What happens in Glasgow?

Web map framework comparison from dynamic data visualisation perspective

Kornel Kotan

August 2016

This dissertation was submitted in part fulfilment of requirements for the degree of MSc Advanced Software Engineering

DEPT. OF COMPUTER AND INFORMATION SCIENCES

UNIVERSITY OF STRATHCLYDE

AUGUST 2016

DECLARATION

This dissertation is submitted in part fulfilment of the requirements for the degree of MSc Advanced Software Engineering of the University of Strathclyde.

I declare that this dissertation embodies the results of my own work and that it has been composed by myself. Following normal academic conventions, I have made due acknowledgement to the work of others.

I declare that I have sought, and received, ethics approval via the Departmental Ethics Committee as appropriate to my research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to provide copies of the dissertation, at cost, to those who may in the future request a copy of the dissertation for private study or research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to place a copy of the dissertation in a publicly available archive.

(please tick) Yes [✓] No [ ]

I declare that the word count for this dissertation (excluding title page, declaration, abstract, acknowledgements, table of contents, list of illustrations, references and appendices) is 17,932 words.

I confirm that I wish this to be assessed as a Type 1 2 ③ 4 5 Dissertation (please circle)

Signature: Kornel Kotan

Date: 29/08/16

Abstract

When it comes to web mapping it is always a pressing challenge to decide which mapping technology fits best the requirements. There are numerous frameworks on the market all having advantages and disadvantages which make them suitable for some purposes better than for the others. There are projects comparing frameworks, but no research has been made on finding the best framework for displaying changing data. This research focused on comparing three web mapping frameworks from dynamic data visualisation perspective: Leaflet, Openlayers and Google Maps. The comparison included both developer's and user's perspective.

To provide a platform for the comparison two maps were built, which displayed different dynamically changing data. The first visualised live flight traffic data above Europe while the second displayed live road traffic events and the car park occupancy levels of Glasgow. The research also measured the effectiveness of the dynamic data visualisation techniques used. The maps used various type of live data, for which different techniques were used such as heatmaps, moving icons, appearing and disappearing icons. Specific constraints were set which the maps needed to meet. The developer side comparison was based on how difficult it was to meet the constraints. To determine which map is the best from the users' perspective a questionnaire was used. The research provided detailed comparisons of the frameworks from dynamic data visualisation perspective as well as revealed the advantages and disadvantages of the frameworks. As an output of the research the effectiveness of the different visualisations was worked out. Also the best frameworks from both developer and user perspectives were identified.

Acknowledgements

I would like to thank to my supervisor Dr. Marc Roper for his guidance throughout researching this dissertation.

Thank you also to my friends and colleagues for providing answers to the questionnaire and supporting me in different ways.

Finally, thanks to my family, for backing me up during the whole year.

Table of Contents

Abstract iii
Acknowledgements iv
Table of Contents v
List of Figures vii
1 Introduction 1
1.1 Background and Context 1
1.2 Scope and Objectives 1
1.3 Achievements 2
1.4 Overview of Dissertation 3
2 State of the art 4
2.1 Online maps 4
2.2 Spatio-temporal data 4
2.3 Research review 5
2.4 Related Applications Review 10
3 Methodology 11
3.1 Methodology justification 11
3.2 Data gathering 11
3.3 Overview and Planning 13
3.4 Development approach – Development strategy 16
3.5 Design 17
3.5.1 Server side architecture 17
3.5.2 Client side architecture 19
3.6 Methodology of comparison 19
4 System documentation 21
4.1 Development environment 21
4.2 System implementation 22
4.2.1 Web service 22
4.2.2 Data collection services 29
4.2.3 Maps – Client Side 29
4.2.3.1 Google Maps 30
4.2.3.2 Leaflet 34
4.2.3.3 Openlayers 38
4.2.3.4 User comparison maps 43

4.2.4 Deployment 44
5 Framework evaluation and comparison 45
5.1 Origin and license 45
5.2 API and support 46
5.3 Matching the constraints 48
5.4 Questionnaire 52
6 Results, conclusion and recommendation 58
6.1 Summary 58
6.2 Key Findings 58
6.3 Weaknesses of the study and recommendations 60
7 Bibliography 62
Appendix 1 – Questionnaire 64

List of Figures

List of Code Snippets

Code Snippet 1 – package.json 23
Code Snippet 2 – Module imports 24
Code Snippet 3 - Application configuration 24
Code Snippet 4 – Application routing 25
Code Snippet 5 - Error handling 25
Code Snippet 6 – Error handling configuration 25
Code Snippet 7 – Database scheme 26
Code Snippet 8 – Response parsing 27
Code Snippet 9 - GeoJSON parsing 27
Code Snippet 10 - Algorithm 28
Code Snippet 11 – Framework import 30
Code Snippet 12 – Map initialisation 30
Code Snippet 13 – Heatmap initialisation 31
Code Snippet 14 - Filtering negative values 31
Code Snippet 15 - Parameterised endpoint call 33
Code Snippet 16 - Map initialisation 35
Code Snippet 17 – Tile loading 35
Code Snippet 18 - Popup binding 36
Code Snippet 19 - Updating heatmap 37
Code Snippet 20 - Map initialisation with transformation 39
Code Snippet 21 - Popup listener 40
Code Snippet 22 - Setting plane orientation 41

List of Images

Figure 1 - Interactive OD map 8
Figure 2 - Geofabrik map compare 10
Figure 3 - System architecture 16
Figure 4 - Web service structure 22
Figure 5 – Google Maps Car park heatmap 32
Figure 6 – Google Maps flight map 33
Figure 7 – Google Maps Combined car park heatmap and traffic events map 34

Figure 8 - Leaflet flight map 36
Figure 9 - Leaflet heatmap 37
Figure 10 - Leaflet combined car park heatmap and traffic events map 38
Figure 11 - Openlayers heatmap 39
Figure 12 - Openlayers flight map 41
Figure 13 - Openlayers combined car park heatmap and traffic events map 42
Figure 14 - Glasgow user comparison map 43
Figure 15 - Flights user comparison map 44

List of Questionnaire Questions

Question 1 53
Question 2 53
Question 3 54
Question 4 54
Question 5 55
Question 6 56
Question 7 56
Question 8 57

List of Tables

Table 1- API and support comparison 48
Table 2 - Constraint based comparison 52

1 Introduction

1.1 Background and Context

When an interactive web based map needs to be created it is always a pressing challenge to decide which mapping framework to use. There are numerous technologies available on the market which provide slightly different features. The frameworks all have different advantages and disadvantages which make them more suitable to some requirements than others. Research has been made on the comparison of different mapping frameworks from numerous perspectives, such as API compatibility or drawing in the overlays. However, since the technologies are quite new and they evolve fast, there are many other perspectives to be considered.

The evolution of networking allowed data providers to continuously update their data, providing live updates of the underlying information. If the data has location information attached, it is called spatio-temporal data which can be visualised on maps to make the interpretation easier. Visualising spatio-temporal data is a relatively new field of research. The aim of this project is to compare map frameworks and find the one which ultimately provides the best and most straightforward way for visualising dynamically changing spatio-temporal data.

1.2 Scope and Objectives

The project will focus on comparing three web mapping frameworks, Leaflet, Openlayers and Google Maps. Each framework will be analysed from a dynamic visualisation perspective to determine the one which is most suitable for this purpose. To provide a platform for the comparison, two maps will be built which will display different dynamically changing data, the first will visualise live flight traffic data above Europe while the second will display live road traffic events and the car park occupancy level of Glasgow. Both maps will be implemented with all three frameworks and to make them easier to review, the same type of maps will be displayed on the same web page. The research will also try to measure the effectiveness of the dynamic data visualisation techniques used. The maps will use different types of live data for which different techniques will be used, such as heatmaps, moving icons, appearing and disappearing icons.

The comparison will include both the developer's and the user's perspective. For the developer side, each framework will be required to meet certain evaluation constraints. After

the development the developer will rank how difficult it was to write the code to meet the constraints based on the results. These numbers will be aggregated with general features of the frameworks, such as support and API complexity, to find the final winner. The user comparison will be based on a questionnaire which the users will be asked to complete after they have spent some time interacting with the map. The questions will focus on collecting subjective opinions on the different frameworks and the effectiveness of the visualisation methods. The users will be asked to rank the map implementations by how much they liked them. The effectiveness of the visualisation techniques will be measured in a similar way.

Based on the outcome of this research, one framework will be chosen from both developer and user perspective as the best web mapping framework for dynamic data visualisation. The visualisation techniques will also be evaluated based on the questionnaire.

With regard to the previous, the research questions of the project are as follows:

- Which framework is the best for dynamic data visualisation from the developer's perspective?
- Which framework is the best for dynamic data visualisation from the user's perspective?
- Which methods are the best for visualising the dynamically changing data?

1.3 Achievements

For each framework, a detailed comparison was made of the dynamic data visualisation from the perspective of both user and developer. The comparison was based on different methodologies which helped analyse the frameworks on different levels. It included the development of an application to compare the frameworks side by side which provided the basis of the developer side comparison along with the literature reviews. The comparison from the user point of view was based on a questionnaire which was filled out by users after their interaction with the map. The questionnaire also collected some feedback on the effectiveness of the different dynamically changing data visualisations.

The research revealed the strengths and weaknesses of each framework, demonstrated how difficult it was to develop the various features which are required for dynamic visualisations and what support was provided by each framework. The questionnaire indicated which framework was preferred by the users and which visualisation type they

considered to be effective. It also suggested some possible improvements and ways in which the application can be developed further.

1.4 Overview of Dissertation

After the introduction, the second chapter is responsible for putting the reader into context with a high level introduction to the topic, followed by a summary of some reviewed literature and application.

The third chapter deals with the different methodologies which are used during the project. The visualised data is introduced among the development strategies as well. This chapter also includes an abstract planning phase, which defines the requirements against the system.

The methodology chapter is followed by a chapter which is responsible for providing a detailed insight into the actual system including the environment, the design and the implementation. In this chapter code snippets with explanation and images from the final application can be seen too.

Chapter five details the actual framework comparison by going through all the previously defined comparison methodologies. First there is a short overview of the frameworks which is followed by a comparison based on the APIs and the frameworks' support. The final two sections compare the frameworks by the developer rating and the users' feedback including some details about the questionnaire.

The final chapter is responsible for summarising the research. It contains an overview of the whole research showing the key findings and observations. The methodologies are evaluated in this chapter. The weaknesses of the study are revealed and some future research possibilities with recommendations are described.

After the final chapter the bibliography and appendixes can be found.

2 State of the art

In this chapter a high level overview of the theoretical background of the study can be read which introduces the topic, and the researches made in the area. The chapter begins by describing some key expressions and concepts which are needed to understand the following research review subchapters.

2.1 Online maps

Making maps used to be a demanding task requiring a broad knowledge in cartography and mapping technologies as well. By today thanks to the revolution of the Geographic Information Systems (GIS) there are numerous tools on the market which have simplified the map-making process (Crickard). A GIS is a system for capturing, displaying or editing location related data on the Earth's surface. It allows the users to see, analyse and understand the patterns and relationships in the data easier (GIS - geographic information systems). Today people can not only customise already existing maps, but also build their own versions much more easily than ever before. The web GIS systems have layered architecture, which means that the applications are divided into different levels of visualisation. They consist of a base, tile layer displaying the actual map tiles, and one or more overlay layers above the tile layer displaying additional information.

2.2 Spatio-temporal data

The data displayed in the overlay layers can be split into two main groups depending on the static nature of the information which it carries. The first group contains the static type of data which doesn't change neither time nor location, for example place of interest locations or static routing information. The second group consists of the dynamic, spatio-temporal data which can again be split into three subgroups based on the kind of changes occurring over time:

- Data which changes in spatial properties: location, shape and/or size, orientation, altitude, height, gradient and volume.
- Data which changes in thematic properties, values of attributes, qualitative changes and changes of ordinal or numeric characteristics value.
- Data which changes in existence in time i.e. appearance and disappearance.

(Andrienko, Andrienko, & Gatalsky, 2003)

2.3 Research review

To gain deeper understanding of the related topics, this chapter delineates and reviews a few existing research articles and applications in the fields of map framework comparison and dynamic data visualisation. Although there are various studies focusing on comparing a few of the frameworks which are used in this research, as well as few cursory online comparisons of all three technologies, to the best of my knowledge, this is the first deep comparative study focusing on the comparison of Leaflet, Openlayers and Google Maps from a dynamic data visualisation perspective.

A Comparison of Maps Application Programming Interfaces

Fernandes et al. (Fernandes, Goulão, & Rodrigues) compared the Google Maps JavaScript API, the ArcGIS API for JavaScript and the Openlayers API from a usability perspective. Since two of these frameworks are involved in this dissertation as well it is worthwhile to review their results focusing on the Google Maps and Openlayers APIs. Their goal with the comparison was to determine how complex and leveraged these map supporting Application Programming Interfaces (API) are. Since the APIs have a high effect on programmers' productivity the evaluation indicates how easy or difficult it is to build maps with the above mentioned frameworks. Two different information sources were used to analyse the APIs.

They built three map prototypes with similar capabilities to demonstrate the basic functionalities, such as zoom, pan, controls and location search, and then measured the complexity of the implementations. To determine the values for each map they created their own metrics system which was based on the constructor, function and property calls. During the development they already discovered that the Openlayers API does not support geocoding and the Google API does not provide wide support for layer management and manipulation. The latter can affect this project as well since I want to visualise dynamic data in the overlay layers.

The other information source was the quantitative evaluation of the APIs. They measured the size of the APIs, the number of objects, the methods, and the properties contained then counted how many additions or deletions were made between the different versions of the frameworks. The analysis of the dimensions of the APIs showed that Google API has less than half of the object of the Openlayers API and is much smaller in terms of the number of methods and properties as well, but thanks to these it has a reduced number of calls. The size of the Google API indicates that it is implemented on a higher abstraction level leading to

easier usage but less customisation which again shows that it can be more difficult to display customized overlay data within the framework.

The analysis of the evolution resulted in the findings that both APIs are growing in the number of added, deleted and kept objects and the migration of applications from one version to another remains maintainable. However, the Google API was the only one which did not have removals of objects between the analysed versions. This means that it performed the best in terms of retro compatibility. Since I only want to make a comparison on a single version of the frameworks this discovery will not really have an effect on my work.

Reviewing this research paper was useful for getting a high level overview of two of the APIs which I plan to work, with as well as revealing a few possible difficulties which need to be dealt with relating to the less customisable Google Maps JavaScript API. Some information was obtained as well on how steep the learning curve will be when working with the frameworks. The results indicate that implementing the dynamic visualisation with Openlayers can take more time, but probably it can easier be done than with Google Maps, where there is a chance that not all the different visualisations can be implemented.

Visualizing the Dynamics of London's Bicycle-Hire Scheme

Wood et al. carried out a research on visualising traffic flows to analyse the the status of docking-stations and the movement of bicycles in the London Bicycle Hire scheme. (Wood, Slingsby, & Dykes, 2011) Transport London (Barclays Cycle Hire Map) provides a scheme which is similar to the other schemes around the world, where the users are able to pick up a bicycle from any docking-station in London, and then drop it off at any other station after they had finished using it. One can track the real-time status with the number of bikes docked currently via the Web interface. Wood et al used this data to reach their goals, which were:

a) to allow an overview for managers for helping to keep the scheme geographically balanced
b) to help users to make the best use of of the facilities
c) to provide an overview of urban mobility and structure of the city

Two datasources were used to generate the visualisations. Firstly, they harvested the station status data every five minutes which indicated the flow but did not provide exact information about the journeys or individual behaviour. Secondly they acquired 540,000 direct, origin-destination (OD) journey data which shows where and where the bicycles were picked up and dropped off.

Since in the paper they are dealing with live visualisation of data which is dynamically changing in value reviewing this paper is useful to identify the issues and techniques of representing this type of data.

With regards to (Rae, 2009) visualising geographically heterogeneous origin destination flows can result in a less meaningful "hair ball" like structure because of the line overlapping and path sharing in big datasets. Following the guidance of Andrienko et al., Wood et al. identified three alternative methods of visualising flows:

- *Edge bundling* tries to show general trends by combining pathways which go to adjacent locations into smoothed paths. (Phan, 2005)
- *Density mapping* to reduce visual noise creates surfaces from the separate flow lines. This technique can be misleading in cases where only the origin and the destination are known since the modelled line can be interpreted as the actual route between the points (Rae, 2009).
- *Origin Destination matrix visualization* is an alternative way to solve the occlusion problem as well as giving equal weight to short and long flows too. Each cell in an origin destination matrix is symbolized with a colour representing the magnitude of flows between origin and destination in the OD pair (Wilkinson & Friendly, 2009). However, user studies made by Ghoniem et. al. (Ghoniem, Fekete, & Cstagliola, 2004) showed that from an interpretation perspective, matrix visualisation is more effective than the previous options. Origin destination matrix visualisation is not a classic mapping technique so It can not be used in this project.

