags, so that only photographs with tags related to flooding are returned. These images then can be added to the photo table in the database via the DB class. The Instagram script consists of two functions: *searchLocation* and *getLocality*. The *getLocality* function is the same function that is documented in the development of the Flickr Script, so only the *searchLocation* function will be documented in this section. This script can only search for recent media due to Instagram's API restrictions. This means if the script is to gather any relevant flooding images it will have to run at regular intervals throughout any days on which a flood has occurred.

4.3.2.1 Search Location Function

The *searchLocation* function accepts the following three parameters:

- Latitude – Latitude of the location to be searched.
- Longitude – Longitude of the location to be searched.

- Search Term – The tag used to filter the media, e.g. “Flood”.

Similar to the Flickr Script, the latitude and longitude parameters are used to create a URL that the Python requests library will use to query the API and retrieve the JSON data response. This JSON data is then parsed using the “simplejson” library[38]. Any images with a tag that is equal to the search term parameter are saved as a photo dictionary object, with the same key-values as seen in the Flickr script. This dictionary is then added to a photo list, the same as the Flickr script. The photo list is then returned which can be used to add the photos to the database by the DB class.

4.3.2.2 Updated Photo Model and Flickr Scripts

A new key-value and field, “source”, was added to the photo dictionary in both the Flickr and Instagram scripts, and to the photo model. This will be used to determine from where a particular photo had been retrieved. A Flickr photo will have a “source” value of “flickr”, but any Instagram photos will have the value “instagram”. All photographs currently in the photo table were given the value “flickr”, this is because no Instagram photos were present at this time. This field is required because the Instagram and Flickr photos require different HTML code to display the photos.

4.3.2.3 Instagram Script Test

The Instagram script was tested by running the script on several different locations and search terms. However, because there was no flooding in the UK during or after the development of this script it could not be tested for the search term “flood” or “flooding”. The script itself contains a print statement which prints out any data that has been gathered to the console. It also prints the URL used to make the request. This way the data gathered by the script can be compared to the actual data returned by the request to see if any data has been missed. An example of this response can be seen in Fig 23.

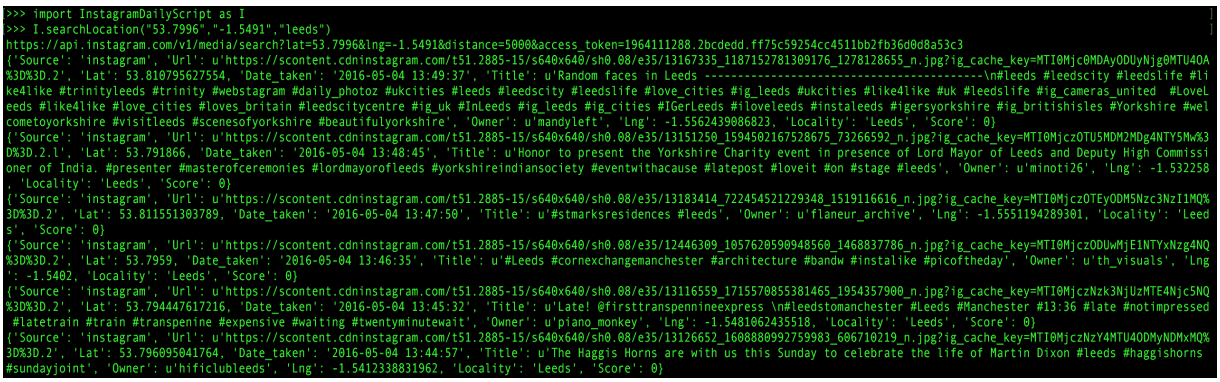


Fig. 23 Console response for 5Km search around Leeds, with tag “leeds”

4.3.2.4 Adding Instagram Data

The timeline page was updated to display Instagram images, and this involved a simple If statement to check the source value of the photo object. This is because the Flickr and Instagram photographs used different URLs when providing links back to the owner's page. To test this was working correctly, ten manually sourced Instagram images where added to the database via the Django admin page. The Instagram photo links were then tested to make sure they directed to the correct page.

4.3.3 Flood Areas Script

The flood areas script uses the flood monitoring API[19] maintained by the UK government. The script consists of four functions: *searchArea*, *getFloodLocation*, *runInstagram* and *dailyExecute*. The *dailyExecute* function searches for any flood warnings within 20km of Leeds by calling the *searchArea* function. If any warnings are found, they will be passed to the *runInstagram* function. This function will search for any Instagram photos related to flooding for the locations found using *getFloodLocation* function. This Instagram search will be executed at 10 minute intervals for 24hours; in order to gather any flooding images that may be present in the recent media. A diagram of how these functions interact can be seen in Fig 24.

4.3.3.1 Search Area Function

The *searchArea* function accepts three parameters:

- Latitude of the area to be searched with a string format.
- Longitude of the area to be searched with a string format.
- Radius, size of the area to be searched in kilometres with a string format.

The search then constructs a URL to query the flood monitoring API using the requests library, and this API then returns a JSON response of flood warnings. This JSON data is then parsed using the python “simplejson” library and the following data is extracted for each flood alert:

- Flood Id: This is another URL which returns detailed information about the flood such as its latitude and longitude.
- Description: This is any information relating to the particular flood, such as areas affected.
- Severity: This is the type of warning issued, such as possible flooding or flood alert.
- Severity Level: This is a numerical value specifying how extreme the flood is, with higher values indicating higher severity.
- Time: The time and date the warning was issued.

This data is saved as a "flood-alert" dictionary object and then added to a list of "flood-alerts" which is then returned by the function.

4.3.3.2 Get Flood Location Function

This function accepts a Flood Id as its parameter, which is a URL string for a particular flood warning. This URL is then queried using the requests library. The JSON response is then parsed using the "simplejson" library and the latitude and longitude are extracted from the JSON data, and returned as a list.

4.3.3.3 Run Instagram Function

This function accepts a list of flood warnings produced by the *searchArea* function as its parameter. The latitude and longitude of each flood warning are then found using the *getFloodLocation* function. The *searchLocation* function taken from the Instagram script is then used to search for any recent media related to flooding; the latitude and longitude of the flood warning is used as its parameters. The Instagram search API is limited to a search area of 5km; this implementation provides a much more targeted search area, since this implementation only areas that are definitely flooded or expected to flood are searched. If any photographs are found using the *searchLocation* function they are added to the photo table in the database. This is completed by an instance of the DB class created earlier in the project.

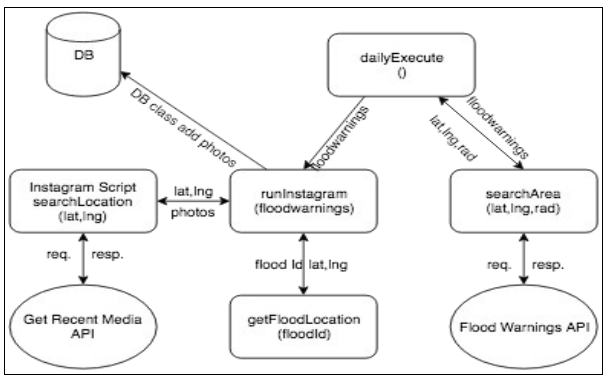


Fig. 24 floodAreas function interactions

4.3.3.4 Testing of Flood Areas Script

Since there was no flooding in the UK during or after the development of the Flood Areas Script, it had to be tested using fake JSON data. This fake JSON data was constructed based on the specified data returned by the environment API documentation[19]. Other similar environment APIs were also used to see how the APIs structured the JSON data. The flood script contains multiple print statements to monitor the results produced; an example of this output can be seen in Fig 25.

```
Michaels-MacBook-Pro:Scripts mike$ python FloodAreas.py
[{'severitylevel': '4', 'severity': 'Warning', 'time': '01/01/2016 16:00:00', 'i
d': 'http://environment.data.gov.uk/flood-monitoring/id/floodAreas/053FWFPUWI06'
, 'description': 'flood warning for leeds'}, {'severitylevel': '3', 'severity':
'Alert', 'time': '01/01/2016 15:00:00', 'id': 'http://environment.data.gov.uk/fl
ood-monitoring/id/floodAreas/061FWF23Goring', 'description': 'flood warning for
Hebden Bride'}]
http://environment.data.gov.uk/flood-monitoring/id/floodAreas/053FWFPUWI06
('location:', [53.222051752689374, -0.5415693979278])
https://api.instagram.com/v1/media/search?lat=53.2220517527&lng=-0.541569397928&
distance=5000&access_token=1964111288.2bcdedd.ff75c59254cc4511bb2fb36d0d8a53c3
('Instagram Data:', [])
Nothing to add to DB.
http://environment.data.gov.uk/flood-monitoring/id/floodAreas/061FWF23Goring
('location:', [51.519009988614556, -1.130658812882548])
https://api.instagram.com/v1/media/search?lat=51.5190099886&lng=-1.13065881288&d
istance=5000&access_token=1964111288.2bcdedd.ff75c59254cc4511bb2fb36d0d8a53c3
('Instagram Data:', [])
Nothing to add to DB.
sleeping 10minutes
```

Fig. 25 Example Console output from Flood Area Script Test

4.4 Twitter Data, Photo Ranking and Map Clustering

This iteration took place after the Open Data Institute in Leeds held their “Flood Hack” event, which provided the Twitter data set outlined in Section 2.2.5. This prompted the creation of a script to extract any tweets related to the floods from this data set, and these tweets where then added to the map and timeline pages. A photo voting system was also implemented on the timeline page; this provided users with a way of ranking photos based on their quality, and relevance to flooding. Lastly, map clustering was added to the map template to improve the visualisation of the data.

