OPTIMAL LOCATION FINDER

April 2019

Project by Addil Afzal

City, University of London

Supervised by Prof. Jason Dykes

Contents

Abstract

The purpose of this project was to simplify the property searching process, by gathering meaningful data that would otherwise be retrieved from various sources, and then representing it through one interactive system. This would allow for the user to make an informative decision much quickly, by limiting the number of different types of research that need to be performed independently. The data collected had one of two purposes: to be used as filters to limit a result-set, and/or to display in relation to a property.

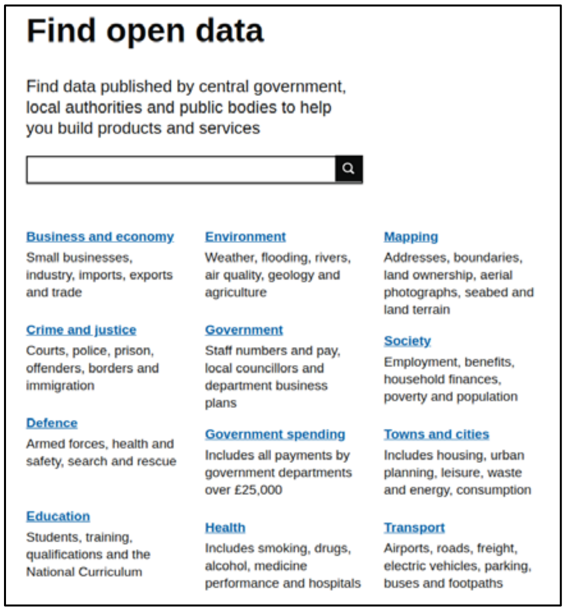
Acknowledgment

To begin, I would like to thank everyone involved at the various stages of this project. I would like to thank the participants that took part during the survey process, for their precious time and feedback. Most importantly, I am grateful for the valuable support that my supervisor, Prof. Jason Dykes, has provided throughout this experience.

Chapter 1: Introduction

Users in the housing market are often faced with an overwhelming number of decisions, when looking to rent or purchase a property. This requires a lot of independent research to be conducted into the factors in which the user finds important in relation to their ideal property. For example, a factor considered by many is the duration of time required to commute to a place of work. Ideal, they would want for it to be as low as possible. However, the challenge arises when also trying to meet a number of other constraints as various resources then need to be used to gather information. This can be a time consuming process, thus an issue for those looking to relocate on short notice.

Data has become increasingly available online, from government sites in the form of open data, and via public APIs provided by some of the large organisations. **Figure 1** shows the categories of data that the official UK government website provides.



Find open data

Find data published by central government, local authorities and public bodies to help you build products and services

Business and economy Small businesses, industry, imports, exports and trade	**Environment** Weather, flooding, rivers, air quality, geology and agriculture	**Mapping** Addresses, boundaries, land ownership, aerial photographs, seabed and land terrain
Crime and justice Courts, police, prison, offenders, borders and immigration	**Government** Staff numbers and pay, local councillors and department business plans	**Society** Employment, benefits, household finances, poverty and population
Defence Armed forces, health and safety, search and rescue	**Government spending** Includes all payments by government departments over £25,000	**Towns and cities** Includes housing, urban planning, leisure, waste and energy, consumption
Education Students, training, qualifications and the National Curriculum	**Health** Includes smoking, drugs, alcohol, medicine performance and hospitals	**Transport** Airports, roads, freight, electric vehicles, parking, buses and footpaths

Figure 1 – Categories of data

1.1 Problem to be solved

There have been many attempts to incorporate data into solutions that aim to help find a home. For example, some of the popular online home-finding websites make use of publicly available data when displaying individual property listings, but do not offer the functionality to filter results based on them. At the same time, there are many user-concerning factors that are not addressed, like the commute example discussed, and local amenities other than public transportation.

The aim of this project was to simplify the home-finding process, making use of much of the data that has become available, to address factors that individuals are generally concerned with, other than those that are being addressed by existing solutions. This had to be represented as a solution that would be intuitive and simple to operate. To accomplish this goal, the author examined past publications and existing system to identify how they chose to implement their functionality, and to seek inspiration.

To create such a solution, during this project the author went through the process of gathering factors that would typically influence those in the market.

These were then converted into requirements, where feasible to implement as functionality. The data was then used in one or two ways: for shortlisting results or to associate with a property.

1.2 Project Objectives

Initially, as described in the PDD, the objectives were focused mostly on the end-result of the final system, and not the activities that would take place surrounding the development, such as gathering data, performing research and design. Upon re-evaluation, the objectives were devised to reflect to the project plan.

To achieve the primary objective, a series of sub-objectives were devised:

1. To analyse existing publications of similar projects that aim to simplify the home-finding process and apply knowledge from their findings.
2. To analyse existing property finders, closely examining the information that they display through their chosen method of representation, and their search process.
3. To find factors that are considered during the home-finding process, and to associate them with publicly available data.
4. To develop and test the functionality of a web system that would allow individuals to find a property based on a set of unique requirements. Where possible, making use of the appropriate technologies to ensure an intuitive design and a structured code base.

To evaluate the effectiveness of the final system, at the end of the project a survey would be conducted to gather feedback from beneficiaries. This would then be used to evaluate how effectively the objectives were attempted.

1.3 Anticipated beneficiaries

When the project started, these were the group of anticipated beneficiaries that had been concluded. However, as the project progressed, this list changed slightly as will be discussed at a later stage.

1. Students that wanted to move out for university were a target for this solution as it would allow them to filter for properties that met their budget and commute time needs.
2. Parents looking to find a reputable school for their child. The solution would use Ofsted school inspection ratings as a method of informing parents of the quality of teaching to expect at each given institution.

3. Anyone looking to ‘optimise’ their daily routine by relocating to a location that provides them with more of the amenities that they require, without having to perform as much independent research.

4. The author, as this opportunity would involve developing research skills when find data, analyse solutions and implement a complex system making use of many different resources.

1.4 Project scope

As this solution was to be developed as a proof of concept, the system was limited to the London area. Finding relevant data sources for information regarding all areas across the UK would have been unrealistic due to the amount of research and time that would be needed. Given that one of the goals of the project was to find factors and associate them with data, the focus was to find as many factors as possible and associate it with sufficient amounts of data, which would only have been achievable for a small region, such as London. The reason why this area was chosen was due to it being the capital, thus expecting for there to be more data available.

1.5 Report structure

Chapter	Description
2. Outputs Summary	A list of output that were devised during the project.
3. Literature Review	A list of resources used to aid the development of the project.
4. Method	An overview and brief reasoning for methods used to conduct the project.
5. Results	The results from conducting the methods. Detailing implementation, design and research.
6. Conclusion	Evaluation of project objectives, and future work/improvements
Glossary	Definition of technical terms used throughout the project.
References	Links to resources used throughout the conduction of the report.
Appendices	Containing code and documents produced during the conduction of this project.

Chapter 2: Outputs Summary

2.1 Application

Description:	The proposed solution. Implemented in two parts. The back-end is used to host APIs and provide static files, split into apps. Each app will The front-end consists of React component classes that are grouped into directories based on their functionality. Front-end code can be found under Core/static.
Usage:	Will be used to run the application.
Beneficiaries:	Everyone.
Link:	Submitted on a USB Flash Drive

2.2 Requirements document

Description:	List of requirements that were devised during the research/analysis stage of the project.
Usage:	This document will be used by the Author to keep track of the requirements to implement, during the use-case diagram construction phase, and when the UI is being designed.
Beneficiaries:	City, University of London, Author.
Link:	Appendix M

2.3 Project log

Description:	A word document to keep track of events that occurred throughout the project, such as meetings, milestones, problems encountered during implementation and decisions made.
Usage:	Inform the user with the authors thoughts and progress at each stage.
Beneficiaries:	City, University of London, Supervisor, Author.
Link:	Appendix P

2.4 Testing Documentation

Description:	A document identifying the requirements that were tested, with their outputs.
Usage:	Inform the reader with the thoughts and progress at each stage.
Link:	Appendix J

2.5 Use-case diagram

Description:	A visual representation of the functionality that was to be implemented. Would later aid the UI development process.
Usage:	City, University of London, Author, Supervisor
Link:	Appendix E

2.6 Questionnaire and Results

Description:	Questionnaire + results. The questionnaire was developed to gather feedback for the developed solution, to determine whether the author's methods were successful at producing an effective system, the results of which were analysed by the author against the objectives.
Usage:	City, University of London, Author, Supervisor
Link:	Appendix J

2.7 Deployment Script

Description:	Segment of code written in bash shell. Purpose was to aid the deployment process when attempting to trial the proposed system with participants.
Usage:	Author
Link:	Appendix R

2.8 Factors analysis document

Description:	A document which gathers factors that are potentially considered by people in the housing market. Used to aid the establishment of requirements. Factors were associated with data where possible, and then decided upon how they could be integrated.
Usage:	Author
Link:	Appendix N

2.9 API Definition Document

Description:	This document contains definitions of APIs for use during the back-end integration stage.
Usage:	Author
Link:	Appendix K

Chapter 3: Literature Review

This stage of the report documents the literature that was referenced through the project. The contents will also cover some of the thought process behind the technologies that were reviewed to accomplish the project.

