Submitted for the Degree of B.Sc. in Computer Science 2019-2020

City Planner

201647252

Leonard Magyar

Supervisor: Clemens Kupke

Except where explicitly stated all the work in this report, including appendices, is my own and was carried out during my final year. It has not been submitted for assessment in any other context.

I agree to this material being made available in whole or in part to benefit the education of future students

Signature:

Abstract

Applications that show the measure of deprivation in data zones all over the UK have been around since 2012. These applications take data collected by the government and rank data zones based on multiple domains. Data zones are groups of 2011 Census output areas which have populations of around 500 to 1,000 residents. City planner uses data from the huge variety of open data sets about data zones in Scotland available through government and data research sites. It also attempts to translate this into valuable information by examining travel times from data zones to a nearest selected key facility and allowing users to place facilities in data zones that are potentially underdeveloped. During the development of this project several interesting insights were discovered, with identifying areas in Glasgow and Edinburgh where an establishment of a key facility could positively benefit hundreds of residents' lives staying in the given neighbourhood. Evaluations of the system proved that the web application created was intuitive and communicated the underlying data to users effectively.

Acknowledgements

Firstly, I would like to thank my supervisor, Clemens Kupke, for all the help and valuable feedback and support he provided me throughout this project. His input was certainly helpful in successfully completing the project and making sure I was on track during the whole process.

Furthermore, I would also like to thank my second marker, Professor Roma Maguire, for her comments and observations on the project during the project progress presentation.

Finally, I would like to thank all of the people who volunteered to take part in the surveys and evaluations.

Table of Contents

1 Project Aims and Objectives

1.1 Aim of Project and General Problem

Extracting useful information from data sets and applying them to solve real-world problems around a city can be challenging. There are a huge variety of open data sets about areas in Glasgow available through government and data research sites.

City Planner will explore the possibility of translating these data sets gathered about the city into valuable information and aid city planners where to improve local services in Glasgow. The tool should help to understand available data sets better and determine where new schools, general practices, post offices and shopping facilities should be built by marking under-provided areas.

Another challenge that this project will attempt to overcome is that data sets represent locations of the facilities with different geography (i.e. data zone, intermediate Zone , postcode area). It is essential that the application will be able to deal with data set(s) that provide locations of facilities in the previously mentioned way as it would significantly increase the number of usable data sets and areas that the application could be used for. Expanding the application so that it would not only look at the area of Glasgow but other main cities in Scotland too would be beneficial and could be looked at during development.

1.2 Main Objectives

Main objectives for the project are as follows:

- Collect useful open government data about areas (data zones, wards, postcode areas) in Glasgow
- Data from collected sets will be used to relatively compare areas in Glasgow based on vicinity of facilities in the area
- Suggest areas for new facilities that have maximum positive impact on the general level of provision

2 Related Work

2.1 Scottish Index of Multiple Deprivation 2016

(https://simd.scot)

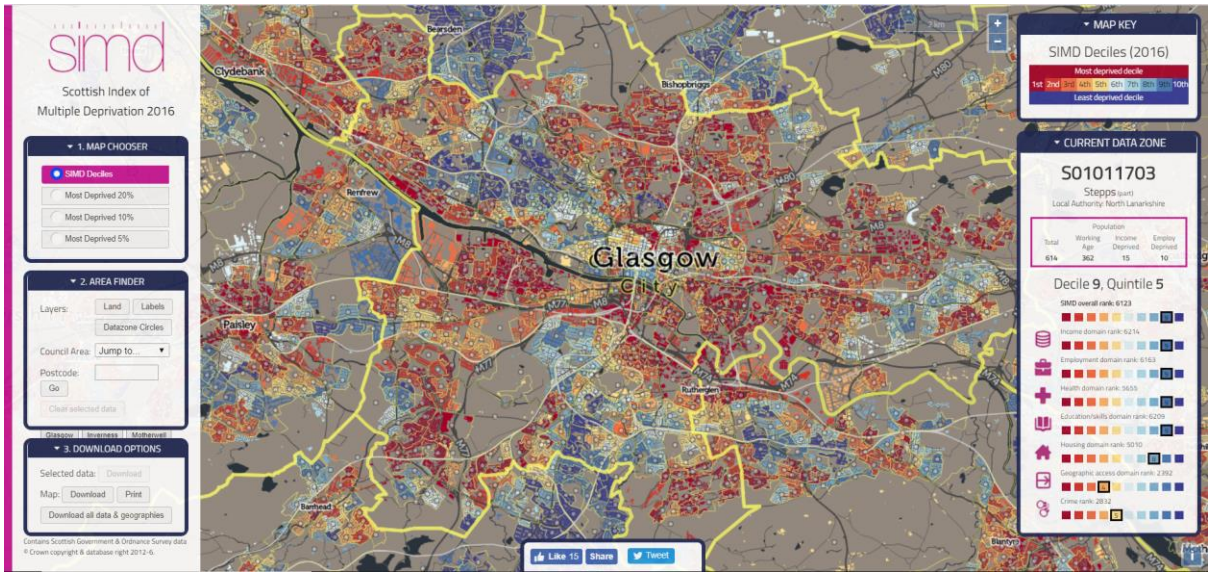


Figure 1: Multiple Deprivation indices in the data zones of Glasgow

2.1.1 Description of related project

SIMD (Scottish Index of Multiple Deprivation) tool consistently analyses multiple deprivation in small are concentrations in all of Scotland.

This tool aids concentrating policies and funding in order to deal with area concentrations of multiple deprivation. The first SIMD was released in 2004 and this is the fifth edition the government published since then. It merges seven domains from which severity of deprivation could be derived. These domains are; income, employment, health, education, skill and training, geographic access to services, crime and housing.

Domains form ranks after they have been measured by several indexes. Data zones are then also ranked from 1 to 6,976 where 1 is the most deprived and 6,976 is the least deprived area. Afterwards, an overall SIMD is calculated by combining all seven domain ranks. This ranking is sufficient enough to inform the user that one data zone is relatively more or less deprived than another data zone. Deep red colour indicates the most deprived data zones, whereas deep blue colour indicated the least deprived data zones [1].

SIMD has three other visible components besides the map itself. It has a side bar on the left and two other windows on the right side of the screen. On the left sidebar there are three different panels where you can modify the map. The first one (Map Chooser) will let you choose to look at all the data zones(default) or the lowest 20%, 10% or 5% of the deprived data zones. In the second panel

(Area finder) you can change the view of the map, jump to council zones and look for post codes. Finally, you can also data from the selected data zones or print out the map. On the right panels you can also see how colours presented on the map correspond to the SIMD ranking, and see the domain and overall ranking of each data zone.

2.1.2 Comparison with City Planner and suggested improvements

City Planner will primarily look at a single domain selected by the user and will not calculate an overall score. It will indicate how well/badly a data zone is equipped with the selected facility. Firstly, it will show only data zones that are in Glasgow, then might look at other main cities in Scotland too.

The City Planner is aiming to be similar in terms of displaying data zones' borders and indicating with a range colours how each zone is ranked in terms of facilities. However, SIMD does not show each data zone in a uniform colour in its default view, not residential areas are marked with grey and residential areas are marked with the colour according to their ranking. If the layer is changed to 'land' the view will be like what City Planner will try achieving. Indicating a data zone with a uniform colour not making difference between residential and not residential areas.

2.2 University of Strathclyde - Institute for Future Cities

(http://ifuturecities.com/glasgowatlas/)

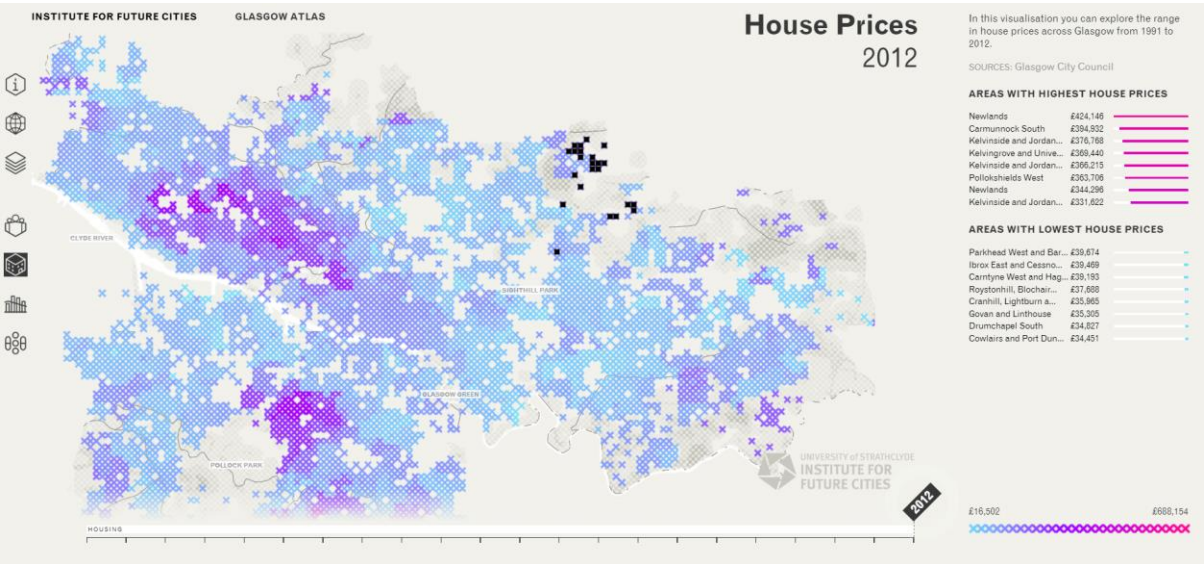


Figure 2: Institute for Future Cities tool displaying housing prices (2012) in the data zones of Glasgow.

2.2.1 Description of related project

This visualisation system created by the Institute for Future Cities is uncovering different types of data that are related to urban environment in Glasgow. This tool is used for splitting those data into layers, so they can be more understandable by city planners and citizens. In the Future Cities tool, we can look at four different domains regarding Glasgow; population, house price, deprivation and

drug misuse. However, we are only able to look at these domains individually and there is not a calculated total score of all domains

There is an additional feature to the website which allows users to look at megacities around the world (A megacity is usually defined as a metropolitan area with a total population in excess of ten million people) on the map and using the previously mentioned slider allows users to see how the population of these cities have changed throughout the last few decades.

2.2.2 Comparison with City Planner and suggested improvements

City Planner will look at different domains, however it will also let users look at single domains and see the corresponding map. City Planner will not look at historical data, therefore won't implement a slide bar similar to this tool.

The main improvement I see regarding Future Cities is the way it displays the data zones. The zones are displayed in pixels therefore it is difficult to differentiate between them and when a user hovers above a data zone the pixels which correspond to the area are not adjacent.

2.3 Indices of Deprivation 2019 explorer

(http://dclgapps.communities.gov.uk/imd/iod_index.html)

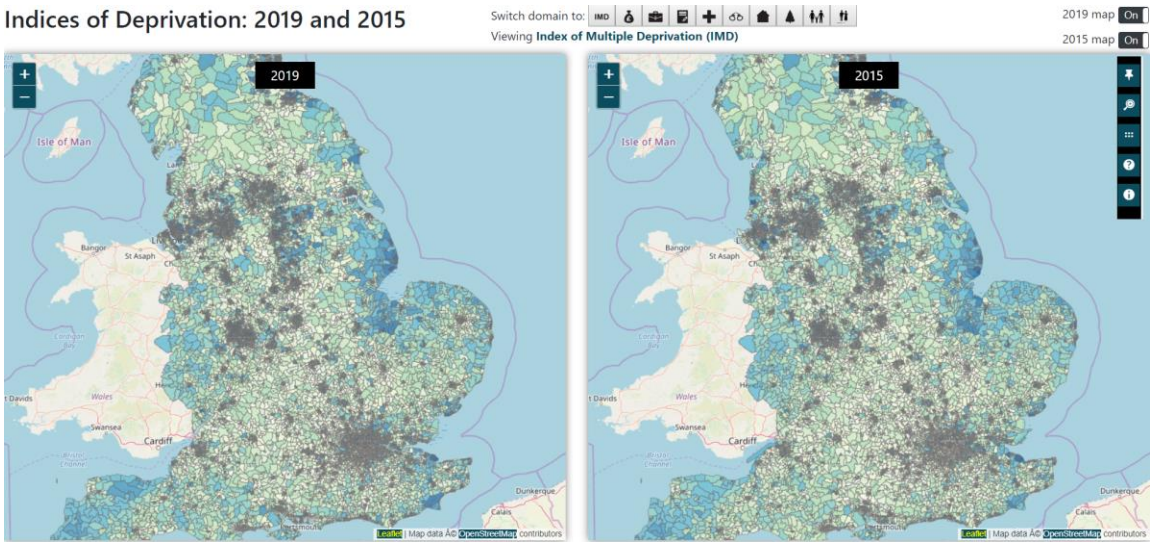


Figure 3: Comparing Indices of Deprivation of LSOAs from 2015 and 2019

2.3.1 Description of related project

Users of this application are shown the deprivation of selected LSOAs (Lower Super Output Areas) in England. Both the 2015 and 2019 indices of deprivation can be seen and compared side by side. It also enables users to look up data by entering postcode.

Lower Layer Super Output areas are similar to Data Zones in Scotland as they are both geographic areas made to aid statistics for smaller areas. Minimum population for a LSOA is 1000 and mean is around 1500 [2].

In conclusion, this application is almost identical to the SIMD application, with the main difference being at able to compare data from two different collection, the area which is being looked at and being able to select individual domains.

2.3.2 Comparison with City Planner and suggested improvements

The explorer includes a dashboard which provides a brief summary of how relatively deprived the area selected is in each iteration. Data can be downloaded directly using this tool. These features won't be implemented in City Planner.

However, users are also able to choose between individual domains and see their corresponding ranking for each LSOA which is a feature that will be available in the City Planner application as well.

2.4 2013 US Area Deprivation Index – Neighborhood Atlas (University of Wisconsin)

(https://www.neighborhoodatlas.medicine.wisc.edu/mapping)

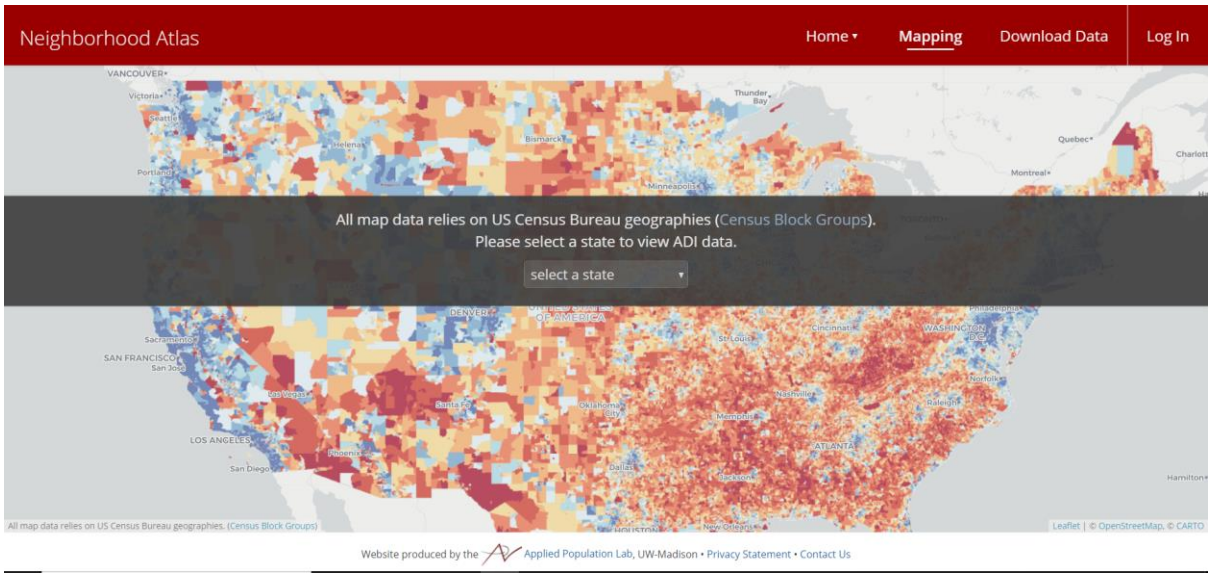


Figure 4: Initial page of 2013 US Area Deprivation Index Atlas

2.4.1 Description of related project

In this application users are enabled to select a state of the USA or the entire country and view the selected area mapped by 2015 ADI (Area Deprivation Index). ADI is similar to SIMD as they both represent a geographic area-based measure for deprivation in a certain area.

More deprived areas are indicated with higher index values. These deprived areas been associated with high risk of disadvantageous health and health care outcomes.

The ADIs on this website can be seen in percentile rankings at the block group level from 1 to 100. Groups that have are ranked 1 are the lowest ADI and groups that have ranked 100 is the highest ADI. The block groups that have the lowest level of deprivation have a ranking of 1, and the block groups that have the highest level of deprivation have a ranking of 100 [3].

2.4.2 Comparison with City Planner and suggested improvements

Upon opening the application, the user is asked which state's ADI data would like to examine. There is also a potential of creating a similar feature in the City Planner application where the users would not only be able to examine data zones and facilities that are located in Glasgow but in a few additional cities too.

Furthermore, both applications are using Leaflet open-source JavaScript library to display the tile layer and map. In overall, Neighbourhood Atlas is almost identical to SIMD application and will have some similarities to City Planner (i.e. how the areas are represented), however it will look at different domains to determine a data zone's level of disadvantage.