4.4.1 Twitter Data Implementation

The twitter data was implemented into the project in the following way:

- Creation of tweets model in models.py.
- Addition of *addtweetdatabase* function to the DB class.
- Development of a Twitter script to extract and save tweets to the database.
- Update *timeline*, *map,* and *flood_event* functions in views.py.
- Implementation of tweets on the timeline and map templates.

4.4.1.1 Tweets Model and DB Class Update

A tweets model was added to models.py, and this model was used to create a tweets_table in the database alongside the photo table. Once the tweets model was created, Django's “migration” function in manage.py was then used to create the table in the database.

The tweets model consists of the following fields:

- Date_taken: The date on which the tweet was posted.
- Lat: The latitude value of the geo-tagged tweet's location.

- Lng: The longitude value of the geo-tagged tweet's location.
- Username: The Twitter username associated with the person who posted the tweet.
- UserId: The userId of the person who posted the tweet.
- TweetId: The id of the specific tweet.
- Tweet: The main body of the tweet, its contents.
- Html: The embed HTML of the tweet, used to display the tweet inside a webpage.

A function was added to the DB class to connect and add any gathered tweets to the tweets table. The *addtweetdatabase* function accepts a list of tweet dictionary objects as its parameter, and it works in the exact same way as the *addphotodatabase* function. The only difference is that the tweets method uses the parameters specified in the model above to update the database.

4.4.1.2 Twitter Script

The Twitter script was used to extract any tweets related to flooding from a csv file of geo-tagged tweets. Once the extraction is complete the tweets are added to the tweets table, which is executed by the DB class. The extraction process is fulfilled by the following methods:

ExtractTweetsCsv Function:

This function opens and reads the csv file using Python's DictReader class. A for loop is used to cycle through each row in the csv file, where each row represents a single tweet. The datetime, latitude, longitude, username, userId, tweet body, and tweetId are then retrieved from the row. The tweet body is then checked to see if it contains the word “flood”. If this word is absent the tweet is ignored; the tweet is also ignored if it contains the word “warning” or “alert”, this is to ignore irrelevant tweets that are usually automatic warnings.

Each tweet's username is also compared to a list of forbidden users; if the tweet's username is present in this list, it is discarded. The forbidden users list is a list of usernames associated with Twitter accounts that post flood alerts. These are discarded as only tweets from citizens are to be shown on the website. If a tweet passes this test, the values specified above are used to construct a tweet dictionary object, and this object is then added to a tweets list. This list of tweet dictionary objects is then returned and passed to the DB class for processing.

GetEmbedHtml Function:

This function is called by the *ExtractTweetsCsv* function to get the embed HTML associated with a tweet. It accepts a tweetId as its parameter; which is used to construct a URL to query Twitter's “oembed” API[39]. The requests library is used to query the URL and receives a JSON data response, and the embed HTML is then extracted from this JSON data using “simplejson”. The HTML is then

“cleaned” of any JavaScript present in the HTML by the use of a library called Beautiful Soup[40]. The JavaScript is removed so the HTML can be safely stored in the database. It will be added back to the HTML when the tweet is displayed on the website. This “clean” HTML is then returned as a string by the function.

Testing the Twitter Script:

The Twitter Script contains multiple print statements to check that only the correct tweets are being added to the database; the twitter script was tested by executing it on small sections of the data-set and then the results produced where checked against the data-set. An example of these console outputs can be seen in Fig 26.


```
DONE: 1
Tweet: {'Html': '<blockquote class="twitter-tweet"><p lang="en" dir="ltr"><a href="https://twitter.com/NYorksRPG">@NYorksRPG</a> <a href="https://twitter.com/hashtag/B1224?src=hash"
>#B1224</a> heavily flooded before Bickerton.</p>&mdash; Vicky Sanderson (@VickySander25) <a href="https://twitter.com/VickySander25/status/680661080192888833">December 26, 2015</a>
</blockquote>', 'Time': '2015-12-26T08:07:02.000', 'Lat': '53.946408679999998', 'Lng': ' -1.31813223', 'TweetId': '680661080192888833', 'UserId': '387634189', 'Tweet': '@nyorksrpg #
b1224 heavily flooded before bickerton.', 'Label': 'VickySander25'}
DONE: 2
Tweet: {'Html': '<blockquote class="twitter-tweet"><p lang="en" dir="ltr"><a href="https://twitter.com/NickAhad">@NickAhad</a> from pics. The flood in Mytholmroyd is worse than 1963
(I lived there then)</p>&mdash; David Peter Woodhead (@DavidWoodhead26) <a href="https://twitter.com/DavidWoodhead26/status/680718791152676864">December 26, 2015</a></blockquote>',
'Time': '2015-12-26T11:56:21.000', 'Lat': '53.803199200000002', 'Lng': ' -1.6476857', 'TweetId': '680718791152676864', 'UserId': '1471944446', 'Tweet': '@nickahad from pics. the flo
od in mytholmroyd is worse than 1963(i lived there then)', 'Label': 'DavidWoodhead26'}
DONE: 3
Tweet: {'Html': '<blockquote class="twitter-tweet"><p lang="en" dir="ltr"><a href="https://twitter.com/NickAhad">@NickAhad</a> <a href="https://twitter.com/hashtag/floodingfilmclub?
src=hash">#floodingfilmclub</a><br />Bring me the Head waters of Alfredo Garcia<br />Any film with River Phoenix</p>&mdash; David Peter Woodhead (@DavidWoodhead26) <a href="https://
twitter.com/DavidWoodhead26/status/680721906035732480">December 26, 2015</a></blockquote>', 'Time': '2015-12-26T12:08:44.000', 'Lat': '53.803195799999997', 'Lng': ' -1.6477092', 'Tw
eetId': '680721906035732480', 'UserId': '1471944446', 'Tweet': '@nickahad #floodingfilmclubbring me the head waters of alfredo garciaany film with river phoenix', 'Label': 'DavidWoo
dhead26'}
DONE: 4
Tweet: {'Html': '<blockquote class="twitter-tweet"><p lang="en" dir="ltr">Strensall under water! <a href="https://twitter.com/hashtag/yorkfloods?src=hash">#yorkfloods</a> <a href="h
ttps://twitter.com/hashtag/boxingday?src=hash">#boxingday</a> @ Strensall, York, United Kingdom <a href="https://t.co/STpJPQadIO">https://t.co/STpJPQadIO</a></p>&mdash; Doug Pearce
(@dougiepearce) <a href="https://twitter.com/dougiepearce/status/680724024981340160">December 26, 2015</a></blockquote>', 'Time': '2015-12-26T12:17:09.000', 'Lat': '54.0499999999999
97', 'Lng': ' -1.0333300000000001', 'TweetId': '680724024981340160', 'UserId': '20979265', 'Tweet': 'strensall under water! #yorkfloods #boxingday @ strensall, york, united kingdom
https://t.co/stpjpqadio', 'Label': 'dougiepearce'}
```


Fig. 26 Example output from the Twitter Script

4.4.1.3 Updating Views.py

To pass the Twitter data from the database to the map and timeline templates, certain functions in views.py had to be updated. Passing the data to the map page is a simple task. The Twitter data is retrieved from the database as a QuerySet, which is then passed to the map template alongside the photo QuerySet. However, passing the Twitter data to the timeline template was more complicated.

Updating *Timeline* Function:

The *timeline* function now retrieves both the Photo and Twitter data from the database as QuerySets; these sets are ordered by the date on which they were taken. Both QuerySets are then iterated by using two separate for loops (one for each QuerySet). During each loop a list is created which contains the date of the tweet, the objects type (“photo” or “tweet”), and finally the photo or tweet object itself. This list is then added to a “combined list” which is made up of both the Tweet and Photo lists. An example of this combined list can be seen in Fig 27. This combined list is then sorted again using the dates added to the lists earlier, so the tweets and photos are in the correct date order, and this list is then passed to the *flood_events* function. The flood events function was also updated to accept and process this combined list. This was so it could find flood event dates based on both the tweets and photos. Finally, the combined list and the flood events list is passed back to the *timeline* function, which passes the lists to the timeline template by the to be displayed.

```
[[date,"photo",photoDictionary],[date,"tweet",tweetDictionary],[date,"photo",photoDictionary]]
```

Fig. 27 Example of a combined list

4.4.1.4 Updating the Templates

Updating the map template to show tweets plotted on the map was simple. This is because it used the same process that was used to add the photos to the map, outlined in Section 4.2.1.1. The only difference was that the *bindPopup* function added the tweet's embed HTML (rather than an image) to the popup.

The addition of the tweets to the timeline was slightly more difficult. Some of the previous template code had to be refactored; rather than using individual QuerySets, the combined list was used instead. This was so the tweets and photos would display in the correct date order and on the correct sides of the timeline. If statements were added to the for loop which cycles through the combined list; these if statements checked the object's type. Objects with the type “photo” are plotted on the left side of the timeline, where as “tweet” objects are plotted on the right. This can be seen in Fig 28.