3.1 Analysis of publications

3.1.1 The Dynamic HomeFinder

The original home finder *[The Dynamic HomeFinder: Evaluating Dynamic Queries in a Real-Estate Information Exploration System (Christopher Williamson and Ben Shneiderman) 1992]* attempted to accomplish much of what the author planned to do with the project, "*to explore a real-estate database and find homes meeting specific search criteria.*", with a similar reasoning behind its need, "*Finding a home is a laborious task for those that have experienced it.*". However, this solution was limited by the technology and the amount of data available at the time. The author evaluated some of the findings in this paper to aid the development of this project.

3.1.2 Home Finder Revisited

The author was referred by Prof. Jason Dykes to another paper *[HomeFinder Revisited: Finding Ideal Homes with Reachability-Centric Multi-Criteria Decision Making (Di Weng , Heming Zhu , Jie Bao , Yu Zheng , Yingcai Wu 2018)]* of a system that also took inspiration from the Dynamic HomeFinder, however, this placed more emphasis on reachability from a selected number of locations. This paper also placed more emphasis on being beneficial to people searching for a home by allowing the user to provide points such as place of work and recreational facilities, as points on a timeline. This was influenced in the map functionality of the final system.

3.2 Analysis of websites

As the proposed system had to add additional filtering functionality on top of what was currently offered, existing systems were observed to see how they had implemented their inputs for their searching functionality and general layout (**Appendix B**). RightMove (https://www.rightmove.co.uk/) and Zoopla (https://www.zoopla.co.uk/) were the main online home-finders in the industry, for which reason were the solutions examined. These solutions were referred to during the interface design process.

3.3 Finding factors

Requirements were gathered during the research stage by attempting to find factors which people generally consider. The HomeOwnersAliance had an article about the factors that are genrelly considered by those in the market. Each factor, when associated with data, was evaluated to determine how it could be integrated into the system.

3.4 Programming languages and frameworks

A vital part of the solution was to design an interface with functionality that was intuitive, thus finding a programming language that was appropriate for this task was crucial. Using a language such as Java or C++ would not have made sense, since in terms of interface design, they were not well suited. Naturally, this task was best accomplished through the means of a web solution given the amount of flexibility with appearance, and cross platform support that would be provided. With any web-based system, there are two main aspects: the front-end and the back-end.

When hosting the back-end, the author chose to use Django, a Python based web framework, with the reasoning being familiarity. Django allowed the project to split different types of data into packages known as apps. Each app had its own set of models, serializers and APIs. For example, functionality relating to Zoopla could be kept in a different package from functionality provided by Google maps; this would force for a more organised layout for code structure.

The front-end only has one programming language, JavaScript, however it has many libraries and frameworks which extend the core functionality. The framework that was selected was react as it was most appropriate for the given task with the ability to class visual elements with their own unique behavior. React was used to create filter blocks on the front-end and manage the general structure of the code.

3.4.1 Map

The chosen method of showing results was via an interactive map. The technology used to accomplish this was a library known as LeafletMaps. The library's documentation (https://leafletjs.com/reference-1.4.0.html) was being used consistently when attempting to define custom map related behavior, such as 'onClick' events and boundaries. An alternative to this would have been Google maps; however, this solution did not offer enough customization and had API call limitations. To information for routes between points of interest, the HereMaps developer API was used. Used to develop the commute filter.

3.4.2 Interface styling framework

Semantic UI and ReactJS are front-end libraries used for development. Since functionality was prioritised over styling, a generic styling framework was used to accomplish a consistent design throughout the application. Elements were copied from the Semantic UI website (https://react.semantic-ui.com/) and placed into React components accordingly. The React website (https://reactjs.org/) was referenced throughout the front-end development phase to retrieve documentation for component related functions such as 'componentDidMount' and 'componentWillMount'. The chosen styling framework was semantic UI, for the reason being that it provided more visual components compared to its competitors, Bootstrap and Foundation, and it had a dedicated library for react.

3.5 Books

The author used *(Dawson, C. (2011). Projects in Computing and Information Systems. Old Tappan: Pearson Education UK.)* as a method of guiding them through the project. In-particularly, section 6.2.2 (Requirements capture) was referenced during the requirements gathering stage. This is where the idea of prioritising tasks with labels such as 'Must', 'Should' and 'Could' was obtained, to manage the importance of a task as would be referenced during the implementation stage.

3.6 GitHub - Version Control System

GitHub is an online version control provider for Git, a version control system. It is used to store source code. This was made use of in the project to back-up the code externally for the unlikely event that the work was lost. This would also allow for the code to be developed in a systematic approach, by using branches to handle the development of different functionality concurrently, as was done with this project. A history of commits can be seen in **appendix L** (http://github.com)

3.7 StackOverflow

StackOverflow is an online community for programmers from all backgrounds, where they can ask questions and expect answers from other members. Some of the solutions on this website were used during the implementation stage when developing the python back-end. In particularly, a KDTree algorithm (https://stackoverflow.com/questions/48126771/nearest-neighbour-search-kdtree) , Haersine distance function (https://stackoverflow.com/questions/25623829/returning-nearby-locations-in-django?lq=1) and a multi-threaded function (https://stackoverflow.com/questions/2632520/what-is-the-fastest-way-to-send-100-000-http-requests-in-python) were re-purposed from examples found on the website.

Chapter 4: Method

The aim of this chapter is to provide a detailed report of the work that was undertaken to accomplish this project. This chapter will describe all the analysis, design, implementation and evaluation activities involved in great depth.

4.1 Development methodology

The chosen methodology for conducting this project was a cross between Agile and water-fall for the reason being that the author envisioned for there to be overlaps between different stages. For example, the implementation and testing stages would be conducted together to save time, as would research and implementation when attempting to design interfaces and when looking for libraries to aid coding.

4.2 Research

4.2.1 Finding property data

The purpose of this task was to find a source of property data that would provide enough information to be able to plot each record. What was required specifically was either a set of Geo-coordinates, or a full address which could be converted to coordinates. At the same time, the data had to be in a format that did not require much manipulation before use, such as via an API that returned a JSON response. This involved looking at some of the most popular online property markets and establishing the resources that each provided.

4.2.2 Finding a directions API

Being able to retrieve commute times and distances between properties and points of interest was considered an important aspect of the system, since the project was based around the idea of finding a property located in an ideal location. Simply returning the distance between two points would not be enough information for the end-user to determine the amount of time required to travel from point A to point B, thus would not allow the user to establish whether the location was ideal. There are many factors which can influence travel time, such as mode of transport, time of day and day of travel. When finding an API, the author had to ensure that those were considered by the service. Whilst performing research, the author briefly evaluated available solutions and the associated costs of using the services if any. Two solutions were looked at GoogleMaps and HereMaps, both of which provided very similar functionality. The only differentiating factor was found to be the number of API calls that were provided. HereMaps was the preferred solution as it provided a massive 250,000 calls per month.

4.2.3 Finding factors

The basis of this project was to create a system that would gather data and display meaningful information to the user, so that they do not have to perform as much independent research as previously necessary. To establish requirements, the author had to undertake the task of finding data that people on the market would generally find useful when looking for a home. The author used online resources such as (HomeOwners Alliance, 2018) and to understand factors which were considered when looking for a property in a area. The factors gathered were written into a spreadsheet document (**Appendix N**).

4.2.4 Analysis of publications

The author went through the process of reviewing existing publications of solutions based around housing and transportation. The aim of this was to identify some of the challenges that others had faced, to identify reasoning behind implementation choices, and to gather requirements that may be of use to accomplish the author's end goal. The websites were visited, and their UI interfaces were analysed, paying close attention to their use of input types to obtain certain types of information. Both the display and filter processes where being observed.