2.5 Atlas by Scottish Government

(https://statistics.gov.scot/atlas/resource?uri=http://statistics.gov.scot/id/statistical-geography/S92000003)

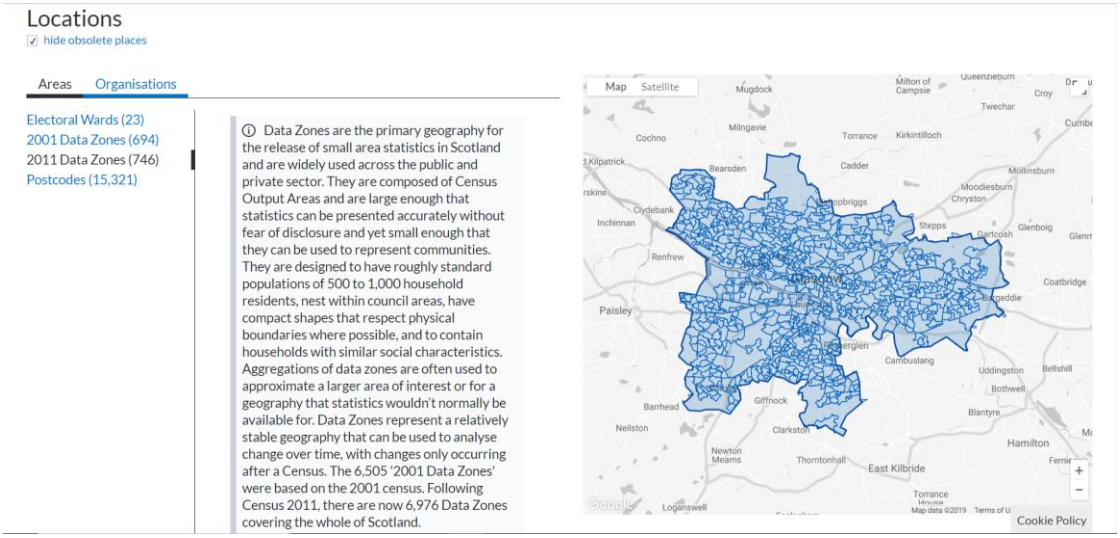


Figure 5: Scottish Government Atlas displaying data zones of Glasgow

2.5.1 Description of related project

This tool is used to explore areas about which the Scottish government publish data. On this website you can view statistics about a selected area and compare it to whole of Scotland, you can browse data sets that include the selected area and also look at the location of the area and select other areas and organisations.

Indicator	Period	Abbeyhill - 01	Abbeyhill	City of Edinburgh	Scotland
Population	2018	1,230	2,893	518,500	5,438,100
Births	2018	7	20	4,899	51,308
Deaths	2017	16	19	4,251	57,290

Figure 6: Scottish Government Atlas comparing Abbeyhill-01 data zone's statistics with its superior geographic areas (Intermediate Zone, Council Area, Country)

2.5.2 Comparison with City Planner and suggested improvements

City Planner's view will be very similar to what we can see in the Atlas tool when (see first picture from the tool) a user selects a Council Area and view its data zones. However, data zones would be coloured according to their facility ranking.

One feature that could be added to the Atlas tool would be a post code search bar, that would return the data zone that corresponds to the looked-up post code so you can view statistics of the data zone and compare them against nationwide data. Also, the Atlas tool uses Google Map to render out the map and the selected areas.

However, when there is too many data zones or areas to display the tool comes back with a warning saying that there are too many items in the list therefore it won't display them. The application also has a full screen mode, which could be useful for the City Planner tool as well as city planners might find it more comfortable and easier to focus on their selected area.

3 Project Specification

3.1 Functional Requirements

3.1.1 Must have:

- Access information about facilities and their data zones
- View map of Glasgow
- View boundary and area of each data zone in Glasgow
- Select different individual domains
- Query appropriate data
- An algorithm which will decide how a colour will correspond to a ranking depending on the selected domain
- An algorithm which assigns a colour to each data zone depending on the selected domain
- View coloured map of Glasgow according to selected domain

3.1.2 Should have:

- A menu which enables user to select different domain's icon
- Add a single facility from the selected domain into a data zone
- View updated map of data zones
- List of least well developed areas in Glasgow when looking at a data set
- A drop down box where a user could select between bigger cities in Scotland(i.e. Glasgow, Edinburgh, Dundee, Inverness) and have same functionalities as mentioned in 'Must have' but with different city's data zones.

3.1.3 Could have:

- An interface to provide opportunity to import data
- Import shapefile/GeoJSON file and look at boundaries and area of imported data
- Import a data set about facilities
- View coloured map of imported location based on imported data
- A logo for the application
- Full Screen mode
- Downloadable data

3.1.4 Won't have

- An overall ranking of the domains
- A comparison/timeline with historical data
- Post code search/Area finder

- Select multiple data zones and compare them
- Mark each individual facility in a category on the map

3.2 Non-Functional Requirements

- Application should take less than 3 second to load initial screen.
- When selecting a domain, it shouldn't take more than 3 second to see the coloured map of Glasgow
- After adding a new facility into the map, it shouldn't take more than 3 second to insert it to the database and see difference in map if appropriate
- No database query should take more than 3 seconds
- Browser compatibility with Chrome, Firefox and IE

3.3 Project Plan and Summary of Progress

3.3.1 Project Plan and Milestones

Main deliverables for City Planner will consist of development milestones, documentation, evaluation and testing progress. Some of these deliverables are dependent on each other, however some aspects can be done independently and in parallel.

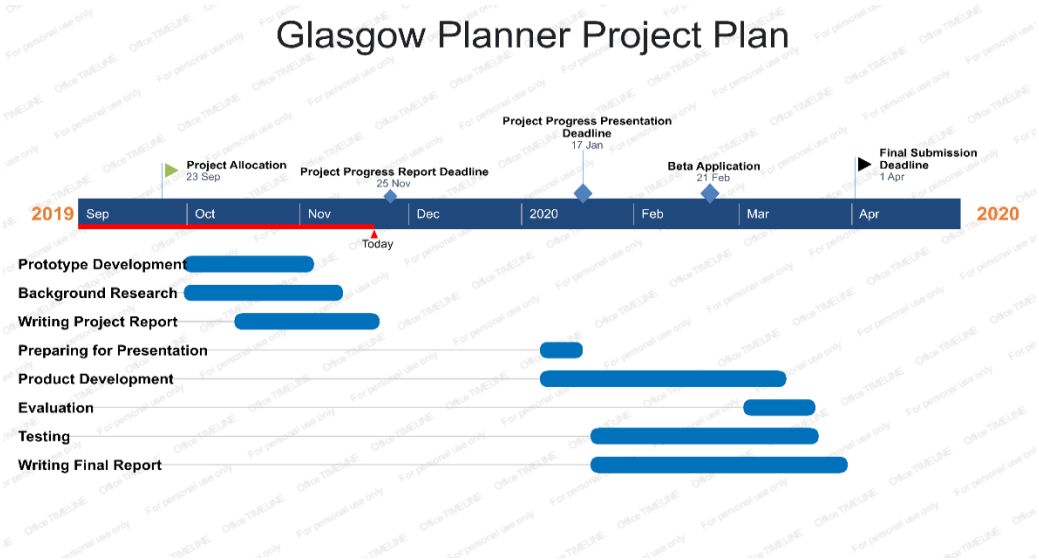


Figure 7: Gantt Chart for City Planer Project

Project plan for the first semester was mainly driven by two objectives. Firstly, to have a project progress report by end of November, secondly, to have a prototype application working by the same time. Application development and background research was started soon after project allocation and creating an initial project report followed shortly.

When writing the project report the initial background search helped significantly with the Related Work chapter as well as with the Project Aims and Objectives chapter. Another essential aspect of background research was looking at available open data sets. There are many resources where open data set can be found about Scotland and Glasgow (Government site, NHS, Glasgow Open Data, Urban Big Data Centre). However, it is essential to use data sets that will add valuable information and validate this project as main the goal is to help identify underdeveloped areas of Glasgow and suggest facilities that can be established in those areas.

After Project Report submission and for the first half of semester two the main milestones will be related to develop a beta application from the later mentioned already existing prototype application in the Summary of Progress. As some of the must-have functionalities already exist in the prototype application, additional must-have features should be developed in the following order by the 21st February (Week 5):

- Ability to select different individual domains on the application page (by Week 1)

- In order to achieve this feature deciding on which domain will be looked at is required prior to implementation as well as storing facilities' locations uniformly in the database. Government employees have been contacted in order to aid this decision and provide more value to the project.

- View coloured map of Glasgow according to selected domain (Week 2 and Week 3)

  - In order to achieve this feature an algorithm will be required which counts all the facilities in each data zone and assigns a colour to each data zone depending on the numbers.

- Allow adding new facilities and update the map accordingly (Week 4 and Week 5)

  - In order to achieve this a menu will need to be implemented and allowing to insert a new facility in the application.

These features align with the all mush-have and most should-have functionalities stated in the Project Specification chapter. Remaining should-have and could-have functionalities will be attempted to implemented between Week 6 and Week 8.

Week 9 and 10 will be spent finishing report and evaluation, alongside with functional testing of final version of the product.

4 System Design

This chapter will discuss how the City Planner application was designed. It will include software design and architecture in addition to development process. Furthermore, it will mention database design and user interface.

4.1 Architecture

City Planner is a single-page application(SPA) aimed to use on desktops. SPAs are dynamically loading information on the current page without having to reload the page during user interaction. Therefore, the application will not ruin the user experience with loading between successive pages.

The front-end of the application, which is dynamically rendering the information to the website was built using React - a JavaScript library. The front-end also includes some HTML and CSS code alongside JavaScript. Originally, the server-side of the application was written in Kotlin – a technology which runs on Java Virtual Machine and communicates with the database running on PostgreSQL engine. However, due to memory leak problems during deployment of the back-end application some architectural changes were required. In the final implementation, there is no back-end application, GeoJSON files are stored on an online JSON storage bin, n:point (see Chapter 5.7.1). The website, npoint.io, enables to create lightweight JSON endpoints, which an application can access using REST API and receive the requested JSON file. This change in architecture has no drawback to performance or user experience, they can still view travel times and add facilities to data zones. However, this alteration makes the application even more lightweight as it does not require a written back-end application.

There are three main components in the architecture of this project as previously mentioned. The front-end of the application which is only talking to the JSON endpoints and display visualisation of the data to the user. The JSON endpoint acts as a server to front-end as it is passing requested data from the client-side. An independent, however, very essential component of the architecture is the database which made handling and uniting the data sets used effortless. The data prepared in the database using some operations (Chapter 5.1) were uploaded the JSON endpoints where the front-end could access them. There is no direct communication between the JSON endpoints and the database as the endpoints just enable developers to upload JSON files and access them via REST API from the client-side.

4.2 Software Design

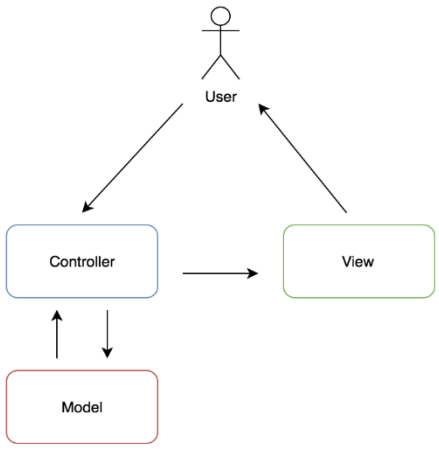


Figure 8: Initial software design: Model – View - Controller

Initially, Model-View-Controller was the software design aiming to be used. However, after the previously mentioned changes in architecture this was no longer a suitable design principle to follow as there was no continuous two-way communication between database and JSON storage endpoints. Although, the view remained unchanged as it is the front-end application which gets data from the JSON endpoints and displays it to the user in the browser. The model now became the endpoints where the view is getting the data from. It provides the view with the data that the user has requested. There is no controller in the final implementation which could be associated with the MVC design.

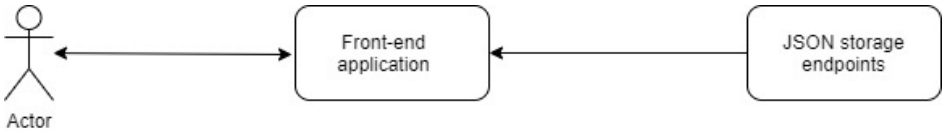


Figure 9: More lightweight software design in final implementation

4.3 Development Methodology and Process

When looking at development methodologies Agile was proposed at the beginning of the project. Working in Agile would mean working with continuous feedback and developing while taking client feedback into consideration. As opposed to Waterfall methodology where the product is being looked at by the user after the developer thinks the product is ready, working with Agile would mean showing progress and development to client frequently, therefore client requirements would be more efficient to achieve.

For proposed methodology an ideal client would have been needed, whose expertise is city planning and development and whose feedback and input would be exceptionally valuable to this project. Unfortunately, this person was not found, even though several attempts were made to reach out to the city council and ask for a contact who would be interested in overlooking the project as a client. Therefore, the main design methodology used was incremental development with some features of agile principles featured as well.

4.3.1 Incremental Development

In Incremental Development the development workload is divided into modules which build on each other, therefore they can be incrementally developed, tested and delivered. The core features will be developed first, then additional features are selected from the requirements depending on priority of the feature and then developed throughout successive versions of the applications.

Incremental Development is based on Waterfall methodology, however it is more flexible, as there is a chance to improve a feature before the delivering the whole product, making sure a feature is satisfactory to customer before building on more features on top of the previous one. In Waterfall method there is only one big process cycle before delivering the final product, while Incremental Development splits the process into smaller cycles based around the features of the application making sure each feature is satisfactory to the customer before moving on to the next one.

4.3.2 Agile

While Agile methodology is more suited to a team of developers than a single person, there are some features of it which still can be beneficial when applied to developing a whole project alone. Agile methodology is somewhat incremental, since there are features that need to be developed on a previously built feature, however there are concreted end dates, to which the feature should be developed by.

During the development of this project, an attempt was made to split development into two-weekly sprints – an Agile principle, and it helped to pace the workflow and helped to keep focusing on a single task or a feature during development process. Two-weekly sprints were felt the most suitable as supervisor meetings were held every two weeks, and this meant after each sprint valuable input and feedback were communicated by the supervisor and it helped to keep the project development on track.

4.3.3 Design Process

After following Incremental Development methodology, the development process was split into three main cycles. Firstly, the main feature and core of the product was implemented, where users could view all the data zones in Glasgow. Secondly, the feature where users could examine the data zones based on their distance to the

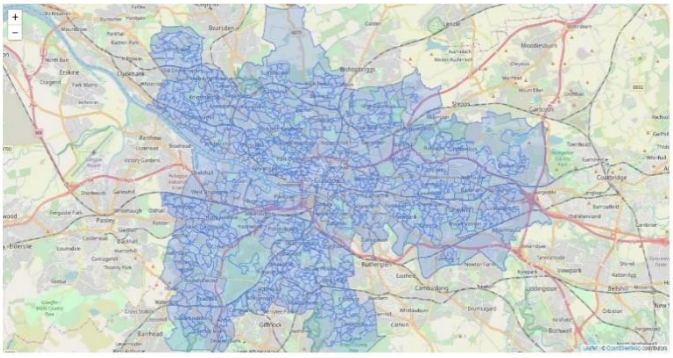


Figure 10: Prototype application displaying data zones in Glasgow

nearest select facility needed to be implemented. Finally, a tool by which the users could add a selected facility to a certain data zone was built. The second feature relies heavily on the first one, while the third one relies on the second one, therefore it was essential that each feature is robust and well maintained. Intermediate goals of each feature were set up and completed in previously mentioned two-weekly sprints. Setting up these intermediate tasks were helpful in order to keep focus on a single task and not to be overwhelmed by other requirements to be implemented at later stages.

4.4 User Interface

When designing the user interface, the focus was on having a clearly visible large map with only the necessary interactive UI components displayed around it which can be easily found and could provide additional value to the system. In order to achieve this, React framework was ideal as it has many features that suit this project. React is component based, where components have their own state and could be nested into each other to create more complex UI components. React is also declarative, which means there are different design views for each state a component has, and automatically re-renders the component when its state has changed. Therefore, it is also efficient because it keeps track of the states and only re-renders the components that have been changed.

Figure 11: Button components based on Semantic React UI

In addition, to React, another framework was used to create a powerful and tidy user interface. Semantic UI React is the official React integration for Semantic UI – a JavaScript library that provides a lightweight user and development experience. Using React Semantic UI significantly sped up the development process of the user interface and helped it easier to maintain and reuse code.

4.5 Database Design

When designing the database architecture, it was important that the front-end and the back-end could communicate easily with sending and receiving GeoJSON files. Therefore, it was essential to have the data zone boundaries and the associated travel times under one file, so the application would not have to query the database more than once to load all the information which is needed.

There are two table database tables that the system actively uses. However, there are many other tables that were used during the development in order create the tables that the application is communicating with.

The main tables are called glasgow and edinburgh and contain GeoJSON data about the two cities respectively. The GeoJSON data also includes the travel times to the nearest facilities as a property which were added, in order to make data query easier and faster. Details of this operation can be read in the next chapter.

Apart from the previously mentioned database tables, there are a number of tables found in the database which were helped to create the main tables. Some these tables include information about data zone boundaries, data zone lookup and travel times to nearest facility from a data zone. A list of all database tables being used during development and their details can be found below:

Table	Column	Type	Details
all_data_zones	boundary	JSONB	Boundary data from all data zones in Scotland
glasgow_data_zones	boundary	JSONB	Boundary data from all data zones in Glasgow
edinburgh_data_zones	boundary	JSONB	Boundary data from all data zones in Edinburgh
glasgow_data_codes	dataCode	text	Code of all data zones in Glasgow
edinburgh_data_codes	dataCode	text	Code of all data zones in Edinburgh
travel_times	dataCode	text	Data code of all data zones
	value	double	Travel time in minutes
	destination	text	Name of facilities
	methodOfTravel	text	By Car or Public Transport
edinburgh_travel_times	dataCode	text	Data code of all data zones in Edinburgh
	value	double	Travel time in minutes
	destination	text	Name of facilities
	methodOfTravel	text	By Car or Public Transport
glasgow_travel_times	dataCode	text	Data code of all data zones in Glasgow
	value	double	Travel time in minutes
	destination	text	Name of facilities
	methodOfTravel	text	By Car or Public Transport
glasgow	Boundary	JSONB	Boundary data from all data zones in Glasgow with travel times
edinburgh	Boundary	JSONB	Boundary data from all data zones in Edinburgh with travel times

5 Detailed Design and Implementation

In this chapter details of the implementation will be discussed alongside with gathering open data and facing main challenges along the development. It will also reason for chosen technologies, data structures and algorithms being used in the application.

5.1 Open Data Gathering

One of the main objectives of this project was to collect open government data about data zones. This was achieved by exploring different sources such as the official government website (statistics.gov.scot), NHS open data website (opendata.nhs.scot), and the website of the Urban Big Data Centre (ubdc.ac.uk) which is a national research and data service based in Glasgow.

The most suitable source for both boundaries of the data zones and determining data zones which might be under-provided with some kind of facility, proved to be the official government website as their data set included the most kind of facilities. Both data sets that are used in the application will be described in the next sections.

5.1.1 Boundaries of Data Zones

Collecting data about boundaries of data zones was the first step in implementing the project. An open data set is available at the official government statistics website which includes the boundary of all data zones in Scotland in shapefile format. However, this data set had four problems which needed to be solved before it could be used in the project.