The previously identified techniques used to visualise the flow data, all use the origin and destination information of the vehicles, which is significantly harder to acquire than data which only represents information about the number of vehicles in a location. These techniques can only be applied if there is access to data which not only offers information about the current value of a location, but also provides details about the origin and destination.

For representing the fullness of the stations Wood et al. used the geographic small multiples technique to create a gridded spatial tree map. They displayed the saturation of all the bicycle stations on London's map by adding squares to the map with different tones of the same colour. Then the geographical background was gradually removed and the squares were ordered into a matrix displaying the distribution and the proportion of bikes currently

docked at stations all over central London. The final result is extremely similar to a heatmap; the only difference is that there are no explicit borders of the fields on the heatmaps so the adjacent values can be washed together on different zoom levels. The gridded spatial tree map shows the changes in the number of bicycles over a time period but does not provide information about the origin or destination of the journey. Therefore, this technique could be used on less complex data without origin destination information as well.

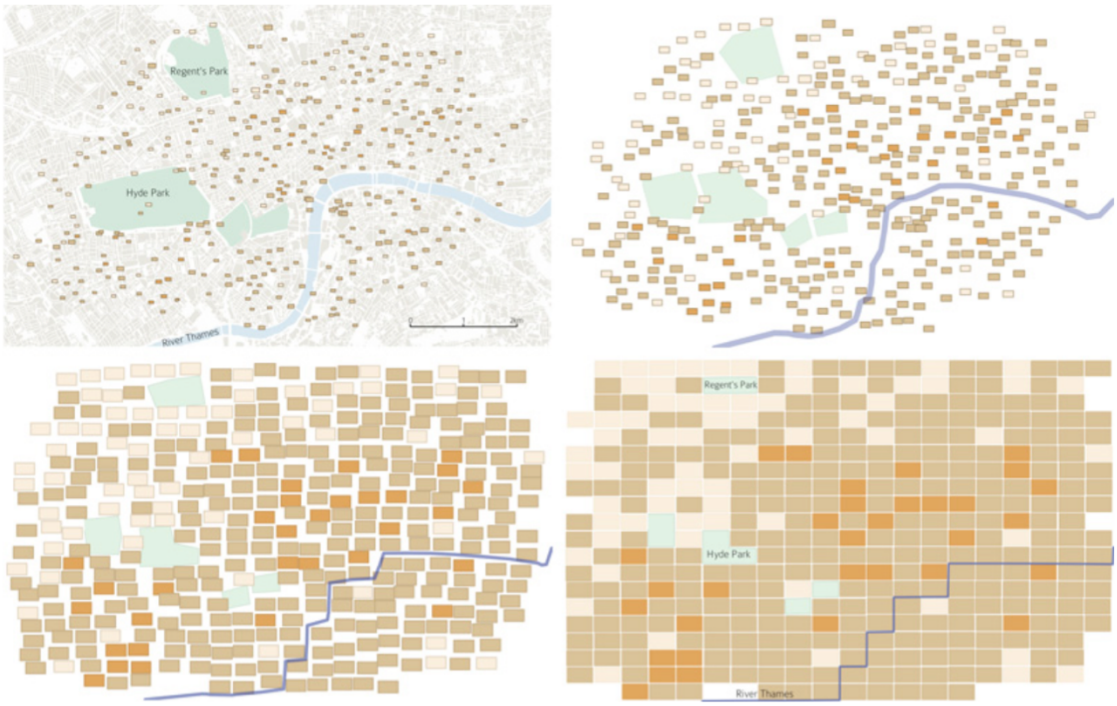


Figure 1 - Interactive OD map

To display the origin destination provided by the flow data Wood et al. extended the gridded spatial tree map to an interactive OD map (Wood, Slingsby, & Dykes, 2011). The map shows a tree map for all the cells displaying the frequency of the destination locations with different colours thus representing the flows. The visualisation helped them to reveal local patterns and source stations in the London Bicycle hire scheme. Although this technique proved to be the most representative way to visualise the flows it requires some familiarization.

Reviewing this research paper provided information about techniques to visualise data which is changing dynamically in value, focusing on a specific subtype which owns origin and destination knowledge as well. Knowledge was obtained about different visualisations with different advantages and disadvantages. This will be useful when determining which techniques should be used to visualise the data in this research project.

Testing web map APIs – Google vs Openlayers vs Leaflet

Robin Lovelace wrote a comparison on Google Maps, Openlayers and Leaflet which included a high level overview of the main attributes of the frameworks and some basic coding examples as well. (Lovelace, 2014)

With regard to Google Maps he highlighted the easy and quick implementation of the basic features and the wide range of functionalities which the framework provides, although he also mentioned the disadvantages caused by the restrictions of the private source code such as the lack of customisation in the background map tiles. He emphasized the great level of support which Google provides, along with the vast server infrastructure, the programmer team behind the software and the improving variety of static and dynamic visualisations in the framework. The latter suggests great promise in this project's spatio-temporal visualisation with the framework as well. Some other Google features are referenced in the paper too, such as the Street View and Google Earth integrations but these are not closely related to this research.

The second framework he analysed was Openlayers, which he defined as the most mature and the most widely used open source client side web mapping tool on the internet. When discussing the advantages of the framework he wrote about the wide range of functionalities in the framework. On the other hand, it was indicated that the size of his version was almost one megabyte which is not ideal for mobile applications. Although, since his review the size of the framework has been reduced but it is still much bigger than the other two frameworks. In addition, he highlighted the significant differences between the version changes of the Openlayers API as Fernandes et al. (Fernandes, Goulão, & Rodrigues) did.

Finally, he described Leaflet as fast moving and lightweight, aiming to allow the developers to quickly create well functioning maps. The number of growing plugins and the modular base of the framework was highlighted which can be useful when looking for visualisation solutions. He also appreciated the GitHub based community support and the mobile compatibility thanks to the small size of the framework.

He concludes by saying that all of the frameworks can most likely provide 80% of the required basic features for web mapping, therefore personal preferences and special requirements should decide which framework to use. Finally, he returns to the Google Maps' license disadvantages but summarized it as a good "off the shelf" quick solution. Although he considered Openlayers as a mature, big and good framework his favourite is Leaflet as the

latest and most exciting mapping technology not only for desktop browsers but for mobile devices as well.

Reviewing this paper provided a good high level overview on all the three frameworks which will be compared, and indicated a few issues and possible weaknesses of the frameworks which will need to be faced.

2.4 Related Applications Review

After reviewing the theoretical background of the research area and collecting some ideas on the goal of the research a way needed to be found to implement and compare the frameworks. While looking for applications with similar goals the Map Compare application of the Geofabrik company was found (Figure 2 - Geofabrik map compare). The main purpose of the tool is to compare different map frameworks by displaying them side into side which is exactly what this project wants to reach. The screen of the application is divided by four equal sized areas which contain maps implemented with different technologies showing the same location. This website provided useful ideas on how to display the maps. (Geofabrik)

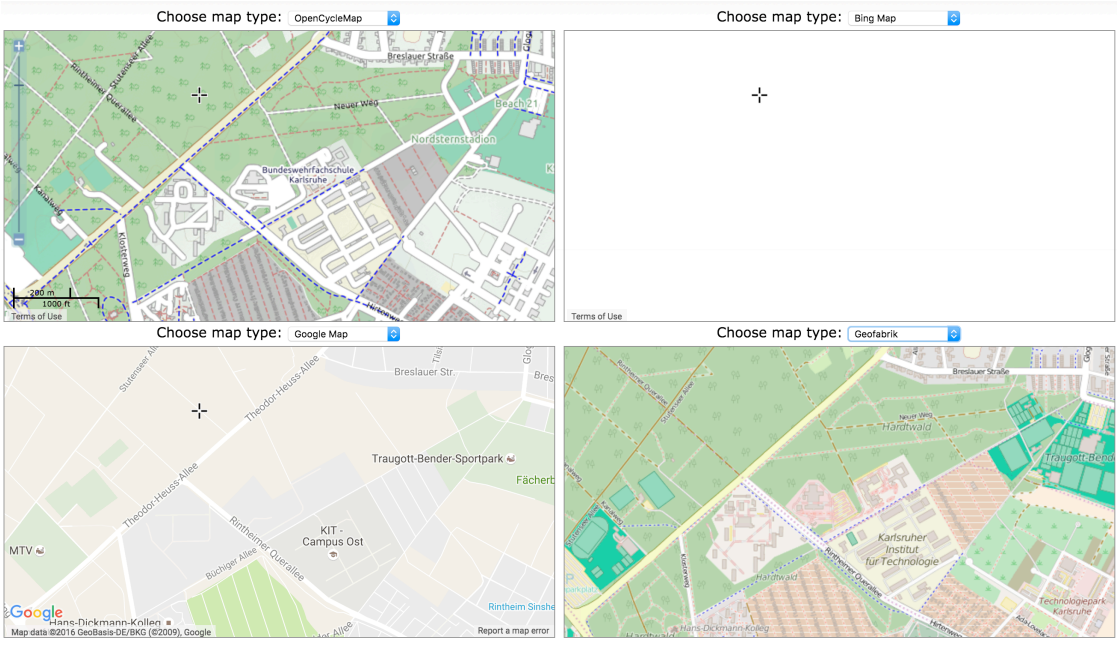


Figure 2 - Geofabrik map compare

3 Methodology

3.1 Methodology justification

The goal of the research is to compare three map frameworks based on their effectiveness in visualising dynamically changing data. The project will include a comparison from both the developer's and the end users' perspective. Since both require a different approach, various research methodologies will be used for pushing through the research. To compare different technologies from a usability perspective, the most straightforward method is to make the developers to use them. So the developers side task will include the creation of a software which will provide the platform for the qualitative research which will be based on a questionnaire. The application will visualise dynamically changing data using different frameworks and techniques, thus allowing the users to test and compare the frameworks.

The first part of this chapter is about the methodologies related to the application development, including the requirements of the system, the data which will be used for the visualisations and the strategy with the process models of the development itself.

The second subchapter presents the design of the application, highlighting the most important design decisions and describing the system's architecture.

The third part of this chapter focuses on the methodology of the comparison. After the system is built, it will be evaluated to decide how it will work with each framework, and how difficult it is to meet the requirements with each framework during the development. The best way to judge whether the visualisations and the maps serve their purpose is to expose the application for user testing and evaluation. To gather feedback from the users, a survey will be created which will be detailed in the second part of the chapter as well.

3.2 Data gathering

In order to test and demonstrate the various abilities of the different map frameworks for dealing with dynamically changing inputs, datasources will be obtained from each type of live data described in section 2.2 and they will be visualised on the maps. The three data types which change in location, value or existence as time passes, more or less cover all the use cases which a typical map framework needs to deal with in connection with dynamically changing data. To gather dynamic data, third party data sources need to be found which allow public access to their live data. In addition, since the project's title is "What happens in Glasgow", the data should be related to Glasgow in some way.

Data changing in location

In the original plan some kind of traffic data for example GPS locations provided by taxi companies, was planned to be used as the data which changes in location.

During the data exploration it unfortunately turned out that there is no company in Glasgow which provides publicly available car location data, although, a web API which provides live flight data feed was found. OpenSky Network is a community-based receiver network which continuously collects air traffic data above Europe and the east coast of the United states to share it with developers free of charge. They provide their data through various interfaces, including a REST API with a simple GET request which allows developers to retrieve all the tracked flights' details in JSON format such as flight number, origin country, current location and even the orientation of the flight. Although there is a time restriction on the API, registered OpenSky users can retrieve data with a time resolution of five seconds which means that fresh data can be obtained almost in real-time. The method to display this type of data on the maps is quite trivial since the location of the flights can be updated quite often. Icons or markers can be placed on the maps and moved to the new position when the coordinate data refreshes. To indicate the direction of the planes and make their movement smoother, animations or direction vectors could be used. (The OpenSky Network API)

Data changing in value

For the data changing in value it was planned to use a datasource which significantly changes its value during a day and represents some kind of behaviour of the inhabitants of the city. Bicycle hire docking stations' state or car parks' occupancy information is perfect for this role. Fortunately, the Glasgow council provides access to around four hundred datasets within the Open Glasgow project, allowing developers and researchers to work with local data.

On the Glasgow Open Data Portal there is a live Glasgow car park feed which provides information about the car parks' location with the name as well as description and their occupancy in Glasgow. The service collects information from sensors installed in the car parks and propagates it to an endpoint in JSON and XML formats every five minutes. Based on the work of Fernandes et al. I decided to visualise this data with the help of heatmaps representing the relative occupancies of the car parks with different colours. This way the map not only displays how many cars are parked in the parking areas but also indicates how busy the various parts of the city are during different periods of the day. (Glasgow Open Data Portal - Datasets)

Data changing in existence

Regarding the prior plans, the twitter API could have been used to get the location of the posts to display on the maps and keep them there for a few seconds before making them disappear. But while looking for the car park feed a road traffic events datasource provided by the city council was found which not only correlates better with the car park data but provides more Glasgow-specific information as well. The data follows a similar structure as the car park feed. It provides location and duration information with a description od specific road traffic events in real time. The feed is refreshed every five minutes but only available in XML format. From a data structure perspective this feed is even more useful than the twitter posts because the traffic events have different duration in time which presents additional information.

Mock data sources

Except of the previously mentioned feeds, other sources will be used during the different development phases either to prove concepts or to back up less reliable datasources. To test static visualisations on the maps, GeoJSON files were used which can be obtained from the government's open data site, containing reported rodent activities in Boston (City of Boston). The data file will be loaded into the database from where it will get served up by the web service to the browser. For testing the dynamic data concept, a website designed for developer purposes will be used. It returns only one point with its coordinates in GeoJSON format and on each request the point's coordinates change. This endpoint is perfect to test a simple movement visualisation (Wanderdrone). In the subsequent phases of development, in case of datasource outages or just to display more changes in the data recorded, data will be replayed from the database which was originally produced by the real data feeds.

3.3 Overview and Planning

Functional system requirements

To display the different visualisations not only the maps need to be implemented but numerous other hosting and data gathering infrastructures have to be developed and set up. The final system needs to be able to collect the dynamically changing data periodically from different data sources then process them and serve them to the clients for displaying in the required format. The collected data also needs to be stored for testing and for improving the visual experience by replaying the more interesting periods. One of the most important requirements of the system is to provide a platform to display the visualisations and let the

users interact with the data. This means that the infrastructure has to be able to render the map tiles and equip them with all the functionalities which are required for a web based geospatial system. To satisfy the research goals the system needs to be structured in a form which allows the users to make a comparison between the different map frameworks and visualisations.

To make the process more tangible some constraints will be defined before starting the development which will provide the basis of the comparison. All the maps need to meet these constraints and it will be tracked how difficult and time consuming it was to implement the code passing the requirement. The constraints will require the frameworks to support commonly used features and protocols for visualisation and web mapping in general.

Constraints

Nowadays GeoJSON is a widely used format in the field of web mapping. The protocol provides support for encoding a variety of geographic data structures. It can represent different types of map features like point, polygon or line and contain additional property information as well. Since GeoJSON is gaining more and more territory in web based mapping nowadays, all the maps should be able to provide some kind of support for automatic GeoJSON loading, or rendering on their API (The GeoJSON format specification).

The map which will be created for visualising the flights should be able to render markers or icons on the locations of the planes, displaying custom icons. In addition, the icons have to be able to display the heading of the plane either by rotating the icons or by any other easily interpretable solution. The plane icons should be clickable and should display the flight number as well as the origin country of the flight. The popup should also move with the plane as the icon's location changes.

The other two data types will be combined and displayed on the same map. The car park occupancy data is displayed as a heatmap for which all the frameworks have to provide a heatmap feature with the ability of setting different weights in this case the occupancy values for the separate heat nodes. The heatmap needs to be able to follow the dynamic value changes as well, so when the underlying data changes the heatmap should refresh and display the new data without showing any delays to the users.

Because of the correlation between the two feeds, the road traffic events are displayed on the same map. The requirements with this data type are that they need to be clickable and

able to display the description of the event. Furthermore, the frameworks should provide a loading feature for the GeoJSON format which automatically displays the events on the map.

Client-Server model

To fulfil the previous requirements, the application will be based on the traditional client-server model. This is a distributed application structure which divides the responsibilities between the service callers and the provider. In the case of a web application there is a centralised server which can host static web pages and provide dynamic services as well. The server runs in a remotely accessible network area and waits for a request which comes from the other participants of the structure, from the clients. The communication is always started by the client, but a server can host resources for multiple clients. The clients in web applications are usually the browsers which send direct requests to URL-s or to other resources with the help of the code running in the client applications. This type of architecture perfectly fits the modern web application's and online interactive map system's structure. Based on the research reviews and my previous knowledge in this area, responsibilities will be distributed between the server and the client to fulfil the requirements as follows.

Server responsibilities:

- Collect, process, store dynamically changing data
- Host collected data and static web pages
- Respond to HTTP requests

Client responsibilities:

- Query and render map tiles
- Query and display dynamic data
- Handle user interaction
- Navigate between the map implementations

The system architecture based on the distributed responsibilities can be seen on Figure 3 - System architecture.