4.2.5 Analysis of existing systems

To identify additional features that could be of use to embed from existing systems, the author went through the process of review current home-finding solutions such as Zoopla and RightMove. The key features that were being observed were how they allowed users to filter for properties, as well as the information that they thought was useful, and how they were presenting them. Much of the analysis was placed on the inputs as well as the data present on the websites.

4.2.6 Eliminating non-feasible factors

Under this stage of the project the author went through the process of eliminating factors for which data could not be found online. As with the properties data, this also had to be in a format that would be easily importable, such as in CSV or SQL, or accessible via an API. A key aspect of the data was to be able to link it to each property via an attribute such as postcode or any other reference to sub-location. Another aspect considered was how the data would be stored and if caching was necessary for when making API calls or when performing computationally heavy tasks. Each factor/data combination was then evaluated to decide how integration into the system could commence. A table was created to summaries the use-cases for each type of data, as shown in Appendix B.

4.3 Establishing requirements

Once research had finished and data had been gathered, the task of establishing feasible requirements was attempted. The requirements devised were either functional or non-functional and were produced from findings gathered during the research stage of the project. A document was produced, listing each requirement with its reasoning, possible methods of implementation and data to use if applicable (**Appendix N**). Requirements were devised by evaluating functionality from websites, and whether it would be of use in the proposed system. Requirements relating to data were devised based on how they could be implemented: as a filter or as a representation on a map/table. Factors for which data could not be found were eliminated from the requirements process.

4.4 System design

An important part of this project was mapping out each of the different functionalities that were proposed in the preceding stage. When designing the system, two tasks were attempted.

4.4.1 Use-case diagram

A use case diagram was developed (**Appendix E**), and a basic system UI had been constructed. The use-case diagram was developed using Visual Paradigm, and the UI was coded with JavaScript libraries such React and SementicUI. The purpose of the use-case diagram was to aid the UI design by allowing for a user-flow to be defined at the different stages of the home-finding process.

4.4.2 Front-end UI design

The author had planned to jump straight into coding a UI rather than to produce sketches or wire-frame diagrams as they believed it would save time, having attempted the task of converting wire-frame diagrams to an interactive UI previously. The building blocks of the UI would be provided by a JavaScript library known as SemanticUI, and once combined with React it allowed for visual components to be added and customised rapidly. The use-case flow diagram formed earlier assisted the process of developing the UI in increments; main flow activities were developed first, followed by alternative flow. The author developed the UI to incorporate placeholder data, which would be overwritten once API end-points had been established during the implementation stage. Many of the sources of data required appropriate attribution which was also considered. Since the basis of the project was to create an intuitive design, the solution would be self-explanatory to use, thus eliminating the need to form a user manual at a later stage.

4.5 Implementation

This stage of the report highlights the activities undertaken during the implementation stage of the report. This involved defining model classes to store data, importing data, creating API end-points, and then integrating the front-end with the end-points.

4.5.1 Models

The first aspect developed were the back-end model classes (**Appendix F**). A model class allows for a database table structure to be defined without having to write SQL code. Each column in a table is represented by an attribute on a model instance, and each row represents an instance of a model. A model was developed for each of the types of data that were going to be import.

4.5.2 Importing data

The next task at hand was to import all the data that had been gathered during the research stage. With the models having been defined earlier, an import method was written for each data file (**Appendix G**). All file-based data was in a CSV format, meaning that each line in a file corresponded to an instance, hence data was imported line by line. Each group of models were developed under a unique app. An app allows for related code to be packaged together for a more structured file layout. Property data was obtained by making a series of calls to the Zoopla API spanning across 2 days, due to the limitations of the API as mentioned earlier.

4.5.3 APIs

To get data from the back-end server to the front-end browser, an API end-point was created for each type of data. Creating an end-point consisted of three main aspects: creating a serializer class, creating filter classes and defining a route. A serializer class allows for response structure to be defined. A single API existed for the filtering stage as the content displayed was a set of static options. Multiple APIs were developed for the results page which displayed data relating to each property. A document was produced to aid the integration stage, as there were numerous end-points that were defined and were a challenge to keep track of. This contained URL, definition location and input/outputs where appropriate.

4.5.4 Front-end integration

With back-end APIs defined, the front-end was updated to make requests and substitute the response into the placeholders created during the design stage. A total of six APIs end-points were defined for the information based components displayed on the results page. When a category of information is selected, an asynchronous call is made to fetch data from the back-end. During this process, a loading icon is shown to inform the user that a task is being executed.

4.5.5 Performance optimisations

As the application was data heavy, performance optimizations had to be made to both the front-end and the back-end. The main concern on the back-end was that responses were taking a very long time to generate. This issue was addressed by performing some research into the Django architecture (https://docs.djangoproject.com/en/2.1/topics/performance/) and establishing that a technique called pre-fetching had to be used when performing database transactions. Each viewset was then updated, with the 'prefetch_related' method call added to each of the base queries. Another optimization made was made to the way in which data was fetched from external APIs. To limit the number of duplicate API calls, caching was introduced into the HereMaps sub-system. This sub-system was optimized even further by using multiple CPU threads to perform calls. Both changes helped speed up response times. Previously, a series of API calls that would take over 30 seconds was optimized to only require 2 seconds, which allowed for API call numbers to be increased. The aspect of the front-end that required optimization was the map displaying results. In many cases, a query would return more than a few thousand results which would take a very long time to render and manipulate. This issue was addressed by clustering nearby points together and then displaying a count at the center. When zooming in, the clusters would be split. Data serializers were updated to return only the requested fields by an API call, since during the implementation stage all fields were being returned. An additional serializer was created for returning data that would be seen when clicking onto a property on the results map. This change in serializations allowed for results to be loaded within a fifth of the time previously required.

4.6 Testing

4.6.1 Functional Testing

The purpose of this type of test was to ensure that functions were returning the correct results/output for a given input. This was mainly in relation to input fields used on the filters page as well as API calls made when trying to load specific information about a property. When developing the back-end, functional testing was being performed as each API was being defined. On the front-end functional testing was performed with the definition of each filter. A document was created to keep track of the tests, with reference to the inputs used, outputs produced and expected output. Another test was to ensure that

4.6.2 Unit testing

The purpose of this type of testing was to ensure that some if the critical back-end methods were performing the operations correctly. Only the core sub-system was tested since two of its methods were being re-used throughout the application.

4.7 Deployment

The system could have been trialed from the authors laptop, however, this would not have been ideal for participants as they were encouraged to take their time to provide well-thought feedback, during which to survey multiple people at a given time. Research was undertaken to find a suitable location to host the web application, that supported the requirements of the web-server and did not involve a hefty fee. The solutions were to either self-host or to find a hosting service. Research was performed into the matter, looking online for recommendations on websites such as quora.

4.8 Survey

A survey was created to find out how useful the filters and data were, in the context of finding a home. This opportunity was also used to gather opinions relating to the way in which functionality and data were being displayed. A three-part survey was created to obtain findings in each of the following areas: filtering options, property related data and user background. The survey went through 3 iterations of the review process by the project supervisor, Prof Jason Dykes. Initially, the intended target audience of the survey was anyone that had been looking for a home recently. However, finding a sample of individuals which met this very specific requirement was not feasible, therefore the survey was made available to everyone. An additional question was added to the survey which asked, "*Have you been looking for a home recently?*". This would allow the author to better evaluate responses. Before the survey could be conducted, the author had to propose a consent form, a participant sheet and complete an ethics checklist for the survey to be approved by their university. The plan was to find 10 participants in the A309 area, a location dedicated to Computer Science students and lecturers. Each participant would then be asked to read a participant sheet and choose whether to be involved with the study. If so, they were given a consent form to complete, after which they would be allowed to trial the proposed system and provide feedback. The participants that took part would spend no-longer than 20 minutes of their time assisting with the study. Upon conducting the survey, only 6 participants were gathered, all of whom were students. The feedback they provided was diverse. A few minor issues were experienced, related to browser that individuals were using.

Chapter 5: Results

This chapter will describe in detail the results that were obtained using the methods as defined in the previous chapter.