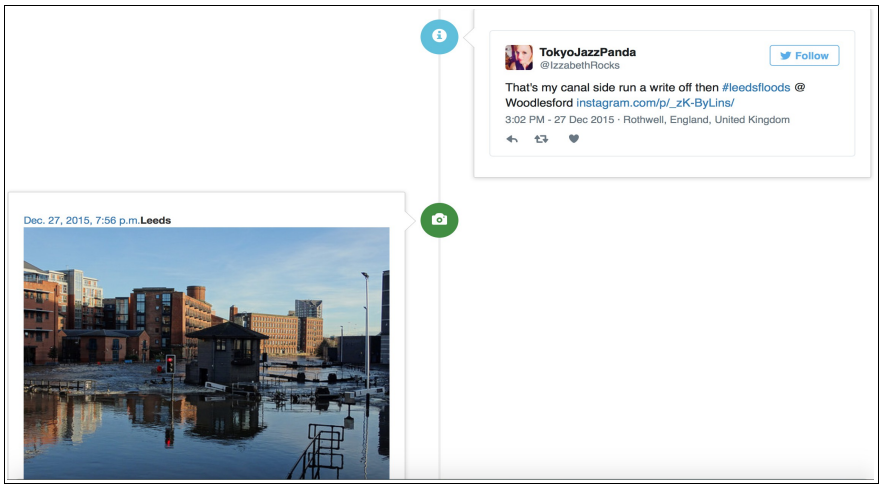


Fig. 28 Example of tweets and photo timeline

4.4.2 Photo Ranking

A solution to the evaluation of the photographic data was to add a ranking system. Up and down buttons, along with a score value, were added to the images displayed on the timeline page. Clicking on a image's up button adds a '1' to its current score, where as clicking the down button does the opposite. A filter was also added to the Django admin page, and this filter allows the administrator to filter the photographs based on these scores. This provides a way for the administrator to remove any negatively scored images which may be unrelated to flooding or that may be of poor quality. The photo ranking system was implemented in the following stages:

- Addition of a 'score' field to the photo model and scripts.
- Implementation of up and down buttons, and a score field to the timeline photos.
- Creation of JavaScript up and down functions to control the buttons.
- Creation of *up* and *down* views in views.py which are used to update the database.

4.4.2.1 Updating the Photo Model and Scripts

A new field “score” was added to the photo model. The “migrations” function was then used to update the photo table, specifying that all current entries in the table are given a score of “0”. The key “Score”, with a default value “0” was then added to the photo dictionaries in the Instagram and “Flickr Daily” Scripts. This ensured that any new photo data retrieved using these scripts would automatically be given a score of “0” before being added to the database. The DB class was also updated to reflect this change.

4.4.2.2 Adding Buttons and the Score Field

Bootstrap buttons and score values where added below each image on the timeline page; as seen in Fig 29. A HTML “onClick” event was also added to each button. This “onClick” event passes the photograph's id value to either the up or down JavaScript function, depending on which button is clicked. These functions then handle the dynamic updating of the photograph's score on the page and in the database.

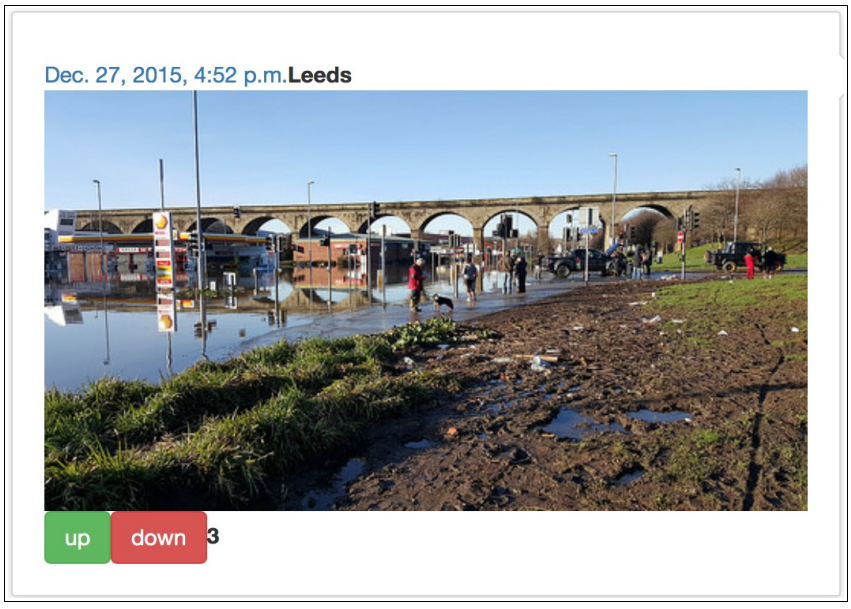


Fig. 29 Timeline Photo with Up/Down buttons and Score

4.4.2.3 JavaScript Button Functions

A “timelinefunctions” JavaScript file was created and stored in the “static/flood/js” Django project directory. This JavaScript file can then be utilised by any template page, as long as a link to the file is added to the template. This JavaScript file contains two functions, *up* and *down*, for each of the buttons. Each function accepts a photograph's Id value as its parameter. The photograph's score is then accessed using it's Id value and the *getElementById.innerHTML* JavaScript function to get the score associated with that Id; this is used to update the score on the page with the new score dynamically. After this, the function then passes the photograph's Id to its corresponding view function in views.py. These views are accessed by sending a URL request with the function's name, and the photo's id as an argument. An example URL can be seen in fig 30.

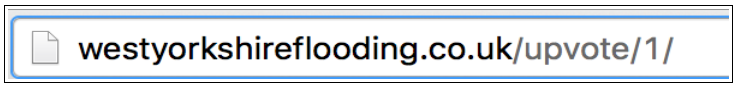


Fig. 30 Upvote Url Example

4.4.2.4 Up and Down, View Functions

Two new view functions named *upvote* and *downvote* were added to “views.py”. These views accept a photograph's Id as its parameter, which is passed to the view functions as an argument in the URL request as seen in Fig 26. This Id is then used to retrieve the photo object associated with the Id, from the photo table in the database. The photograph's score is then decreased or increased by 1 depending on which view was called and the new score value is saved to the database. The view then returns a HttpResponse of “204”; this response ensures that the user stays on the timeline page. However, this method could lead to abuse; continuous requests can be sent to the view functions to manipulate a photograph's score. For a fully developed website a different approach to this photo ranking would have to be implemented. Some possible solutions could be to use Ajax and Jquery to make the database requests along with requiring the user to sign up to the website, before the buttons can be accessed.

4.4.3 Map Clustering and Unique Icons

MapBox's leaflet map clustering[36], groups together the markers on the map page to give the map a cleaner look. Clustering also improves the way the data is visualised; it provides extra insight into how heavily certain areas are affected by the floods, based on the size of each cluster. An example of this clustering can be seen in Fig 31, where as the map before clustering can be seen in Fig 15.

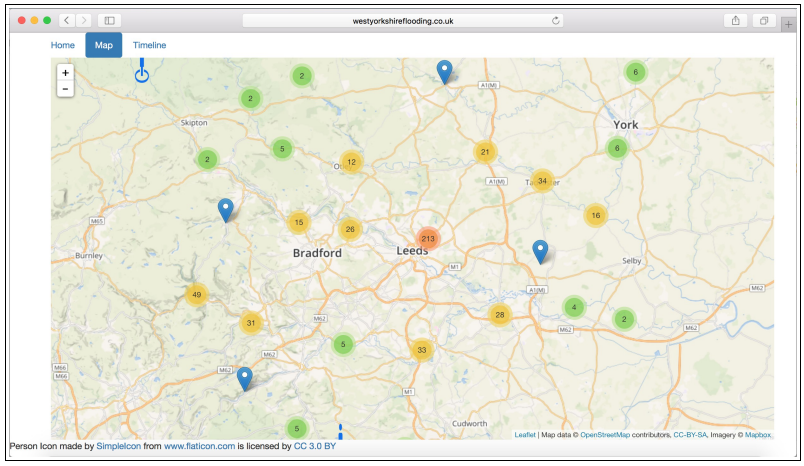


Fig. 31 Map page with clustering

The leaflet clustering also displays a bounding box that highlights which areas on the map the clusters apply to. This is accessed by hovering over the cluster with the mouse pointer, these clusters also zoom into the area highlighted when a cluster is clicked. An example of this bounding box can be seen in Fig 32.

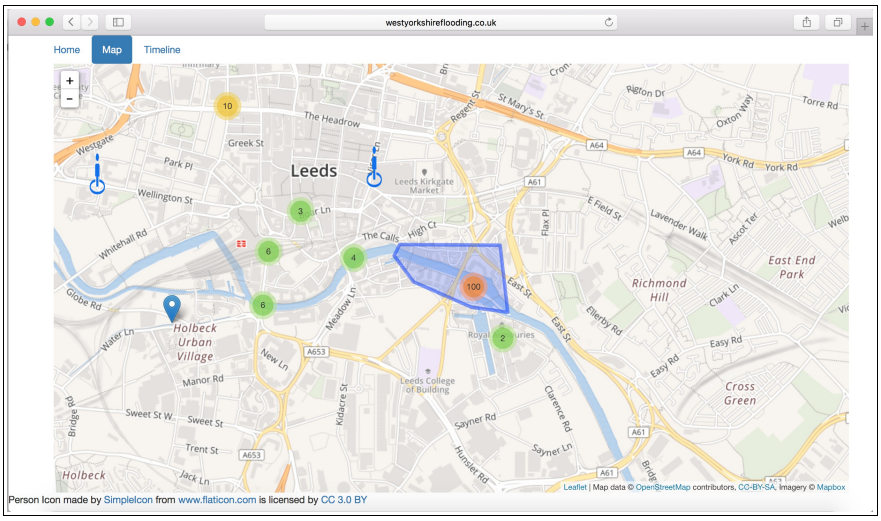


Fig. 32 Map page with bounding box display

The map clustering was implemented by adding the MapBox JavaScript clustering library[36] to the map template. Then rather than plotting any markers straight to the map they were first added to a "MarkerClusterGroup" which is then plotted to the map instead of the individual markers. The clustering library then handles how the clusters are displayed and operated.