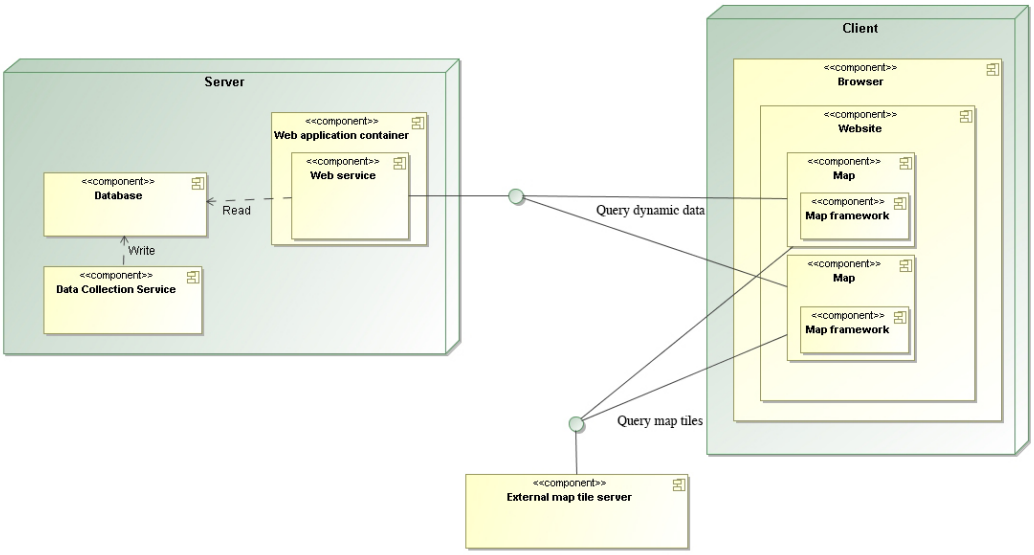


Figure 3 - System architecture

3.4 Development approach – Development strategy

Technology determination

The design stage of the project started sooner than the actual development. During my Personal Studies university course I learnt about a JavaScript based stack including NodeJS and MongoDB. NodeJS is a JavaScript runtime environment for developing server side Web applications while MongoDB is an open source, document-oriented, NoSQL database which allows the user to store the data as JSON objects instead of rows (The MongoDB 3.2 Manual). In addition to that they were convenient to work with and perfect for creating lightweight web application backend, I could exploit code snippets and use the previously obtained knowledge as well so I decided to use this stack as the server side technology of the dissertation project too.

For the client side map and the visualisations, I wanted to locate the three most frequently used frameworks. After reviewing some research papers and applications in the area and consulting with my supervisor and colleagues, the final decision was to use Openlayers as the most popular, Leaflet.js as the fastest evolving open source framework, and the Google Maps JavaScript API as the biggest, most well known map provider.

Development strategy

At the start of the development process there wasn't an exact plan as to how the final application should look. To accommodate this as a development model the Prototyping model was chosen. This technique is usually used when the client is not very clear about the requirements of the project. The developers create an initial prototype which is demonstrated to the client. Then the development process continues with refining the prototypes taking into account the clients' feedback. This iteration is repeated until the prototype outgrows itself as the final product (RNSInformatics). In the case of this project, a basic prototype will be developed with all of the three frameworks, then they will be extended by new features and additional data sources until reaching the final state.

Testing

During the development the application will be tested manually. Configurable data sources will be used so the server is able to send back predefined data which allows the developer to validate the displayed data.

To avoid data related issues mock data was created for the prototypes during all phases of the development. In the beginning only simple data was used, which was returned from the services or local files, then the database helped to serve up a real data like structure, and finally live data was recorded from the original sources and replayed from the database. Since two of the datasources are too static and do not change fast enough for demonstrating the functionalities, the recorded mock data was used for the final maps as well to display the features of the visualisations. Since the incoming data will be under control using this method, it will allow the developer to test whether the application displays the expected output even if there were no integration tests set up for the project.

3.5 Design

3.5.1 Server side architecture

To make the application less fragile and to use the best fitting technologies for each task, the server side responsibilities will be divided into different components.

Web service

A web service will be the core of the server side application, it will provide endpoints where the clients will be able to connect to and query data from. All the datasources will be reachable on different endpoints of the web service in two ways, one providing live while the other providing recorded data. On live request the service will get the fresh data from the dynamic data providers, process it and then forward it in a converted format which can be

handled by the clients. When operating in live mode, the server will behave more or less like a proxy with data translating capabilities. Depending on the required behaviour, recorded data can be queried from the server as well, since the web service will be able to serve up data, such as database or local files, from different sources in the same way.

Database

The most straightforward solution for saving and reloading data is a database, so for satisfying the data storing requirements a database server will be implemented. It will be responsible for permanently storing the data gathered by the data collection services which can be queried by the web service.

Data collection services

Since the local datasources' reliability is quite low the data will be recorded from them for later use. The recorded data will not only be good in case of system outages but will provide the ability to replay certain situations in time which are worth visualising, such as significant changes in car park occupancy at the end of the working hours. The live data arriving from the Glasgow data does not really display the advantages of the dynamic visualisation because of the nearly static nature of the car park and traffic events datasources so the data collection functionality will be broadened and the maps will be extended with a feature to replay the recorded data. To make the architecture less coupled the data collection responsibility will be detached from the main web service. Separate standalone services will be developed for each datasource to query data samples. The services will start up periodically to capture the status of the datasources at that time and save the data images into the database with a timestamp. With this method, historical data can be retrieved from any point of time from the database and can be sent back to the client with various frequency by the web service. For the client this will result in faster changes in the dynamic data which basically will look like time went faster.

Web application container

The web application container will be responsible for providing a platform for the web service to run in and for making its port accessible on the network. The container will not only contain the service but will serve up all the static HTML, JavaScript and CSS files which will be downloaded to the end user's browser for executing more requests towards the service.

3.5.2 Client side architecture

The client in this application, as in any typical web application is the code living in the web browser. In the case of this application the client will contain different web based mapping frameworks which will provide the interface for the users to interact with the services and with the data placed on the server.

Map frameworks

When talking about interactive maps, the interface for user interaction is the map framework itself, on which the users can perform different actions such as panning or zooming to manipulate the underlying map tiles and data. To be able to display the map the frameworks need to display the map tile layer first which can be implemented in different ways. For example, the map features can be rendered from the coordinates in the browser, but the most popular and probably the most efficient way is to download the pre-rendered map tiles on demand from a server. This application will follow the latter approach as well and will connect to publicly available map tile servers to download and then assemble the map tile layer.

Once the base layer is ready, each framework will provide ways to extend the maps with additional information such as text, markers or additional feature layers. These functionalities will be used to visualise the dynamically changing data coming from the web service.

3.6 Methodology of comparison

Developer perspective

The comparison will start with an overall high level framework overview listing all the advantages and disadvantages of the frameworks based on the literature reviews and on the developer's experience during the development phase. At the end of the section the frameworks will be ranked by the the basic attributes related to the API complexity and technological support.

The overall comparison will be followed by a more detailed one, related to the visualisation of dynamically changing data, based on the concrete application development. To set the standards for the comparison, above the basic map features such as panning and zooming some custom constraints were defined related to the visualisations which should be accomplished by all the three map frameworks. The comparison will be based on how easy or difficult it was to write the code meeting these constraints.

User perspective - Questionnaire

In addition to the technology comparison, to analyse the application from a user interface perspective the application will be exposed to a user testing stage. The final version of the website will be deployed to a publicly accessible server thus allowing the users to access it by only typing the IP address into the browser. A questionnaire will be created and handed out to twenty people for completion. Since the users will be asked to interact with the maps in different ways such as panning, zooming and clicking on objects, the questionnaire will imply the beta testing of the application as well. After the testing they will be able to leave bug notes and provide suggestions on additional required features for the maps too. In the questionnaire users can provide subjective feedback on the maps and they are asked to order the different frameworks by different perspectives such as usability and look and feel. The results of the questionnaire are processed and combined with the results of the technology comparison allowing me to write a deductive conclusion and final ranking of the maps.

4 System documentation

This chapter contains the documentation of the final application. It starts with the introduction of the development environment which includes some technology description as well. The last subsection is a detailed overview of the implementation, highlighting the most challenging or important coding structures.

4.1 Development environment

Before starting the project, the development environment needed to be set up which included getting the licenses, as well as installing and configuring the required applications. As a development machine a Macbook Pro laptop with an Intel Core i5 processor operating on 2.7 GHz with 8 GB ram memory was used. The OSX operating system was a straightforward solution because almost all the technologies which were used, have satisfying support and numerous tutorials for the Apple's system.

The server side technology stack which was used does not require too much prerequisites or preparation the installers could be downloaded from the official websites and installed directly on the operating system. The 3.2 version of MongoDB Community Edition for OS X was used as a database server. The release came with a command line database client which was integrated with the terminal for the easier usage by adding it to the PATH variables. On the top of the database a NodeJS v4.2.6 based service was responsible for providing the business logic. The technology uses the Googles v8 JavaScript engine for compiling the source code. NodeJS provides an integrated HTTP server as well so there was no need to use a third party web server such as Apache Tomcat or Jetty.

In the beginning Sublime text 2 text editor was used for manipulating the code then it got compiled and ran with the help of the node command line tool. For debugging purposes, the NodeJS v8 Inspector Google Chrome extension was chosen but soon I needed to realise that this toolkit is not effective enough for me as a NodeJS newbie. To gain more support for the development the JetBrains Webstorm 2016.1.1 Integrated Development Environment (IDE) got installed which not only provides all the previous functionalities but includes useful JavaScript tools for the more effective development such as refactor and autocomplete. The software originally is not free but JetBrains provides a complimentary student license for one year. However, this IDE was used for editing the client side JavaScript and HTML code, the testing and debugging took place in the browser for which the Google Chrome's 51.0 version with integrated developer tools was applied.

To version control the source code the GitHub, web-based Git repository was used. Since this was my first time using Git, instead of working with the command line tool I found easier to run the official GitHub Desktop client application. The application was connected to the project's remote repository and provided a graphical user interface to maintain the code versions.

4.2 System implementation

In this subchapter a detailed description of the implementation can be read. The chapter is divided to subchapters based on the different components of the architecture. For each subchapter the key technologies, achievements and design decisions are listed. The explanation is supported with code snippets which describe any code that is particularly interesting or was challenging to develop.

4.2.1 Web service

To create the basic structure of the web application backend Express.js was used which is a minimal and flexible Node.js framework providing features for creating web applications. The framework comes with a tool called express-generator which is responsible for quickly creating web application skeletons by creating the default folder structure and including the additional node packages which are needed to set up a well designed server (Express JS Documentation). The generator was used to create the application structure which can be seen on Figure 4 - Web service structure. After scaffolding out the skeleton and setting some configurations, basically the web application container is ready to start up.

```
.
├── app.js
├── bin
|   └── www
├── package.json
├── public
|   ├── images
|   ├── javascripts
|   └── stylesheets
|       └── style.css
├── routes
|   ├── index.js
|   └── users.js
└── views
    ├── error.jade
    ├── index.jade
    └── layout.jade
```

Figure 4 - Web service structure

The **package.json** file is responsible for the dependency handling. The name and version of the required packages for the application can be written in the file along with the location and name of the start script and the version of the application. The Node Package Manager (npm) is used in JavaScript to share and reuse the packages between the developers. If an "npm install" is ran in the projects root directory the node

package manager will look for the *package.json* (Code Snippet 1 – package.json) file and install the named dependencies into the *node_modules* folder. The packages can be referenced from there in the application. If an "npm start" command is run to start up the application in the root directory of the server, on default the node package manager will read the location and name of the start up script from this file as well.

The *bin* is the folder where the start up scripts are defined. The start command points to a file called *www* in the *bin* folder. This script is the entry point of the web server. It contains configuration information about the web server container such as, which file to load, which file to include and which port to use on the start of web applications. The scripts also initialise basic logging settings and set how to handle invalid request and server errors.

The *www* script will load the *app.js* file as the main file of the application which is responsible for loading the resources, dependencies and the other files of the application.

```
{
  "name": "whathappensinglasgow",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "body-parser": "^1.13.3",
    "concat-stream": "^1.5.1",
    "cookie-parser": "~1.3.5",
    "debug": "~2.2.0",
    "express": "~4.13.1",
    "geojson": "^0.3.0",
    "jade": "~1.11.0",
    "method-override": "^2.3.6",
    "mongodb": "^2.2.5",
    "mongoose": "^4.5.0",
    "mongoose-geojson-schema": "^2.1.1",
    "morgan": "~1.6.1",
    "request": "^2.72.0",
    "serve-favicon": "~2.3.0",
    "xml2json": "^0.9.1"
  }
}
```

Code Snippet 1 – package.json

External Imports

The "require" keyword is used to import the components into variables which can be used later (Code Snippet 2). The description of the more important ones can be read below.

Express – Imports the express.js framework

Path – A core node module which is responsible for path handling. It helps to create a path from strings both statically and dynamically.

Morgan – This module is responsible for logging the incoming requests and the responses. Useful during the development phase.

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
var db = require('./model/db');
var feature = require('./model/features');
var routes = require('./routes/index');
var features = require('./routes/features');
var flights = require('./routes/flights');
var carparks = require('./routes/carparks');
var trafficevents = require('./routes/trafficevents');
```

Code Snippet 2 – Module imports

Body-parser – Is an express middleware to support POST HTTP request. It adds a body object to the incoming inquiry thus allowing the developer to access the request parameters (Soeters, 2016).

The remaining imports reference local files which responsibilities are described in the next sections.

Application configuration

After the require section there are the application configurations which are set by using the "*app.use()*" function (Code Snippet 3 - Application configuration). This function tells the application to use the parameter which is passed as global configuration. The most important ones can be seen below.

```
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(express.static(path.join(__dirname, 'public')));
```

Code Snippet 3 - Application configuration

Logger('dev') – Defines the logging framework. The *dev* parameter indicates that the application logs debug information about the arriving requests, such as the method, status code and response time.

bodyParser.json() – Allows the application to parse JSON files.

bodyParser.urlencoded({ extended: false}) – Allows the application to automatically read and parse data from URLs with GET requests.

Express.static(path.join(__dirname, 'public')) – Configures the *public* folder as a static directory where the client side files and resources can be stored.

The next four lines on Code Snippet 4 – Application routing are the routing methods, helping to redirect the arriving requests (Code Snippet 4 – Application routing). The first parameter is the name of the endpoint while the second is the file where the request will be redirected to. Basically these are the registered endpoints of the application. The purpose of the redirection is to keep the code clean, and sort out the sub routing in different files from the application configuration.

```
app.use('/features', features);
app.use('/flights', flights);
app.use('/carparks', carparks);
app.use('/trafficevents', trafficevents);
```

Code Snippet 4 – Application routing

In order to handle the wrong HTTP requests the following function was written, where the error message and code is configured as well (Code Snippet 5 - Error handling).

```
// catch 404 and forward to error handler
app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);
});
```

Code Snippet 5 - Error handling

The next two functions are responsible for the general error handling. The difference between them is that in the development environment the error message is sent back with the error code while in production the stack traces should not be leaked to the users (Code Snippet 6 – Error handling configuration).

```
// development error handler
// will print stacktrace
if (app.get('env') === 'development') {
  app.use(function(err, req, res, next) {
    res.status(err.status || 500);
    res.render('error', {
      message: err.message,
      error: err
    });
  });
}

// production error handler
// no stacktraces leaked to user
app.use(function(err, req, res, next) {
  res.status(err.status || 500);
  res.render('error', {
    message: err.message,
    error: {}
  });
});
```

Code Snippet 6 – Error handling configuration

Model Folder

```
var mongoose = require('mongoose');
var schema = new mongoose.Schema({
  icao24: mongoose.Schema.Types.String,
  callsign: mongoose.Schema.Types.String,
  origin_country: mongoose.Schema.Types.String,
  time_position: mongoose.Schema.Types.float,
  time_velocity: mongoose.Schema.Types.float,
  longitude: mongoose.Schema.Types.float,
  latitude: mongoose.Schema.Types.float,
  altitude: mongoose.Schema.Types.float,
  on_ground: mongoose.Schema.Types.GeometryCollection,
  feature: mongoose.Schema.Types.Feature,
  featurecollection: mongoose.Schema.Types.FeatureCollection
});
var Features = mongoose.model('Feature', schema);
module.exports = Features;
```

Code Snippet 7 – Database scheme

In the *model* folder the mongoose database scheme files can be found (Code Snippet 7 – Database scheme). These are responsible for adding scheme restrictions to the noSQL scheme less database. The model schemes help in writing the database queries by using the exported scheme.

The view folder contains the template files of the application. There are different template engines compatible with *express.js* from which *jade* is used by the application. It can replace variables in the template files to actual values during runtime and create a view by translating the templates to HTML (Express). This feature is only used for creating the error page in this software.

As mentioned previously the incoming HTTP requests will be redirected to the corresponding files of the **routes folder**. For example, if the application receives a GET request to the *flights* endpoint, regarding to the mapping, it will be forwarded to the *flights.js* file. Which will handle, process the request and send the response back.

As the configuration showed above, the *public* folder takes care of hosting the static content of the server. Basically everything what is in this folder can be accessed by concatenating the folder's or file's name to the end of the base URL of the server.

Endpoints

In the final application there are four endpoints from which three are used.

The */rodents* endpoint which is responsible for providing static data about the reported rodents activities in Boston. The data was recorded from the Boston city council's page and then replayed from the database in order to test the frameworks' capabilities to display GeoJSON files.

The */flights* endpoint is responsible for serving real time flight data to the clients. When a query arrives to the flight data endpoint it will send a HTTP get request to the flight data provider's API which sends back the response. The response then is piped into a function which is responsible to process the data. To send the request, the *request* module is used which is a simplified HTTP request client for node.js (Simo). Since the queried data arrives back in a bite array, as a first step it needs to be translated to String with the help of a custom function. The transformed file can be parsed and processed as JSON which contains only an array of strings and numbers without the object keys. For the easier usage the array data is looped through and with the help of the API documentation, the data which is used, such as the flight Id, coordinates and the origin country are copied into a structured object with the proper field names (Code Snippet 8 – Response parsing).