5.1.1.1 Coordinate System Conversion

The shapefile stored coordinates of the boundaries as Ordnance Survey National Grid reference (also known as OSGB 1936) - a system of geographic grid references used in Great Britain, opposed to latitudes and longitudes. Therefore, a conversion, to the more widely used and uniform World Geodetic System (WGS 1984) was needed, so that it would be compatible with Leaflet framework which helps to put the boundaries on the map, and it would be more easily maintainable as well.

For achieving this, a tool called OGR2OGR was used. OGR2OGR is a command line tool which enables users to change projection and coordinate system of a map layer. As OGR2OGR is part of an open-source translator library called GDAL, it had to be installed first locally. After OGR2OGR was ready to use, the following command had to be run to convert shapefile coordinate system to WGS 1984:

```
ogr2ogr -t_srs EPSG:4326 map_wgs84.shp map_original.shp
```

The shapefile called map_wgs84.shp now contained data zone boundary information in the desired coordinate system.

5.1.1.2 Simplification

Secondly, as this file was 18 MB in size it needed to be simplified to make database queries and front-end to back-end communication faster. In order to achieve this, an online tool website, mapshaper.org was used. After importing the shapefile of boundaries in this tool, it allowed to simplify the shapefile to the 41.5% of the original file before it started cutting off certain of edges and blocks of data zones. Therefore, the file size is now reduced by 58.5%, and it resulted in the total size of 7.74 MB without significantly affecting the boundaries of data zones in bigger cities like Glasgow and Edinburgh.

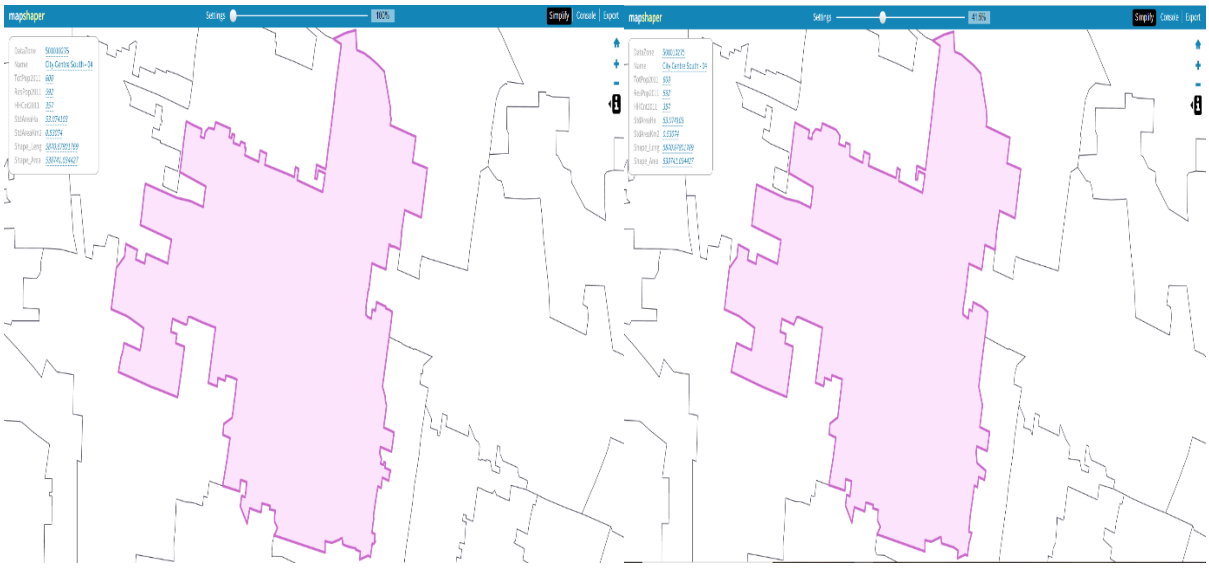


Figure 12 and 13: Comparing boundaries of City Centre South-04 data zone in Glasgow with different simplification levels – 100% (original file - left) and simplified to 41.5% (used in City Planner application - right)

5.1.1.3 File Format Conversion

However, the ideal format to use in this project was not shapefile, as it contains different files and is more difficult handle in a database and web application. GeoJSON was the desired format as it is designed as one BLOB (collection of binary data stored as a single entity in a database [4]) and makes it easier to pass between server and client. It also has higher level of parsing support than shapefile as it is a subset of JSON [5]. Fortunately, the previously mentioned mapshaper online tool had an option to export shapefiles in GeoJSON format. After exporting the simplified shapefile as a GeoJSON, the only modification left to perform on the data set was to eliminate all the data zones that do not belong to neither of the cities being examined in this project.

In order to do this, the GeoJSON had to be imported to the PostgreSQL database table ‘data_zones’ and make appropriate queries on the table containing the file. After importing a council zone to the data zone lookup table also from the government site, all data zone codes of Glasgow and Edinburgh

could be found and inserted into separate tables ‘glasgow_data_codes’ and ‘edinburgh_data_codes’. The query that lists and inserts only data zones boundaries in Glasgow to a table ‘glasgow_data_zones’ is the following:

INSERT into glasgow_data_zones SELECT "data" from data_zones INNER JOIN glasgow_data_codes ON data_zones like CONCAT ('%', glasgow_data_codes."dataZone2011", '%');

After running this query, the GeoJSON file in the database table was ready to use in the application to display data zones in Glasgow.

5.1.2 Travel times

The other main objective in this project was to determine under-provided areas of Glasgow with the help of collected open government data. When browsing for data sets on different sources the initial aim was to find a data set with the list of locations of different facilities (i.e. hospitals, GPs, schools, shops, libraries).

Many of these data sets were found, however these data sets stored the locations nonuniformly. Some of them stored locations only as postcodes, some of them had data zone codes too, but from the 2001 census as opposed to the 2011 one (which is used in the application). Furthermore, it would have been beneficial to have a data set which has data about all data zones in Scotland for potentially providing more value to the application. Also having one data set for each facility would have slowed down development significantly and would have made maintenance more inefficient.

The most ideal data set that has been found and eventually used by the application consists of travel times (in minutes) to nearest key services by car or public transport from every data zone in Scotland. An example entry of the data set can be seen below:

FeatureCode	*DateCode*	*Measurement*	*Units*	*Value*	*Destination*	*Method*
S01010241	*2015*	*Mean*	*Minutes*	*1.7*	*GP*	*Car*

This means that from the data zone S01010241 (Alexandra Parade – 03) to go the nearest GP the mean travel time by car is 1.7 minutes. Travel times are calculated by the government at multiple times at different times of the day. Main points of their methodology for calculating travel times are as follows [27]:

“

- Time windows were chosen to represent access throughout the day, in terms of two journeys from the origin to the service destination and two journeys from the service destination back to the origin (both peak and off peak).

- Drive times for schools were calculated for only two time windows, one for the morning peak journey, and another for the evening. This aims to better represent the regular 'school run' pattern.
- Public transport times were not calculated for schools because public transport trips to school tend to use bespoke transport for which there is no national data.
- As public transport availability varies by time of day and day of the week, multiple time windows and both outbound and inbound trips were considered. All calculations were performed for a Tuesday, as this was deemed to be the most representative weekday.
- Automatically substitute walk times where quicker than public transport times or where public transport is not available.

"

After importing this data set to the database table 'travel_times' a query had to be run it to receive the lines which include travel times only related to Glasgow.

5.1.3 Combining the two data sets

In order to efficiently use the boundaries and travel times of the data zones they needed to be united under one file. Travel time information of the data zones had to be added to the GeoJSON boundaries as an extra property so that the client would only have to request one file from the server and no further loading would be necessary when the user is switching domains. Also, it is beneficial to have the travel times under a JSON property as accessing it and modifying it made development faster and maintenance easier.

After exporting both main tables 'glasgow_data_zones' and 'glasgow_travel_times', writing a python script to read and write two files was decided upon. Please see script in Appendix B.

After running the script and importing the GeoJSON file back to the database all the data modifications have been done and development on the back-end server were started.

5.2 Implementation Languages

City Planner being a web application was discussed while gathering initial requirements. The main reason behind that was that city planners could easily gain access to the application without them having to install any kind of software as they can just run it on their browser. This section will discuss the choices regarding implementation technologies.

5.2.1 JavaScript

JavaScript is a scripting language and it is one of most popular choices when it comes to developing a front-end of a web application. When combined with HTML it can provide dynamic interactivity and flexibility to websites. Third party APIs, libraries and frameworks are widely available to JavaScript and provide additional functionality with minimum effort[6]. I took advantage of this feature of the language, by using many frameworks and tools during development of this project.

5.2.1.1 React

React is a JavaScript library and used for building user interfaces especially single-page applications. Web applications that were created using React change data without having to reload the whole page. This allow developers to create smooth user experiences without excessive loading screens. The main benefits of React are that it is fast, scalable and reusable. As React is being used to create user interfaces it corresponds to view in the MVC template [7]. It can be used with a combination of other JavaScript libraries or frameworks, such as Leaflet or Semantic UI React. An alternative framework was AngularJS, however, React is more suitable to use in a single-page application and focuses more on building intuitive UI components [8].

5.2.1.2 Semantic UI React

As mentioned in Chapter Semantic UI React is the official React integration for Semantic UI. This framework significantly made development progress faster and more efficient with a wide-range of pre-defined and modern UI components. The developer only has to load the components he utilises from the library and it offers a great range of customization too [9]. It was chosen over the alternative Bootstrap because it has offered a more suitable range of components such as icons.

5.2.1.3 Leaflet

Leaflet is an open-source JavaScript library to help developers create interactive maps for web- and mobile-applications. It is a lightweight tool as its total size is 38kB of JavaScript code. Leaflet works efficiently across all major desktop and mobile platforms. Main advantages of Leaflet are the speed which it loads up the map to the page and its range of map tiles the developer can choose from [10]. Layers can be added on the map tiles and it also has an option to add GeoJSON data to map as layer. The combination of being lightweight and ability to add GeoJSON to the map are main reasons why Leaflet was chosen to this project.

5.2.1.4 Turf.js

Turf is a JavaScript library for spatial analysis. It includes traditional spatial operations, helper functions for creating GeoJSON data, and data classification and statistics tools [11]. Turf can be added to any website as a client-side plugin or can be run Turf server-side with React.

5.2.1.5 Axios

Axios is a promise based HTTP that enables users to make GET and POST request from the browser. It is lightweight, easy to use and is compatible with many frameworks including React [12]. It also provides support for auto-conversion to JSON which is why the main reason this technology was chosen for this project. In this project it is used from make GET request from front-end to back-end to acquire GeoJSON data of the cities.

5.2.1.6 Mocha/Chai

Mocha is a JavaScript testing framework running on both front-end and in the browser, making asynchronous testing more efficient. Assertion libraries are often required in addition to tests [13]. Chai is an assertion library for front-end and the browser that is compatible with any JavaScript testing framework such as Mocha [14]. Both libraries support Behaviour Driven Development and Test Driven Development which is why they were chosen over alternatives like Jasmine [15].

5.2.1.7 Enzyme

Enzyme is a JavaScript Testing utility for React that makes it easier to test React Components. It also enables developers to manipulate, traverse, and in some ways simulate runtime given the output. Enzyme's API is meant to be intuitive and flexible by mimicking jQuery's API for DOM manipulation and traversal [16].

5.2.2 HTML

HTML is the basic building block of creating websites. It gives structure and meaning to content on the website. HTML generally used alongside with other technologies such as CSS or JavaScript.

5.2.3 CSS

CSS is a style sheet language which is responsible for the presentation and styling of a document written in a markup language like HTML. CSS is a main technology used in web application development alongside HTML and JavaScript.

5.2.4 Kotlin

Kotlin is an open source, statically-typed programming language which runs on JVM (Java Virtual Machine) and Android. It has both object-oriented and functional programming features. Kotlin is

ideal for developing server-side code as it allows developers to write concise and powerful code. [17] When comparing Kotlin to Java, it has the advantage of null-safety and better readability.

5.2.4.1 Exposed

Exposed is a lightweight SQL library used with Kotlin. It enables developers to connect to database and query tables in an efficient way.

5.2.4.2 HTTP4K

HTTP4K is a HTTP framework for Kotlin. It is written in pure Kotlin and allows serving and consuming HTTP services in a logical way [18]. It handles GET requests from client and responds back to front end with data extracted from database tables.

5.2.5 Python

Python is a very versatile programming language. It can be used for server-side programming, connecting to databases and performing complex mathematical calculations. In this project Python was chosen because of its file reading, altering and writing abilities. It efficiently added the travel times to the GeoJSON boundaries as extra properties [19].

5.2.6 SQL

SQL is a language for handling the data stored in a relational database management system. SQL's main functionality lays in extracting and manipulating the data, which is stored in a database, but it also allows users to create, structure and maintain data [20]. These functionalities were the reason behind choosing SQL to include in the implementation.

5.3 TravelTime Platrform API

The TravelTime API helps users find locations by journey time rather than using 'as the crow flies' distance. Time-based searching gives users more opportunities for personalisation and delivers a more relevant search. Returns routing information between source and destinations [21].

5.4 n:point

n:point.io is an online JSON storage which helps developers set up a lightweight JSON endpoint in seconds, while developing their application or website [22].

5.5 Back-end

The main role of the initial back-end in this application was to provide the front-end with GeoJSON files. This was achieved by connecting to the database and establishing server routes. Then the server-application made a query to the appropriate table and post it to the appropriate route as a

response. As mentioned, the initial back-end application was written in Kotlin with the use of Exposed and HTTP4K frameworks.

Development on back-end could be split into two part. Firstly, creating routes and establishing a server was done using HTTP4K framework. There were two routes created, one for each city's boundaries data. Then server was started locally on port 8000. Therefore, the routes could be accessed during development after starting the back-end application on localhost:8000.

```
fun main(args: Array<String>) {

    val app: HttpHandler = routes( ...list:
            "/glasgow" bind Method.GET to { Request :Request -> Response(OK).body(getData( s: "glasgow")).header( name: "Access-Control-Allow-Origin", value: "*")},
            "/edinburgh" bind Method.GET to { Request :Request -> Response(OK).body(getData( s: "edinburgh")).header( name: "Access-Control-Allow-Origin", value: "*")}
    )
    app.asServer(Jetty( port: 8000)).start()
}
```

Figure 14: Establishing server routes and starting application as a server on back-end

If the routes received a GET request, they were answering back with data collected from the getData() function.

In order to get data from the server, connection to the database and variables to reference the tables were needed to be set up. Connection was rather simple due to the framework being used.

```
fun getData(s: String): String {

    Database.connect( url: "jdbc:postgresql://localhost:5432/postgres", driver = "org.postgresql.Driver",
            user = "postgres", password = "**********");
```

Figure 15: Connecting to database from back-end

After connection was made the application had to query the database tables. This was done by use of the earlier mentioned getData() function.

```
    if (s.equals("glasgow")) {
        // 'select *' SQL: SELECT glasgow.data
        for (line in glasgow.selectAll()) {
            result = result.plus( other: "${line[glasgow.data]}")
        }
    }

    else{
        for (line in edinburgh.selectAll()) {
            result = result.plus( other: "${line[edinburgh.data]}")
        }

    }
}
return result;
```

Figure 16: Querying the database table and returning GeoJSON in getData function

The return value of this function is the GeoJSON boundaries of either city with travel times. This result got sent back to the appropriate route and it displays the information on the server in the following manner:

```
"features": [
  {
    "geometry": { … }, // 2 items
    "type": "Feature",
    "properties": {
        "StdAreaHa": 7.849029,
        "HHCnt2011": 242,
        "DataZone": "S01009758",
        "Shape_Area": 78490.3090084,
        "Name": "Darnley East - 01",
        "ResPop2011": 547,
        "StdAreaKm2": 0.078491,
        "CarTravelTimes": {
            "GP": "2.7",
            "SecondarySchool": "4.3",
            "PrimarySchool": "2.9",
            "PostOffice": "2.5",
            "ShoppingFacilities": "2.9",
            "PetrolStation": "1.8"
        },
        "TotPop2011": 598,
        "PublicTransportTravelTimes": {
            "PostOffice": "10.0",
            "ShoppingFacilities": "10.6",
            "GP": "11.0"
        },
        "Shape_Leng": 1474.15202765
    }
  },
```


Figure 17: Server-side application replying to GET Request by browser and information about data zones in GeoJSON format

However, back-end application is no longer used in the implementation due to deployment issues further discussed in Chapter 5.7.1.

5.6 Front-end

During development most efforts were focused on creating an intuitive and powerful front-end as this project is a map-based interactive application. To successfully achieve requirements, React library was used which significantly sped up development and made it easier to maintain codebase.

The main role of this part of the application to translate data gathered by the back-end to valuable information on a map and enable user interaction with displayed data zones.

5.6.1 Components

Components are reusable pieces of code and work independently. They serve the same purpose as JavaScript functions, but work in isolation and return HTML elements via a render function[23]. Since React's main focus is on creating user interface components each of these found in the implementation will be discussed with main emphasis on their role, how they communicate between each other and any key algorithms found in the codebase.

5.6.1.1 CityMap.js

This component is the root component in the project, this means that all the other components are nested inside this one. It also holds all the states, as almost all the other components are stateless. States are plain JavaScript objects and when a state changes it causes a re-render of the page. This

was a key concept to utilise during implementation. The most important states this component has are:

- isFetching (bool) – It indicates whether the application is getting data from the back-end. When it is true the 'Spinner' component is loaded in the place of the map showing the client that the page is loading. When it is false, map is shown to the user.
- geo (GeoJSON) – After the application has finished fetching data from server, geo will have the value of the server response. The boundaries of data zones of either city with the travel times.
- domain (String) – Indicates which facility is being examined at the moment by the user (i.e. GP, Primary School, Post Office…).
- method (String) – Indicates whether the travel times displayed on the map are by car or public transport.
- city (String) – Indicates which city is being looked at by the user.
- changesList(List) – A list of dictionaries which keep track of facilities added to the map.

It also has the functions to change each state:

- changeCity(city) – changes 'city', 'geo', and 'isFetching' state
- changeMethod(method) – changes 'method' state
- changeDomain(domain) – changes 'domain' state

The main role of this component is to render the map correctly according to the state the component is in. It gets the GeoJSON from the server-side application and renders all the other child components too. The algorithm which decides what colour each data zone will have is contained inside the getStlye() function. It loops through the data zones in the GeoJSON file and checks the travel times depending on the domain and travel method currently selected by the user.

The colour categories represent different range of travel times in minutes:

	Travelling with car:	Travelling with public transport:
- Dark blue:	0-1	0-3
- Light blue:	1-2	3-6
- Green:	2-3	6-9
- Yellow:	3-4	9-12
- Orange:	4-5	12-15
- Red:	5+	15+

Since a component can only modify its own state, in order to communicate with other child components, the parent component must pass down state and functions to be used by child components. In React, so-called props are used in achieving this. Props are similar to function arguments in JavaScript however they are passed between components. They are passed to each other via HTML attributes.