It was also at this stage that a new icon was added to the map; this icon was used to distinguish between photos and tweets on the map. This was implemented by saving the icon's image in the static directory and then creating a new leaflet icon (named "tweetIcon") from this image. The JavaScript code used to create tweet markers was then modified to use the tweet icon instead of the default icon. The icon's image was taken from a free online resource, and credit to the creator of the icon was added to the map page. These icons can be seen in Fig 33.

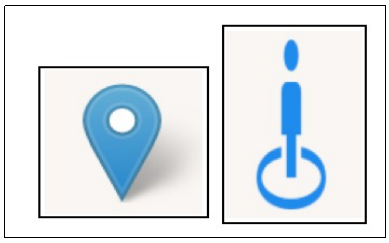


Fig. 33 Photo Icon and Tweet Icon

4.5 Implementing River Level Data and Updating the Home Page

In this iteration Rain and River Level data provided by Leeds City Council and the Environmental Data API were added to the timeline page. After this the website's home page was updated to display live flood warnings and information about the website.

4.5.1 River Level Data

The river level data displayed on the timeline page was collected from a government archive[41]. The csv data-set provided by this archive is usually fairly large; it consists of river level readings taken at 15 minute intervals for every river level gauge in the UK. Each csv data-set represents one day, with the archive having records dating back to the 15 January 2015.

The River Level data was implemented in the following stages:

- Addition of a River Level model to models.py.
- Addition of the *addRiverLevel* function to the DB class.
- Creation of a River Level script to gather the data from the csv files.
- Updating the timeline template and views to display and query the data.

4.5.1.1 River Level Model and DB Class Update

A new model named “RiverLevel” was added to the Django models file; this model was used to create a new table in the database to house the river levels. Django's “migration” functions in manage.py were used to complete these changes to the database.

The River Level Model has the following fields:

- date_taken: This is the date-time value of when the river level was recorded.
- river_level: This is the value of the river level recording.
- place: The area location for the river level, e.g. “Leeds”.
- lat: The latitude value of the location of the river measure.
- lng: The longitude value of the location of the river measure.

A new function *addRiverLevel* was then added to the DB class. This is similar to the other functions in the class; it accepts a list of river level dictionary objects as its parameter. Each river level object is then added to the river level table using the fields specified above.

4.5.1.2 River Level Script

The river level script downloads all river level archives between two specified dates. Each csv file is then parsed and the river level data for five different river measures in West Yorkshire are saved as a comma-separated text file and added to the database table. An example of a csv file's data and the text file produced after processing can be seen in Fig 34 and 35.

dateTime	measure	value
2016-05-04T00:00:00Z	http://environment.data.gov.uk/flood-monitoring/id/measures/3625TH-level-stage-i-15_min-mASD	0.091
2016-05-04T00:00:00Z	http://environment.data.gov.uk/flood-monitoring/id/measures/E660-level-stage-i-15_min-mAOD	24.567
2016-05-04T00:00:00Z	http://environment.data.gov.uk/flood-monitoring/id/measures/4040TH-level-stage-i-15_min-mASD	0.936
2016-05-04T00:00:00Z	http://environment.data.gov.uk/flood-monitoring/id/measures/3815TH-level-stage-i-15_min-mASD	0.849
2016-05-04T00:00:00Z	http://environment.data.gov.uk/flood-monitoring/id/measures/2029-level-stage-i-15_min-mASD	0.614
2016-05-04T00:00:00Z	http://environment.data.gov.uk/flood-monitoring/id/measures/E15230-level-stage-i-15_min-mAOD	102.793
2016-05-04T00:00:00Z	http://environment.data.gov.uk/flood-monitoring/id/measures/E11321-level-stage-i-15_min-mASD	0.07
2016-05-04T00:00:00Z	http://environment.data.gov.uk/flood-monitoring/id/measures/0903TH-level-stage-i-15_min-mASD	0.044

Fig. 34 River Level CSV file Data

```
place,date_taken,river_level,lat,lng
mytholmroyd,2015-01-15T01:30:00Z,1.264,53.73128,-1.983282
castleford,2015-01-15T01:15:00Z,1.558,53.731002,-1.358177
Leeds,2015-01-15T01:15:00Z,0.711,53.794197,-1.535433
Kirkstall,2015-01-15T01:15:00Z,1.703,53.819613,-1.605025
tadcaster,2015-01-15T01:15:00Z,0.210,53.677173,-1.491306
Leeds,2015-01-15T00:00:00Z,0.701,53.794197,-1.535433
tadcaster,2015-01-15T01:00:00Z,0.203,53.677173,-1.491306
Kirkstall,2015-01-15T00:45:00Z,1.705,53.819613,-1.605025
mytholmroyd,2015-01-15T00:45:00Z,1.159,53.73128,-1.983282
castleford,2015-01-15T00:30:00Z,1.554,53.731002,-1.358177
```

Fig. 35 Example Text File Data

As can be seen in Fig 34, the csv file only lists the date, measure URL, and river level value. However, each measure URL, when queried, provides extra information about the measure, such as its latitude, longitude, and area which that measure applies to.

Rather than querying every measure URL to check this information, the environmental measures API[42] was used to search for all measuring stations within 20km of Leeds. This response was then manually searched to find 5 measuring locations in West Yorkshire that best represented the photo data gathered. Each location's measure URL, latitude, and longitude were then saved as variables in the script. These measure URL's could then be used to filter the data, and an example of these URL's can be seen in Fig 36.

```
leeds = "http://environment.data.gov.uk/flood-monitoring/id/measures/L1707-level-stage-i-15_min-m"
kirkstall = "http://environment.data.gov.uk/flood-monitoring/id/measures/L1708-level-stage-i-15_min-m"
castleford = "http://environment.data.gov.uk/flood-monitoring/id/measures/L1703-level-stage-i-15_min-m"
tadcaster = "http://environment.data.gov.uk/flood-monitoring/id/measures/L1304-level-stage-i-15_min-m"
mytholmroyd = "http://environment.data.gov.uk/flood-monitoring/id/measures/L1204-level-stage-i-15_min-m"
```

Fig. 36 Measure URLs used to filter the data

The river level script contains two functions: *DownloadRiverData* and *readFromFile*, which are used to download and process the data.

DownloadRiverData Function:

The download river data function accepts two date objects (start date and end date) as its parameters; a for loop is then used to cycle through every date between these two date objects. Each date is used to construct a URL, the request library is then used to download the csv file related to this particular date. Another for loop is then used to extract the measure URL, date-time, and river level value for each row in the csv file. The measure URL is then compared to each location measure URL variable seen in Fig 36. If the current measure URL being examined is equal to one of these variables, its location, value, latitude, longitude and date-time is used to create a river level dictionary object. This river level dictionary is then added to a list of river level dictionary objects. This list is then passed to an instance of the DB class when all of the dates requested have been processed; the DB class instance then adds the river levels to the database. The river level dictionary object's values are also added to a text file as seen in Fig 35, which can be used to add the river level objects to the database at a later date using the *readFromFile* function. Finally, this function was tested by running it on particular csv files and checking that the correct data has been added to the text files. There are also multiple print statements which are used to see if the program is running correctly.

readFromFile Function:

This function accepts a string of a text file's name and location. Each line in the file is then examined using a for loop; the location, latitude, longitude, date-time, and river level value are then extracted. This data is then used to create a river level dictionary object, which is added to a list. Once all of the lines in the text file have been processed the final version of the list is passed to an instance of the DB class to add the data to the database. This function contains multiple print statements for test purposes, to see if the data is processed correctly.

4.5.1.3 Updating Views.py for River Level Data

To pass the river level data from the timeline view to the timeline template, the *getFloodEvents* function in the Django views file was updated. The else statement in the function's for loop now retrieves a filtered QuerySet of river levels, that is filtered based on the start date and end date of a flood event. This QuerySet of river level objects is then added as the third element in the flood_event list, which is then added to the flood_events list that is passed to the timeline view. This ensures that only river level data related to any flood events is passed to the timeline template. Originally, a QuerySet of all river level objects was passed from the timeline view to the template,

where the template would then perform any date filtering. However, this stopped the template from loading properly in the browser window. This was because the river level QuerySet was too large to be processed alongside the photo and Twitter data. So the flood events solution detailed above was implemented. It was also possible this problem was also due to an inefficient Django query. Or it could be because the machine running the website needs to be upgraded, as the current machine has minimal RAM. This was for cost reasons, a fully developed website would be ran on a much more powerful machine.

4.5.1.4 Timeline Template River Level Display

The river level data was displayed on the timeline page by using the Plotly.js library[43] to create a graph from the river level data. The graph displays the river level data for every possible river level gauge location (such as "Leeds", "Castleford", etc). Bootstrap buttons have also been added above the graph which can filter the data using these locations. The graph can also be filtered by date using the flood events; this filter uses the dates of the flood event that is being viewed to filter the river level data.

Any example of the graph, and the location filtering, can be seen in Fig 37 and 38. The timeline page also displays a message above the graph, this message changes depending on the availability of the river level data.