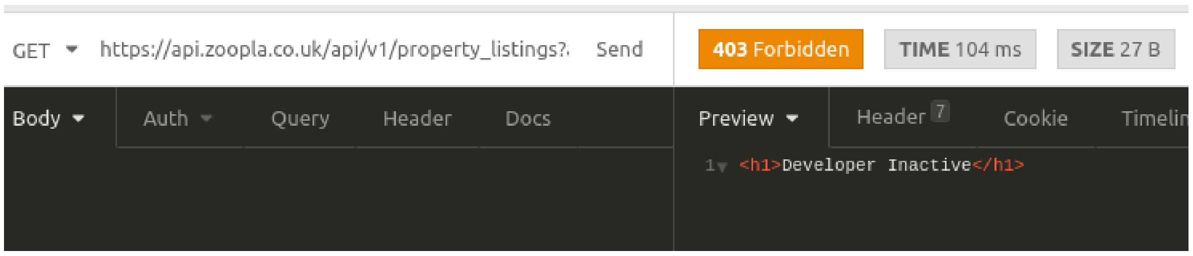
5.1 Research

5.1.1 Finding property data

The first task was to find a source of data for property listings. Each property listing would need to have some information that would allow for it to be plotted onto a map. Look at Zoopla, they provided an API that allowed for searches to be performed. The information that the API had to provide was anything that would allow for an accurate plot for each location, such as a postcode or a set of coordinates. The API did not provide a postcode however it did provide a set of coordinates (latitude and longitude) which would allow for each location to be plot accurately enough to approximate where a property was located on a given street. The exact precision of accuracy of the locations could not be tested as a door numbers were not provided with the data. To use this service, an API key had to be requested, which was provided instantly. The key terms of service were to ensure that users were directed to Zoopla, and not the estate agent when displaying listings, and that when they were being stored, that they had be kept up-to-date. Not meeting these requirements would result in the API key being deactivated, and Zoopla could then request for the data kept being deleted.

A limiting factor of the API was that it would only allow for at most 100 requests to be made each hour, and each call would return at most 100 results meaning that multiple requests had to be made to fetch all results associated with a search. For example, if a search query consisted of 2,100 results then 21 calls would need to be made to fetch all listings; this was a limiting factor.

Another more critical issue experienced was that during the early testing phase, the author's API key was disabled a day after registration. Making an API call would yield a response as shown in **Figure 2**.



```
<h1>Developer Inactive</h1>
```

Figure 2 - Response when requesting data via the API

To request a new key meant registering for a new account and generating another. Upon going through the procedure again, the same issue was encountered.

An attempt to contact Zoopla about the matter was made via their contact page (https://www.zoopla.co.uk/contact/), however, a reply was never received. Checking the developer site, there happens to exist a forum, but users are unable to post there. When searching Google for “Zoopla API”, the author came across an article (Mediumcom. 2018. How not to run an API (looking at you, Zoopla). [Online]. [10 December 2018]. Available from: https://medium.com/@mariomenti/how-not-to-run-an-api-looking-at-you-zoopla-bda247e27d15) explaining that the service had been “abandoned”, suggested that the API was no-longer being maintained or supported, which did not answer why the service was still online. This was problematic since there did not exist any alternative APIs from other large providers such RightMove or PurpleBricks, from where the author could get property data.

To counteract the problem, an alternative method was to scrape data from a property listing website that allowed web-crawlers, which would have been difficult to find. RightMove and Zoopla both prohibit the use of automated means of scanning their websites, as stated in their terms of use, thus making the approach challenging to attempt. Another problem with this solution was that it would be very time consuming to query for results and would take a lot of time to write a script that would download a page and extract all the needed data.

A more feasible solution was to simply limit the system to a smaller region, such as London, and download as much data as possible for all properties that fell within, still using the API but with a new key. This would then mean that the system would no-longer display accurate results as listings would eventually become outdated and require for the data to be fetched periodically with a new account and API key. Storing data was allowed by the service, however, enforced that developers update the data frequently as mentioned before. Keeping the data up-to-date was not possible since the API key was being disabled shortly after creation, without any explanation from Zoopla, thus preventing updates from occurring. This was not seen as being much of an issue since the data was only used to demo a proof of concept system, thus not being displayed to influence potential of purchase and was being kept under the other required conditions. If the providers did request for the retained data to be removed, which had not been attempted during the project, then the author would have had to oblige. This was a potential risk but given that the organisation failed to respond to the earlier query and did not attempt to contact the author, it was still an option that could be attempted.

This was the preferred method, thus selected, as it would provide quickest method of getting a working solution running. The author could have chosen to simply create a new account and request a new API key each time it expired as it was not prohibited in the terms of use, however, there was uncertainty as to how long the service would remain operational.

When fetching the data, it would not have been efficient to fetch all the listings as it would simply take too long, for which reason the proposed system was limited to the Greater London area. Searching the API for properties within "London" would yield only 100 pages of results, and given that each page only contained 100 listings, fetching all pages would return only 10,000 results, which would not have gathered enough data. The approach to fetching data for all London based properties was to query the API by postcodes that fall within the city, more specifically the first 2-3 characters, as the number of pages of results would not exceed 100. A list of London based postcode districts was found on Doogal. The process of fetching data from the Zoopla API was attempted in December 2018 and took numerous days to complete.

5.1.2 Finding a directions API

Upon conducting research, two feasible solutions were found: Google maps and Here maps, both of which provided very similar functionality. Even though google maps had been around for much longer, HereMaps had matured relatively quickly, evident by the amount of documentation available online. Both solutions performed relatively fast at returning results and offered matrix-based queries for non-public transport routes. They both had support for being able to set departure and arrival times up to a week in the future. The key requirement which the solutions had to satisfy was the ability to provide directions for public transport routes and provide enough API calls to be able to make use of the resource. The Google maps API would provide only 40,000 monthly calls, whereas the HereMaps API offered 250,000, for which reason it was the API which ended up being used.

5.1.3 Finding factors

The next step in the research process was to find data that would be meaningful to display and/or use as a filter for limiting results. The reason for filters was because the system could return thousands of results, making the home finding process time consuming, and introducing a filtering criterion would bring the count down, making the results more manageable to view.

The data gathered would most likely be for local areas and less about individual properties, since almost all the property specific data would be taken from Zoopla's API, for which reason the author acknowledged that they would be performing research to gather area specific data. Upon conducting research, the author found an article on the HOA website that rounded up a few area specific factors for home buyers to consider. Many of the factors were in the form of a question, which happened to relate to styles of living such as "*Do you want to be in a happening place, with lots of life?*" and "*Do you relish a laidback lifestyle or will you get bored?*", for which finding data would be difficult. However, there were other factors for which data could be found, such as: crime rate statistics, sporting facilities, schools, health care amenities and transportation links. The author chose to construct a table showing each factor with a potential data source, as shown in **appendix 1.** It was quickly established that not all factors were logical to use as filters. For example, filtering by accessibility from local health care services would not be very useful since almost all properties within a city are within reach of such services, therefore would be more useful as information displayed when viewing details for a property.

5.1.4 Analysis of existing publications

5.1.4.1 The Dynamic Home-finder

This solution placed more emphasis on the ability to quickly change search parameters. This was evident given that the solution used sliders and select boxes, portraying that this solution was aimed more at analysts looking for patterns rather than people looking for a home.

The author considered the idea to incorporate filter options beside a map, as present with this publication, however, with the quantity of data that was envisioned they assumed that the solution would not be capable of plotting results in real time, for which reason planned to separate the filter and display functionalities. An advantage seen at the time was that both the functionalities could have more space to be represented, which was considered important for the map as it would be needed in situations where many results are being displayed. In much the same way, when clicking onto a house on the map, the author wanted to show further information about the location. They had considered showing the information when hovering over a location but figured that it would be difficult for the end-user if locations were plotted very small and closely to each other.

The paper mentioned that the interface was found as being 'fun' to use by the subjects, with it having acted as a 'motivational factor' to wanting to use the system and concluded that it was due to the range of animations used throughout. This was something that would be taken into consideration during the design stage.

5.1.4.2 HomeFinder Revisited

This solution attempted to address a single factor in the home finding process, being the reachability from key points of interest. Each was represented on a timeline, with a link showing duration between the points.

The system that the author envisioned would allow for the user to define travel times from a selected number of locations via chosen modes of transport, however would not illustrate the functionality to sequence journeys. The intend solution would allow for the user to define a travel time radius, thus the overlapping regions would contain the properties that meet all travel constraints. When displaying reachability, the system uses paths drawn on a map to show routes between key points of interest. This helped to visualise a summary of the route.

5.1.5 Analysis of existing systems

Functionality and design choices of existing home-finding services were investigated, selecting those that would be feasible to implement. The first solution analysed was Zoopla. The layout of the search page was very clean and quick to navigate with minimum typing, as was illustrated with the use of auto-completion and drop-down boxes, as shown in **figure 3**.