An example of passing down state and function as props to child components in the CityMap.js component:

```
<Header
    city = {this.state.city}
    method = {this.state.method}
    changeMethod = {this.changeMethod}
    changeCity = {this.changeCity}
/>
```

Figure 18: Passing down state and function as props to child Component

In this example 'city' and 'method' states are passed down alongside with changeMethod(method) and changeCity(city) method. In the child component Header, they can be accessed, and functions can be called with feeding the parent's as an argument. This way the parent's function will get called which is allowed to modify its own state. It was important to keep the main states in the root component as this component gets the GeoJSON and the getStlye() method which is responsible for colouring data zones need to have access to boundaries data, currently selected city, domain and method of travel at the same time.

Therefore, this component also acts a centralised state store. Also, CityMap.js does not call any of its own state changing methods directly. Those methods always get called from child components as props.

5.6.1.2 Header.js

One of the child components include Header.js, which is responsible for displaying the header of the page, which includes the title of the page with an icon, two group of Button elements from Semantic UI for navigating between cities and between methods of travel and also the import/export buttons.

Figure 19: Header component being displayed in the application

Therefore, the main two parent methods that are called from this component are changeDomain(domain) and changeCity(city). These methods are responsible for changing the states method and city in the CityMap.js component which will then cause a re-render and getStyle() function called which will change GeoJSON data depending on what city is selected and the colour of the data zones depending the selected method.

5.6.1.3 SideButtons.js

The main task of SideButton component is to render the buttons which are related to selecting domains. It allows users to switch between facilities and therefore view travel times to the nearest selected key service. See Figure 10 in Chapter 4.4 to see how the component looks like in the application.

If the selected method of travel is car the component will render five buttons, however when the method is public transport it only renders three buttons as the underlying data available is limited and it does not store information about travel times to schools by public transport. Each button has a onClick method assigned to it which is Map component's changeDomain(domain) function. Different buttons pass in different values to function depending on the facility.

5.6.1.4 AddButton.js

The main task of this component is to display a modal window after the user clicked the plus button located on the top right corner of the map and handle adding a selected facility by the user to a selected data zone. The main algorithm dealing with adding facility and calculating new travel time from data zones is also implemented here. The component has four states:

- modalOpen(Boolean) : This state indicates whether the modal window should be closed or opened
- addFacility(String): This state holds the type of the facility that user has selected to add.
- addDatazone(String): This state holds the name of the data zone the user selected to add the facility into.
- affectedDatazones(list): This state holds a list of recently inserted facilities to the map. Each list item is a dictionary which holds data about how the added facility affected travel times in the data zone it was added into and its neighbours.

Most important functions located in this component are:

- populateDropdown(GeoJSON): Populates the dropdowns located in the modal which helps users to select facility and data zone
- handleSubmit(GeoJSON): Once a user has decided on the facility and data zone to add to the map and clicks on the 'Done' button located in the modal, this method will be called. It calls the methods below sequentially.

- getNeighbours(): Returns the data code of all data zones that are adjacent to the data zone that has been selected by the user.
- getCoordinates(arrayOfNeighbours): Returns a list of coordinate points. Each list entry is the centroid coordinate points for each data zone that is in the given array as an input.
- calculateCarTravelTimes(sourceCentroid, destCentroids, GeoJSON): Calculates travel times between the sourceCentroid (centroid coordinates of selected data zone) and each entry in the destCentroids which are the centroid coordinate points of the neighbouring data zones.
- calclatePublicTransportTravelTimes(sourceCentroid, destCentroids, GeoJSON): Same functionality as the previous method but calculates travel times by using public transport rather than driving.
- checkChanges(newTravelTimes, GeoJSON): It creates a dictionary changesDictionary with the following keys: Datazone, Facility, NeighbouringDataZones where the first two key values are equal to addDataZone and addFacility state. NeighbouringDataZones' values hold data about the new travel times of the neighbouring data zones.

The modal window - a dialog box/popup window that is displayed on top of the current page is appeared after the user has clicked the plus button and has two dropdown bars one for selecting a facility and one for selecting a data zone to put the facility in. To populate the data zone dropdown with all the options populateDropdown() is invoked.

Figure 20: Modal allowing users to select facility to add to their chosen data zone

It loops through the GeoJSON boundaries and creates an array of dictionaries with the name of the data zones as a key for each entry. Each dictionary has also a text and value property. Text is what is being displayed in the dropdown and it also has the code of the data zone shown next to name. The value property is used in setting the state addDatazone which is helping to establish the later created affectedDatazone list state.

After clicking on the ‘Done’ button, handleSubmit() method is called which calls a series of functions which firstly calculates the centroid coordinate points of the selected data zone using Turf.js. Furthermore, it calculates the neighbouring data zones of the selected data zone by calling the getNeighbours() function. It loops through the GeoJSON boundary data and if it finds a common coordinate point between a data zone and the selected data zone it adds it to an array. This result array gets input into the function which calculates the coordinate points of the centroid of each neighbouring data zone. Then the application calculates travel times by using calculateCarTravelTimes() and calculatePublicTransportTravelTimes(). Both functions take the centroid of the selected data zone, the array of centroids of the neighbouring data zones and the GeoJSON data as parameters. Then it calls TravelTimes Platform API with the coordinates of the source centroid and one if its neighbouring data zones centroid coordinates at a time (Figure 19.). After the travel times have been calculated checkChanges() function is called, which checks if the travel time calculated by any of the previous functions are better than the one present in the current GeoJSON data. If travel time is improved, then it overwrites the current travel time. After this function is finished, the updated GeoJSON data is passed back to the parent component, CityMap.js alongside with affectedDatazones list to keep track of changes made to the GeoJSON data by adding facilities.

```
let dataPost1 = {
    "locations": [
        {
            "id": "Facility",
            "coords": {
                "lat": sourcepoint.geometry.coordinates[1],
                "lng": sourcepoint.geometry.coordinates[0]
            }
        },
        {
            "id": "DataZone centroid",
            "coords": {
                "lat": destpoints[key].geometry.coordinates[1],
                "lng": destpoints[key].geometry.coordinates[0]
            }
        }
    ],
    "departure_searches": [
        {
            "id": "departure search example",
            "departure_location_id": "Facility",
            "arrival_location_ids": [
                "DataZone centroid"
            ],
            "transportation": {
                "type": "public_transport"
            },
            "departure_time": "2020-03-20T10:00:00Z",
            "properties": ["travel_time", "distance", "route"]
        }
    ]
};
```

Figure 21: An example request to TravelTime Platform for calculating travel time.

The main challenge encountered when implementing the feature to add a facility to a data zone was waiting for API responses. Making sure that all responses are received by the application before continuing with the rest of the program was essential in order to accurately update the map and display new travel times calculated by the API. To overcome this issue, the concept of promises has been utilised. Promises are suitable to deal with asynchronous operations in JavaScript. They are easy to manage when handling multiple asynchronous operations where callbacks can often lead to unmanageable code [24]. Both functions that include API calls are using promises in order to wait for the requests to finish. Once a promise has been resolved (in this case an API response received), only then the program continues. In the screenshot below you can see utilising promises in two places.

Firstly, when an individual response has been received, the travel time is extracted from the response and placed in the dictionary. Secondly, the program will only continue to the function checkChanges() when all API calls made by this function have received the response.

```
    promises.push(
        axios.post( url: 'https://api.traveltimeapp.com/v4/routes', dataPost1, axiosConfig)
            .then((res) => {
                let minute = (res.data.results[0].locations[0].properties[0].travel_time) / 60;

                let time = Math.round( x: minute * 10) / 10;
                times[key]["PT"] = time;

            })
            .catch((err) => {
                console.log("AXIOS ERROR: ", err.response);
            })
    );
}

Promise.all(promises).then(() => this.checkChanges(times, geo));
```

Figure 22: JavaScript Promises being utilised in CalculatePublicTransportTravelTimes() function.

5.6.1.5 Scale.js

Main purpose of this component is to aid users determining travel times from data zones by providing a coloured scale where each colour is associated with a range of travel times in minutes. This component is stateless and does not have any methods.

Figure 23 and 24: Scale component when travel times are displayed using car and when using public transport.

The component is using a variable called timeDiff which is set for 1 if the selected method of travel is car and is set for 3 if the selected method of travel is public transport. When setting the actual text in the coloured columns, timeDiff and its multiples are used rather than concrete numbers, therefore it makes for better maintainability.

5.6.1.6 ChangeList.js

The main task of this component is to render out a panel on the right hand side of the map and dynamically display which facilities have been added to which data zone and how this change affected neighbouring data zones. This component is stateless and has one method which is responsible for populating the panel with data about travel time changes. ChangeList component

gets the Map component's changesList state as a prop and loops through it. It creates a string of multiple lines for each added facility to show users how it affected neighbouring data zones. Then the string will be passed to the Accordion Semantic UI React component as a prop, therefore displaying it.

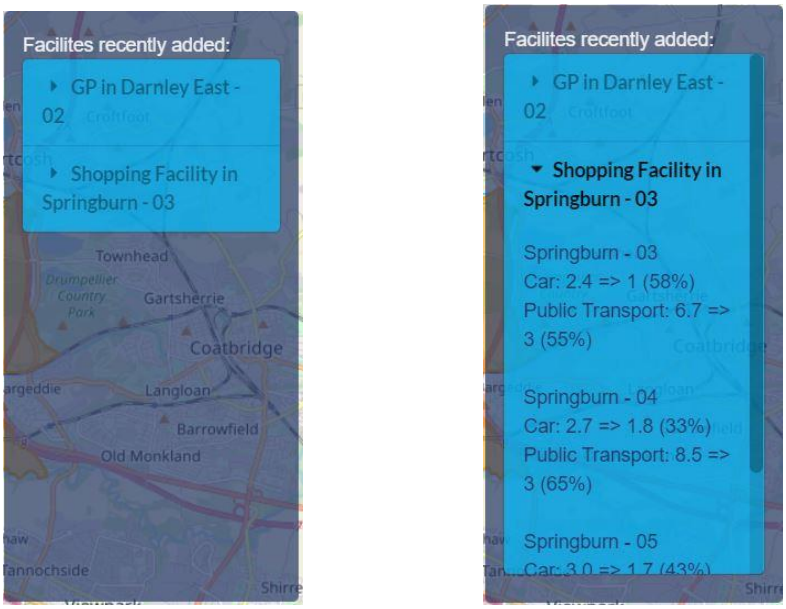


Figure 25 and 26: ChangeList component shown after two facilities have been added, and effects of second facility being examined after opening collapsible panel

5.6.1.7 Import.js

This component is responsible for handling importing and exporting GeoJSON files to/from the application. When a user decides that after adding some facilities to data zones, the underlying GeoJSON data needs saving, export feature of the application allows users to download that data. The program uses URL Web APIs createObjectURL() function from which a link is being created and on click this link is triggered. As a result, the download of the JSON file through browser is starting.

If the user at a later stage decides that previously downloaded GeoJSON file needs to be loaded in the application, import feature of the application allows users to do so. Import function is using FileReader class. Once the FileReader has finished reading the file it passes back the data to the parent component CityMap.js which sets its geo state to the recently read in file.

5.7 Deployment

In order to enable multiple users to concurrently access the application on the internet both back-end and front-end application had to be deployed on a suitable platform.

5.7.1 Back-end

Initially the back-end application was deployed on Heroku. Heroku allows developers to build, run and operate applications as a platform as a service on the cloud. The main building blocks in Heroku are called dynos, which are isolated, virtualised Linux containers [25]. However, after deployment, the build was facing a memory quota exceeded error. This error is related to the dyno's memory which at a free tier instance is 512 MB. Memory space was exceeded when the application was requesting and receiving data from the database. As, there was no instance available with bigger memory for free, other options were sought.

Amazon Web Services was the second option, however after configuring system environment variables to AWS credentials, the application was not able to access the database outside the AWS cloud environment. An RDS (Relational Database Service) instance set up on AWS would have been required, however looking at the scope of the project, a more lightweight solution was implemented.

As previously mentioned in Chapter 4, the two GeoJSON files were uploaded to npoint.io, to an online JSON bin storage, making the application work consistently without a back-end server.

5.7.2 Front-end

The front-end application was deployed to GitHub Pages. GitHub Pages takes a GitHub users HTML, CSS and JavaScript file from their repository and runs the files through a build process and publishes a website [26]. Therefore, GitHub Pages enables continuous integration as at every build run it looks at the latest version of the repository. Deployment to GitHub Pages was fairly straightforward, it included installing an npm package called gh-pages, adding two lines in the packages.json file and deployment was done running one command, npm deploy. The city planner application is available at: https://magylenny.github.io/CTYPLNR/.

6 Validation and Verification

This chapter will discuss what testing strategies have been applied to ensure the project satisfied project specifications mentioned in Chapter 3. Testing also played an essential part in identifying and fixing bugs and errors that appeared during development.

6.1 Testing Overview

Testing was vital in the development of this project. After the completion of each feature, test cases were developed and performed making sure that the recently built feature is robust and other features can be safely integrated into the project. This concept is one of the main principles of Agile development methodology. After finishing the development, test files were created to allow testing the code independently from the rest of the application. The file which includes all 32 tests are in the test folder of the project and can be run with running the 'npm test' command in the terminal. See the result of each test in Appendix B.

6.2 Test Cases

All of the test cases were built after examining the functional requirements in Chapter 3. Each of the test cases found below was created after the completion of associated functionality. Detailed test case descriptions can be found in Appendix B.

Name of test case	Details	Test case type
Render components	All components except map should be rendered on the page.	Unit test
Load the map with boundaries of data zones	Map of the selected city should be rendered with boundaries of data zones	Unit test
Change domain	After changing domain, the appropriate travel times should be displayed.	Integration test
Change method of travel	After changing method of travel, the appropriate travel times should be displayed.	Integration test
Change city	After changing, the appropriate map and travel times should be displayed.	Integration test
Add a facility	After adding a facility, travel times should be updated for the affected data zones.	Integration test
Export JSON file	After exporting underlying GeoJSON data, it should be downloaded to the user's device.	Integration test

Import JSON file	After importing previously exported GeoJSON data, map should display appropriate travel times.	Integration test

6.3 Unit testing

Several Unit Tests were developed to help testing code and functionalities in isolation ensuring the application is without bugs and errors. The application was tested using Enzyme, Mocha and Chai frameworks. To allow complete isolation for unit testing Enzyme's shallow method was utilised. Shallow rendering is useful to isolate a component as a unit, and to ensure that the tests aren't indirectly asserting on behaviour of child. Shallow() not only allows to test functions written in the class but also to test if the child components are being rendered too without their contents.

```
const wrapper = shallow(<CityMap/>);
it('should change CityMap method', () => {

    expect(wrapper.instance().state.method).equal( actual: "Car");
    wrapper.instance().changeMethod( method: "PublicTransport");
    expect(wrapper.instance().state.method).equal( actual: "PublicTransport");

});
```

Figure 27: Unit testing CityMap.js component's changeMethod() function

```
it('should render SideButtons component', () => {
    expect(wrapper.find(SideButtons)).to.have.length(1);
});
```

Figure 28: Testing CityMap.js component's render() function

6.4 Integration Testing

Integration tests decides if units that have developed separately work appropriately when they are connected to each other. For developing integration tests Enzyme's mount() method was used. As opposed to shallow(), mount() can be used for testing the component alongside with its child components. This kind of testing allows to test the behaviour of a parent component based on the child component's action. For example, it enables to test the situation when a child component's button is clicked and changes the parent's component state (see Figure 29).

```
    const wrapper = mount(<CityMap/>);
    let headerWrapper = shallow(<Header changeMethod = {wrapper.instance().changeMethod}
                                        changeDomain = {wrapper.instance().changeDomain}/>);
it('should change parent`s method state', () => {
    expect(wrapper.state().method).equal( actual: "Car");
    headerWrapper.find(Button).at(1).simulate('click');
    expect(wrapper.state().method).equal( actual: "PublicTransport");
});
```

Figure 29: Integration testing with CityMap.js component and Header.js component

7 Results and Evaluation

This chapter will firstly discuss the final product made by the end of the project and how the original project specifications were compared to the final application. The second part of this chapter will detail the valuable input and suggested improvements offered by the user evaluation.

7.1 Final Application

The final design and implementation of the project from a user-end view is shown below. Functionalities of the application are highlighted in the picture and detailed in the next paragraph.

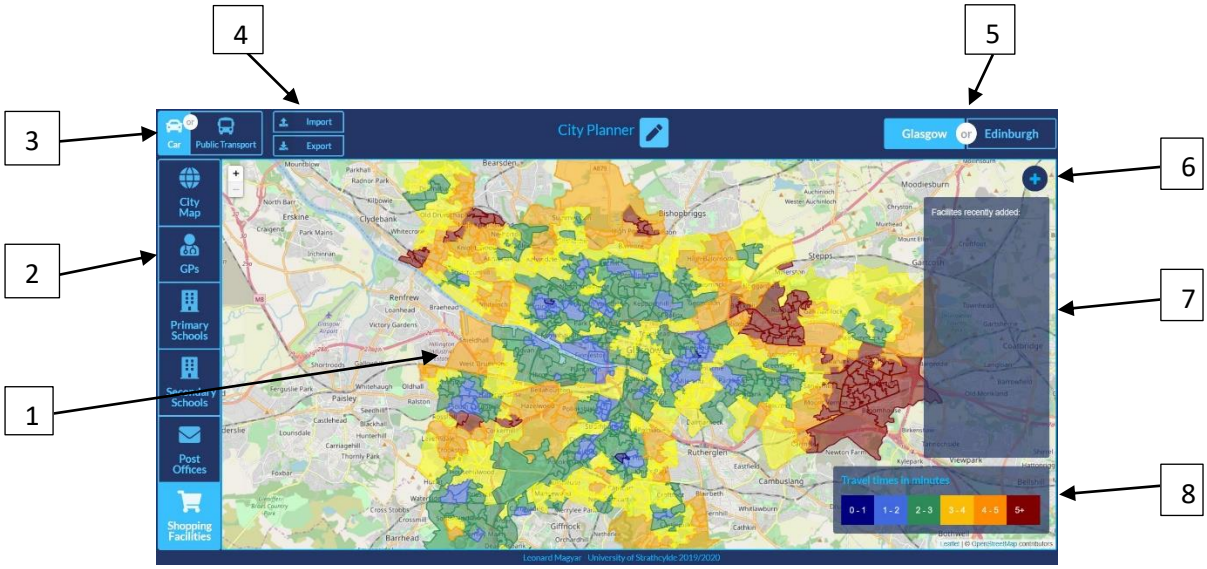


Figure 30: Screenshot of the final application

1. Map of currently selected city with its data zones is displayed and coloured according to travel times to the nearest currently selected type of facility.