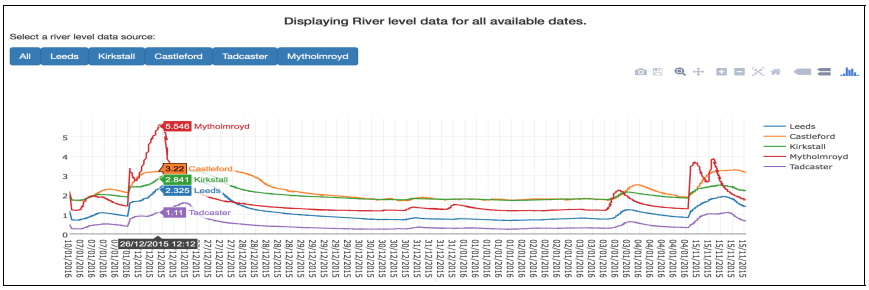


Fig. 37 River level graph with mouse hover and no filtering

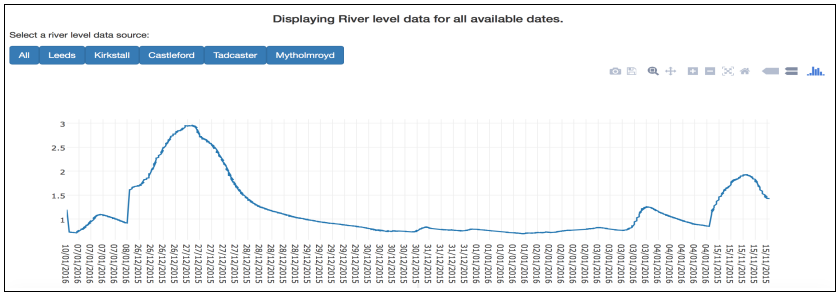


Fig. 38 River level graph with the "Leeds" filter

River Level Graph Implementation:

The graph was implemented by refactoring the example code provided in the Plotly.js "scatter graph" documentation[43]. First "traces" are created for each river gauge location; a "trace" is a

JavaScript dictionary. Each “trace” has a trace type, name, and, X and Y arrays which are used to plot the data on the X and Y axes. For the river level graph, the X and Y arrays corresponded to the river level dates and values respectively, and the trace type was set to “scatter” (for a scatter graph). The river level objects received from the timeline view are then iterated through using a for loop. The current iteration's river level data was then added to the different traces, based on the level's location. For example, a river level with the location “leeds”, has its date and river level value added to the X and Y arrays in the “leeds” trace. Once the different traces have been filled they are added to another array, which is an array of traces to be plotted on the map. This array is then used to add the data to the graph using Plotly's *newPlot* function. The creation of the graph, and the plotting of the data on the graph, is all handled by the Plotly JavaScript library which is imported in the timeline template's HTML.

The Bootstrap buttons filter the data by using an “onClick” event to reset and plot new data the graph. The *newPlot* function is called by the “onClick” event and the function plots only the selected button's trace to the graph after reseting the graph. For example, clicking the “Leeds” button plots the Leeds trace on the graph. The date filtering is applied in the same way as the photo and tweet date filtering on the timeline. The river level's date is simply compared to the selected flood event dates with an If statement; if the river level object is outside of these dates, it is not added to any of the location traces.

4.5.1.5 Rain Level Data

Originally, the Plotly graphs used rain fall data instead of river level data. This rain fall data was sourced from a rain gauge data-set provided by the Leeds City Council[44] and implemented using the same steps outlined in Section 4.5.1.4. The only difference in the implementation was the script and model used to gather and house the rain level data. However, after a discussion with my project supervisor, the rain fall data and graph was removed from the website. This is because the rain level data was not as good an indication of flooding compared to river levels. Periods of heavy rainfall do not always indicate a flood risk, unlike high river levels which do indicate a risk. The rain level data was also only relevant to Leeds, unlike the river level data that was taken from various locations throughout West Yorkshire.

4.5.2 Home Page Update

Up until this stage in the project the home page being displayed on the website was the same as in Fig 13. In this iteration a more appropriate home page was created and developed. The home page now displays any current flood warnings within 20km of Leeds, and information about the website

in Bootstrap collapsible boxes. This new home page can be seen in Fig 39 and 40.

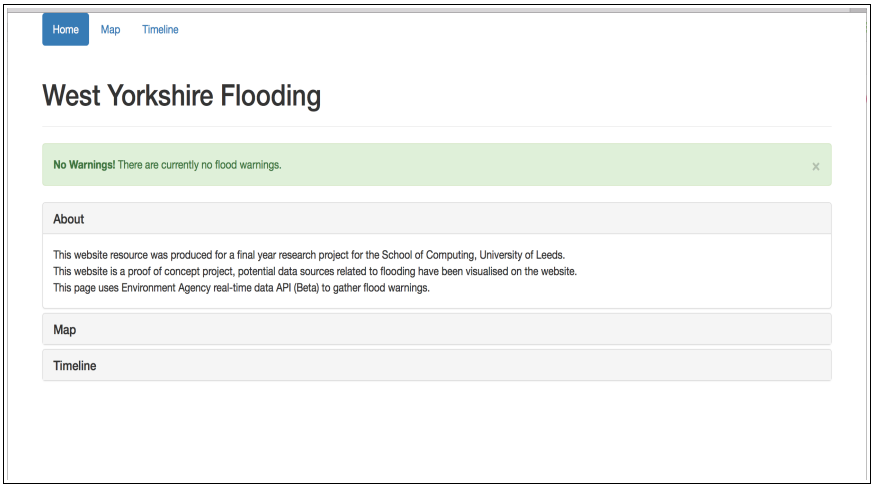


Fig. 39 Updated Home Page

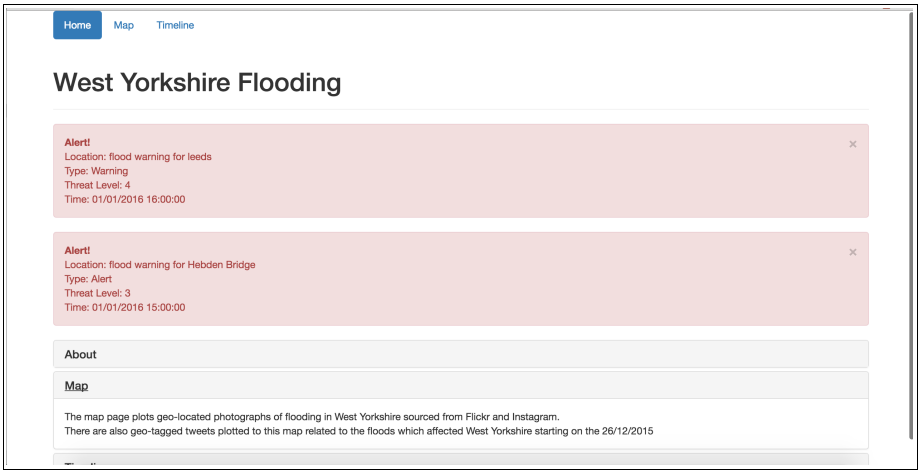


Fig. 40 Homepage with warnings using test JSON data

The flood warnings in Fig 40 are generated by the FloodAreas script documented in Section 4.3.3. The index view in Django's views.py file calls the *searchArea* function in the FloodAreas script to search for any flood warnings within 20km of Leeds. The Python list generated by this function is then passed to the index template from the index view. The template page then checks for an empty list. If the list is empty the message in Fig 39 is shown, otherwise, the warnings in the list are displayed as seen in Fig 40. This means that whenever a user accesses the home page the *searchArea* function is called, this provides the user with the latest flood warnings.

Chapter 5: Evaluation

This chapter details the evaluation of the project. User evaluations of the final version of the website were completed. This was followed by a self evaluation of the website and the project, and a discussion of possible extensions for the project, and lastly, a final conclusion and reflection.

5.1 User Evaluation

User evaluations were completed to gauge the functionality and user experience of the website. The methods, procedure, results, and conclusion are detailed below. The user evaluation conclusion focuses on what was learnt from the user evaluations and the possible improvements that could be made based on these evaluations.

5.1.1 Method and Procedure

Two participants were recruited to complete an online survey to assess their understanding of the website, it's purpose, and their overall experience of using the website. The survey consisted of questions about the design, functionality, and purpose of the website. The survey that was used can be seen in Appendix C. Lastly, at the end of the survey, the participants were given the opportunity to suggest possible improvements that could be made to the website.

5.1.2 Results

The questions asked and the participant's responses can be found in the tables below.

Participant 1:

Question	Answer	Additional Comments
You have a clear understanding of the website's purpose	Strongly Agree	-
The website is easy to navigate	Agree	It is easy, but it could be designed in a simpler way so it doesn't take so much time to understand where to navigate and why. For example having one bold and concise message on the homepage, with three direct links in the centre of the page.
You have a better understanding of floods affecting West Yorkshire.	Strongly Agree	-
What do you think of the website design and organisation?	As mentioned above, the first message should be simpler and shouldn't take so long to understand the purpose of the site. Also instead of 'timeline' I might use 'live updates' or something so it is clearer what the page offers.	N/A
Is the Map page easy to use? Did you have any difficulties?	yes. no issues. it might be interesting if it was possible to integrate also news updates from google with links to the source, as well as personal twitter feeds. I also found some difficulties when viewing the photos. I had to drag the map down to fully view the image.	N/A

Is it clear what data is available on the Map page?	Yes	-
Is the timeline page easy to use? Is the data displayed clearly and do you understand how to use the different filters available?	Yes, see question 4.	N/A
Are there any improvements you could suggest for the website?	Functionality is fine, I think from a design perspective it could be improved to enhance the user interface/ experience	N/A

Participant 2:

Question	Answer	Additional Comments
You have a clear understanding of the website's purpose	Strongly Agree	-
The website is easy to navigate	Strongly Agree	-
You have a better understanding of floods affecting West Yorkshire.	Agree	If plotly allows, maybe add the data for 16-25th December as well? To understand how the floods escalated
What do you think of the website design and organisation?	Neat and easy to use.	N/A
Is the Map page easy to use? Did you have any difficulties?	Would be nice to hover of the location pins and know what year the photo inside is from, especially when a large number of pins from different years are highly concentrated in a certain region.	N/A
Is it clear what data is available on the Map page?	Yes	A legend would be even better
Is the timeline page easy to use? Is the data displayed clearly and do you understand how to use the different filters available?	The river filters are easy. The Flood Event button takes a few minutes to drop down though.	N/A
Are there any improvements you could suggest for the website?	-	-

5.1.3 User Evaluation Conclusion

From the results, it can be seen that the information displayed on the home page is sufficient enough to explain the purpose of the website and how to use it. Although, the explanation of the website's purpose could be more concise and explicit. Both the participants have also left the website with a greater knowledge about the flooding which has affected West Yorkshire. This shows that the data and its visualisation helped the users learn new information about the floods. Each participant also thought that the website design, navigation, and user experience, could be improved. The participants also thought that the functionality of the timeline page could have been better developed, this is also covered in Section 5.2. These issues would have to be addressed if a fully developed website was to be implemented.