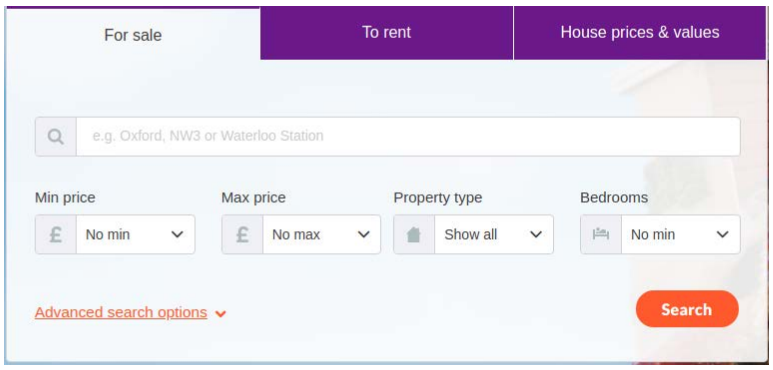


Figure 3 - Zoopla interface

When proceeding with a search and selecting a property, four categories of information were shown: Property details, Floor-plan, Map & Nearby and Market stats. Each category was listed as a menu heading on which the user clicked to show corresponding contents.

Something that was present with the design of each category was that the information was well spaced and kept brief, getting straight to the point. The reasoning behind this may have been due to the overload of information that was already on the page, and leaving gaps showed clearer distinctions between the information and functionality. The functionality that was of most interest was the map, which featured distances from nearby schools and train stations, pin pointed. When viewing the map, all points of interest (POI) were displayed at the same time, which seemed to work in this situation since there was a limited number of points to illustrate; each showed a clear distinction from its background by using colours darker than those used on the map.

Another website that was examined was RightMove. The searching functionality was very similar to that of Zoopla, however, in this case it was split between two pages. The area is entered on the first page, and then upon submission a redirection occurs to another page where the user is asked to provide additional information. The reasoning behind this may have been to keep the homepage looking clean due to the presence of other non-related information and functionality, not a situation which should be encountered in the project. The types of fields used to express the inputs remained the same. When viewing information for a specific property, the categories of information were very similar, where the only difference was that RightMove had 'SchoolChecker' as an option as opposed to 'Floorplan', which is embedded into the description section. Distances from transplantation links were present again, illustrating that this was an important factor taken into consideration.

In conclusion, the take away was that the proposed solution needed to have an organised layout for displaying information, ensuring that contents were quick to understand by using brief texts and by spacing contents accordingly. Another finding was that the system needed to be quick to navigate, requiring minimum user input where necessary. Displaying transportation links was also important, however, the author believed that it was not necessary in this project since commute-based data was being used to provide more of a perspective thus being more meaningful.

5.2 System design

This stage of the report covers the design aspects of the project. The author chose to develop only a use-case diagram and not to form a class diagram for the reason being that they expected for the back-end data structure to change rapidly during implementation, when bringing all functionality together.

5.2.1 Use-case diagram

A use-case diagram was developed to aid the UI development process (**Appendix E**). Each category of data would be located under a different sub-system during implementation to allow for related functionality to be kept together and for a more organised architecture. For example, a dedicated sub-system would exist for Zoopla, and another for HereMaps since they did not share any functionality. Even though in code they appear separated, to the user they would seem to be closely related as they would all function and be presented similarly, as illustrated by the use-case diagram. There were two main aspects that had to be covered in the use-case diagram: the ability to filter properties through compulsory and optional fields, and the ability to view the different types of information associated with a property when selected through the map.

5.2.2 User-interface design

During the first stage of implementation, the front-end was designed and developed at the same time, with functionality and layout having the highest priority.

5.2.2.1 Filters

The first aspect of the front-end that was developed was the UI for the filtering functionality. The initial plan was to develop a page with a series of forms, where each form would represent the filtering functionality for a category of data, however, this quickly changed as the author saw that there was a problem with too many options being displayed simultaneously, which would confuse the end-user.

A better solution devised was to contain form features into segregated blocks. Each block would be in one of two states: open or closed. The open state would show all fields and other filter options, whereas the closed state would show a summary of the contents within, as can be seen in Figure 4. Since the filters were being designed with similar functionality, a base component, rather like a class, was declared to contain abstract functionality.

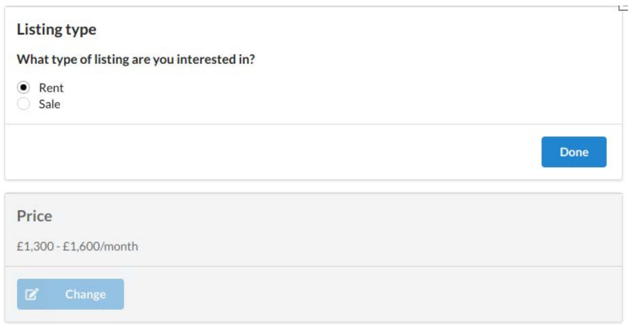


Figure 4 - Filter options as blocks

The aim of the interface was to allow the user to configure the options to their requirements, being able to select only those filters which they want to apply. Having analysed the input interfaces of RightMove and Zoopla, it was found that they always required price and area as inputs, as they were frequently used. Based on this, the author decided to include compulsory fields to reflect the research. Price, property type and listing type were made compulsory. The necessity of area was less of a priority as the system was already being limited to London. The property-type field was introduced to allow the user to select a building type if they had a preference, since the property listings could contain types such as ‘land’ and ‘garage’. Given that these fields were referring to different aspects relating to a property, each was designed to be contained within its own block. The contents of the blocks would only be shown when expanded, forcing all other blocks to close.

Once the generic design for the display of filter options had been established, and the options had been broken down, they were then developed according to requirements that had been defined earlier.

Listing Type

This filter would only have one field: a radio select button with two options. The options were 'Rent' and 'Sale' of which only one could be selected at a given time. The validation for this filter was to select one of the options, after which the 'Done' button would become enabled. (**Appendix S.4**)

Price

Two values were required from the user: a min and max price. This could have easily been accomplished through the use of a drop-down menu, as present on the Zoopla and RightMove websites, however, the author believed that the fields could be represented through a more intuitive design. Since the user was to define a 'range', the author felt that it was most appropriate to use a slider, as shown in **Figure 5**.

When listing type was selected as 'Rent' in the previous filter, an additional field would show up allowing the user to indicate whether the price range was per week or per month. Since the range of prices was to be different for rental and sales listings, this was also considered when creating the slider, so that it would update accordingly. By default, the full range would be selected, with the per-month option, if if listing type was set to 'Rent'. At a later stage, when data had been integrated, the author established a problem: the user could select a price range for which properties did not exists. This was addressed by displaying a price histogram positioned directly above the slider, where the X-axis would correspond with the slider values. Validation was not necessarily since default values were set. (**Appendix S.5**)

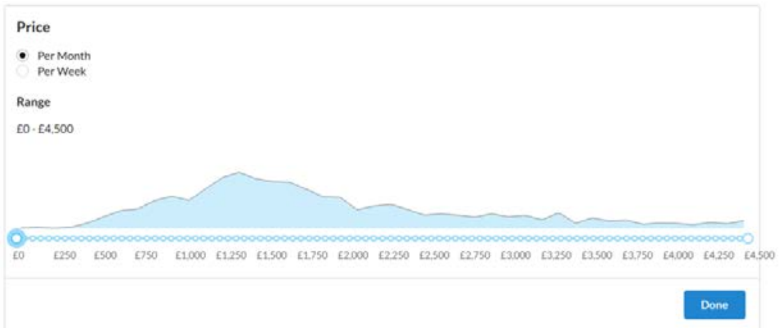


Figure 5 - Price filter interface

Property type

This filter would consist of one field, which would allow the user to select multiple values. On both RightMove and Zoopla, the filter would only allow for one value to this field, however, the values were grouped together into categories where applicable. For example, semi-detached, mid-terraced and detached homes were grouped into 'houses'. The author chose not to develop the functionality in a similarly way as they wanted to provide the user with more control over the filters and allow for the filters to be as specific as possible, thus did not enforce grouping. This field was represented as a multi-select drop-down menu. By default, all property types were selected. This would represent that no filter was being applied. The validation enforced on this block was to ensure that at least one property type was selected in the drop-down menu. (**Appendix S.6**)