2. Navigation panel to select a facility. Colour of each data zone will adjust to reflect travel times to selected facility.

3. Navigation panel to select method of transport. Colour of each data zone will adjust to reflect travel times with selected method.

4. Import button to load GeoJSON data to the application. Mainly recommended to load data which was previously saved from the application. Export button to save underlying GeoJSON data from the application is also located here.

5. Navigation panel to select city. Map will show map of city and its coloured data zones according to selected city.

6. Button to add facility, it will invoke a panel where selecting facility to place and data zone is available.

7. Panel where facilities under one session are displayed. It also displays other data zones that are affected by the change and how their travel times changed.

8. Legend to indicate travel time to colour encoding. It changes with the selected method of travel to enable relative comparison between data zones.

7.2 Valuable Outcome

The main result of this application is that certain areas in both cities can be identified by users where there are many adjacent red data zones. This means that potentially these areas are underdeveloped and have a shortage of a certain facility. These areas could be examined by city planners and council workers to ensure lack of a certain facility does not influence residents negatively in any aspect. They could also simulate how adding a facility in a certain data zone could affect the neighbouring data zones, therefore optimising travel times in the underdeveloped area. Example of potentially underdeveloped area and how travel times would be affected in the area by utilising the application is shown below:

The potentially overlooked area is marked by red; the area of Summerston in Glasgow when examining travel times to nearest the GP. This might indicate that there is not a GP in the whole are of Summerston.

Figure 31 (left) and 32(right): Map indicating travel times before and after placing a GP in 'Summerston Central and West - 02'

The details how travel times have been affected in neighbouring data zones after adding a GP in 'Summerston Central and West– 02' can be seen below and also examined in the application.

Name of affected data zone	Change in travel times (minutes)	Change in travel times (%)
Summerston Central and West - 02	Car: 5.7 => 1 Public Transport: 15.6 => 3	Car: -82% Public Transport: -81%

Summerston Central and West - 03	Car: 5.5 => 1.8 Public Transport: 18.4 => 3.9	Car: -67% Public Transport: -79%
Summerston Central and West - 04	Car: 6.3 => 1.6 Public Transport: 20.1 => 2.6	Car: -75% Public Transport: -87%
Summerston North - 03	Car: 6.2 => 2.8 Public Transport: 14.9 => 2.8	Car: -55% Public Transport: -81%
Summerston North - 04	Car: 5.4 => 1.2 Public Transport: 12.4 => 1.9	Car: -78% Public Transport: -85%

In some data zones, travel times to the nearest GP has reduced by nearly 90% making a close to twenty minute journey to as short as a couple of minutes.

7.3 User Evaluation

To aid the evaluation of the project user feedback was required. User evaluation consisted of three parts. Firstly, an introductory form where volunteers were explained the main aim of the application. Secondly, a task sheet where users were asked to perform a series of tasks (i.e. identifying a data zone from where travelling to the nearest GP would take more than 5 minutes, then adding a GP to this data zone) using the application and examine the outcome of these tasks. Finally, users were asked to fill out a survey, which asked about the usability of the application and what features could be improved or implemented.

A link to the application was included in the evaluation form so volunteers could easily access the website. The user evaluation form was circulated through friends and fellow students. In total, 12 responses were provided and only one of them have heard about data zones before. Age of users were spread between 21 and 33. 58% of the people were male and 42% female.

7.3.1 Negatives

The main aspect of the application that users rated negatively was how affected data zones and their travel times were displayed in the 'Facilities recently added:' panel after adding a facility. Also, the panel would sometimes overlap with the 'Travel time in minutes' panel if there were too many affected data zones. Two responders also would have preferred to add a facility by clicking on the selected data zone. Two users also mentioned that increasing the size of the map would be beneficial to the application.

To address the first issue, lines in the panel have been separated more to aid easier readability and adding travel times changes in percentage were implemented. Maximum height of the panel was also set, to avoid overlapping with the other panel and enable scrolling.

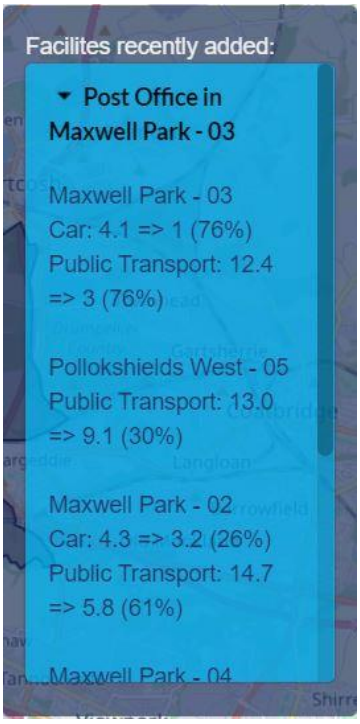


Figure 33 and Figure 34: Change in displaying affected data zones and their travel times after user evaluation.

7.3.2 Positives

When users were asked what they liked the most about the application most responders highlighted the design of the interface and ease of usability. Users mentioned that the use of colours made it easy to understand what data set they are exactly looking at and to navigate between domains. Some responders also mentioned how responsive and quick was the application when switching between domains, method of travel or cities.

11 responders either agreed or strongly agreed that using this application would help city planners and land developers decide where to establish new facilities in the city. One of the users stayed neutral to this statement. 75% of the responders strongly agreed that they found navigating between cities, travelling methods and facilities straightforward while the remaining 25% of them agreed. These numbers are the same when they were asked if they found identifying the travel times for a data zone straightforward. This confirms that all responders were satisfied with identifying travel times of data zones and navigating between options.

7.3.3 Suggested improvements

When asked about features that could be added to the application two users suggested addition of other facilities such as banks, cinemas, parks. Other suggestions include collapsible side panels so that size of map would increase, a search field to find a data zone through typing, examining the actual values of travel time by clicking or hovering over a certain data zone. Some of these features could be implemented into the application either on a short-term or on a long-term and will be further discussed in the next chapter.

8 Summary and Conclusions

This chapter will discuss the success of the project in the aspect of achieving the previously stated aims and objectives. It will also detail any potential future improvements.

8.1 Summary

The final product of this project allows users to examine travel times from data zones to the nearest selected key facility, deduce areas on the map which in certain aspects potentially underdeveloped and place facilities where the user thinks it would have the maximum impact on hundreds of residents' lives who stay in that area. Users can see the visualisation of an existing government data set being translated it into valuable information.

Furthermore, the project heavily relied on the open government data sets that could be found on the internet, therefore the number of examinable facilities were limited, however the application still provides enough data to potentially solve real-world problems.

In summary, the project was successful in providing valuable meaning to an open government data set and as user evaluation showed the application is intuitive and easy to use, however improvements still could be made to aid city planners job choosing where facilities should be exactly placed in a certain area to maximise impact.

8.2 Future Work

This section will discuss potential future improvements for the application. These will include uncompleted tasks from the project specifications, suggestions from user evaluation and any opportunities noticed during development.

8.2.1 Short-term development ideas

- Collapsible navigation panels: Buttons for selecting domains, cities, and method of travel could be collapsed into a narrow bar which will extend on mouse hover, therefore map size would increase after the user has changed domain.

- Collapsible display of affected data zones: In the application's current state, after placing a facility, the change is added to panel on the right-hand side as a collapsible item. However, the children of this item (each affected data zone) is not collapsible and can take up a lot of space with each of their new travel time being displayed too.

- Displaying changes after importing a GeoJSON file: After importing a suitable GeoJSON file to the application, it does not display the changes made to the data compared to the original data in

the ‘Facilities recently added’ panel. For example, if a user places in a GP in Anderston – 05, and decides to export the file, and import it back later on, the recent change will not be displayed in the panel, however colour marking and travel time will be correct.

- Adding facility by clicking on data zone: Adding a facility to a data zone could be made more straightforward by just clicking on the on the selected data zone, and a facility of the currently selected type would be added to the data zone. Currently, if the user clicks on a data zone, a popup appears. This feature could be displayed upon mouse hover.

8.2.2 Long-term development ideas

- Data harvesting: Adding more kind of facilities would significantly increase the depth of the application and could be potentially done by a potential collaboration with the government or by harvesting other data sets that include the locations of a certain key facility and calculating the travel time separately.

- Automate adding more cities: Adding more cities to the application can be done fairly straightforward as both used data sets include data about the whole of Scotland, however uniting the two data sets (boundaries and travel times) requires some operations in the database and running a python script (see Appendix A). This could be overcome by building a back-end application which would perform those database operations and run the python file, therefore travel times from all data zones of Scotland could be examined.

- Algorithm calculating new travel time: As a free-tier account in TravelTime Platform API only allows 10 API requests per minute and 3000 a month, the number of requests made in the algorithm were limited. It only calculates the travel time at one time period (outbound Tuesday 10am) as opposed to the calculations’ of the government where travel times are measured multiple times at different times of the day.[27]

8.3 Conclusion

Throughout this project a single-page web application ‘City Planner’ had been implemented, developed and evaluated. This system allows users to examine travel times from data zones to the nearest selected key facility in Glasgow and Edinburgh. It also helps users to identify potentially underdeveloped areas and inspect how travel times could be reduced in several data zones by adding a facility to the desired area. It could also be extended in the future to show data for all data zones in Scotland, providing more depth.

9 References

[1] UK Government (2018). *Scottish Index of Multiple Deprivation (SIMD) 2016*. Location :https://data.gov.uk/dataset/a448dd2a-9197-4ea0-8357-c2c9b3c29591/scottish-index-of-multiple-deprivation-simd-2016

[2] NHS (2019). *Lower Layer Super Output Area.* Location: https://www.datadictionary.nhs.uk/data_dictionary/nhs_business_definitions/l/lower_layer_super_output_area_de.asp?shownav=1

[3] Department of Medicine University of Wisconsin (2018) . *Neighborhood Atlas.* Location*:* https://www.neighborhoodatlas.medicine.wisc.edu/

[4] TechTerms. Blob. Location: https://techterms.com/definition/blob

[5] Juho Hame (2019). Shapefile vs. GeoJSON vs. GeoPack Location: https://feed.terramonitor.com/shapefile-vs-geopackage-vs-geojson/

[6] W3Schools.com. JavaScript Tutorial. Location: https://www.w3schools.com/js/

[7] React. Homepage. Location: https://reactjs.org/

[8] Tech Magic(2019). React vs Angular vs Vue.js. Location: https://medium.com/@TechMagic/reactjs-vs-angular5-vs-vue-js-what-to-choose-in-2018-b91e028fa91d

[9] GitHub – Levi Thomason. Semantic-UI-React. Location: https://github.com/Semantic-Org/Semantic-UI-React

[10] GitHub. Leaflet. Location: https://github.com/Leaflet/Leaflet

[11] GitHub. Turf.js. Location: https://github.com/Turfjs/turf

[12] GitHub. Axios. Location: https://github.com/axios/axios

[13] Mocha. Homepage. Location: https://mochajs.org/

[14] Chai.js Homepage. Location: https://www.chaijs.com/

[15] CodeMentor Team (2016). JavaScript Testing Framework Comparison: Jasmine vs Mocha. Location: https://www.codementor.io/@codementorteam/javascript-testing-framework-comparison-jasmine-vs-mocha-8s9k27za3

[16] GitHub. Enzyme. Location: https://enzymejs.github.io/enzyme/

[17] Martin Heller (2017). What is Kotlin? The Java alternative explained. Location: https://www.infoworld.com/article/3224868/what-is-kotlin-the-java-alternative-explained.html

[18] GitHub. HTTP4K. Location: https://github.com/http4k/http4k

[19] GitHub. Python. Location: https://github.com/python/cpython

[20] SQLCourse.com What is SQL? Location: http://www.sqlcourse.com/intro.html

[21] TravelTime Platform. TravelTime Search API. Location: https://docs.traveltimeplatform.com/overview/introduction

[22] n:point. Homepage. Location: https://www.npoint.io/

[23] W3Schools.com. React Components. Location: https://www.w3schools.com/react/react_components.asp

[24]GeeksForGeeks. JavaScript Promises. Location: https://www.geeksforgeeks.org/javascript-promises/

[25] Heroku. Heroku Dynos. Location: https://www.heroku.com/dynos

[26] GitHub. About GitHub Pages. Location: https://help.github.com/en/github/working-with-github-pages/about-github-pages

[27] Kate Trafford, GIS Analyst. Scottish Government (2012). Scottish Index of Multiple Deprivation 2012. Geographic Access Domain Methodology Location: https://www2.gov.scot/Topics/Statistics/SIMD/AccessMethodologyPaper

10 Appendix A - Python Script

Below is the code of the python program that helped combining the two data sets into one unique JSON file.

```
import json
Public = {}
Car = {}
dictionary = {}

##setup [Car, Public] for every datazone
##loop through all lines and datazone and assign list
f = open("C:/Users/magyl/Desktop/UNI Document/4th year/edinburghtraveltimes.txt", "r")
for line in f:
  (zone, rest, rest1, rest2) = line.split()
  dictionary[zone] = [Public.copy(), Car.copy()]
f.close()

f = open("C:/Users/magyl/Desktop/UNI Document/4th year/edinburghtraveltimes.txt", "r")
for line in f:
  (zone, method,destination, minute) = line.split()
  #print zone, method, destination, minute
  if method == "Car":
    dictionary[zone][0][destination] = minute
  else:
    dictionary[zone][1][destination] = minute

#opening JSON

with open('C:/Users/magyl/Desktop/UNI Document/4th year/edinburghgeojson.txt') as json_file:
    data = json.load(json_file)

#looping through JSON and looking up dictionary values
for feature in data['features']:

        feature['properties']['CarTravelTimes'] = dictionary[feature['properties']['DataZone']][0]
        feature['properties']['PublicTransportTravelTimes'] = dictionary[feature['properties']['DataZone']][1]

with open('data.txt', 'w') as outfile:
    json.dump(data, outfile)
f.close()

print ("Success!")
```

11 Appendix B – Testing results

11.1 Detailed Test Cases

Render components

Description: All components except map should be rendered on the page.

Preconditions:

- The website has completed the initial load

Steps:

1. Navigate to https://magylenny.github.io/CTYPLNR/ in the browser

Expected Result: All components are rendered on the site, instead of map, spinner component is present while loading getting data from JSON storage.

Actual Result: As expected

Load the map with boundaries of data zones

Description: Map of the selected city should be rendered with boundaries of data zones

Preconditions:

- The application received the GeoJSON data from JSON storage

Steps:

1. View screen

Expected Result: Map of Glasgow is rendered with data zones displayed on it

Actual Result: As expected

Change domain

Description: After changing domain, the appropriate travel times should be displayed.

Preconditions:

- The website has completed the initial load
- Application has received data from JSON storage

Steps:

1. Select any kind of facility by clicking on the associated button on the left-hand side.

Expected Result: Data zones will be coloured based on their travel times to the selected facility.

Actual Result: As expected

Change method of travel

Description: After changing method of travel, the appropriate travel times should be displayed.

Preconditions:

- The website has completed the initial load
- Application has received data from JSON storage

Steps:

1. Select method of travel by clicking on the associated button on the top left.

Expected Result: Data zones will be coloured based on their travel times to the selected facility and selected method of travel.

Actual Result: As expected

Change city

Description: After changing, the map of the selected city and appropriate travel times should be displayed.

Preconditions:

- The website has completed the initial load
- Application has received data from JSON storage

Steps:

1. Select either Glasgow or Edinburgh by clicking on the associated button on the top right.

Expected Result: Spinner component should replace map while the application is getting data from JSON storage, then display data zones coloured based on the currently selected domain and method of travel.

Actual Result: As expected

Add a facility

Description: After adding a facility, travel times should be updated for the affected data zones.

Preconditions:

- The website has completed the initial load
- Application has received data from JSON storage

Steps:

1. Click on the ‘plus’ button on the top right corner of the map.
2. Select data zone and facility to add.
3. Click on ‘Done’ button.

Expected Result:

1. Modal window should appear with two dropdown panels.
2. Selected data zone’s name and type of facility should be set in dropdown panel.
3. New travel times are calculated and colouring changed if appropriate for the selected data zone and its neighbours.

Actual Result: As expected

Export JSON file

Description: After exporting underlying GeoJSON data, it should be downloaded to the user’s device.

Preconditions:

- The website has completed the initial load
- Application has received data from JSON storage

Steps:

1. Click on ‘Export’ button located next to buttons associated with methods of travel.

Expected Result: Download of the underlying JSON data will be started.

Actual Result: As expected

Import JSON file

Description: After importing previously exported GeoJSON data, map should display appropriate travel times.

Preconditions:

- The website has completed the initial load
- Application has received data from JSON storage
- Previously exported file from application

Steps:

1. Click on ‘Import’ button located next to the button associated with methods of travel.
2. Select previously downloaded JSON file.

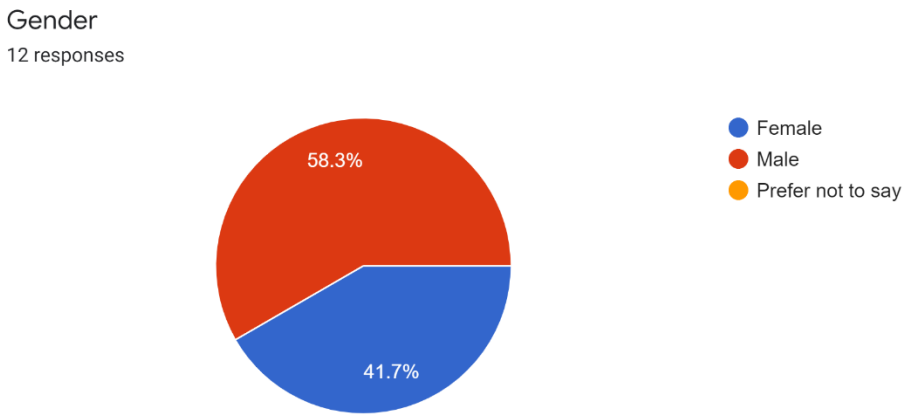
Expected Result: Travel times from data zones and will be based on the recently imported file.