The second participant also stated that the river levels between the 16th December 2015 and 26th December 2015, should be added to the timeline graph for a greater understanding of how the floods escalated. This data is actually present on the graph but the labelling of the graph could be improved to make this more clear. It was also possible that the participant was viewing a flood

event, date filtered, river level graph. If this was the case, it could provide a better visualisation of the floods if the river level data of a few days before and after a flood event was included on the date filtered graphs.

The first participant suggested including news stories related to the floods on the website. This was overlooked as a potential data source, but it could prove to be an interesting and informative source. Researching and integrating that data source could be a good possible extension of the project.

The design of the map page could be improved as well. participant two's idea of adding a hover function over the icons to display the date of the popup is interesting. The second participant seemed to want a way of filtering the markers by their date. This could be solved by adding a date filtering dropdown box to the map (similar to the flood events on the timeline page), or the solution the participant suggested could be implemented. Participant one also stated they had trouble viewing the images on the map, and they had to drag the map down to properly view the image. If more time was available this would be fixed by having the map re-centre on any popup that is clicked, so the popups contents are centred on the page.

Overall, it seems the participants had a positive experience of the site and the main pitfalls was the design and user experience. This would have to be addressed if the website was to ever be released as a full software product.

5.2 Self Evaluation

This section details the self-assessment of the website and the project as a whole. It details any problems or flaws that were found with the website and their possible improvements.

5.2.1 Website Performance Evaluation

The performance of the website was evaluated using Pingdom's full page test tool[45]. This tool analyses the speed and size of a web page and then ranks each page against every other tested website. It also provides a breakdown of the factors affecting the page speed and any possible improvements that could be made. The loading time of a page is important. A study on tolerable waiting time[46] revealed that a waiting time of 2 seconds or less was needed to “ensure smooth” interactions between a user and a web page. It also goes on to state that a user would leave a page that took longer than 15 seconds to load, if the page did not provide any feedback.

5.2.1.1 Home Page Performance

The home page performed well on the Pingdom test and had a load time of 1.10 seconds and a page size of 192kB; this ranked it as faster than 88% of all websites tested by the tool.

5.2.1.2 Map Page Performance

The map page was tested and had a load time of 2.68 seconds and a page size of 1MB; this ranked the page faster than 59% of all tested websites. After examining the breakdown provided, it appears the extra load time is caused by requesting the map tiles from the MapBox server. Each tile request has a wait time of around 300ms. According to Pingdom this page speed could be improved by implementing browser caching for these map tiles.

5.2.1.3 Timeline Page Performance

The Timeline page performed poorly on these tests. It had a load time of 24.22 seconds and a page size of around 10MB. This meant that the timeline page was slower than 93% of all tested websites. After examining the breakdown seen in Fig 41, it can be seen that the large load time is caused by a wait time of 19.76 seconds from the server hosting the project's website. The extra 5 seconds of load time appears to be caused by the wait time associated with requesting data from Flickr, Instagram, and Twitter.

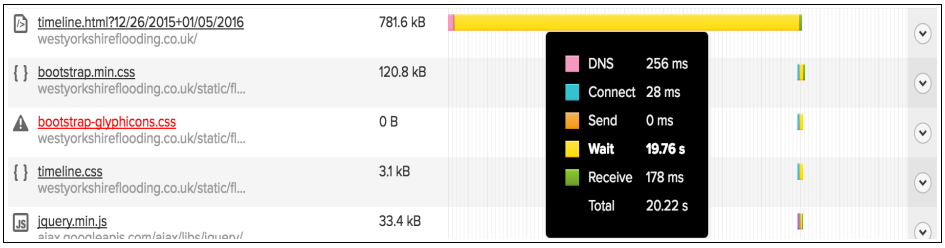


Fig. 41 Timeline page speed breakdown

This 5 seconds could be improved by implementing browser caching for the images displayed on the timeline. The page size and speed could also be reduced by using Jquery to only load extra tweets and images on request by the user. For example, Jquery could be used to load more of the tweets and images as the user scrolls down the page. This would remove the need to load all of the data during the initial page load, which would reduce the size and load time of the page.

However, the serious problem is the website server wait time of 19.76 seconds. This extra load time only occurred after implementing the river level data on the website, which was completed late in the project. After reviewing the project code, the problem was narrowed down to the *getFloodEvents* function in views.py and the Django query used to retrieve the river level data. As the river level data-set is rather large the Django query can take some time to complete. This could be reduced by possibly tweaking the query used to request the data, or perhaps, using raw SQL to make the request rather than the built in Django functions. The flood events function can also take a while to return the data that is to be passed to the timeline template. This could possibly be improved by moving the *getFloodEvents* function from the server to the client. The data required by

the function could first be passed to the template, then the flood event function can be executed on the client side as a JavaScript function after the page has loaded.
To avoid losing users due to the excessive load time a landing page could also be implemented; this page could have a progress bar, or a loading wheel to provide feedback to the user. This would inform the user that the page is still being processed and that it hasn't crashed.

5.2.2 Website Display Evaluation

The display and usability of each page of the website was evaluated for desktop and mobile devices. A 15inch Macbook Pro was used to test the desktop display and a 4.6inch mobile was used to test the mobile display.

5.2.2.1 Home Page Display

The home page performed well on both devices. The buttons and navigation bar was easy to use on each device and the text information was also easy to read. The home page was also responsive to any resizing of the screen on a desktop device.

5.2.2.2 Map Page Display

The map page displays and resizes well on the desktop device but a few problems were encountered on the mobile device. It can be a little difficult to use the navigation bar on a mobile device with a relatively small screen, another problem is that the map does not fill the mobile screen and the popups can be quite difficult to click on due to their small size. This could be fixed by changing the size of the leaflet icons for mobile devices. An example of this mobile display can be seen in Fig 42.

5.2.2.3 Timeline Page Display

The timeline page displays and resizes fine on the desktop screen but the mobile device suffers from similar problems as the map page. The navbar, tweets, images, and graph appear rather small, and the screen has to be zoomed in to see any proper detail. This could be fixed by reconfiguring Bootstrap to scale the mobile pages with a different display for the timeline. These problems do not affect the usability of the website to greatly, however, if a fully developed website was to be created these minor problems would have to be rectified. Examples of the timeline page can be seen in Fig 42.

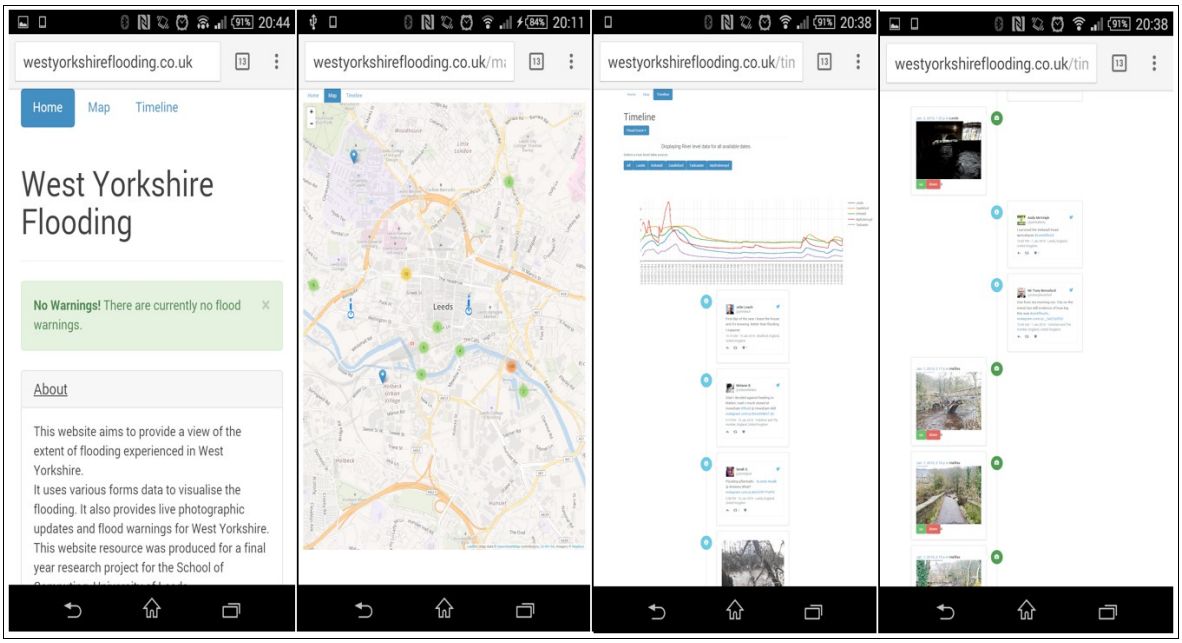


Fig. 42 Home, map and timeline pages on mobile display

5.2.3 Meeting The Requirements

Investigate publicly available photographic, social media and environmental data:

This objective was completed during Chapter 2 and 4; the research, design, and implementation of the Flickr, Twitter, Instagram, River Levels, and Flood warning scripts covers this topic.