Area

This filter would allow the user to select an area where properties should be located. During the requirements stage, two methods of implementation had been devised. The first method involved allowing the user to search using a text field, with the aid of auto-complete suggestions, however, this was not a very intuitive design, as was required for **R14**. The second approach was to use a map and let the user define an area by selecting a point. This would draw a circle over the selected region to indicate the bounds of the search. The radius would then be adjusted in real-time using a slider, displayed below the map. The was the desired approach as it would provide the user with more information than the search-box implementation. A challenge the author faced with this task was deciding how to best integrate the map controls. The three controls that the map itself had to provide were zoom, select and pan. To keep this filter simple, the author chose to only allow a single area to be defined, simplifying the controls. A short click on the map would drop a pin at the center of a circle, and then clicking elsewhere would shift the circle center. A click and drag was used to pan around, and zooming in and out was performed using scroll. The approach to validation was to ensure that a point had been defined on the map, otherwise the user would not be able proceed. (**Appendix S.1**)

Rooms

The purpose of this filter was to offer room filtering functionality, on a room type basis, as needed to meet requirement 3. The API provided the count of three different types of rooms: bedrooms, bathrooms and receptions. Filter each of these fields would allow for more control over the search parameters, compared to existing solutions. Filtering for each type of room required two values: min and max; used to define a range. The author decided that this was best expressed using a slider as done so previously. The values of each slider ranged from 0-10+, where selecting '10+' would represent '10 or more rooms' for a given type. By default, the full range was selected for each of the sliders. This would represent that the filter was not being applied. This filter did not require any validation since an option would always be selected. (**Appendix S.7**)

Commute

This would offer the capability to filter properties based on a commute time to a single location, or multiple locations, via public transport. This was implemented with a map and a search field. The field would allow the user to search for a point of interest, aided by auto-complete suggestions, and the map would then display them. To apply the filter, the user had to provide at least one POI. A table was used to display each POI defined, with the functionality to edit/remove them. A departure/arrival time field would have been a great addition, however was not implemented due to lack of time. Instead, the system was implemented to retrieve commute times for departure in one hour from the current time (default). (**Appendix S.3**)

School Filter

The purpose here was to allow for properties to be filtered by their proximity to nearby schools. The end-user would be given the option to customise their preference based on attributes associated with the schools. These attributes were those that existed with the data-set found, such as education type, gender and admission type. Each of the fields were implemented with choices; education type was a multi-select, whereas admission type and gender were single choice only. The default value for the education type field was to leave each box un-checked, allow the user to define which category of education was of interest. Gender was set as mixed and admission type was set as any, specifying no preference. The validation function was defined to ensure that at least one education type was selected. (**Appendix S.8**)

Upon completing this page, the user would click a 'Submit' button to proceed to the map of results. This button was implemented with a loading animation that would replace the text upon click, with the purpose being to provide the user with feedback, and to make the system more engaging/fun.

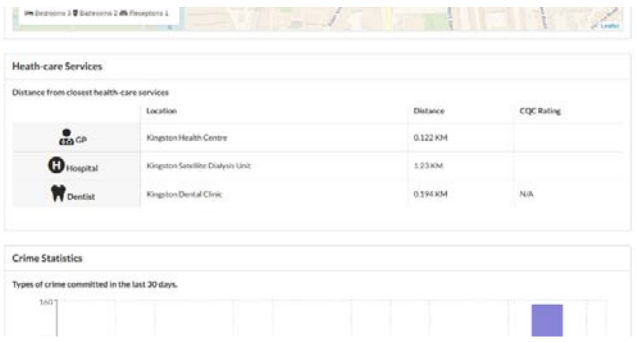
5.2.2.2 Map

When displaying properties, the plan was to pin-point each on a map, with additional behavior being triggered once selected. Since the system would be limited to London, the author chose to force the map to stay in the region of the City, so that the user could not scroll to any areas outside of a defined boundary. To set a boundary, the map required a set of coordinates, which would be generated at a later stage.

The second part of the UI was to display information that would be returned from the API call; this would include the list of properties plotted onto a map, along with categories of area based data, as will be discussed later. In theory, the idea was to implement a map larger enough to be able to view all points. A map of height 700px accomplished this comfortably however this was not suitable for smaller screen sizes. To be more compatible, a better implementation was to set the size of the map dynamically so that it would occupy almost the entire height of the browser's viewport every time the page rendered or when a re-size event occured. If the height of the viewport fell under a minimum threshold then a fixed size of 400px would be used.

5.2.2.3 Information

Figure 6 - Information stacked



Heath-care Services

Distance from closest health-care services

	Location	Distance	CQC Rating
GP	Kingston Health Centre	0.122 KM	
Hospital	Kingston Satellite Dialysis Unit	1.23 KM	
Dentist	Kingston Dental Clinic	0.194 KM	N/A

Crime Statistics

Types of crime committed in the last 30 days.

The data associated with the area surrounding each property was displayed under the map, and not beside as this would decrease the map's overall width significantly. Initially, the structure for displaying property related data was to show everything together right under the map, as shown in **Figure 6** however this changed as it became clear that there was too much information being presented. Instead, the information was split by category and each could be selected via a horizontal menu, similarly to the presentation of the data shown on RightMove and Zoopla, as found during the analysis stage. The menu was hidden until a location was selected from the map.

Summary

The summary information was kept brief, showing the property description, key dates and a link to the Zoopla listing page from where the user could find out more. A key aspect of the system was to provide information other than that already provided by Zoopla, as the user could simply go to the website to find out more.

Crime

The purpose of this information was to inform the user of the types of crime occurring in the area surrounding each property (**R5)**. This would in theory allow the user to evaluate the safeness of the area. Since the data was being returned by an external API and not processed locally, comparisons could not be performed relative to the surrounding city, which may have been more meaningful for evaluating. The data returned by the API was plotted onto the map when the category was selected to show where exactly the crimes were occurring, and a bar chart was shown to visually illustrate the frequency of each crime type. When displaying individual crimes on a map, it was found that in some areas there were over 1,000 occurrences of crime, which would have been impractical to plot due to performance issues, as discussed earlier. As with the property data, each of the points was clustered. This would improve performance (**R16**), but would sacrifice the ability to accurately portray the locations of each crime unless zoomed in.

Health Services

The purpose of this information was to inform the user of the health facilities nearby, as well as the quality of service that they provided (**R6**). This would allow the user to evaluate whether the property was suitable for them, based on whether the health-care facilities met their distance and quality needs. This was represented as a table with three rows, where each row would show a type of health-care facility with its distance and CQC rating. Upon selecting the category, the locations would be plotted onto the map, showing the user exactly where each was located.

Restaurants

The purpose of this information was to inform the user of the distance from nearby places to eat, the types of food that they provided and their ratings if provided (**R10**). This was implemented as a table, due to the number of attributes that were associated with each location. Each restaurant was plotted into the map when the category was selected. This would allow the user to visually find where restaurant hot-spots were (**R14**). In the table, the rating for each restaurant was colour coded to quickly illustrate the quality of service at each location with just a glimpse. The colours where those that were defined by Zomato when submitting an API request. A link to the menu was added, however was implemented so that it would open in a new tab and not redirect the user away from the current page otherwise it would reset.

Sporting Facilities

The purpose of this tab was to inform the user of sporting facilities relative to a property, which would be of help to those looking to practice a particular type of activity (**R7**). This information was represented as a table and on the map. The table simply detailed each of the qualities associated with each facility and contact details where appropriate. Opening times were not displayed where applicable, as is it was information that could be found by visiting the provider's website and was only needed if the user was interested. The map simply plotted the estimated location of each facility using the same icon. The icon could have been customised based on the type of facility, which would have allowed users to visually find the facility they need (**R14**), however was not attempted due to limited time. The number of points displayed was limited to the 6 nearest, as opposed to displaying those that fell within a distance radius since then there could be too many or too little results depending on the area of the property selected.

Demographics

This tab would simply show area specific statistics represented as charts. The aim was to display both ethnic background and age based data, however, only the first was implemented due to time constraints (**R9**). The aim with this data was to build an image of the type of people that were living within the local area. A bar chart was used to represent the data, initially with ethnic backgrounds along the x-axis and frequency along the y-axis. A problem here was that the labels were too long, for which reason the axis' were swapped. The bars where then shown in descending frequency so that the population of each ethnic group could be ranked.