```
var str = Utf8ArrayToStr(response);

var incominDataInJSON = JSON.parse(str);
var arr = [];

for (var i in incominDataInJSON.states) {
    var structured = {
        "icao": incominDataInJSON.states[i][0],
        "originCountry": incominDataInJSON.states[i][2],
        "longitude": incominDataInJSON.states[i][5],
        "latitude": incominDataInJSON.states[i][6],
        "heading": incominDataInJSON.states[i][10],
    };
```

Code Snippet 8 – Response parsing

As a final step the structured object is parsed into GeoJSON format with the helps of the *GeoJSON* module. The module can convert any object with location information into GeoJSON format which is the agreed communication protocol format between the server and the client. The response is sent back as an object, containing the GeoJSON files with the additional fields included in the properties field of the converted object (Code Snippet 9 - GeoJSON parsing).

```
response = GeoJSON.parse(structured, {Point: ['latitude', 'longitude']});
```

Code Snippet 9 - GeoJSON parsing

The flight datasource usually returns more than one thousand planes and to maintain the continuous movement the endpoint needs to be queried every five seconds. If we multiply this with the three maps, it not only results in massive network traffic but easily overloads the computers memory and CPU as well, since the computer needs to maintain and move all the planes on the maps. Resolving this issue, the /flights endpoint can be parameterised with a southwest and a northeast point. If the clients call this endpoint with the bottom left and top right coordinates of their current view, the service will filter out all the flight which can not be seen on the current screen thus reducing the resource usage and the network traffic as well.

Car parks, Traffic events

The */carparks* and */trafficevents* endpoints' structure is really similar to the flight endpoint's. Of course they use distinct services to acquire the data, and the object parsing is slightly different as well. The traffic event data arrives in XML so as an extra step an XML to JSON translation is added to the process, using the xml2Json module.

The more significant difference is that these router files are responsible for handling another endpoint which is the */{endpoint-name}/recorded*. On this endpoint the historical data can be accessed for replaying purposes. When a request arrives to the endpoint the router connects to the database and queries all the objects from the specific collection. The retrieved objects contain a timestamp and information about all the car parks or traffic events at the recording's time. An algorithm was written to replay all the records from the database within one minute, not depending on the number of records. The length of the retrieved array is divided by sixty then multiplied with the current second of the minute. The result of the previous calculation is used to determine from which position of the array should the record be sent back (Code Snippet 10 - Algorithm). By adding the current second of the minute into the algorithm instead of using counters keeps the service completely stateless. Since there is no state stored on the backend side, even if there are more clients querying the data, it will not cause any interference.

```
var secondOfMinute = new Date().getSeconds();
var unit  = trafficEventList.length/60;
var numberOfImage = Math.round(secondOfMinute * unit);

res.json(trafficEventList[numberOfImage]);
```

Code Snippet 10 - Algorithm

With this method from the client's perspective receiving data from the live or from the recorded endpoint is entirely transient, so no client side changes were needed to allow the clients displaying historical data.

4.2.2 Data collection services

To collect and record data from the car park occupancy and traffic events datasources, two standalone services were developed. These services continuously ran for a week on the cloud based server to collect and save the data to the database. The structure of the applications is really simple. A scheduled task wakes them up in every 15 minutes when they connect to the datasources on the same way as the web service does. But instead of hosting the data on an endpoint, it will just get saved into the database with a timestamp, using a simple database insert method. This data will be used later by the */recorded* endpoints of the server.

4.2.3 Maps – Client Side

This subsection contains the detailed description of the client side components. The section starts with the structure of the client side then one can read about the different map framework implementations. The

JavaScript and HTML components

The runtime environment for the map frameworks is provided by JavaScript components which are embedded into static HTML pages supplemented with CSS styling in some cases.

When the user types the web page's address in the browser it sends an HTTP get request to the web server which will result in downloading the index HTML site. The main page provides access to the different maps by redirecting the users to the other areas of the server. In the map subdirectories there are other HTML pages which are responsible for loading the map frameworks and other libraries. Once the whole system is loaded, the JavaScript components query the data from the web service and pass it to the map framework via its application programming interface.

Subdirectories

In the final application there are four subdirectories in the public folder. Three of them are for the different map frameworks and the fourth contains the comparison map just by referencing the JavaScript files from the other folders. All the technology folders consist of four subfolders containing the maps for the flights, car parks, traffic events and one for the combined Glasgow car park and traffic event map. In the following sections detailed

documentation can be read about all the twelve maps which demonstrate the behaviour of the frameworks as well.

4.2.3.1 Google Maps

Car park Map

When the folder got referenced, the *index.html* gets loaded first which is responsible for creating the *div* HTML element in which the map will be initialised. This file does the loading of all the JavaScript files including the custom map files and the Google Maps framework itself too. The loading of the Google Maps framework can be seen on Code Snippet 11. It is loaded in an asynchronous way with the *initMap* callback function. The key variable defines the application's private key, which needed to be generated on the Google's API key creation site. This is the way how Google and the developers can monitor the applications' usage. If the usage reaches a limit, the key might need to be upgraded by buying a business license (Google, Google Maps APIs). For implementing the car park occupancy visualisation, heatmap technology was used. The library, which is responsible for the visualisation, is contained by the *geometry* and *visualisation* Google Maps libraries. These libraries are not part of the main framework so the explicit import is passed to the API as the last parameter of the framework import call.

```
<script async defer
        src="https://maps.googleapis.com/maps/api/js
?key=AIzaSyBl8Rjx-k57khpDrH4iRwozQTH1zjObLYI
&callback=initMap
&libraries=geometry,visualization">
</script>
```

Code Snippet 11 – Framework import

When the framework is loaded it will call the *initMap* callback function which is in the *carparkmap.js* file. The script starts with loading the map into the *div* which can be seen on Code Snippet 12. On initialisation the zoom level and the starting coordinates are set as well.

```
var mapDiv = document.getElementById('map');
var map = new google.maps.Map(mapDiv, {
    center: {lat: 55.8642, lng: -4.2518},
    zoom: 13
});
```

Code Snippet 12 – Map initialisation

The next step is the heatmap initialisation, which is demonstrated on Code Snippet 13. It is quite simple with the integrated support, only the datasource and a few configuration settings are needed.

```
var heatmap = new google.maps.visualization.HeatmapLayer({
    data: heatmapData,
    dissipating: true,
    radius: 20,
    map: map
});
```

Code Snippet 13 – Heatmap initialisation

The datasource of the heatmap is a special *MVCArray* which fires events on change, so the heatmap layer will be notified when the underlying data changes. After the layers are initialised the *setInterval* JavaScript function is used to execute the *requestLatestData* function once in every minute. The function sends an Ajax call to the web service's car park endpoint to query the car park information. If the call was successful, the data will be passed to the *refreshMapData* function which first clears the datasource of the heatmap layer then iterates through the received data, parses and adds it as the new datasource. For reaching the different colours on the map the weight property of each datasource object needs to be set to the car parks' occupancy level. During this some filtering needs to be done, because some car parks has negative occupancies in the data which resulted in an error on the display (Code Snippet 14 - Filtering negative values).

```
function refreshHeatmapData(results) {
    heatmapData.clear();
    for (var i = 0; i < results.features.length; i++) {
        var coords = results.features[i].geometry.coordinates;
        var latLng = new google.maps.LatLng(coords[1], coords[0]);
        var occupancy = results.features[i].properties.occupancy;
        var weightedLoc = {
            location: latLng,
            weight: occupancy
        };
        if (occupancy > 0)
            heatmapData.push(weightedLoc);
    }
}
```

Code Snippet 14 - Filtering negative values

As a result, a Google Maps based map appeared with the live Glasgow carparks' occupancy data displayed on the overlays as a heatmap (Figure 5).

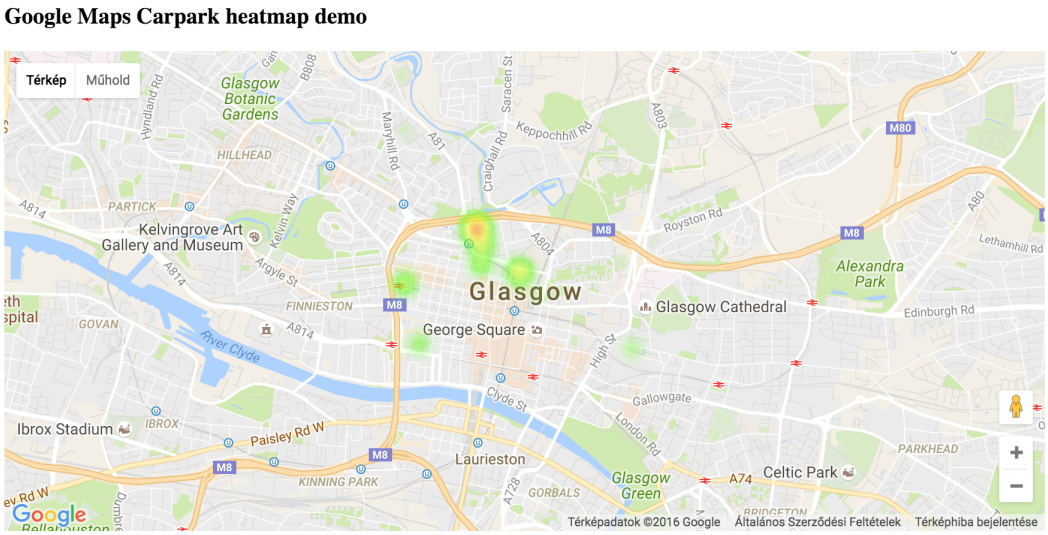


Figure 5 – Google Maps Car park heatmap

Flight map

The *index.html* of the flight map is really similar to the car park's *index* file but instead of loading the visualisation libraries, there are three other JavaScript files: *Slidingmarker.js*, *markerAnimate.js* and *jquery.easing.1.3.js*. These files are the part of SlidingMarker GitHub plugin for Google Maps. The script enables the Google Maps markers to move from its original position to a destination in an animated way (Viskin).

When the *flightmap.js* is loaded the initialisation starts with the SlidingMarker's global initialisation which overrides the original markers and enables them to behave as animated ones. The map itself is initialised in the same way as it was seen at the car park map and the data loading works in a similar way as well. The icon which represents the planes is loaded on initialisation as well. The Google Maps framework is not able to rotate images as icons, so an SVG vector image needed to be created instead of a normal image file.

Originally the application used to query all the flight data information in every request which leads to high network traffic and long rendering time. To optimise the solution, the flight endpoint was made parameterised. The URL for getting the flight data can accept two points, the bottom left and top right coordinates of the currently displayed map thus returning only those flights which can currently be seen on the map. This change reduced the loading time and increased the rendering speed significantly (Code Snippet 15 - Parameterised endpoint call).

```
function requestLatestData() {
    var bounds = map.getBounds();
    $.ajax({
        url: '../../flights?swlat=' + bounds.f.f + '&swlon=' + bounds.b.b
+ '&nelat=' + bounds.f.b + '&nelon=' + bounds.b.f,
        async: true,
        success: processData
    });
}
```

Code Snippet 15 - Parameterised endpoint call

When the data arrives the icon's rotation will be set to the direction of the plane's heading. Then it is tested if it is a new flight or the map already contains the plane for the incoming data. If the plane is not on the display a new Marker is created with the properties of the flight and the icon. On creation not only the duration of the animation is set but a listener is added to the marker as well which is responsible for showing an info window on the click event of the icon, displaying the flight number and the origin country.

If the map contains a plane with the same flight number, the marker for the specific identifier is queried from the map then the icon and the location get overridden with the new data. Therefore, the plane will move to the new location with the new rotation in an animated way. The map queries the new data in every five seconds and since the length of the animation is set to the same duration it results in am almost continuous movement of the planes.

Google maps flight map

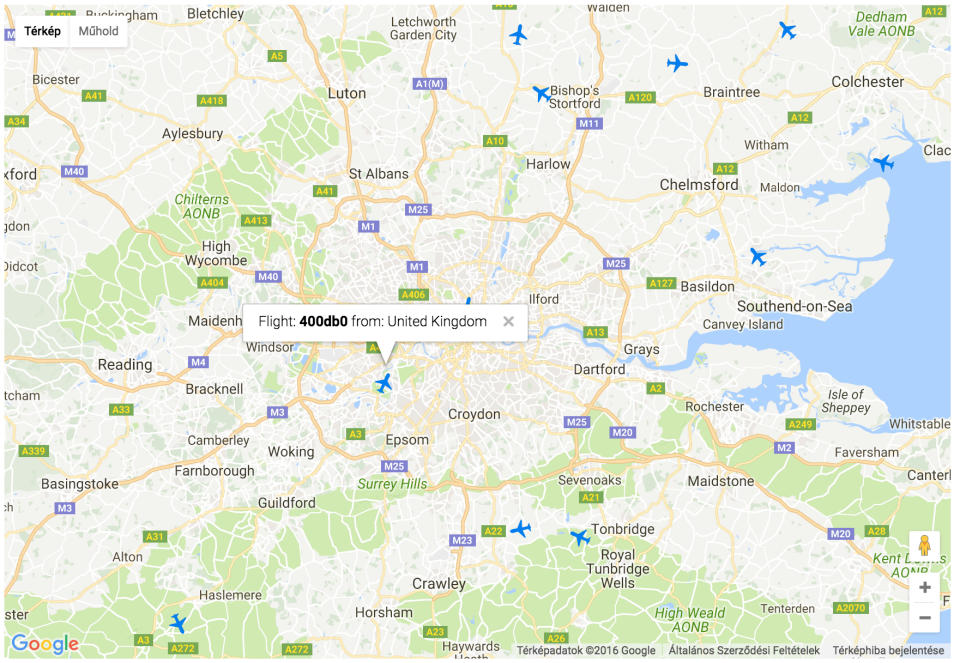


Figure 6 – Google Maps flight map

Combined

The last Google Maps map is responsible for showing the car park and the traffic events data on the same map. It connects to both endpoints in an asynchronous way and loads the data into the same map object. However, the map uses the aggregated codebase of the previous two maps, there are some configuration changes in the data loading frequency and URLs. The combined map uses the recorded data provided by the web service and to display the faster changing data, the time between two queries is reduced to one second. This way the map displays the changes in the car park occupancy in a smooth, gradient way.

Google Maps Combined carpark heatmap and traffic events map

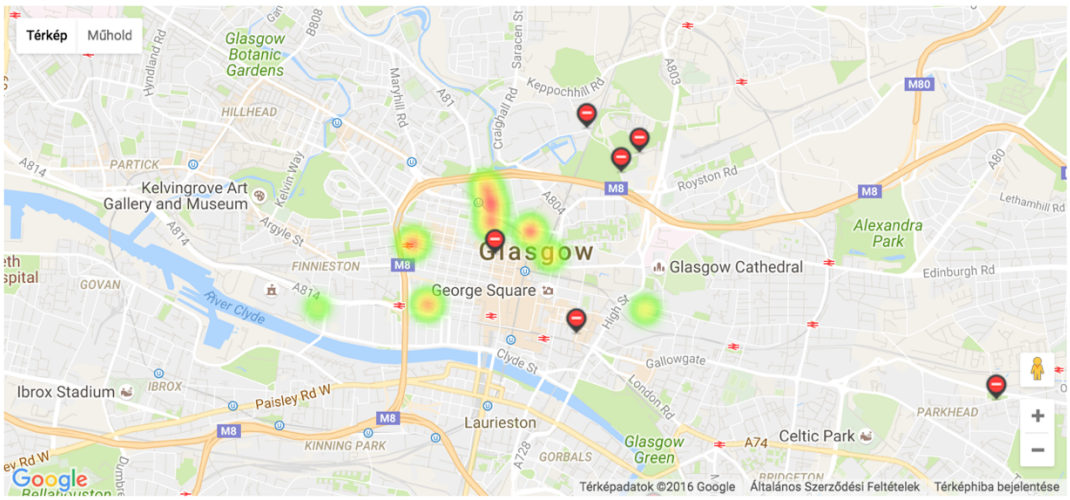


Figure 7 – Google Maps Combined car park heatmap and traffic events map

4.2.3.2 Leaflet

Flight map

When navigating to the Leaflet *flightmap* folder the *index.html* will be loaded first. It is not only responsible for loading the stylesheets and the leaflet framework but for importing all the plugins from the *plugins* subfolder as well.

The Leaflet Realtime is a library which helps to put realtime data on a Leaflet map by reading and displaying GeoJSON from a provided source (Perliedman) while the Leaflet Rotated Marker enables the rotation of marker icons in leaflet (Bbecquet). Once the *flightmap.js* file is loaded, the initialisation process starts similarly to the Google Maps' structure. As a first step the map object is created with centre and zoom level settings (Code Snippet 16 - Map initialisation).