Investigate similar existing solutions to the problem:

This objective was completed in Chapter 2; the features and design of Peta Jakarta was investigated.

Implement different data visualisation tools and methods:

This objective was met in Chapter 4; the map page, timeline, and river level graphs are some of the visualisation tools implemented in this project.

Investigate and if possible, implement a cloud based vision API:

This objective was only partially completed. Cloud based vision APIs where investigated in Chapter 2 but they proved to be very poor at detecting flooding images. However a cloud vision solution for removing harmful or offensive images could have been implemented if more time was available.

Design and develop a website to display and organise the visualised flooding data:

This objective was also concluded in Chapter 3 and 4; the development tools used to create and the website is outlined in Chapter 3, where as the design and implementation of the website is detailed in Chapter 4.

5.2.4 Possible Extensions

The website created for this project is not a product ready website. A lot of extra work is still required to make it a fully functional website. Some extra features or data sources could still be implemented, and some of the improvements needed have been listed as part of the self and user evaluations, however, some other future work and possible extensions are:

- Develop a fast, responsive, and functional website with a UI that works well on all devices.
- Extra filtering on the Map page, such as by date or data type.
- Update DB class to use Django models to update the database, this will be useful if the website is to be moved to a new server and database.
- Add river level graph popups to the map page by using the latitude and longitude of each river gauge to plot them on the map. This could give the user a better idea of what areas the river levels apply to.
- Implement “cron jobs” on the website server to execute the Flickr and Flood scripts daily or when a flood is happening (they currently have to be executed manually).
- Design Twitter and river level scripts to gather data daily using API's specified in the background research, they currently only gather data from specific data-sets.
- Investigate and implement LiDAR data[47] on the website to help model the flooding.

5.3 Conclusion and Reflection

Overall, the results of this project show that a fully developed website resource to analyse and inform people of the floods affecting West Yorkshire, which can also provide live flooding updates for West Yorkshire, is viable. This project clearly demonstrates that a large variety of quality data is available to model and track any floods affecting West Yorkshire. This project also demonstrated that there are tools available to effectively visualise these data sources in a way that informs and educates the user of the website.

Personally, I have deeply enjoyed completing this project and I have learnt many new skills in the process, such as, web development, big data collection and processing, as well as gaining a greater understanding of the flooding affecting West Yorkshire.

References

1. PetaJakarta.com, About Section, Accessed Online 21 April 2016, https://petajakarta.org/banjir/en/
2. PetaJakarta.com, Research Page, Accessed Online 21 April 2016, https://petajakarta.org/banjir/en/research/
3. The Open Data Institute, What is the Definition?, Accessed Online 23 April 2016 http://theodi.org/what-is-open-data
4. Flickr.net, Blog Post, Accessed Online 23 April 2016, http://blog.flickr.net/en/2015/06/10/thank-you-flickr-community/
5. Flickr.net, Blog Post, Accessed Online 23 April 2016, http://blog.flickr.net/en/2015/05/07/flickr-unified-search/
6. Flickr.com, Things to do, Accessed Online 23 April 2016, https://www.flickr.com/help/guidelines/
7. Flickr.com, API Developer Guide, Accessed Online 23 April 2016, https://www.flickr.com/services/developer/api/
8. Flickr.com, Search API Documentation, Accessed Online 23 April 2016, https://www.flickr.com/services/api/flickr.photos.search.html
9. Flickr.com, Location API Documentation, Accessed Online 23 April 2016, https://www.flickr.com/services/api/explore/flickr.photos.geo.getLocation
10. Flickr.com, Get Info API Documentation, Accessed Online 29 April 2016, https://www.flickr.com/services/api/explore/flickr.photos.getInfo
11. Flickr.com, URL Guidelines, Accessed Online 24 April 2016, https://www.flickr.com/services/api/misc.urls.html
12. Twitter.com, Advanced search tool, Accessed Online 24 April 2016, https://twitter.com/search-advanced
13. Twitter.com, Developer API Rate Limits, Accessed Online 26 April 2016, https://dev.twitter.com/rest/public/rate-limits
14. Twitter.com, Search API Documentation, Accessed Online 26 April 2016, https://dev.twitter.com/rest/public/search
15. Instagram.com, Our Story, Accessed Online 27 April 2016, https://www.instagram.com/press/?hl=en
16. Instagram.com, Permission Changes, Accessed Online 27 April 2016, https://www.instagram.com/developer/changelog/
17. Instagram.com, API Rate Limits, Accessed Online 27 April 2016, https://www.instagram.com/developer/limits/
18. Instagram.com, Recent Media API, Accessed Online 27 April 2016 https://www.instagram.com/developer/endpoints/media/
19. Data.gov.uk, Flood Warnings API, Accessed Online 27 April 2016, http://environment.data.gov.uk/flood-monitoring/doc/reference#flood-warnings
20. Data.gov.uk, River Level Readings API, Accessed Online, 29 April 2016, http://environment.data.gov.uk/flood-monitoring/doc/reference#readings
21. Microsoft.com, Image Analysis API, Accessed Online, 22 April 2016, https://www.microsoft.com/cognitive-services/en-us/computer-vision-api/documentation/analyzeimage
22. Microsoft.com, Image Categorisation, Accessed Online 22 April 2016, https://www.microsoft.com/cognitive-services/en-us/computer-vision-api/documentation/images/86categories
23. Google.com, Cloud Vision API, Accessed Online 22 April 2016, https://cloud.google.com/vision/

24. Djangoproject.com, Meet Django, Accessed Online 04 May 2016, https://www.djangoproject.com/
25. Djangoproject.com, Models Documentation, Accessed Online 04 May 2016, https://docs.djangoproject.com/en/1.9/topics/db/models/
26. Djangoproject.com, Views Documentation, Accessed Online 04 May 2016, https://docs.djangoproject.com/en/1.9/topics/http/views/
27. Djangoproject.com, Templates Documentation, Accessed Online 04 May 2016, https://docs.djangoproject.com/en/1.9/topics/templates/
28. Getbootstrap.com, Compatible Browsers, Accessed Online 04 May 2016, http://v4-alpha.getbootstrap.com/getting-started/browsers-devices/
29. Getbootstrap.com, Home Page, Accessed Online 04 May 2016, http://getbootstrap.com/
30. Ofcom August 2015, Communications Market Report, Section 5.2.3, Take-up and use of internet-enabled devices, Accessed Online 06 May 2016 via: http://stakeholders.ofcom.org.uk/binaries/research/cmr/cmr15/CMR_UK_2015.pdf
31. Digitalocean.com, Django one click install Information, Accessed Online 05 May 2016, https://www.digitalocean.com/features/one-click-apps/django/
32. Python-requests.org, Documentation of Requests library, Accessed Online 04 May 2016, http://docs.python-requests.org/en/master/
33. Google.com, Geocode API Documentation, Accessed Online 01 May 2016, https://developers.google.com/maps/documentation/geocoding/intro#Geocoding
34. Mapbox.com, API and Library Documentation, Accessed Online 01 May 2016, https://www.mapbox.com/mapbox-gl-js/api/
35. Leafletjs.com, Overview, Accessed Online 01 May 2016, http://leafletjs.com/index.html
36. Mapbox.com, Marker Cluster Documentation, Accessed Online, 03 May 2016, https://www.mapbox.com/mapbox.js/example/v1.0.0/leaflet-markercluster/
37. Bootsnip.com, Bootstrap Timeline template, Accessed Online 02 May 2016, http://bootsnipp.com/snippets/featured/timeline-responsive
38. Readthedocs.io, Simple Json Documentation, Accessed Online 04 May 2016, http://simplejson.readthedocs.io/en/latest/
39. Twitter.com, Oembed API Documentation, Accessed Online 03 May 2016, https://dev.twitter.com/rest/reference/get/statuses/oembed
40. Crummy.com, Beautiful Soup Documentation, Accessed Online 04 May 2016, https://www.crummy.com/software/BeautifulSoup/bs4/doc/
41. Data.gov.uk, River Level Archive, Accessed Online 08 May 2016, http://environment.data.gov.uk/flood-monitoring/archive
42. Data.gov.uk River Measures API, Accessed Online 08 May 2016, http://environment.data.gov.uk/flood-monitoring/doc/reference#measures
43. Plot.ly, Scatter Graph and Library Documentation, Accessed Online 07 May 2016, https://plot.ly/javascript/line-and-scatter/
44. Leedsdatamill.org, Rain Level Data-Set, Accessed Online 05 May 2016, http://leedsdatamill.org/dataset/pottery-fields-rain-gauge-rainfall-data
45. Pingdom.com, Page Test tool, Accessed Online 06 May 2016, http://tools.pingdom.com/fpt/
46. Fiona Fui-Hoon Nah,Study on tolerable waiting times, Section 5, Accessed Online via: http://sighci.org/uploads/published_papers/bit04/BIT_Nah.pdf
47. Gov.uk, LiDAR data and Flood Risk, Accessed Online 07 May 2016, https://environmentagency.blog.gov.uk/2015/06/16/free-mapping-data-will-elevate-flood-risk-knowledge/

Appendix A: External Materials

Some external HTML code has been used to create the timeline and map pages, however these sources have been referenced in the report.