Schools

The purpose of this category of information was to show the quality of education that was available locally. The aim of this information was to allow parents/guardians to evaluate whether the educational resources were suitable for their child/children (**R11**). The information was expressed as a table and on the map. The table showed a list of schools, with attributes such as distance, ofsted rating and the level of education provided (primary, secondary or post 16). The map, again, would plot each of the locations to allow for the user to get a feel of the proximity from nearby schools (**R14**). If a school filter was applied, then showing only the schools that met the defined conditions would have been useful, however, due to lack of time, it was considered an optional task, thus not attempted,

Commute

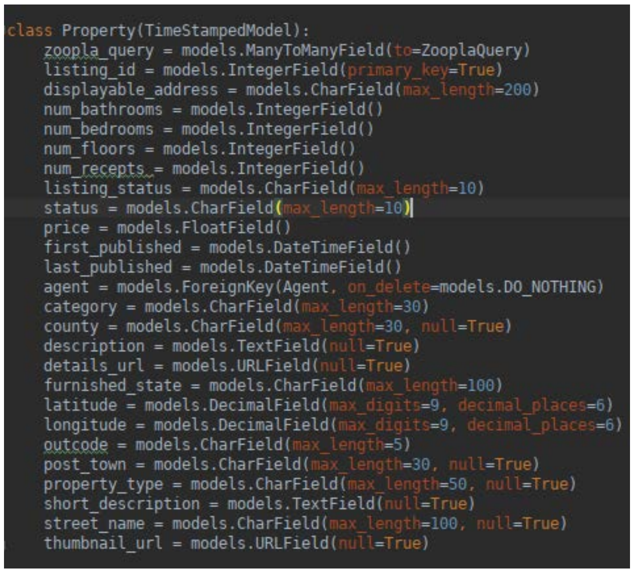
The purpose of this information was to show the amount of time that it would take to reach points of interest from the selected property (**R8**). This tab would only be selectable if the user had provided points of interest during the filter definition stage. The information was shown as a list and as points on a map. This would simply show each location as a block, similar to the filters, where the collapsed view would show a summary of the commute. The routes would appear as paths on the map,

5.3 Implementation

5.3.1 Model declaration

Once the layout for the UI was defined, the back-end was then developed. The first objective was to define models for each of the sub-systems so that data could be imported and then retrieved by the front-end. A total of six apps were declared.

Based on the Zoopla API schema (**Appendix B**), the purpose of these models was to hold data that would have been otherwise fetched from the API in real-time. The Zoopla models would simply act as a local cache for when the API key gets deactivated, so that local filtering could be performed. There were many fields which had to be translated to either attributes or relationships to other models. Based on the attributes that were determined as needed (that would be used somewhere) a total of five models were declared for the app. A 'Property' model was created, as shown in **Figure 7**, to which 'RentalPrice', 'PriceHistory', 'PropertyImage' and 'Author' had associations. As this suggests, the property model would hold data that was specific to a single listing, such as address, description and number of each room. Associations were then defined to store data that had a one-to-many relationship with the 'Property' model. For example, a single listing could have a history of prices. Each attribute would store data of a specific type, with constraints such as length if known. There were some fields for which data did not exists, as was discovered during the importing, for which fields were defined as "nullable" using 'null=True" on the attribute. The model was implemented to extend the 'TimeStampedModel' base class, giving it an additional two attributes for storing information about when the instances was created and when it was last updated. (**Appendix F.6**)



```
class Property(TimeStampedModel):
    zoopla_query = models.ManyToManyField(to=ZooplaQuery)
    listing_id = models.IntegerField(primary_key=True)
    displayable_address = models.CharField(max_length=200)
    num_bathrooms = models.IntegerField()
    num_bedrooms = models.IntegerField()
    num_floors = models.IntegerField()
    num_recepts = models.IntegerField()
    listing_status = models.CharField(max_length=10)
    status = models.CharField(max_length=10)
    price = models.FloatField()
    first_published = models.DateTimeField()
    last_published = models.DateTimeField()
    agent = models.ForeignKey(Agent, on_delete=models.DO_NOTHING)
    category = models.CharField(max_length=30)
    county = models.CharField(max_length=30, null=True)
    description = models.TextField(null=True)
    details_url = models.URLField(null=True)
    furnished_state = models.CharField(max_length=100)
    latitude = models.DecimalField(max_digits=9, decimal_places=6)
    longitude = models.DecimalField(max_digits=9, decimal_places=6)
    outcode = models.CharField(max_length=5)
    post_town = models.CharField(max_length=30, null=True)
    property_type = models.CharField(max_length=50, null=True)
    short_description = models.TextField(null=True)
    street_name = models.CharField(max_length=100, null=True)
    thumbnail_url = models.URLField(null=True)
```

Figure 7 - Model declaration

The HereMaps app defined two model classes, each with the purpose of caching a type of request. This was a necessarily step since the API provided a limited number of calls. A `ReverseGeoCodeCache` model was defined to store the name of a location at a given set of coordinates, for when a pin is dropped in the area filtering front-end functionality. This text would act as a label and would be more meaningful to display to the end-user than a set of geo-coordinates which cannot be interpreted. Given that a cache is generally stored for a defined period before it expires, an instance in this case would continue to exist in the database since locations names are unlikely to change. Another model called `RouteCache` was created to store information specific to routes between two points of interest. (**Appendix F.4**)

A `School` app was created to store school related data, such as name, geo coordinates and other filterable data. The `School` model was created to hold only needed data attributes from the spreadsheet. An `OfsteadInspection` model was defined to store data regarding outcome of ofsted inspections that had previously occurred at each school. (**Appendix F.5**)

A CQC app was created to store data specific to health-care locations. The model was defined as `CQC Location`. The only information required to be stored in the model were the coordinates, address, type of location, outcome of the last inspection, and website if provided. Only a single model existed under this app. (**Appendix F.3**)

To store active place specific data, a `ActivePlace` app was defined with seven models. Not all the data extracted during the import stage would be used, however, each of the attributes was defined with the assumption that they may be needed later. An `ActivePlace` model was defined to hold location related variables. An additional six associations were created to store data with a one-to-many relationship, with the following model names: `Equipment`, `Activity`, `Contacts`, `Disability`, `Facility` and `OpeningTimes`. (**Appendix F.1**)

Having identified that many of the different types of locations of this system were using similar attributes, an abstract `Location` model was created and referenced in previously declared models.

5.3.2 Data import methods

Zoopla

The import method was written to fetch as much data as possible using the API, for London only. Initially, the it was written to fetch all entries for the search term "*London*", however, this would not return all the results but rather only the first 100 pages. Since a page can consist of at most 100 listings, this would result in only the first 10,000 results being returned. The author came up with the idea to search for properties by sub-regions. Searching by borough would still yield too many results, for which reason the author concluded that the search area had to be much more specific to return less results. The final method ended up performing search queries by postcode districts. A list of these was found online at Wikipedia. The postcodes were gathered and added to a python list (**Appendix G.1.1**), which was then used in a loop to fetch all pages for a query of a given postcode district (**Appendix G.1.2**). Each page as a response was received as XML, which then had to be parsed into a python dictionary format. With the data parsed, corresponding models were created to store the needed data. This method was run across two days to fetch approximately 130,000 listings in and around the London area.

Schools

Importing data for schools was done so from a CSV file. The data could have been dumped straight into the database using an SQL script, however, to be compatible with the defined models the author choose to attempt the import using python. The import process was split into two stages. The first stage involved importing school specific data only, using the data file as retrieved from the GOV UK website. The first row of the file was noted and removed, and then the file was processed line by line. Each line would consist of school specific attributes separated by commas. Using this, new model instances were created. Approximately 26,300 schools were imported into the database. The next part of this import process was to import data for Ofsted school inspections **(Appendix G.2.1**). To associate an inspection with a school, a relating attribute needed to be established. The attribute used was `URN`, an identifier that is unique to each school and present with each Ofsted inspection record. Just over 21,000 Ofsted inspection records were imported. The school data file did not contain coordinates for each location, however, they did have an associated postal code. The author wrote a python method, calling an online API at Postcodes.io to fetch and then add a set of Geo-coordinates to each school. Initially, this method of setting coordinates for each of the many listings was slow, as it was performing a single look up at a time; much of the delay happened to occur waiting for the API to return a response. The author noticed that the API had support for looking up multiple postcodes within a single request, which could help lower the time taken by the process. This was then the method implemented (**Appendix G.2.2 - Retrieving Geo-coordinates**). The process took less than 20 minutes to complete.

CQC Locations

The CQC website provided a CSV file that contained almost 50,000 locations, where each had 14 dimensions, however, only 5 of the attributes were needed. The data set contained other locations that would not be integrated into the system, such as nursing homes, but were still imported as they could be filtered out via queries at a later stage. A simple method was constructed (**Appendix G.3.1**) to only create a single instance on each iteration of the loop, since there was not a lot of data. The locations were imported within a few minutes, and once complete, a set of coordinates was added to each using a method (**Appendix G.3.1**) like that used in the schools import. By this time, the author had generalized the coordinate fetching functionality into a generic method (**Appendix G.3.2**) so that there was minimal code duplication. Upon further consideration, the author realised that ratings for each location may also be useful to display to end-users, as they provide an insight into the level of care available, allowing for a better judgement of quality relating to health-care services in the local vicinity. This data was available via another file on the CQC website, for which reason an additional import method had to be defined.