```
var leafletMap = L.map('leafletmap', {
        center: [51.505, -0.09],
        zoom: 9
    })
```

Code Snippet 16 - Map initialisation

The icon uses a *.png* image as the datasource because not like Google Maps, Leaflet is able to rotate images as icons as well. The framework does not use an integrated map tile source, so it needs to set to an external provider manually. For all the Leaflet maps the OpenStreetMap based Wikimedia map server is used. OpenStreetMap is an initiative to create and provide geographic data such as maps to anyone for free (OpenStreetMap Foundation). The *x, y, z* variables on Code Snippet 17in the URL path are the coordinates and the zoom level to load the proper tiles.

```
L.tileLayer(' https://maps.wikimedia.org/osm-intl/{z}/{x}/{y}.png', {
    attribution: '&copy; <a href="http://osm.org/copyright">OpenStreetMap
</a> contributors'
}).addTo(leafletMap);
```

Code Snippet 17 – Tile loading

When the map layer is ready, the Leaflet Realtime starts to operate. The plugin's *reqwest* function is called with the parameterised URL to the */flights* endpoint. This function enables the developer to load data from GeoJSON format to the map automatically. Some configuration parameters are set such as the querying interval and the name of the *featureId* field from the response data. Thus the framework can maintain which flights are new, and which were there before already. The new ones are added to the map layer, and those which were there already only need to be moved to their updated location.

The Leaflet's *pointToLayer* parameter is set to override the default marker icon to the plane icon. To set the rotation of the icon the Leaflet Marker Rotation plugin's *setRotationAngle* method is used. The popup window creation happens in a similar simple way with the help of the Leaflet Marker's *bindPopup* feature with the required details on Code Snippet 18 - Popup binding

```
interval: 5 * 1000,
getFeatureId: function (f) {
    return f.properties.icao;
},
pointToLayer: function (feature, latLng) {
    return L.marker(latLng,
        {icon: plane},
        {title: feature.properties.icao})
        .setRotationAngle(feature.properties.heading)
        .bindPopup('Flight: <b>' + feature.properties.icao + '</b> from '
+ feature.properties.originCountry);
},
```

Code Snippet 18 - Popup binding

The Realtime handles the location updates of the flights but does not update the rotation on default, to enable this feature the Realtime's *updateFeature* function is used where the old layer's rotation is updated.

The animation of the marker's movement was solved with the help of a CSS transition code snippet which was provided by the Realtime plugin as well.

The final result can be seen on the Figure 8 - Leaflet flight map below.

Leaflet flight map

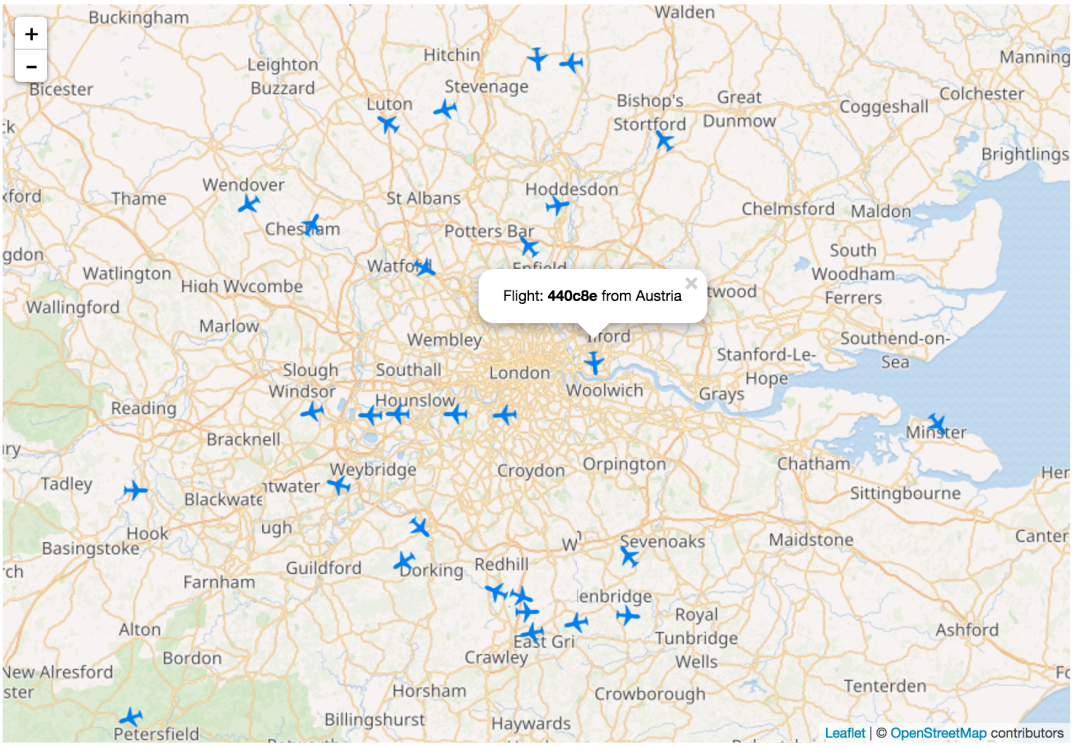


Figure 8 - Leaflet flight map

Car park

The index file of the car park map is really similar to the flight map but here besides the Realtime, the Leaflet Heat and the Leaflet divHeatmap Layer plugins are loaded which are responsible for creating the heatmap.

After loading the *carparkmap.js*, in addition to the standard map initialisation a heat layer is created by the Leaflet Heat plugin with a datasource and a radius parameter.

When the data arrives back from the Realtime *reqwest* it is passed to the *geoJsonToheat* function which transforms the data into an array containing the coordinates and the occupancy of the car park. This array is set as the datasource of the heatmap layer during every iteration.

```
.then(function (data) {
    for (var i in data.features) {
        heatData = geoJson2heat(data);
    }
    heatmap.setLatLngs(heatData);
})
```

Code Snippet 19 - Updating heatmap

The car park map can be seen on the Figure 9 - Leaflet heatmap.

Leaflet carpark heatmap

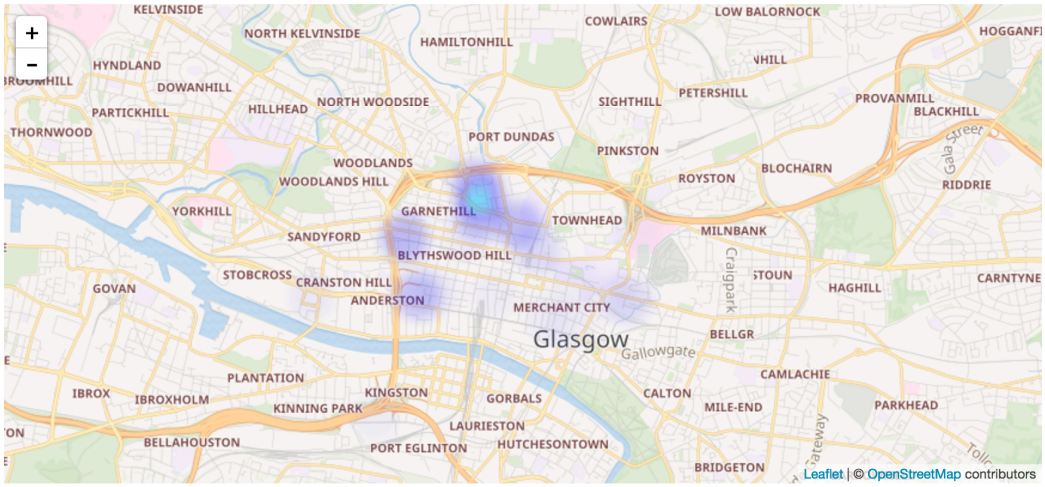


Figure 9 - Leaflet heatmap

Combined

The Leaflet combined map has the same responsibility as the Google Maps one. It displays together the car park occupancy and the traffic events data from the recorded endpoints with a higher querying frequency by using the codebase of the separate maps. Figure 10 - Leaflet combined car park heatmap and traffic events map

Leaflet combined carpark heatmap and traffic events map

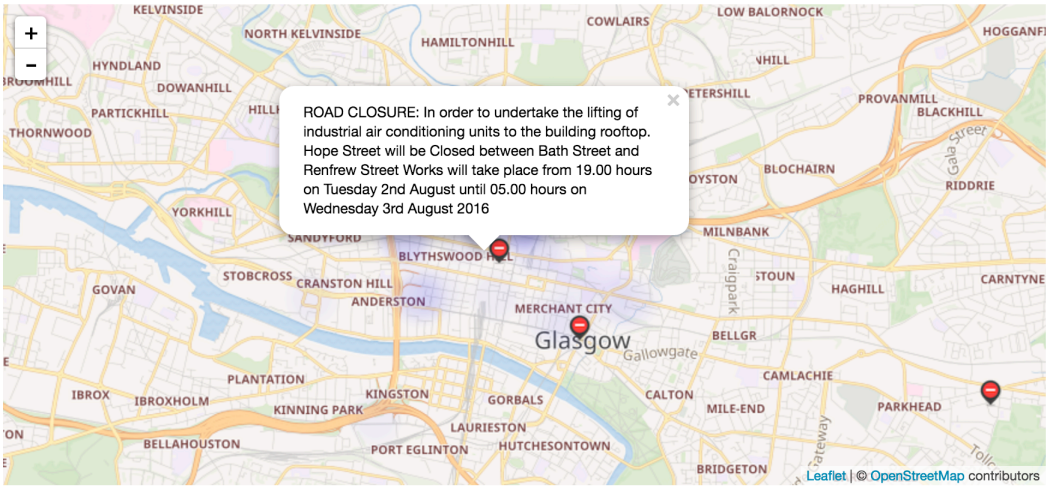


Figure 10 - Leaflet combined car park heatmap and traffic events map

4.2.3.3 Openlayers

Car park

As it could be seen at the previous frameworks, when navigating to the map's folder, the *index.html* is loaded first. The file imports the Openlayers framework and the *carpark.js* JavaScript file as usually. This framework does not require any extra import for the heatmap visualisation.

The structure of the *carpark.js* file is similar to the Google Maps' one although the initialisation here starts with the heatmap loading. As a first step a *Vector* type datasource is defined which is passed to the integrated Openlayers heatmap layer as a source parameter.

On initialisation the Openlayers map takes the layers as parameters. In this case the heatmap layer and a tile layer with the OpenStreetMap datasource are passed. These layers are rendered on the top of each other in the order of the parameters.

The Openlayers framework uses a different type of projection from the previous systems (EPSG:4326). Since the project's data is collected from a system with the Mercator Projection

(EPSG:3857) all the coordinates need to go through a transformation before adding them to the map (Web Mercator). The transformation takes place in the initialisation function of the map (Code Snippet 20 - Map initialisation with transformation)

```
var map = new ol.Map({
    layers:
[new ol.layer.Tile({source: new ol.source.OSM()}), heatmapLayer],
    target: document.getElementById('map'),
    view: new ol.View({
        center:
 ol.proj.transform([-4.2518, 55.8642], 'EPSG:4326', 'EPSG:3857'),
        zoom: 13
    })
});
```

Code Snippet 20 - Map initialisation with transformation

After the layers are initialised the script uses an Ajax request to query the data from the car park endpoint. The response data goes through the projection transformation then it gets loaded into an array with the help of the integrated Openlayers GeoJSON *readfeatures* function. After the vector source gets emptied, the feature objects' weight property is set to the car park occupancy value. Finally, they are added as features to the vector source. At the end of the process a layer changed event is fired on the heatmap for notifying the map to redraw the layer. The process is scheduled with the help of the *setInterval* function as in the Google Maps. The final result can be seen on Figure 11 - Openlayers heatmap.

Openlayers car park occupancy map

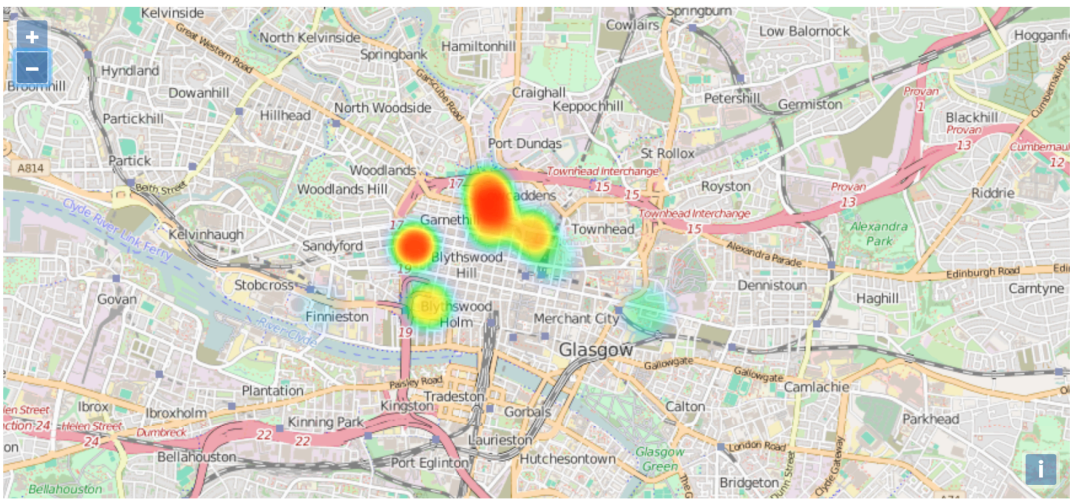


Figure 11 - Openlayers heatmap

Flight map

The *index.html* of the flight map is the same as the car park map's except of that the previous contains a *div* element and a *css include* for a popup window. Since the Openlayers framework does not have a built in solution for displaying popups, a custom *div* element is moved on the screen and populated with information as the flight details popup.

The *flightmap.js* after initialising the sources and the map, creates a vector layer which is responsible for displaying the planes and an overlay layer which contains the popup div as an element. To handle the click events on the layer and on the map a few event listeners are registered. If a click event fires on the map all the features on the map are iterated through and checked if any of them contains the pixel which was clicked. If the result is true, the popup will be moved to the positon and displayed with the required data. The popup will be hidden if the evaluation's result is false so the click was a random click on the map. (Code Snippet 21 - Popup listener)

```
map.on('click', function (evt) {
    var feature = map.forEachFeatureAtPixel(evt.pixel,
        function (feature, layer) {
            return feature;
        });
    if (feature) {
        var geometry = feature.getGeometry();
        var coord = geometry.getCoordinates();
        popup.setPosition(coord);
        content.innerHTML = 'Flight: <b>' + feature.attributes.name +
            '</b> from: ' + feature.attributes.origin;
        $(element).popover('show');
    } else {
        $(element).popover('destroy');
    }
});
```

Code Snippet 21 - Popup listener

To change the mouse icon to a pointer when it is hovered over an icon, a mouse move register is added in a similar way. The last listener is registered on the close icon of the popup window allowing the users to close the window with the red X in the corner.

Once the initialisation and the registers are ready an Ajax call is sent to the parameterised flights URL. The map boundaries are calculated and translated to the right projection before passed as parameters to the request. The logic for moving and adding the planes to the map is similar to the Google Maps implementation. If a flight is new it will be added to the vector

layer as an Openlayers *Feature* object, if it is not the position and the style of the object will be updated.

To display the icons as planes a new *Style* object needs to be created with and *Icon* object as its image attribute. For changing the planes' orientation, the rotation property needs to be set in radians (Code Snippet 22 - Setting plane orientation). The style's rotation is updated with every response then it is added to the right feature.