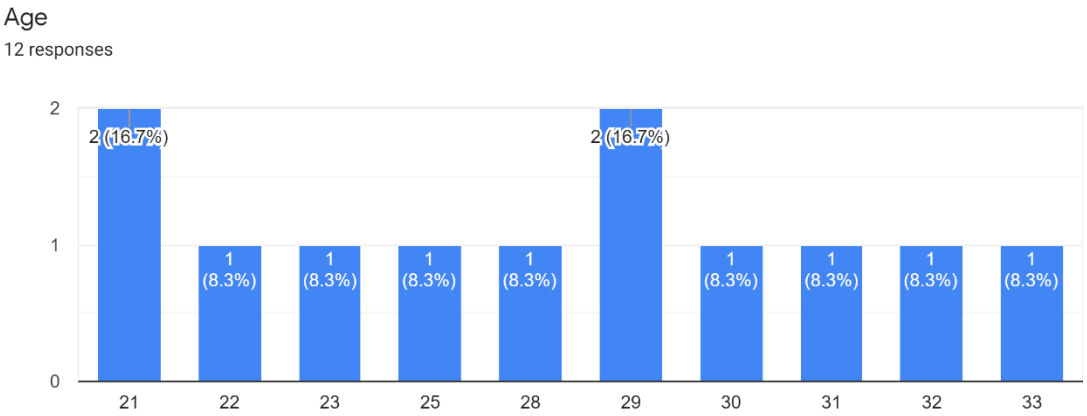
Actual Result: As expected

11.2 Testing Results

```
CityMap Unit/Integration(shallow)
  √ should fetch data (glasgow by default) (198ms)
  √ should change CityMap`s method of travel state by component method to Public Transport
  √ should change CityMap`s domain state by component method
  √ should change CityMap`s city state to Edinburgh by component method  (17003ms)
  √ should render Header`s component (42ms)
  √ should render SideButtons` component
  √ should render ChangeList component
  √ should render Scale component
  √ should render ChangeList component
  √ should render AddButton component
  √ should render GeoJSON component
  √ should change SideButtons` method prop
  √ should change Header`s method prop
  √ should change SideButtons` activeDomain prop by component method

Header.js Unit/Integration
  √ should render Grid component
  √ should render Grid.Column component
  √ should render Button component
  √ should render Import component
  √ should render Title component
  √ should change parent`s method state by clicking on `Car`
  √ should change parent`s city state back to Glasgow by clicking on `Glasgow` (17012ms)
  √ should change data by clicking on `Edinburgh` (17275ms)

Import.js Unit
  √ should render Button component (41ms)
  √ should render hidden input component

Title.js Unit
  √ should render Grid component
  √ should render Grid.Column component
  √ should render Icon component

SideButtons.js Unit
  √ should render Button components - Car
  √ should change parent`s method state
  √ should render Button components - Public Transport

ChangeList.js Unit
  √ should render Control component
  √ should render Accordion component


32 passing (2m)
```

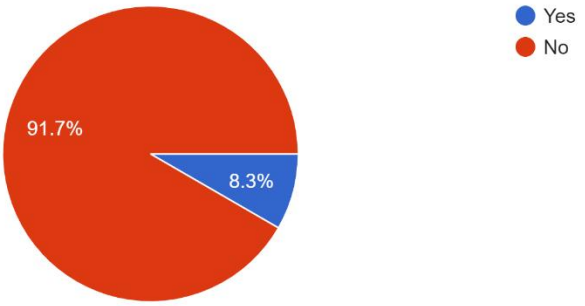
12 Appendix C – User Evaluation Results





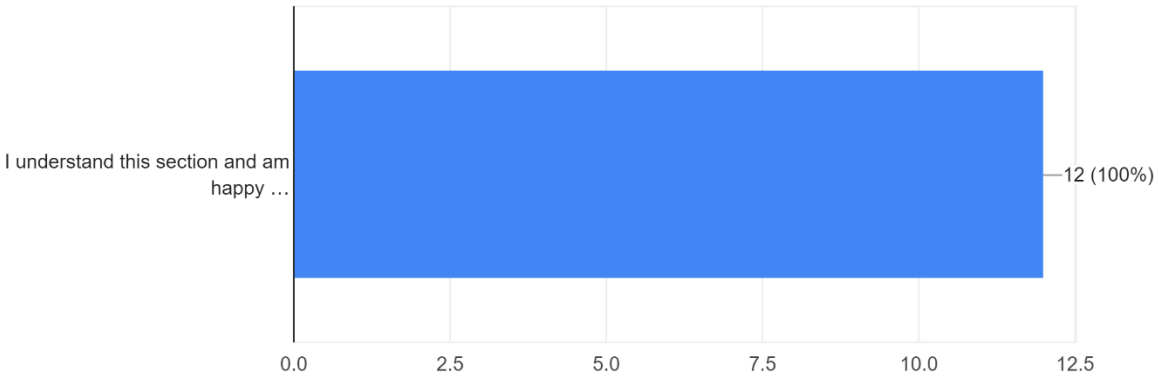
Have you heard about data zones before?

12 responses



Would you like to participate and continue with the study?

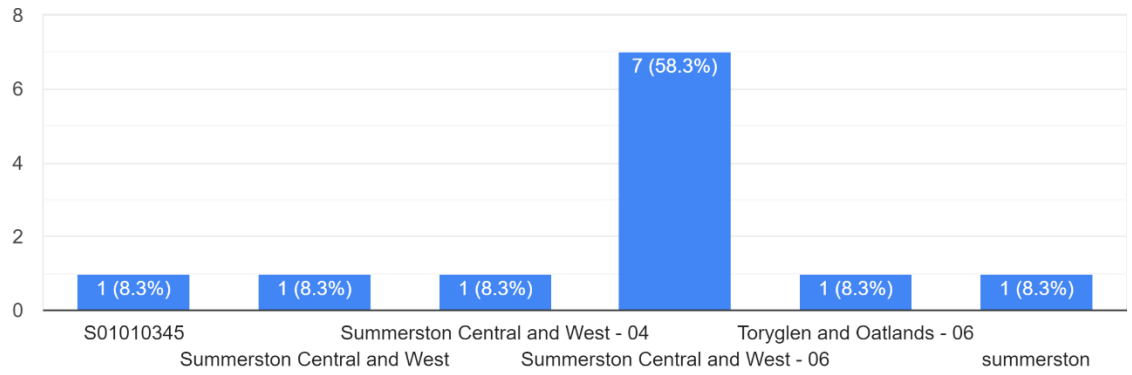
12 responses



CityPlanner Application Study: Task Sheet

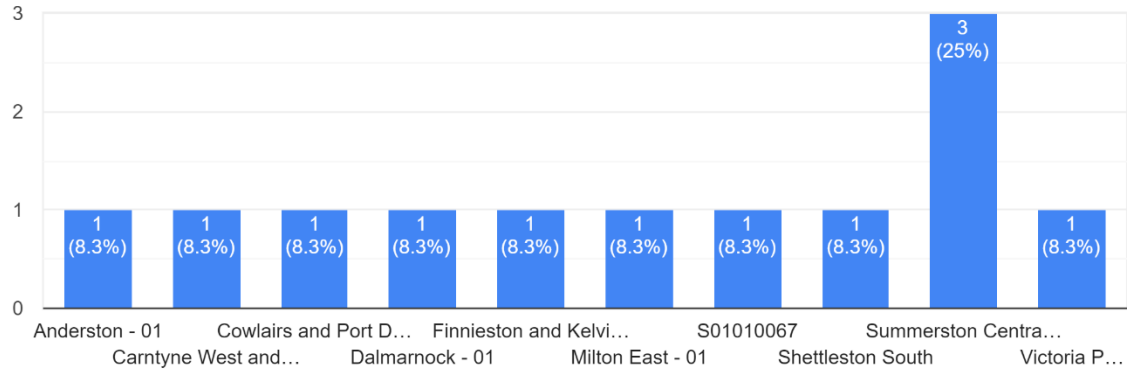
1. Identify a data zone in Glasgow where reaching the nearest GP by car would take more than five minutes.

12 responses



2. Identify a data zone in Glasgow where reaching the nearest primary school by car would take between two to three minutes.

12 responses



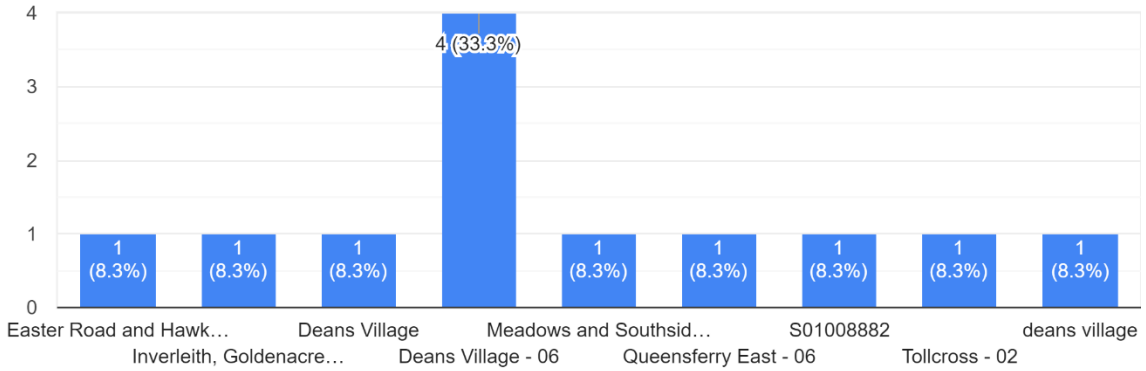
3. Identify a data zone in Edinburgh where reaching the nearest post office by public transport would take between nine and twelve minutes. 12 responses

Ratho, Ingliston and Gogar - 01
Trinity - 02
Bonaly and The Pentlands
Ingliston and Gogar - 01
Comely Bank - 01
Ratho, Ingliston and Gogar - 01
Currie West - 01
S01008991
Inverleith, Goldenacre and Warriston - 01
Western Harbour and Leith Docks - 01

costorphine north
Dalmeny, Kirkliston and Newbridge - 06

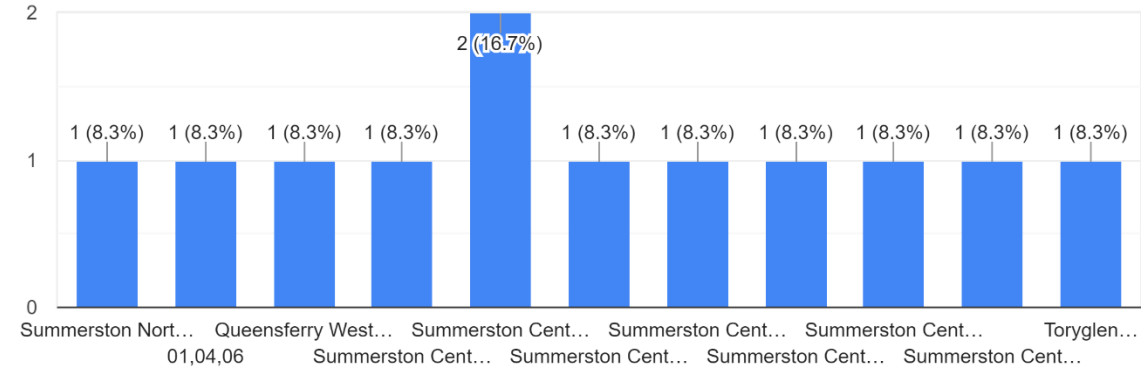
4. Identify a data zone in Edinburgh where reaching the nearest shopping facility by public transport would take between three and six minutes.

12 responses



5. Add a GP to the previously identified data zone in Task 1. List data zones effected by the change.

12 responses



6. Freely select a data zone in Glasgow where you think placing a certain facility would have the maximum effect on travel times in the area. Please state chosen data zone and selected facility.12 responses

GP in Summerston Central and West - 06
Yoker South - 05 Shopping facilities
Possil Park - 06, Secondary School
Secondary School in City Centre South - 04
Laurieston and Tradeston - 05 , Secondary School
Cowlairs and Port Dundas - 02 secondary school
Shopping Facility in Baillieston East - 01

S01010345 adding GP
Summerston Central and West - 04 GP facility
Summerston Central and West - 06, GPs
primary school - city centre south
Secondary School in Possil Park - 06

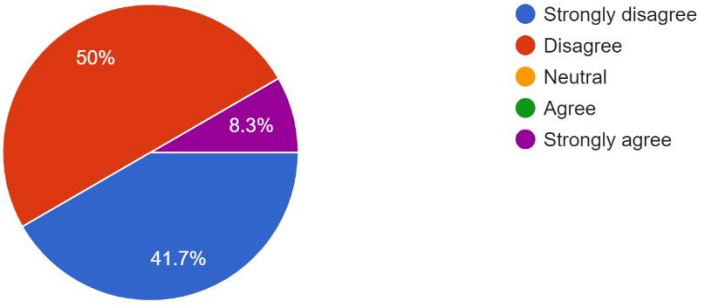
7. Freely select a data zone in Edinburgh where you think placing a certain facility would have the maximum effect on travel times in the area. Please state chosen data zone and selected facility. 12 responses

GP in Balerno and Bonnington Village - 02
South Gyle - 06 Post Office
Liberton West and Braid Hills - 03, Secondary School
Secondary School in Comiston and Swanston - 06
Corstorphine - 04, Secondary School
Meadowbank and Abbeyhill North - 02 secondary school
Shopping Facility in Colinton and Kingsknowe - 02
S01008997 adding GP
South Gyle - 06 GP facility
Bonaly and The Pentlands - 01, Shopping Facilities
shopping facility in bonaly
Corstorphine North - 03, Secondary school

CityPlanner Application Study: Post task form

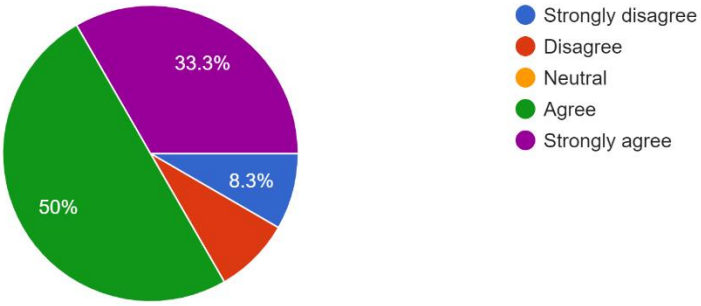
1. I found the system unnecessarily complex

12 responses



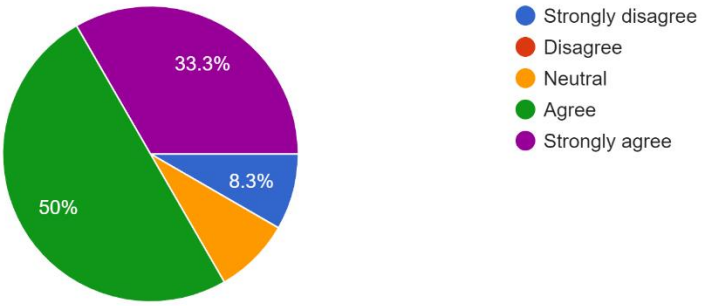
2. I thought the system was easy to use

12 responses



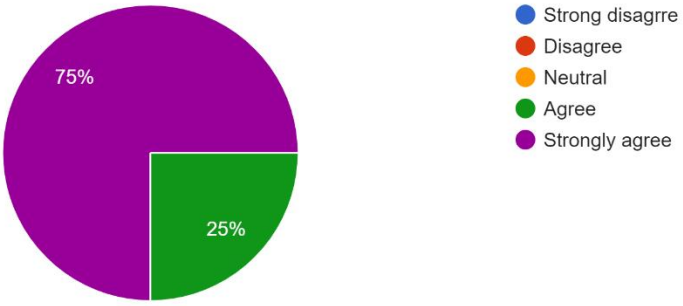
3. I found the various functions in this system were well integrated

12 responses



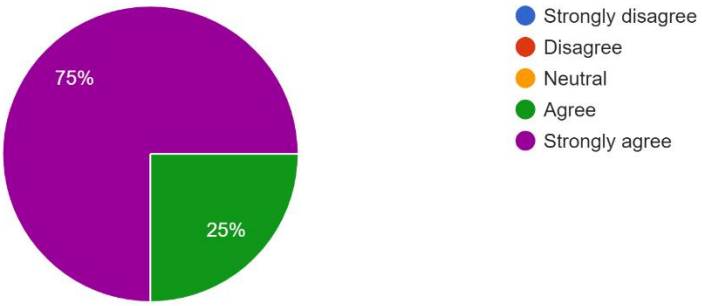
4. I found navigating between cities, travelling methods and facilities straightforward

12 responses



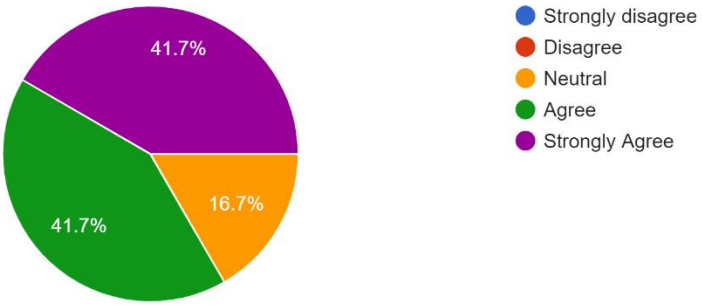
5. I found identifying the travel times for a data zone straightforward.