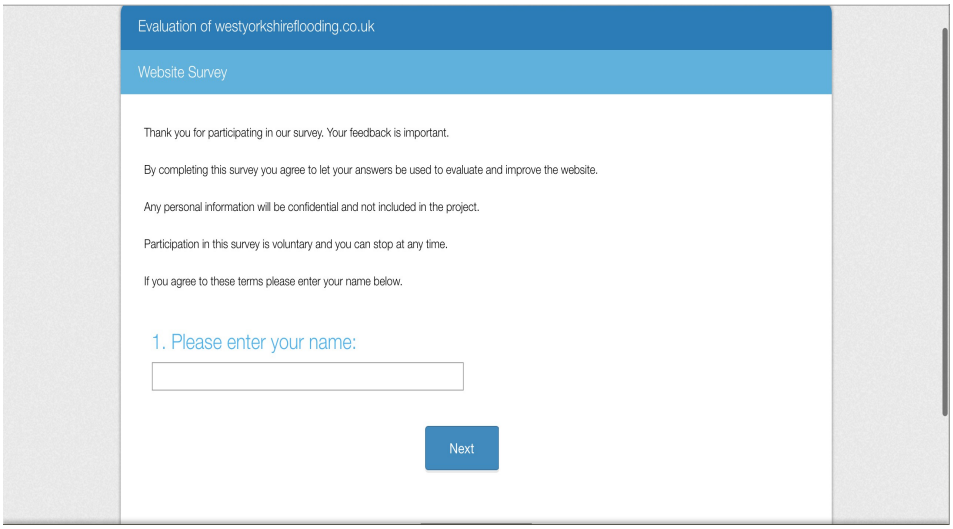
External data-sets have also been used such as the Tweets data-sets provided by Andy Burgin at the Flood hack and the River Levels data-set provided by the UK Government. These data-sets have either been credited in the report or on the website.

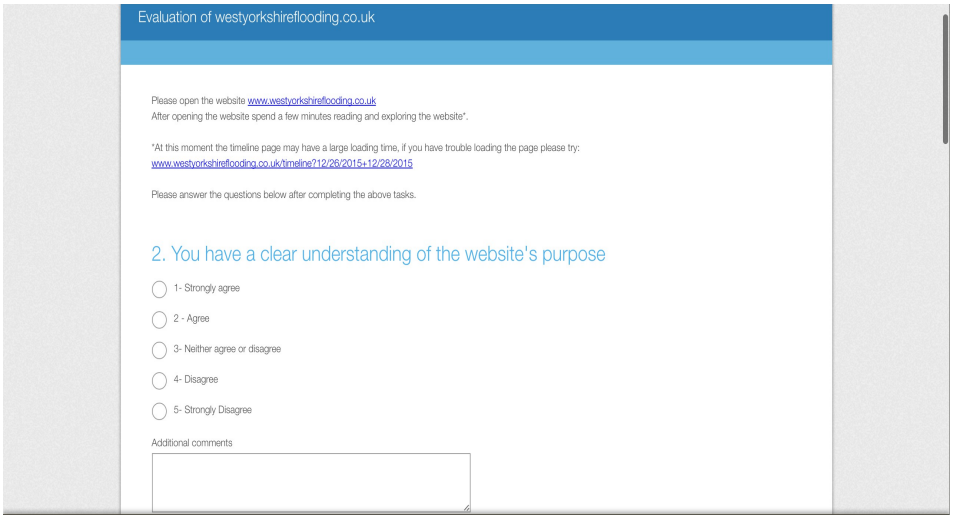
Fig. 3 Is taken from the Peta Jakarta research page: https://petajakarta.org/banjir/en/research/

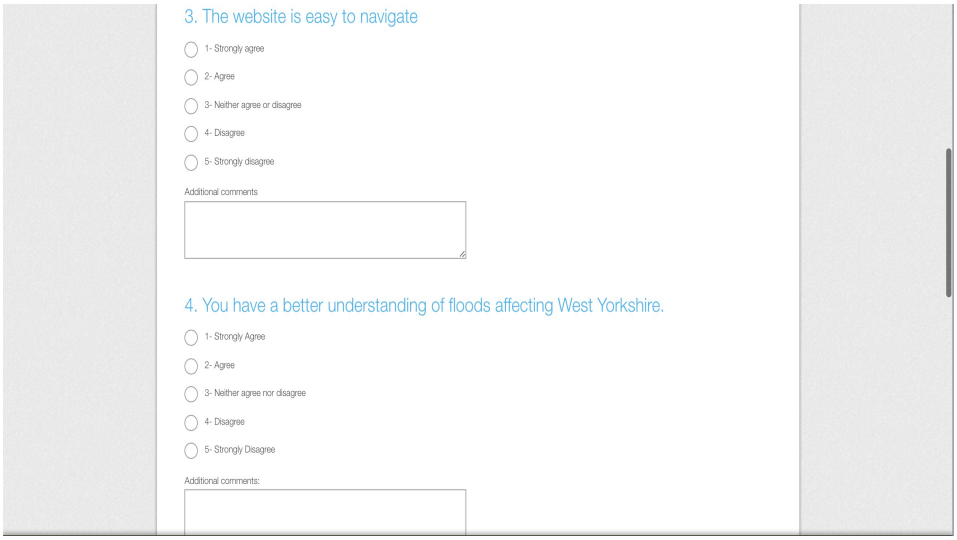
Appendix B: Ethical Issues

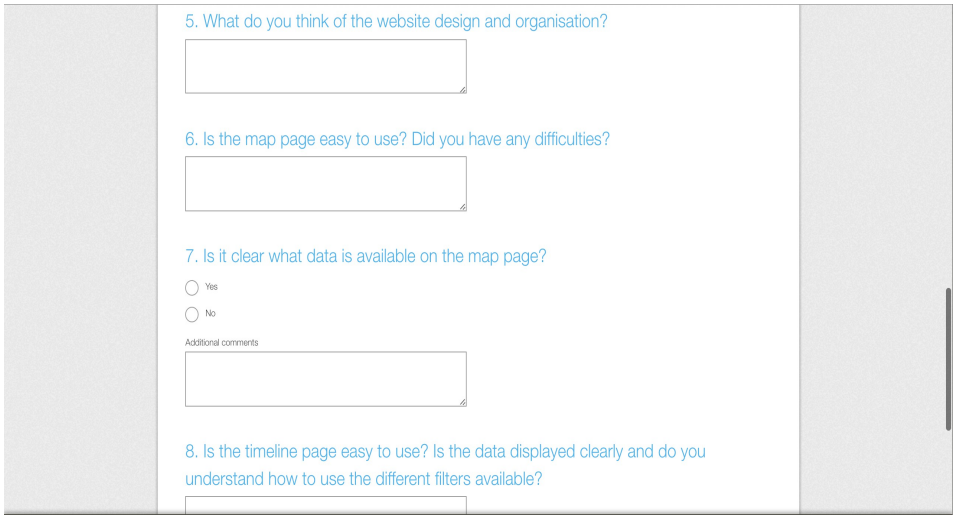
Both participants who took part in the user evaluations agreed that their evaluations could be used in this report. The Participants also agreed to allow their feedback to be used for analysis of the website and that any private information would be confidential and they would remain anonymous. All the data used in this project was publicly available and free to use; no private data has been used. Credit of the data has been given when it is needed, and the website homepage provides an email which can be contacted if any of the data owners wish for their content to be removed from the website.

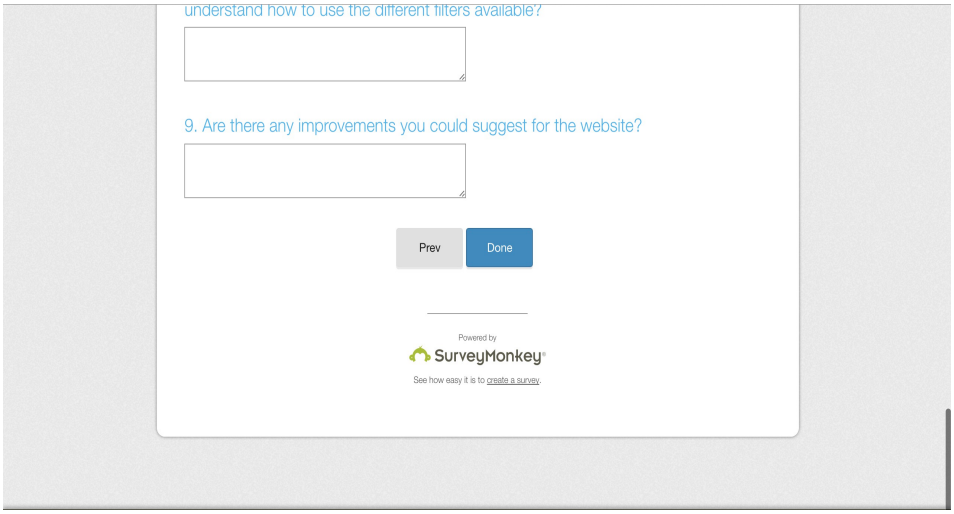
Appendix C: Online Survey








5. What do you think of the website design and organisation?
6. Is the map page easy to use? Did you have any difficulties?
7. Is it clear what data is available on the map page?
Yes
No
Additional comments
8. Is the timeline page easy to use? Is the data displayed clearly and do you understand how to use the different filters available?


understand how to use the different filters available?
9. Are there any improvements you could suggest for the website?
Prev
Done
Powered by
SurveyMonkey
See how easy it is to create a survey.