```
var planeStyle = new ol.style.Style({
    image: new ol.style.Icon(({
        anchor: [0.5, 0.5],
        scale: 0.9,
        opacity: 1,
        src: 'icon.png',
        rotation: results.features[i].properties.heading *
0.0174532925,//converting the angle to radian

    }))
});
```

Code Snippet 22 - Setting plane orientation

When the data processing is finished a changed event is called on the vector layer to indicate to the map that the layers need to be redrawn. For animating the movement of the features there was not any integrated support or library provided by the Openlayers framework so the planes' location change is solved by moving them from one place to the other without any animation. The final result can be seen on Figure 12 - Openlayers flight map.

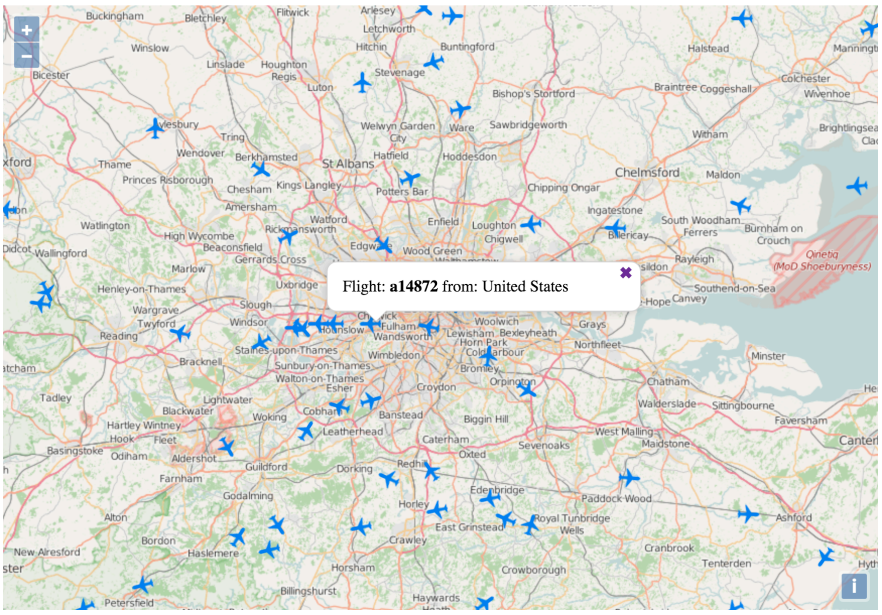


Figure 12 - Openlayers flight map

Combined

The Openlayers combined map displays the car park occupancy and the traffic events data on the same map. It uses the same codebase as the separate maps, except of that this map queries the recorded endpoints with a higher frequency. (Figure 13 - Openlayers combined car park heatmap and traffic events map)

Openlayers combined carpark heatmap and traffic events map

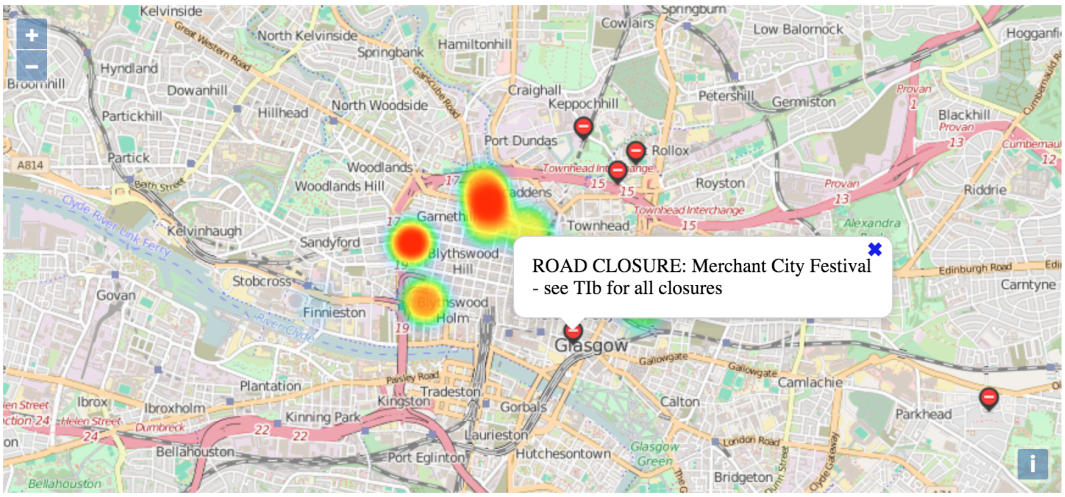


Figure 13 - Openlayers combined car park heatmap and traffic events map

4.2.3.4 User comparison maps

two comparison maps were created to display the same data thus allowing the easier comparison for the users. These maps show a slice from the maps created with all the three technologies. It is crucial that all the maps have to have the same configurations such as the initial view coordinates, the zoom level and the data querying frequency. The different map framework implementations are imported with all their dependencies to three *div* elements under each other in a HTML file. The users are allowed to interact separately with each of them.

Obviously the flight map (Figure 15 - Flights user comparison map) uses the three flight map implementations while the Glasgow map (Figure 14 - Glasgow user comparison map) imports the combined car park and traffic event maps. Since the Glasgow map uses the recorded datasource, a label was added to show the time and date of the data which is currently displayed by the map.

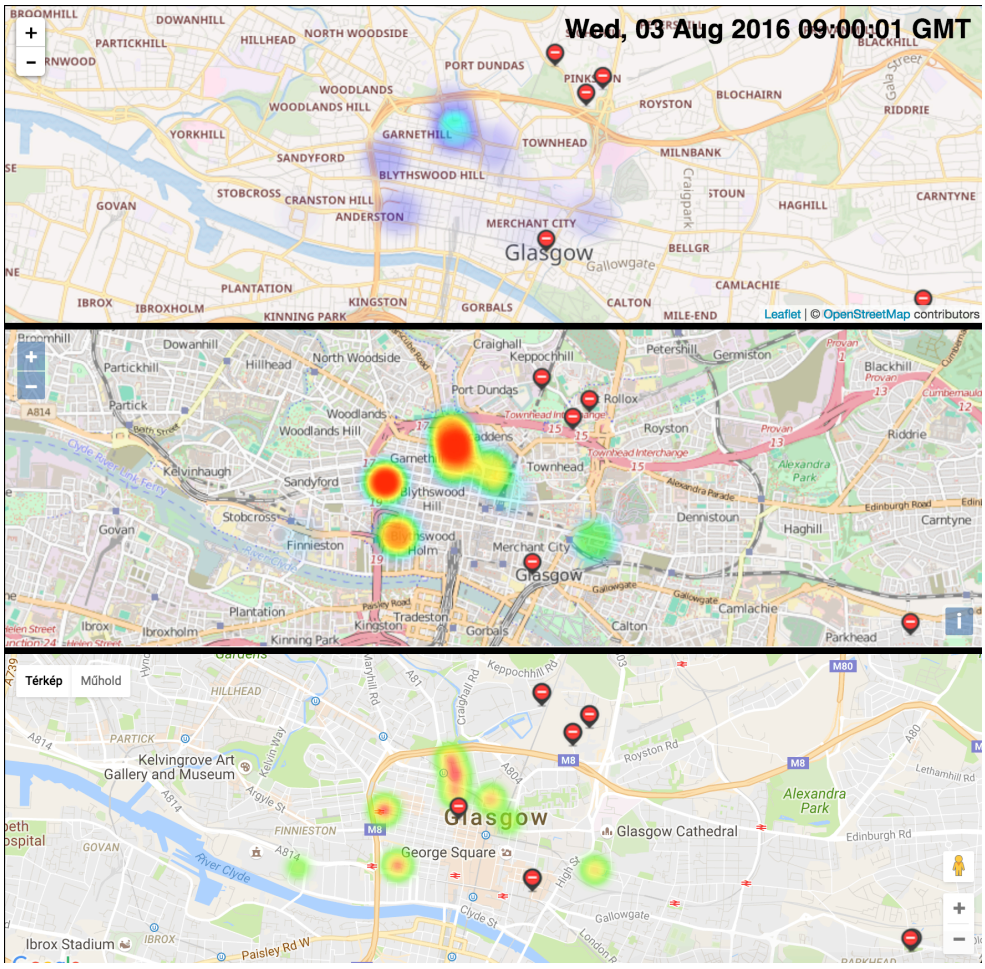


Figure 14 - Glasgow user comparison map

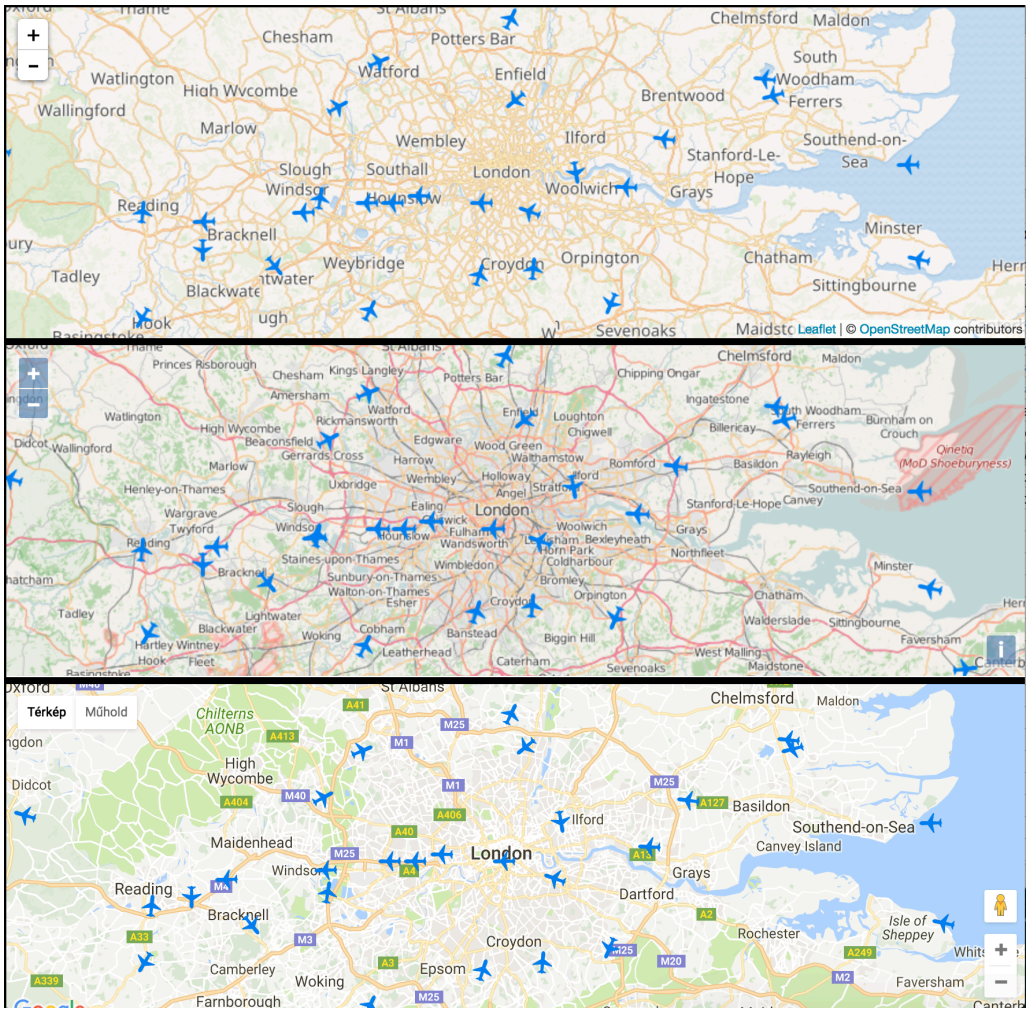


Figure 15 - Flights user comparison map

4.2.4 Deployment

- **Deployment**

During the development phase the application mostly ran on the local development environment but for enabling the users to test the product it needed to be installed on a remotely accessible server. As a first approach the application was installed to a CentOS Linux system in the Amazon EC2 cloud which is a pay as you go based cloud server platform. But as the development pushed forward it needed to be realised that the platform is not appropriate for the project's purposes. The bundle which was used, charged the fee after the number of requests which was not the best option for the polling based application, making numerous requests every minute. At the end the application was deployed to a Windows server in the CenturyLink cloud which provided fix monthly billing.

5 Framework evaluation and comparison

This chapter provides a detailed framework comparison by going through o all the previously defined comparison methodologies. First there is a short overview of the frameworks which is followed by a comparison based on the APIs' and the frameworks' support. At the end of the second subchapter the frameworks are ranked by the the basic attributes related to the API complexity and technological support. The overall comparison is followed by a more detailed one, related to the visualisation of dynamically changing data, based on the concrete application development. The last section compares the frameworks by the users' feedback including describing some details about the questionnaire.

5.1 Origin and license

This subchapter summarises the background of the frameworks and describes the type of license with which the technology is released.

Google Maps

The Google Maps application released in 2005 was the first from the three frameworks in 2005. Soon after, Google created their publicly available Google Maps JavaScript API which has been extended by other platforms since then such as Flash and Android (Google, Our history in details). The API comes under Google's license with different billing plans. Contrary to the common belief, the product usage is free within the framework of the standard usage plan. This includes twenty-five thousand map loads per twenty-four hours which is a reasonable number for smaller applications like this. The other Google services such as street view and the Directions API are not included in this plan (Google, Google Maps JavaScript API Usage Limits). The usage is monitored with a unique API key which can be obtained from Google's website for each application.

Openlayers

The Google Maps was followed by the Openlayers framework's first edition in 2006. It was originally developed by MetaCarta, but since November 2007 it has been an Open Source Geospatial Foundation project (Gratier, Spencer, & Hazzard, 2015). The current version of the API is the OpenLayers3.

Leaflet

In 2011 Leaflet was the last, as an open source project. The library was originally developed by Vladimir Agafonkin but currently it is an open source GitHub project with over hundred contributors.

5.2 API and support

This subchapter is responsible for comparing the frameworks based on general features and the complexity of the APIs. It also includes the summary and the comparison of the support of the frameworks.

Leaflet

As Fernandes et al. (Fernandes, Goulão, & Rodrigues) indicated the structure and the concept behind the API-s are slightly different. Since being the most recent framework Leaflet uses the latest JavaScript and HTML features to implement the different functionalities. The API itself is a very simple to use, tiny, 65Kb sized JavaScript file. However, the core framework only meets the basic mapping requirements as built-in features, the system does not lack any functionalities. The architecture is based on decoupled components instead of a massive core. On the official Leaflet website currently there are more than two hundred different plugins available providing wide variety of functionalities, and this number is growing fast. This modular based architecture allows different developers to focus on one particular feature and make it as good as it is possible. The community support of the framework is outstanding. On average, two to three different plugins can be found for every functionality which was required for the project. The documentation and the quality of the plugins are adequate. The heatmap plugin which is used for displaying the car park occupancy was written by Agafonkin himself. (Ortega, 2015)

The Leaflet official website contains around ten simple tutorials to understand and implement the most used features of the framework. The core documentation is structured thematically and provides usage examples in quite a high number of the cases.

The plugins which can be chosen from the catalogue are accessible through the official site too. After choosing the required plugin the developer is redirected to the GitHub directory of the feature, where the documentation and usually a live demo can be found. Although the plugins which were used during the project were appropriate, since they are maintained by different developers they might not kept up to date with the new versions of the core framework.

Openlayers

Openlayers uses a significantly different architectural concept than Leaflet, it tries to include all the features which a developer might require. The core application contains a wide range of built in features and functions which increases the size. Openlayers2 was almost a megabyte but the developers managed to reduce the size in order to increase the mobile usability. The minified version of the current, core Openlayers3 library is 495Kb which is still more than ten times larger than Leaflet. Because of this change in the size, the API has changed a lot too, and not only between Openlayers2 and Openlayers3 but between the smaller version changes as well. Fernandes et al. revealed that the compatibility between the different versions can be low, but it was not suspected at that time that it would effect this project. However, it was not the case, because many of the example codes and the demos which can be found for Openlayers are not compatible with the current version. Even though there are 149 examples on the official site, not all of them are useful. Not to mention the community based support websites such as Stackoverflow or the GIS Stackexchange where the features which are mentioned in the few years old examples are not even part of the API anymore. Because of this it is crucial to use the API documentation. The official documentation is good quality, containing a few examples too, but since the whole architecture is more complex, the Openlayers' documentation requires deeper knowledge of the area than Leaflet does.

Google Maps

Google's core architecture provides many different functionalities, and is very extensible, more like Openlayers. Since Google Maps is the most widely used mapping framework, it is robustly tested and supported too. On the official website there are plenty of well designed, working tutorials which explain and demonstrate the usage of the framework. As Fernandes et al. indicated the API changes between the different versions of Google Maps are negligible so the older examples provided by the community largely worked well. The API provides a higher level of abstraction which makes the use of the even more complex features relatively simple, but on the other hand, it allows less customisation.

Even though Google's API is publicly available the source code is not, so the flexibility is much lower compared to the open source frameworks. This also affected this project work. The original plan was to make all three maps look exactly the same by using the same map tile server for all of them. Unfortunately, this feature is not configurable in the Google Maps API, and since the source is not open, it is not even editable.

Marking

Finally, the different attributes of the frameworks based on the previous discoveries were marked on a scale between 1-3 for easier comparison. Number 1 is the weakest mark showing that the framework has weaknesses in the measured aspect, while number 3 indicates that the attribute is a strong point of the framework. This is a clearer way to represent the results and more tangible to compare the frameworks.

Table 1- API and support comparison

	Leaflet	Openlayers	Google Maps
API Documentation	3	3	3
Tutorials	2	1	3
Community support	3	1	3
License	3	3	2
Total	**11**	**8**	**11**

The results clearly indicate the disadvantages of each framework. In the end Google Maps and Leaflet got the same score by losing only one point in slightly different areas. These two frameworks won this comparison with the good support and flexible API. Leaflet did not get the maximum point because of the lack of tutorials while Google Maps lost a point because of the less flexible license. As was indicated earlier, because of the weak community support and the significant differences between the API version, it was much more difficult to work with Openlayers than with the other two frameworks. That's why it only collected eight points.

5.3 Matching the constraints

In this subchapter the developer side comparison can be read. The main focus of this subchapter is to describe how the constraints were met with all the frameworks. To help the comparison, evaluation comments and explanations are added to the description. At the end of the section a comparison table can be seen displaying how many points the frameworks got on each constraint.

Leaflet

Thanks to the custom plugins, Leaflet provided some really simple implementations to meet all the constraints which were set. It was relatively straightforward to find the right plugins through the official Leaflet website.

The car park map's code was relatively simple. The Realtime plugin takes care of the data loading, only a few configuration fields are needed to be set. To make the heatmap feature look similar to the others, two different plugins were tried. To swap between the implementations, only another import statements and small changes to the datasource were needed. This demonstrates well the simplicity and flexibility of the framework. The heatmap plugin takes three inputs, the coordinates and the weight of the point by default. The heatlayer's datasource is overridden on every data refresh which implies the redrawing of the layer automatically.

Similar high level functionality was provided for the flight tracking as well. The Realtime plugin can maintain, add, remove and move the objects arriving from the endpoints automatically. The only configuration which was needed to set was the name of the identifier field, to let the plugin know which property identifies the objects. It solved the data loading and object displaying constraints. The icons were clickable by default. To find a solution for implementing the icon changes with the popup window, only the official Leaflet documentation needed to be looked up. The popup window is a built in feature in Leaflet which is bound to the object, so it moves together with the icon if its location changes. To enable the icon rotation on update, was slightly more complicated task but there were useful comments in the Realtime plugin's GitHub repository which helped to satisfy the constraint.

The Realtime plugin accepts inputs in GeoJSON format as well, so the automatic loading of this format was solved as well, without the need of writing custom code.

Openlayers

To work with the Openlayers framework was not as straightforward as with Leaflet even though finally, almost all the constraints were met.

Despite there not being an integrated feature like the Leaflet Realtime for loading data periodically from an endpoint, the final car park heatmap solution ended up being fairly simple. Openlayers contains a built in heatmap feature which was convenient to use. It only requires a vector as a datasource with the weight attributes set to display the different