12 responses



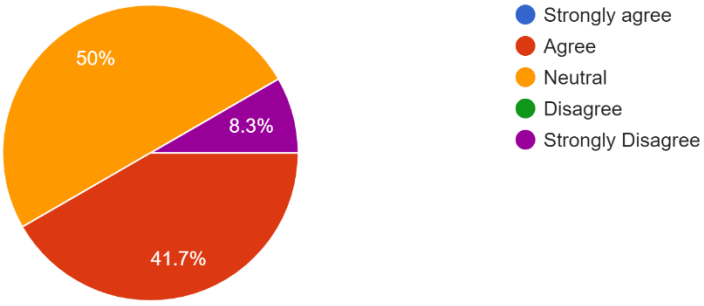
6. I found adding facilities to a data zone straightforward

12 responses



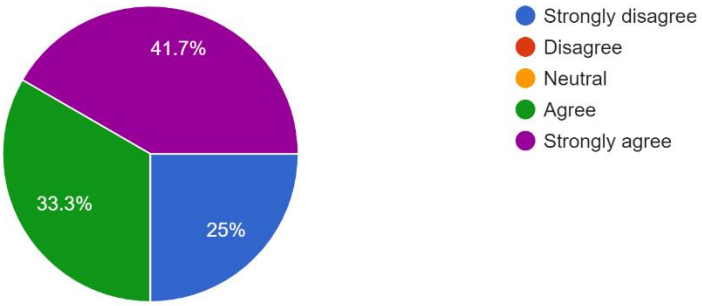
7. I found examining the effects to travel times after adding a facility straightforward

12 responses



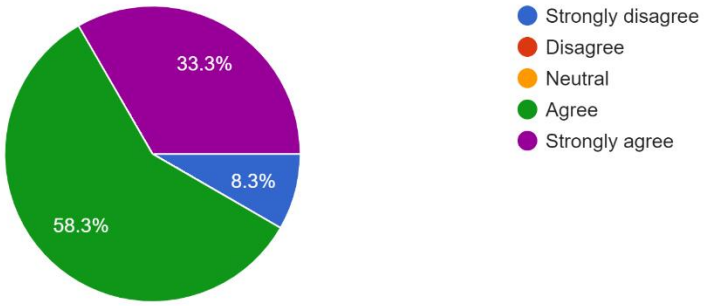
8. I would imagine that most people would learn to use this system very quickly

12 responses



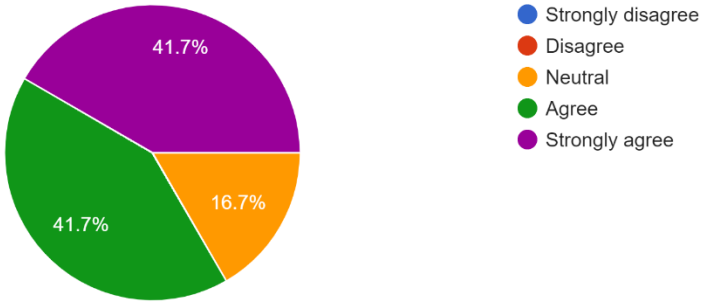
9. I felt very confident using the system

12 responses



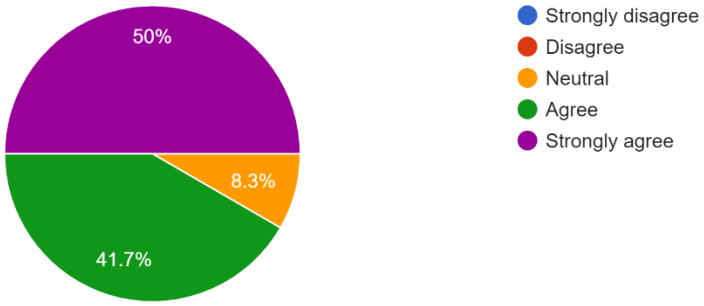
10. I think that I would like to use this on appropriate occasions (i.e. when choosing a new area to live in)

12 responses



11. I think that using this application would help city planners and land developers decide where to establish new facilities in the city

12 responses



12. Which features could be improved? 11 responses

The view that shows how data zones are affected by the addition new facilities could be improved slightly to aid readability.

Possibilities to add more than 2 facilities. When trying to add more, previous options got deleted

Quite hard to read the changes in travel times when adding a facility, add new line or use colour to distinguish between them, maybe highlight the numbers that are changing so they stand out. Also map slightly cuts off zones on the edge, maybe allow zoom out a little more if possible. I know you can move the map anyway so not a big problem

I could not read the list of added facilities properly because of the scale for travel time. Maybe it is just my browser(used Chrome)

overlapping of two panels

It should be able to add facilities to an area by clicking on that area. Also, the colours on the map do not correspond entirely to the colours in the legend, which is a bit confusing.

Adding facility by clicking a location on the map, displaying impact on nearby areas on the map

Facilities recently added could be presented a bit more clearly

When adding a facility to a zone, the data zones could be ordered alphabetically instead of data code.

area fragmentation and adding facility as it doesn't improve travelling time from nearby area when adding facilities

I am not colourblind but some testing should be done with colourblindpeople (some cannot distinguish between green and red)

3. What feature you would like to add to the application? 12 responses

A drag-and-drop feature to add new facilities.

More explanation on how to use it

When you select a zone, give a list of all of the times for each facility for that zone

visualization of affected areas by adding new facilities, zones where travel time was affected could light up on the map or have some kind of indication that they have changed (as the

color might not change if the difference is minor)
Removing the facilities from the list on the right-hand side.

include more type of facilities , collapsible side panels

It would be nice to see the actual values of travel time by clicking or resting with the pointer on a certain area for a short period of time

Adding facility by clicking a location on the map, displaying impact on nearby areas on the map

Add more facilities such as cinemas, bank branches, parks

1. A search field to find an data zone through typing. 2. To be able to drag travel times in minutes box and/or the facilities recently added box because the two can overlap if the addition is not collapsed.

travelling time from areas to city centres

When you add a facility, it would be nice to highlight the areas affected by the change with a new colour.

14. What did you like the most about the application? 11 responses

Easy of use and clean user interface.

Graphic and looks easy to change options

very simple to understand, great design, great use of colours, dynamic loading screen

The design is very nice.

easy to use application

The application is intuitive and useful. However, some improvements are required (see above) to make it more user-friendly. Colour coding is good and helpful.

It was visually pleasant and easy to use

The colours made it easy to understand. It's a fun and useful app

Very responsive and quick. Clear and concise despite the complexity of the underlying data.

how intuitive it is

Clear and straightforward

5. What did you like the least about the application? 11 responses

nothing

None

I find it very difficult to understand how different zones get affected by changes

That i could not read the table on the right-hand side

map size could be improved

Some areas are too large to give a realistic forecast of travel time when placing a new facility.

Adding facilities feature

There was no data for primary schools and public transports

Font size of the frame. However, the font size of the description pop-up box of a data zone otherwise is fine.

n/a

16. Any other comments? 6 responses

great web app

It is not clear, from where are these travel times measured? from the middle of each data zone or the city centre...or?

well done

The concept is neat. Would be nice to have the possibility to zoom into individual areas of the map and see travel time zone's at a lower scale, to allow for even more accurate planning of new facilities.

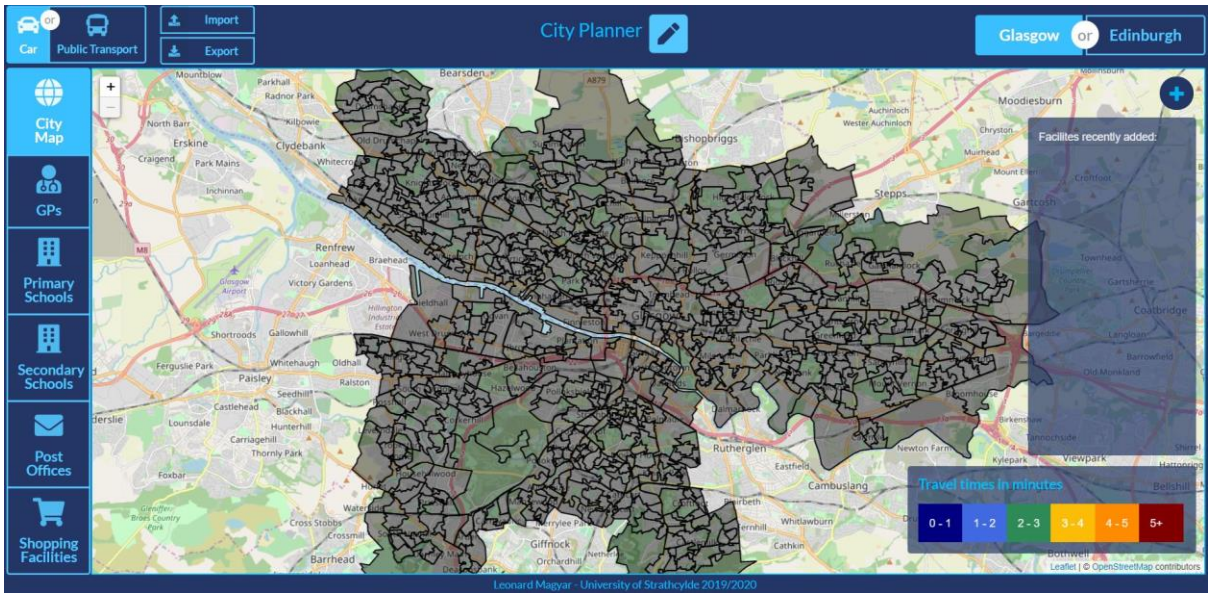
The data zones could be in alphabetical order when adding.

n/a

13 Appendix D – User Guide

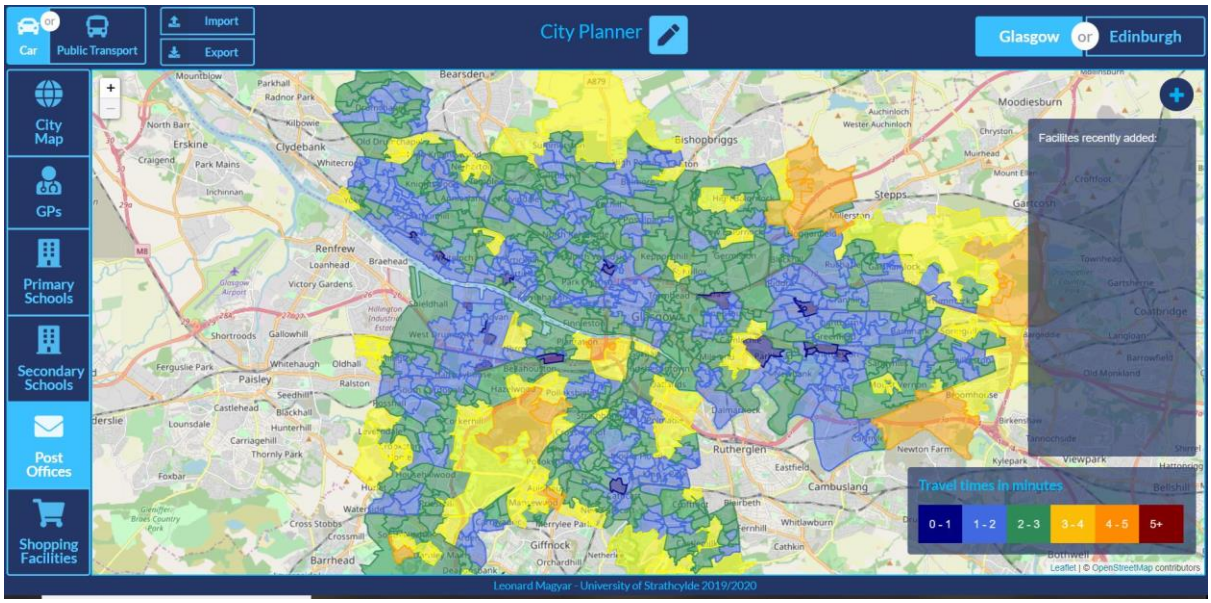
13.1 Viewing the boundaries of data zones

Upon opening the application, the boundaries of data zones in Glasgow will be presented. This view does not represent any travel times, it is just to get the user familiar with the concept of data zones.



13.2 Changing Domain

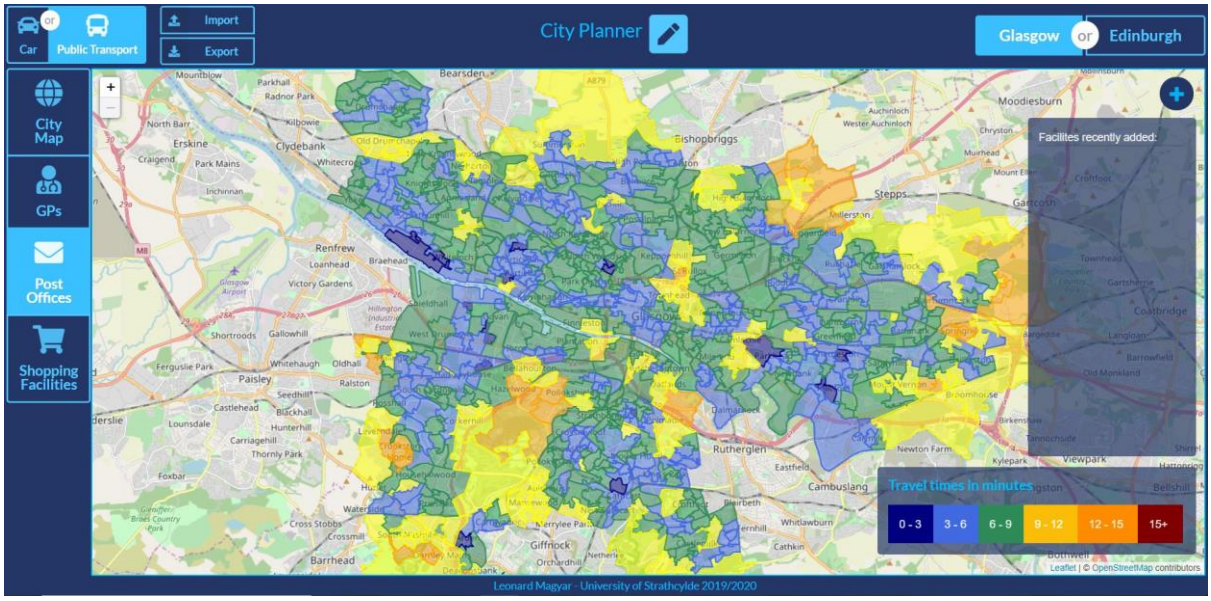
To view more valuable information about travel times from data zones to nearest key facilities, a type of facility needs to be selected. This can be done by clicking on one of the buttons on the panel on the left-hand side.



Travel times can be identified by looking at the panel 'Travel times in minutes' on the right bottom corner of the map.

13.3 Changing Method of Travel

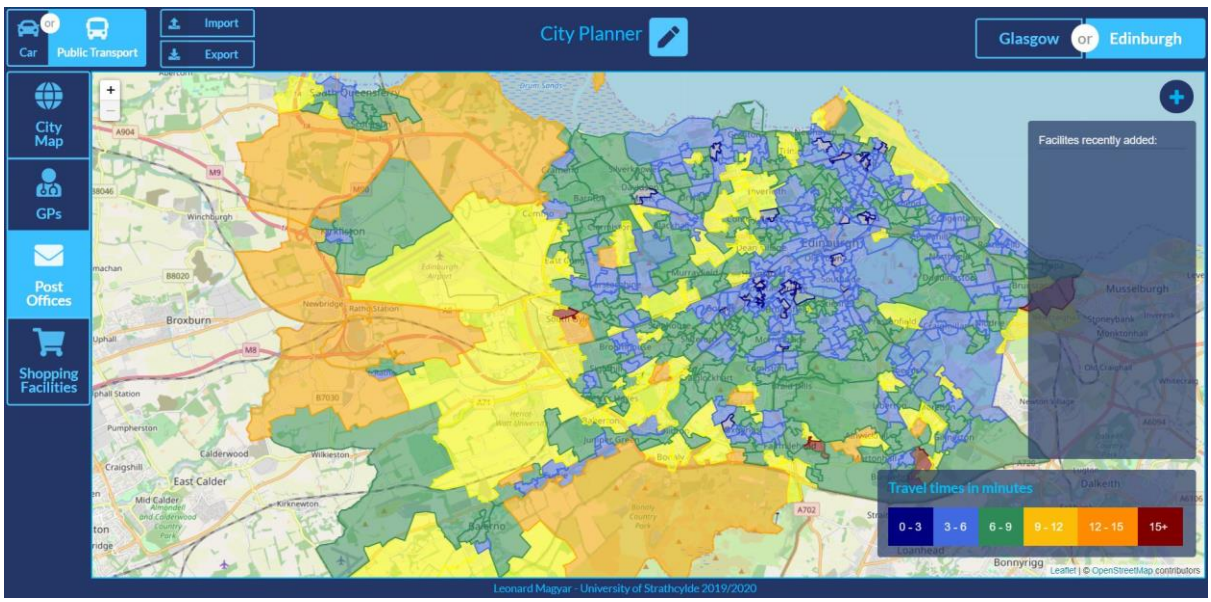
By default, travel times displayed are representing journeys made by car. To change this, 'Public Transport' button can be clicked on the top left corner on the site.



Travel times will be updated to represent journeys made by public transport or walking. The 'Travel times in minutes' panel will be also update and will change the associated numbers for colours.

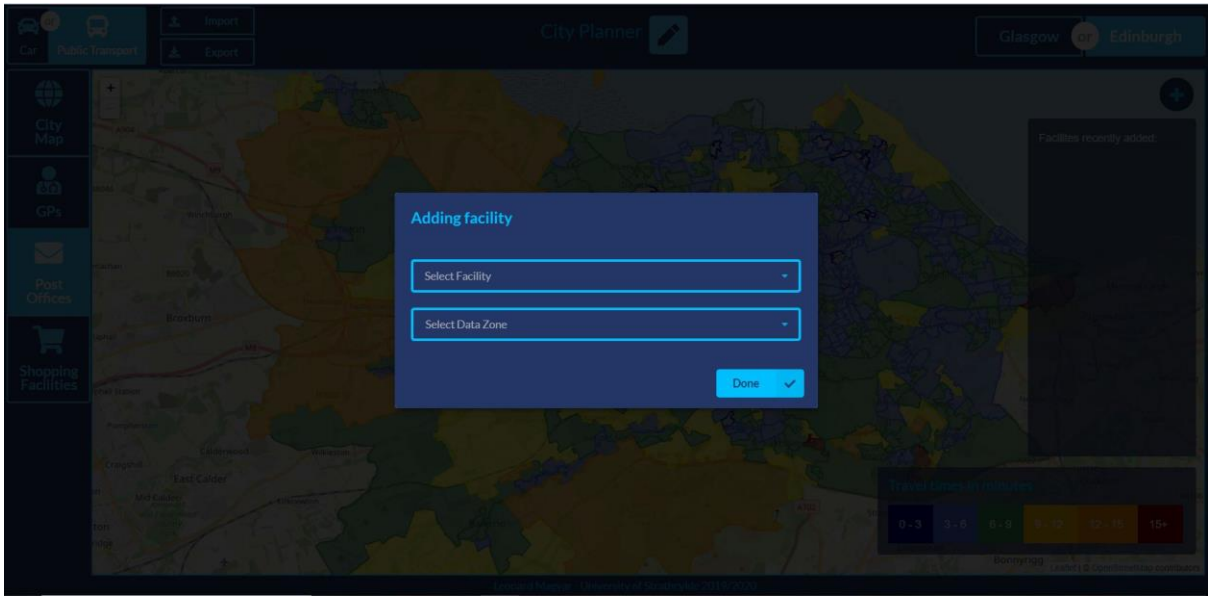
13.4 Changing City

To change between Glasgow or Edinburgh, the buttons are located on the top right corner of the page.