occupancies. To pick up the changes and redraw the layer, the changed event needed to be called manually.

The flight map's code is much longer and more complicated than the car park map's or the flight map's written with the other two frameworks. There is no built in feature for creating markers in Openlayers. To add a marker like object to the map, a Point object needs to be created, obviously with the transformed coordinates. Then this Point has to be passed as a parameter to a Feature object, and then the feature object gets a Style object which will behave as the Openlayers' marker. Although this process has the same result as adding a Marker object, it is more complicated than with the other frameworks.

Openlayers currently did not provide a popup feature either, so a custom solution needed to be created. The popup is simply a positioned HTML element which is added to the map as an overlay. Then all the listeners needed to be written manually allowing the icons to be clickable and for showing and destroying the popups. Since the popups are not bound directly to the map object they do not follow them while their location changes so this constraint is not met.

No reasonable way was found to add animation to the moving flight features either so the map is not animated which fails another constraint.

To change the icons and make them rotated with the orientation of the planes a new Style object has to be created for every data response and either update the already existing planes or set it as the style of a new Feature.

Since Openlayers uses a different projection than the general standard, issues needed to be overcome repeatedly during the development by translating the coordinates from one projection into the other. Even the rotation angle needed to be changed to radians from degrees. The other main problem was the out of date tutorials and community references. Only every fourth or fifth example was compatible with the used version. Openlayers provided basic support for all the required features, but with quite low abstraction and weaker support.

Google maps

There is a built in feature in Google Maps which is responsible for loading GeoJSON files from a remote URL and adding the received data to the map as markers, although it was not used at the end because of customisation issues. The heatmap is not included in the core Google Maps library but only by adding the "visualisation" keyword to the end of the import

URL, another framework extension will be loaded which contains some extra features supporting the visualisations. It was fairly simple to implement the heatmap thanks to the tutorials on the Google's site. The framework provides an MVC object which automatically sends change events every time the underlying data refreshes so the heatmap layer knows when it needs to be redrawn. For adding different intensity only, the weight property needed to be set on the heatmap layers input.

For adding the planes, the Google Maps integrated Marker class was used. On creation the icon can be configured and listeners can be added. A click listener needed to be added which opens an info window. The Infowindow is a built-in feature too, and since it is bound to its marker, it moves if the marker changes its location.

The icon rotation caused some issues since the Google Maps framework is not able to rotate images as the other two frameworks. Scalable Vector Graphic (SVG) needed to be created from the image which was used, to make the rotation property work on the markers.

To animate the movement of the markers a third party Google Maps extensions was used. The MarkerAnimate and the SlidingMarker are two small JavaScript files which helped to change the global behaviour of the markers. By importing the scripts and adding an extra duration property to the creation of the markers, the animation constraint was easily met.

It was not expected that external plugins would be used for the Google Maps framework, but there are quite a few which provide higher abstraction level solutions for those few common problems to which Google has not yet given a solution.

Comparison Table

To compare the results from development perspective a table was created where the different requirements are listed which the maps needed to fulfil. In the table it is compared how difficult it was to satisfy the constraints. For this the time taken to find guidance for the solution, the time taken to implement the solution, the complexity of the code and the quality of the solution were taken into account. At the end these values were aggregated and displayed on a list from 0-3 where 0 means that the requirement is not met while 3 means that it was easy to find guidance and during a short time a good quality but simple solution could be developed.

Table 2 - Constraint based comparison

	Leaflet	Openlayers	Google Maps
Automatic GeoJSON loading	3	2	3
Heatmap API	3	3	3
Heatmap node weighting	3	2	3
Heatmap dynamic update	3	3	3
Custom icon display	3	2	3
Icon rotation	2	2	1
Clickable icons	3	1	3
Popups	3	1	3
Popups moving with icons	3	0	3
Animated movement	3	0	2
Total	**29**	**16**	**27**

Openlayers ended up in the last position again. This was due mostly to the lack of integrated marker and popup support, along with everything taking much longer with this framework. Openlayers definitely lost the competition in this comparison, since in the last two contstraints, most complex functionalities could not even be implemented. Leaflet and the Google Maps finished quite close to each other. Both lost points on the icon rotation and since Google Maps can not rotate images just vector graphics, which makes the animation speed slower, it only came second. Both frameworks provided sufficient support for meeting the constraints and there were no big difficulties during the development either. It can be said that neither of the last two frameworks is a bad option for dynamic data visualisation purposes.

5.4 Questionnaire

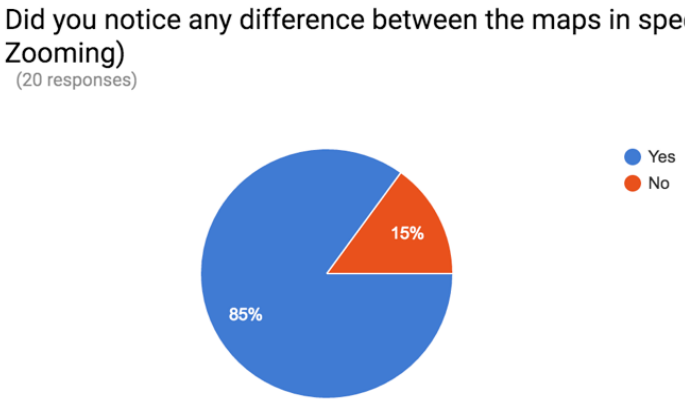
The questionnaire is responsible for helping the map evaluation from user perspective. The questions were mostly focused on getting subjective user feedback on both the framework comparison and the effectiveness of the visualisations as well. The maps were not

named after the frameworks but were called only Top Map, Middle Map and Bottom Map thus trying to avoid any preconceptions. During the questionnaire evaluation the same names will be used, the corresponding frameworks are: Leaflet, Openlayers and Google Maps in order.

The questionnaire was completed by twenty people, from which a significant number has previously worked with maps. This was not a prerequisite but helped to get useful feedback and suggestions on the maps. The questionnaire directs the tester to the main page of the website and provides basic information about what they need to do.

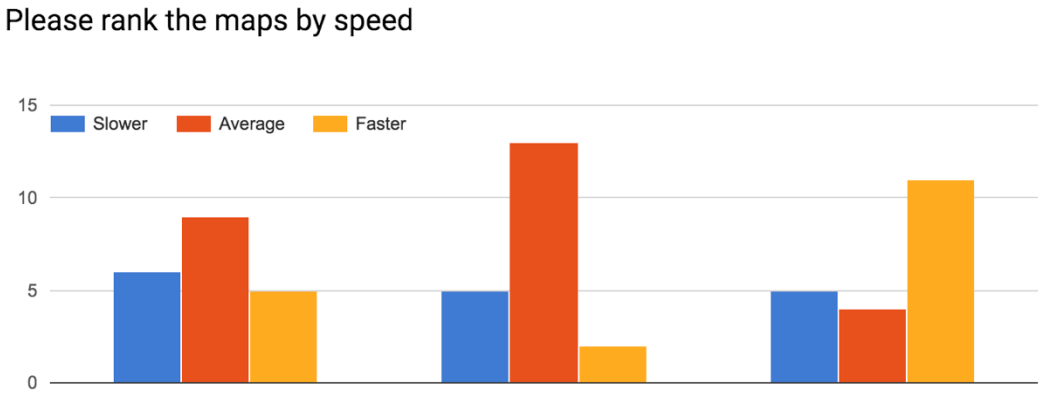
Flight map

The first six questions of the questionnaire are related to the flight map after directing the users to make some basic interactions with the maps such as finding flights.



Question 1

Although 85% of the testers experienced difference in speed between the maps, the ranking results are not that obvious (Question 1). Leaflet was said to be slower than the other two maps by six people but at the same time five tester felt like it was faster than the others.

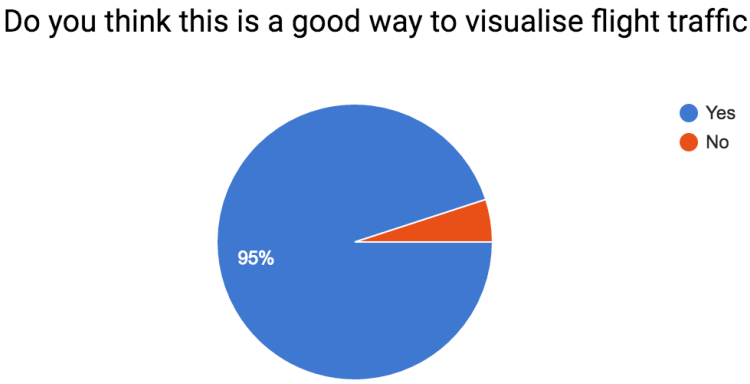


Question 2

As it can be seen on Question 2, Openlayers received thirteen average votes which is the most uniform result. Finally, it seems to be the most number of users felt Google Maps is faster than the others, since it got eleven polls in the “faster” row.

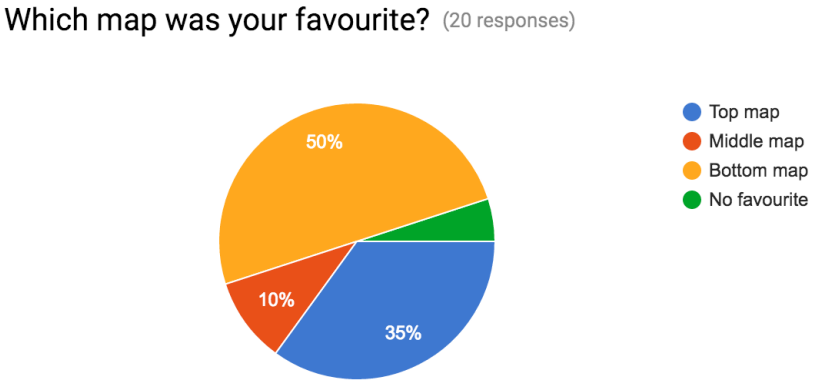
The results show that Google Maps performed slightly better while Leaflet produced slightly worse results in the speed comparison, although as it can be seen this test provided quite scattered results.

The answers for the Question 3 were more unanimous. It shows that 95% of the testers think that the moving plane icons are a good way to visualise air traffic. Based on the comments the users were fond of the real time behaviour and the simplicity of the map both from understanding and from usage perspective.



Question 3

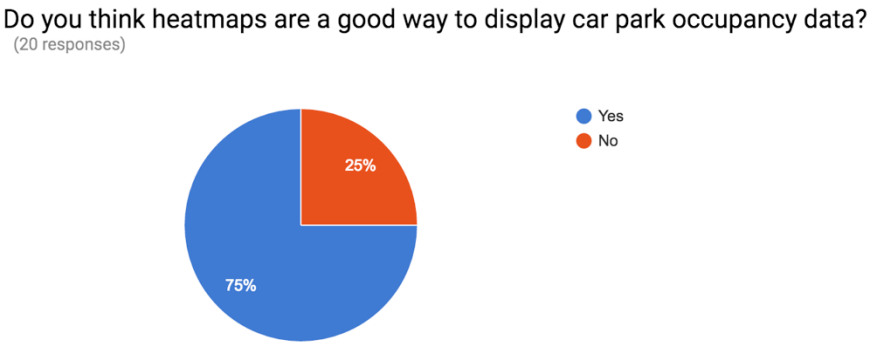
In the last question (Question 4) the users needed to choose a favourite map, where Google Maps seemed to be the most popular with half of the votes. While Leaflet got respectable 35%, Openlayers was not too popular between the testers. Only two people responded that they liked the Middle Map the most.



Question 4

Glasgow map

The second part of the questionnaire focused on the recorded Glasgow car park occupancy and traffic events map asking similar questions as in the first part.



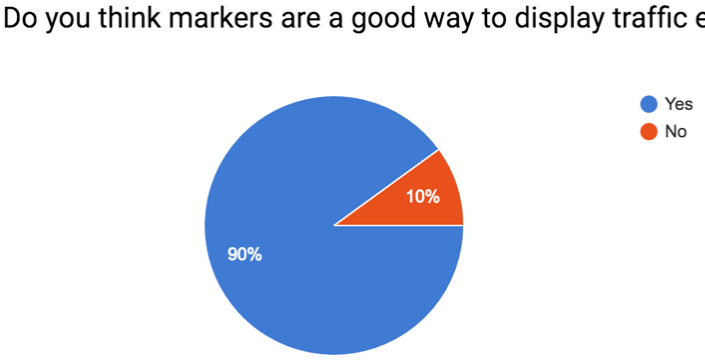
Question 5

The heatmaps got the lowest effectiveness score from the visualisation techniques. Only 75% of the responders think that the heatmaps are a good way to visualise car park occupancy data (Question 5).

Based on the user feedbacks such as:" *It is difficult to make a quantitative decision based on a colour at a single point in time. Would have been more useful if there was also a percentage occupancy too.*" It can be said that in general the users liked the idea of displaying the busier parts of the city but, the exact locations with the exact number of free spaces or occupancy percentages were missed from the maps. The testing indicated as well, that the car parks close to each other on higher zoom level can dilute or pollute each other's data, making their individual occupancy levels look inaccurate.

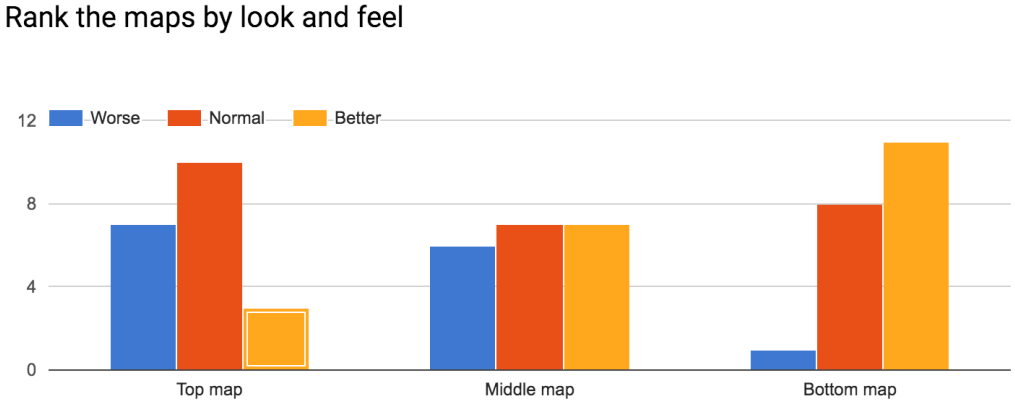
Markers

The markers for visualising road traffic events were more coherent according to the users, since 90% agreed with this technique being good for the purpose, as can be seen in Question 6. Although a few comments arrived indicating that without clicking them it is difficult to figure out what the markers' goal is. "*It's not the most obvious what they are until you click them*".



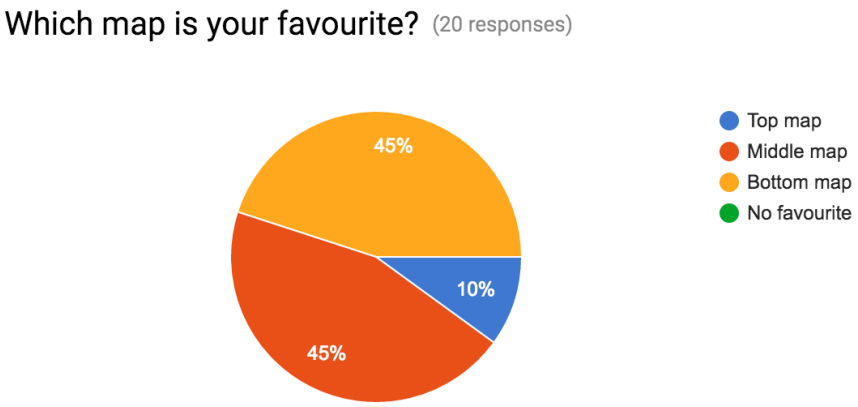
Question 6

As a part of Question 7 the users needed to rank the maps by look and feel. This comparison resulted in a clearer answer. Openlayers became quite average with almost the same amount of people voting it worse as normal or better. While Leaflet got a little more "worse" votes than the Openlayers, it received half as many "better" ones so it finished on the third place. Google Maps got the best results in this comparison with only one "worse" vote and with more "better" ones than the two other frameworks together.



Question 7

The previous results more or less reflect the outcome of the last question, Question 8. Only two people chose Leaflet as the favourite map while Google Maps shared the remaining votes equally with Openlayers.



Question 8

In general, the results indicated that Google Maps was the most popular from the client's perspective. It finished in first place in almost every question which can be related to that the fact that users probably are familiar with the Google Maps map tiles. Leaflet was the least popular from the Glasgow maps. Based on the user feedback such as "*I like the base map, but think the colours on the heatmap would be better going from green to red*" the Leaflet was not that successful. The different, less intense heatmap colours which were used, were not appreciated on the Leaflet map. This could be the reason why it received "worse" votes from the testers.

6 Results, conclusion and recommendation

6.1 Summary

During the research a detailed comparison was made on the Leaflet, Openlayers and Google Maps frameworks from a dynamic data visualisation perspective for both the developer and user. The comparison was based on different methodologies which helped analyse the frameworks on different levels. It included the development of an application to compare the frameworks side by side which provided the basis of the developer side comparison along with the literature reviews. The comparison from the user point of view was based on a questionnaire which was filled in after interacting with the map. The questionnaire also collected some feedback on the effectiveness of the different dynamically changing data visualisations.

The research revealed the strengths and the weaknesses of each framework and demonstrated how difficult it was to develop the various features which were needed for the dynamic visualisations and what support was provided by the frameworks. The questionnaire indicated which framework was preferred by the users and which visualisation type they considered to be effective. It also suggested some possible improvements and ways in which the application can be developed further.

6.2 Key Findings

This subchapter highlights the strengths and the weaknesses of each framework, evokes the key findings during the research and states a general outcome of the comparison. The framework also summarises the evaluation of the different visualisation effectiveness.

Openlayers

Openlayers is the most used open source mapping framework with a wide range of integrated features. Although it provides support for almost all the requirements which are needed for visualising dynamically changing data, there were some features which could not be implemented with Openlayers. On the other hand, the size of the framework is bigger than the other two, including the API's and it is more complex as well. Moreover, Fernandes et al. (Fernandes, Goulão, & Rodrigues) in the reviewed research and the application development revealed that the developers can face difficulties with the Openlayers framework because of the weak support and the significant API changes between the

different versions. These findings can be considered as primary data which proved the conclusion of Fernandes et al. Since the framework uses different projections than the other two frameworks, transforming the coordinates slowed down the development process as well. Even taking into account that the framework finished in the second place on the client side popularity comparison, when aggregating all of the results, Openlayers is the least recommended from the three frameworks for dynamic data visualisations.

Leaflet

Leaflet is the fastest evolving open source mapping framework with a lightweight framework. Although the core framework has limited functionality it is easily extendable with any required features by the wide plugin system. It was straightforward, easy to find, and easy to use plugins for implementing the constraints. For the developer side evaluation, it received the best results out of all three frameworks thanks to the good community support and the flexible architecture. Although the framework got satisfying results on the flight map comparison, it was the least popular Glasgow map because the users didn't like that the heatmap didn't use the standard colours. In general, if one prefers to work with an open source technology for dynamic data visualisation the Leaflet framework is definitely a good choice.

Google Maps

Google Maps as the biggest and most widely used map framework provides a decent sized framework with high abstraction and low complexity API. At the developer side comparison Google Maps finished with good results, just missing first place because of the lack of image rotation. Working with the framework is pretty fast thanks to the good API documentation and the tutorials. From the client side, people liked Google Maps the most. Unlike what was deducted from the work of Fernandes et al. (Fernandes, Goulão, & Rodrigues) the framework is flexible enough for implementing the visualisations. In general, if someone doesn't mind working, or above a certain usage even paying for the licensed framework Google Maps with Google behind it is probably one of the best options.

Effectiveness of visualisations

The other question which the research aimed to answer is related to the effectiveness of the different dynamic data visualisations. The questionnaire tried to find an answer to the previous as well.

Based on the feedbacks the moving icons are a good way to visualise moving traffic such as flights. The movement can be tracked in real time and if the icons are clickable they can provide additional information about traffic participants. To display the more static traffic events, the data which is changing in existence, appearing and disappearing icons seem to be a satisfying solution as well. Unlike the previous two the car park occupancy visualisation as heatmaps wasn't that successful. The users missed the more accurate information, although most of them liked the basic, colouring concept which was based on the work of Wood et al (Wood, Slingsby, & Dykes, 2011). Most likely the technique could be improved with clickable locations or a layer which can be switched on and off to display more details.

6.3 Weaknesses of the study and recommendations

Since this research is the first in the area of web mapping framework comparison from a dynamic visualisation perspective, it is likely that the project's methodology could be improved as there are plenty of other ways to approach the analysis differently.

The comparison based on the development of the application revealed the main strengths and weaknesses of the frameworks but since it was done only by one developer, it doesn't provide wide coverage. To validate the results, similar testing could be done by multiple developers.

Application improvements

In general, the comparison could be made easier if all the maps moved together just by interacting with one. For example, if a user zooms or pans on the top map all the three would do the same.

The users who tested the application could leave feedback and suggestions on how the maps could be improved. The feedback is processed here as well, providing ideas for the possible improvements.

The users were more or less satisfied with the flight map but there were few features which could be added to widen the functionality. A useful improvement would be to add a search field, similar to the search field of standard street maps. The field could be used for tracking the flights by their flight number or other properties.

Regarding the Glasgow map, a feature could be added to indicate more specific information about the car parks and the occupancy, such as location and occupancy percentage or the number of free spaces. As another direction, the heatmap technology could be replaced with something similar. For example, different coloured or sized objects such as circles could indicate the occupancy levels of the car parks.

Moreover, as the original plan was, real time car traffic data could be displayed to increase the usability of the maps. It not only would provide useful information but would correlate better with the car park occupancy and with the traffic events information as well.

In general, the application could be enhanced with other types of live data, showing different types of visualisations too. Another area of future development could be to extend the whole research by adding new frameworks to the comparison, since there are plenty other web mapping frameworks out on the market.

Questionnaire improvements

Even if it can be said that the right questions were asked in the questionnaire, there are possible improvements from the user side evaluation perspective. It is possible that the testers voted on the Google Maps implementations as a favourite map because the tile layers were more familiar than the other two. To eliminate all the differences between the maps the same map tile layers could be used. In this case the users would only evaluate the framework differences not the framework and map tile differences together.

Although 85% of the responders indicated that they noticed speed difference between the maps, the results didn’t show a clear order between them. It is most likely that the maps have performance effects on each other, so a better, automated method could be developed to test the speed differences. This could be used on the separate maps to produce exact numbers for the loading and using speed of the maps.

7 Bibliography

Andrienko, G. e. (2008). Geovisualization of dynamics, movement and change: key issues and developing approaches in visualization research. *Information Visualization 7.3-4* , 173.

Andrienko, N., Andrienko, G., & Gatalsky, P. (2003). Exploratory spatio-temporal visualization: an analytical review. *Journal of Visual Languages and Computing* .

Barclays Cycle Hire Map. (n.d.). Retrieved from Transport London: https://web.barclayscyclehire.tfl.gov.uk/maps

Bbecquet. (n.d.). *Leaflet Rotated Marker*. Retrieved 2016, from Github: https://github.com/bbecquet/Leaflet.RotatedMarker

City of Boston. (n.d.). Retrieved from Data Boston: https://data.cityofboston.gov/City-Services/Rodent-Activity-open-cases-9-4-13/ynt4-n6g9

Crickard, P. *Leaflet.js Essentials*. Packt publishing.

Express. (n.d.). Retrieved 2016, from Using template engines with Express: https://expressjs.com/en/guide/using-template-engines.html

Express JS Documentation. (n.d.). Retrieved 2016, from https://expressjs.com/

Fernandes, A. I., Goulão, M., & Rodrigues, A. (n.d.). A Comparison of Maps Application Programming Interfaces.

Geofabrik. (n.d.). *Map Compare*. Retrieved 2016, from GEOFABRIK tools: http://tools.geofabrik.de/mc

Ghoniem, M., Fekete, J.-D., & Cstagliola, P. (2004). A comparison of the readability of graphs using node-link and matrix-based representations. INFOVIS.

GIS - geographic information systems. (n.d.). Retrieved from http://nationalgeographic.org/encyclopedia/geographic-information-system-gis/

Glasgow Open Data Portal - Datasets. (n.d.). Retrieved from Glasgow Open Data Portal: https://data.glasgow.gov.uk/dataset

Google. (n.d.). *Google Maps APIs*. Retrieved 2016, from https://developers.google.com/maps/documentation/javascript/get-api-key#key

Google. (n.d.). *Google Maps JavaScript API Usage Limits*. Retrieved 2016, from Google Maps API: https://developers.google.com/maps/documentation/javascript/usage

Google. (n.d.). *Our history in details*. Retrieved 2016, from Google : https://www.google.co.uk/about/company/history/#2005

Gratier, T., Spencer, P., & Hazzard, E. (2015). *OpenLayers 3 : Beginner's Guide*. Birmingham: Packt Publishing Ltd.

Lovelace, R. (2014, March 4). *Testing web map APIs - Google vs Openlayers vs Leaflet*. Retrieved 2016, from http://robinlovelace.net/software/2014/03/05/webmap-test.html

OpenStreetMap Foundation. (n.d.). Retrieved 2016, from https://wiki.osmfoundation.org/wiki/Main_Page

Ortega, I. S. (2015). Leaflet vs Openlayers: which is the best for our indoor maps? *FOSS4G 2015 Seoul*. Seoul.

Perliedman. (n.d.). *Leaflet Realtime*. Retrieved 2016, from Github: https://github.com/perliedman/leaflet-realtime

Phan, D. e. (2005). Flow map layout. *IEEE Symposium on Information Visualization* (pp. 219-224). IEEE.

Rae, A. (2009). From spatial interaction data to spatial interaction information? Geovisualisation and spatial structures of migration from the 2001 UK census. *Computers, Environment and Urban Systems 33.3* , 161-178.

RNSInformatics. (n.d.). *Development Models*. Retrieved 2016, from http://www.rnsinformatics.com/development-models.shtml

Simo, V. (n.d.). *Request - Github*. Retrieved from https://github.com/request/request

Soeters, J. (2016). *JavaScript, AngularJS, React, Node, Express and all things web*. Retrieved from Understanding the Express app.js: http://jilles.me/getting-the-express-app-js/

The GeoJSON format specification. (n.d.). Retrieved 2016, from GeoJSON: http://geojson.org/geojson-spec.html

The MongoDB 3.2 Manual. (n.d.). Retrieved 2016, from MongoB Documentation: https://docs.mongodb.com/manual/

The OpenSky Network API. (n.d.). Retrieved 2016, from OpenSky Network: https://opensky-network.org/apidoc/rest.html

Viskin. (n.d.). *marker-animate-unobtrusive*. Retrieved 2016, from Github: https://github.com/terikon/marker-animate-unobtrusive

Wanderdrone. (n.d.). Retrieved 2016, from https://wanderdrone.appspot.com/

Web Mercator. (n.d.). Retrieved 2016, from Wikipedia: https://en.wikipedia.org/wiki/Web_Mercator

Wilkinson, L., & Friendly, M. (2009). The history of the cluster heat map. *The American Statistician* , 179-84.

Wood, J., Slingsby, A., & Dykes, J. (2011). Visualizing the Dynamics of London's Bicycle-Hire Scheme. In *Cartographica* (pp. 239–251).

Appendix 1 – Questionnaire

Map comparison

As a part of my dissertation I examine the efficiency and usability of various real time data visualisations on different web based maps. The aim of the survey is to collect subjective user opinion of the maps.
The survey takes approximately five minutes.
Your participation is voluntary and you can abandon the questionnaire any time if you want.
Your responses are completely anonymous and will only be processed in the dissertation report.

NEXT

Never submit passwords through Google Forms.

Map comparison

Open the map in a desktop browser (not IE)

http://66.155.19.223/

Please open the application's main screen in another window.

BACK NEXT

Never submit passwords through Google Forms.

Map comparison

Flight map

The map displays realtime flight traffic information

Click on the flight map

Please interact with all the maps

For example zoom in on a flight.
Locate a flight which took off in the UK.
Locate a flight above the sea.

BACK NEXT

Never submit passwords through Google Forms.

Map comparison

*Required

Did you notice any difference between the maps in speed? (Loading, Panning, Zooming) *

- ◯ Yes
- ◯ No

BACK NEXT

Never submit passwords through Google Forms.

Map comparison

*Required

Please rank the maps by speed *

	Slower	Average	Faster
Top Map	○	○	○
Middle Map	○	○	○
Bottom Map	○	○	○

BACK NEXT

Never submit passwords through Google Forms.

Map comparison

Issues, suggestions

Please record any issues or make suggestions on possible improvements or additions (leave blank if you don't have any)

Overall

Your answer

Top map

Your answer

Middle map

Your answer

Bottom map

Your answer

BACK NEXT

Never submit passwords through Google Forms.

Map comparison

*Required

Do you think this is a good way to visualise flight traffic data? *

- ○ Yes
- ○ No

BACK NEXT

Never submit passwords through Google Forms.

Map comparison

Why? (Optional)

Your answer

BACK NEXT

Never submit passwords through Google Forms.

Map comparison

*Required

Which map was your favourite? *

- ◯ Top map
- ◯ Middle map
- ◯ Bottom map
- ◯ No favourite

BACK NEXT

Never submit passwords through Google Forms.

Map comparison

Now go back to main screen of the map application and choose the Glasgow map

BACK NEXT

Never submit passwords through Google Forms.

Map comparison

Car park occupancy and traffic events map

This map displays recorded car park occupancy as a heatmap and traffic events as icons. The animation is speeded up and the time in the top right corner displays when the data was recorded.

Please interact with all the maps

Look at how the carpark occupancy changes. Examine the carparks from different zoom levels.
Look at the traffic events. Read the description of one event.

BACK NEXT

Never submit passwords through Google Forms.

Map comparison

*Required

Do you think heatmaps are a good way to display car park occupancy data? *

- ○ Yes
- ○ No

BACK NEXT

Never submit passwords through Google Forms.

Map comparison

Why? (Optional)

Your answer

BACK NEXT

Never submit passwords through Google Forms.

Map comparison

*Required

Do you think markers are a good way to display traffic events? *

- Yes
- No

BACK NEXT

Never submit passwords through Google Forms.

Map comparison

Why? (Optional)

Your answer

BACK NEXT

Never submit passwords through Google Forms.

Map comparison

*Required

Rank the maps by look and feel *

	Worse	Normal	Better
Top map	○	○	○
Middle map	○	○	○
Bottom map	○	○	○

BACK NEXT

Never submit passwords through Google Forms.

Map comparison

Issues, suggestions

Please record any issues or make suggestions on possible improvements or additions (leave blank if you don't have any)

Overall

Your answer

Top map

Your answer

Middle map

Your answer

Bottom map

Your answer

BACK NEXT

Never submit passwords through Google Forms.

Map comparison

*Required

Which map is your favourite? *

- ○ Top map
- ○ Middle map
- ○ Bottom map
- ○ No favourite

BACK NEXT

Never submit passwords through Google Forms.

Map comparison

Thank you!

After submitting please don't forget to close the maps!

BACK SUBMIT

Never submit passwords through Google Forms.