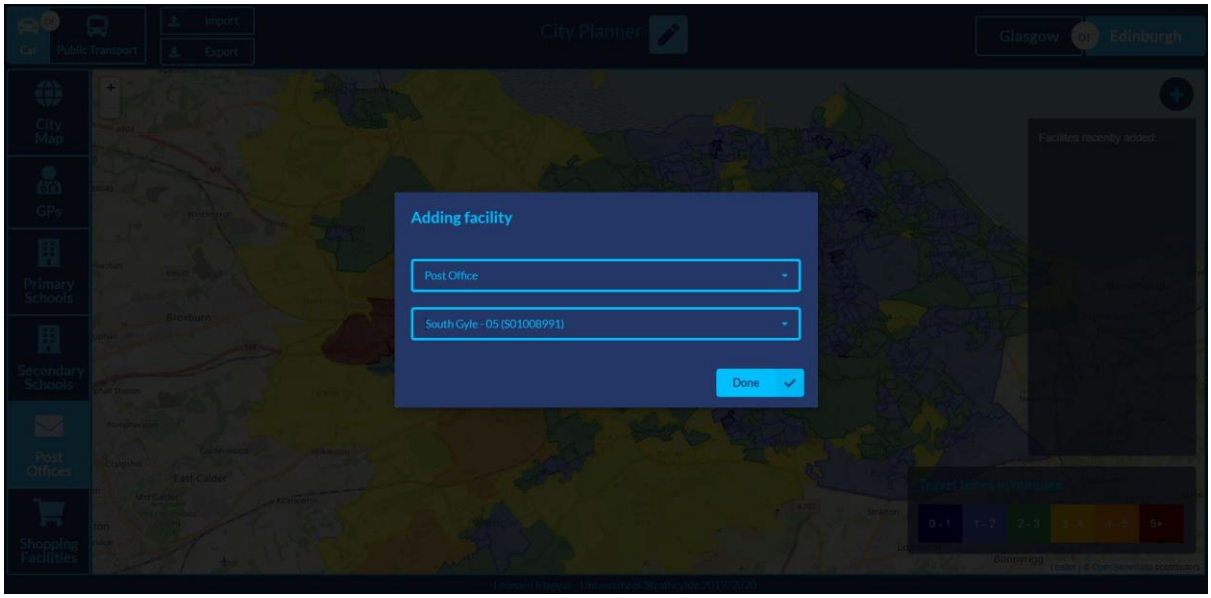
13.5 Adding a Facility

To add a facility, first click on the plus button located on the top right corner of the map, after this the following window will be presented.

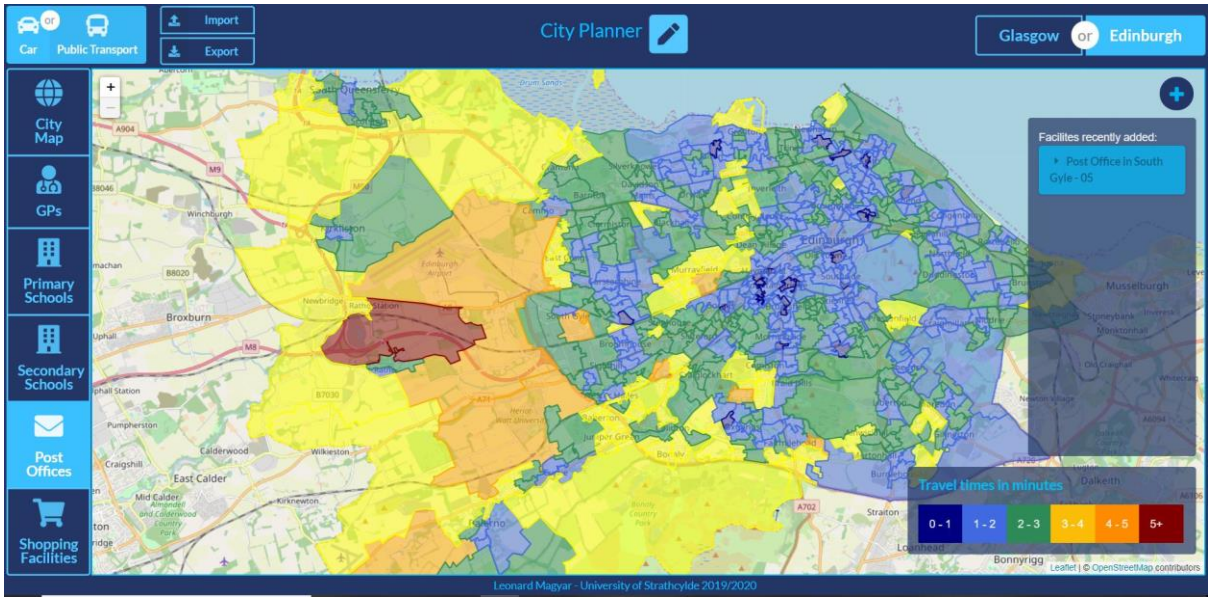


In this window select the kind of facility you would like and the data zone you would like to add it to.

Name of the data zone can be typed in the search box, and matching data zones will be presented which are clickable.

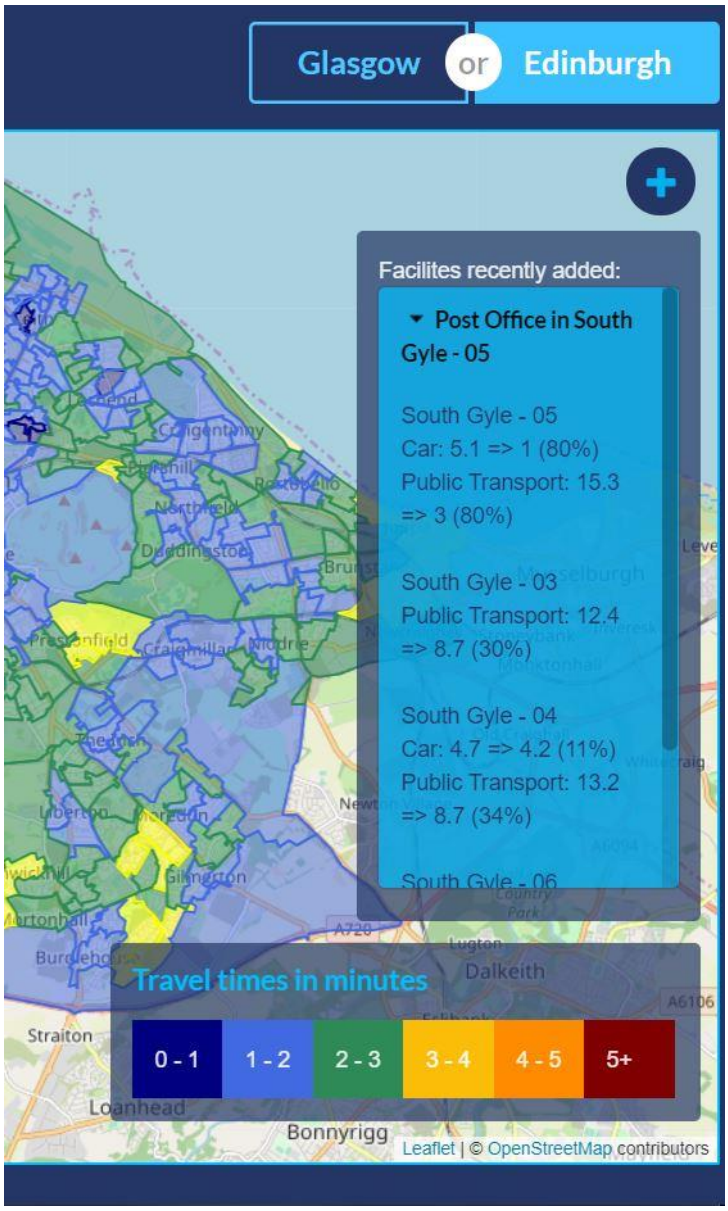


After clicking the ‘Done’ button, travel times will be calculated and the map will be updated.



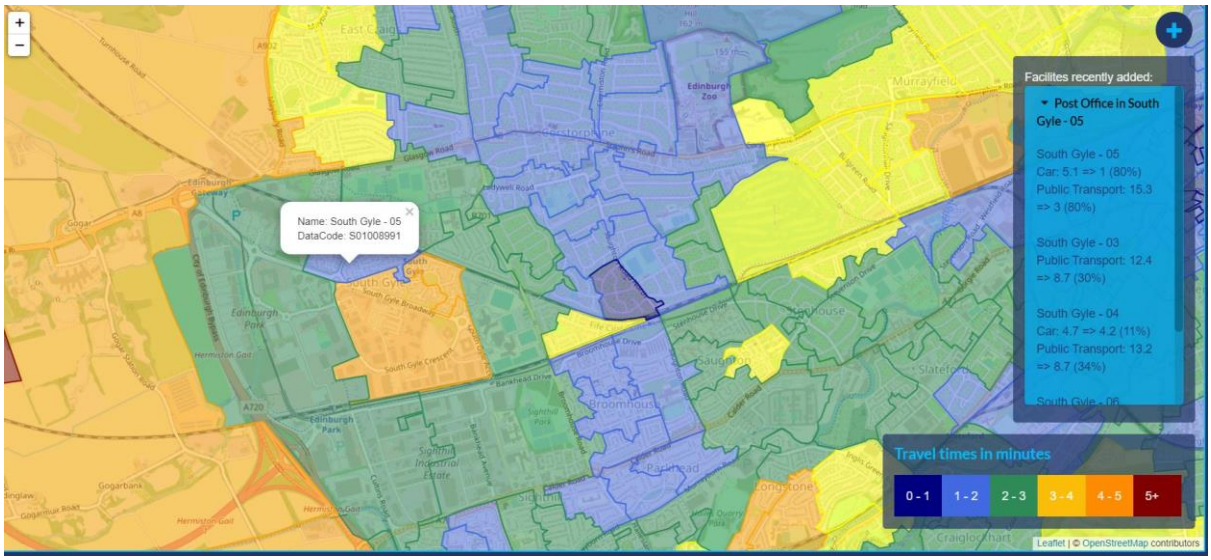
13.6 View New Travel Times

As seen on the previous screenshot, the 'Facilities recently added' panel displays the change. By clicking on the newly added light blue panel, changes to travel times can be examined.



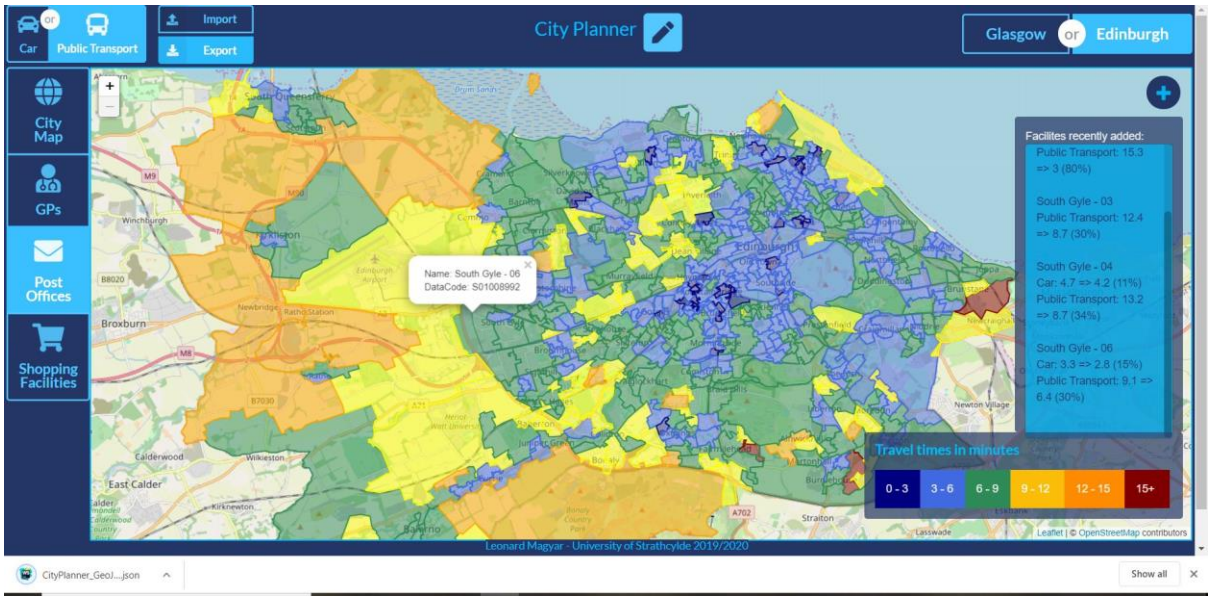
To check these travel times are represented on the map as well examine the area of the data zone where the facility was added to. By clicking on a data zone a pop up appears with the name of the

data zone and data code.



13.7 Export JSON Data

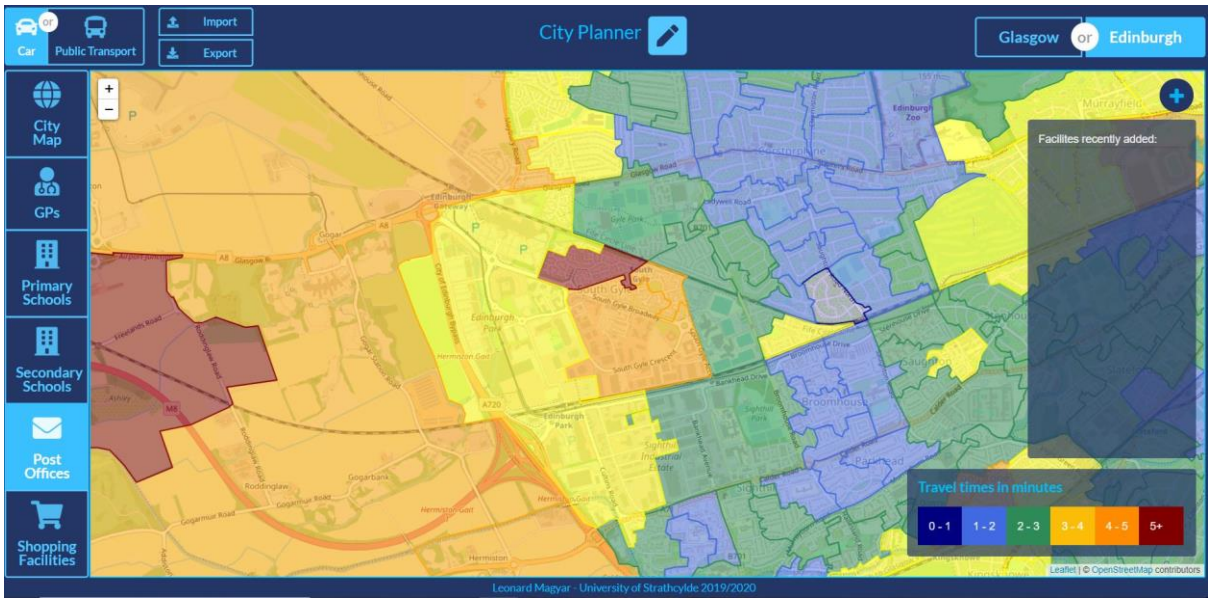
To save underlying JSON data Export button can be used.



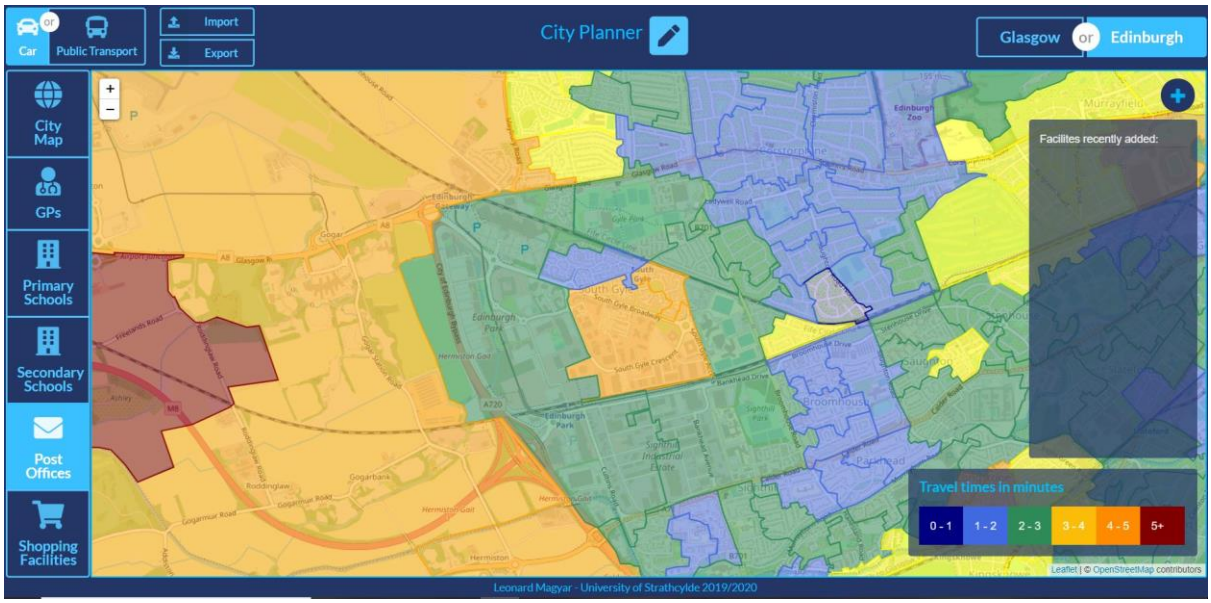
13.8 Import JSON Data

The previously exported file could be imported back to the application. To reset changes made to the application, refresh the page.

After refreshing, before importing data back to the application:



After importing previously downloaded data back to the application by using the ‘Import’ button:



14 Appendix E – Maintenance Guide

This section contains additional information helpful regarding maintenance of the application.

The application is stored at https://magylenny.github.io/CTYPLNR/. The JSON endpoint for boundaries of data zones in Glasgow and relative travel times can be found at: https://api.npoint.io/aa3a9094c684db09d0f8. The JSON endpoint for boundaries of data zones in Edinburgh and relative travel times can be found at: https://api.npoint.io/f7c3649ae02eea7f5e92. The database tables are only available locally, so it would need deploying before other users could gain access to it. However, access to the database would only be needed if another city is being added to the project.

To run the application locally, the local device needs to have Node.js installed on it. After Node.js is installed, open the source and public folder of the project alongside with the package.json in your preferred editor. Alternatively, the project could be also be checkout from version control using this url: https://github.com/magylenny/CTYPLNR.git. After this, navigate to the folder and run 'npm install' to install all the dependencies located in the package.json file. Finally, you will be able to run the application locally by running 'npm start' command.