The second method was created to update existing CQC locations with outcomes of the most recent inspections. The file contained all CQC ratings that had ever been assigned, meaning that there were multiple ratings for each location at a given time. The system was not intended to show a history of ratings, thus only the most recent ratings were imported.

Demographics

This data was retrieved from the GOV.UK website and provided as a CSV file. The dataset was provided as unique rows for population per ethnicity/year/borough/age. Dating back to 2011, with estimations till 2050. Instead of modifying the data structure, the data was imported as is to save time. Not all yearly values were extracted from the file, only those for 2018 and 2019 since those were going to be displayed. (**Appendix G.5**)

Active Places

This data was retrieved from the Active Places Power website, available for download in both CSV and JSON formats. The authors preferred choice was the latter as it would involve less processing and was already structured, which would mean for a quicker import. The format of the JSON showed that each location had associations of type one-to-many, which had to be reflected as instance relationships when importing the data. All location objects had a property '*deleted*' to represent whether the location still existed. In the method definition (**Appendix G.4.1**), for situation where the value was '*true'* the data was not imported as it was outdated, thus would not be useful to the user. The object contained 39,761 entries, of which 39,235 were imported.

5.3.3 Filter backends

The next step of implementation was to define filter backends that would be used by the property searching API at a later stage. The purpose of these filters was to look at individual attributes of a queryset of instances and filter out those that did not meet certain conditions. A total of seven filter blocks had been defined, and for each there had to exist a filter backend. A filter was defined as either a class or a method. Each was tested whilst being developed, using the python console for debugging.

5.3.3.1 BasicPropertyFilter

This was implemented as a django filter class (**Appendix H.1**), to provide the functionality required by two of the options on the front-end (listing and property type filters), thus this filter would only look at two fields belonging to an instance: listing_status and property_type. Both fields were of type char and required an exact match look-up operation (iexact) to be performed. '*listing_type*' would receive and match against one of two values, being '*rent'* or '*sale*, where as the *'property_type'* field would take a list of values and an instance had to match against one.

5.3.3.2 RoomFilter

This functionality was implemented as a class (**Appendix H.1**) and was only to be used for one purpose: filtering for properties that featured a number of rooms, of each type, within a defined range. This would correspond with the optional functionality on the front-end. Originally, the filter was developed to also include the field '*num_floors*', however, the author soon came to the realisation that searching for properties by number of floors would not be a practical option as it would limit results dramatically. The definition for this was relatively short as most of the code was made abstract through the Django Filters library, using a RangeField.

5.3.3.3 PriceFIlter

This filter, much like the previous, was written to filter field values based on a range. Again, it was implemented as a class (**Appendix H.3**), however, this time some extra logic was written. Due to the way in which prices were being stored on different models depending on the listing type, to find a price, two different models had to be checked. For listing type '*sale*', the price was stored on the '*Property'* model, however, for listing type '*rent',* the price was stored in an association called '*rentalprice*' which contained different prices for monthly and weekly rental contracts.

5.3.2.4 AreaFilter

Although a class-based implementation was defined (**Appendix H.4**), only a single sub-filter existed. The purpose of this class was to receive a queryset of locations and the bounds of an area, and remove those that fell outside. The bounds were defined by a circle for which the radius and center-point would be provided by the end-user. The approach was simple, as the logic was based on distance from a set of coordinates. The author attempted to implement a distance finding method in Python, however, upon testing realised that the solution was compute intensive and relatively slow. An alternative method, discovered on StackOverflow, relied heavily on the database performing the trigonometry operations, and once tested, it was found as the quickest approach.

5.3.2.5 filter_properties_for_schools

This filter was implemented as a method (**Appendix H.5**), with the purpose of returning properties that fell within a certain distance of schools that met specific conditions. This, much like the previous method, involved computing distances, but here it was implemented much differently. The major difference here was that a queryset of schools had to be filtered first, after which distances had to be computed with each of the properties. Having tested the database approach to computing distances with this application, this solution worked, however, would not meet **R16** as the method was not quick enough, taking over 10 seconds. The author then went through the process of finding a more optimal method and found a reasonable solution on StackOverflow involving a KDTree. This example was adapted to check within a distance radius as opposed to returning the k-nearest results, as seen in the example. The radius was defined as 0.5 KM and If a property had at least one school within the area then it would be added to the list of properties to return. After testing the performance of this solution, the author found that it was exponentially faster, taking less than one second to compute.

5.3.2.6 filter_properties_by_commute

This filter was implemented as a method (**Appendix H.6**), re-using some of the techniques discovered during the development of the previous filter. The purpose of this was to take a queryset of properties and return those that met time restrictions from user-defined points of interest. A key requirement of this functionality was the HereMaps API, is it would be used to retrieve am estimated commute time between two points. Due to time constraints, the author chose to implement this option only for public transportation journeys. Requesting distances from all locations to each point of interest was not feasible as it would quickly consume the allowance of API calls, for which reason a selected number of locations were checked. Using the KDTree approach learned earlier, a list of 100 of the nearest properties was created; these properties were those nearest to all points of interest. This, therefore, would return more results meeting the required time constraints, as travel time decreases with distance. One of the initial issues that the author faced was that performing 100 API calls was taking too long, which failed to meet requirement 14. Having searched for a solution, the author found a snippet of code on StackOverflow which aimed to solve this problem; this solution was based on the idea of performing API calls asynchronously. Once implemented, it was very effective and cut filtering times down to less than two seconds, as opposed to the almost 20 seconds before. With this improvement, the author increased the number of API calls to 300, allowing for more results to be found.

5.3.4 Defining API end-points and methods

The purpose of defining API end-points was to get data to the front-end so that it could be displayed. Through the use of APIs, data would be fetched and displayed only when needed, which in theory would cut down initial loading times when the results page first begins to render. APIs were defined in '*api.py*' under each app directory. Each API was defined as a view, a python method which takes a request object as a parameter and returns a '*Response*' object. This response, in respect to this project, can be in one of two forms: JSON, which would typically encode data that is returned by an API, or HTML, that would be used to render a page. Found at **Appendix K** is a list showing each of the end-points defined.

5.3.4.1 API definitions

Numerous endpoints needed to be defined to serve data associated with the categories of information on the results page. Five of the API end-points were based on the idea of needing to return data for a set of unique coordinates, of which four were required to perform a KDTree search looking for nearby locations. To prevent code duplication, this operation was transformed into a generic method called 'get_closest_locations', and can be found in **Appendix I.2.1.** A main endpoint was developed to perform the task of filtering based on the parameters defined using the filter UI. This would simply return the listing id and exact location of each property, and nothing more, to improve performance. A second API was developed to complement this, by returning more details for an individual property. This would be called when the user selects a property on the map. This was much faster than sending all the data during the filtering stage. A further two APIs were defined (**Appendix I.1.3**) to return values for the price histogram that would be displayed during the price range selection process. The implementation was relatively simple, making use of a library called NumPy to perform the necessary calculations. Finally, another API was created to allow for auto-complete responses to be displayed during the search process of the commute filter. This API was not compulsory to implement, however, was done so to prevent over use of the limited API calls that the HereMaps service was providing, by using caching.

Keeping track of each API was a difficult task as there were various definitions, each of which with unique parameters, thus documentation was produced to aid the integration process with the front-end. This can be found in **Appendix K.** Each definition listed the URL, purpose, reference, and input/outputs for convenience.

5.3.5 Performance optimisations

Splitting the data into multiple serializers and only accessing data when needed allowed for the map to display results in less than 3 seconds, as opposed to the 20 seconds taken for the same query prior to modifications. This was due to there being the need to transmit less data, thus taking less time.

Performing API calls to the HereMaps API at a rate of 50 calls at a time allowed for much quicker displaying and processing **R16**, so much so that the author increased the number of properties to perform route checks for to 300. This created a database related issue as too many connections were being made simultaneously. This problem was addressed by increasing the default limit of 100 to 600.

5.4 Testing

5.4.1 Functional testing

To test the functionality of the front-end behavior, a series of functional tests were produced. These tests were aimed at testing the complex logic behind the filtering functionality. A test document was produced, showing the elements tested and their outcomes. Where a test failed, the issues were addressed, and the tests were reattempted. Many of the cases were to ensure that default logic was being applied where applicable. For example, tests were written to ensure that certain values were pre-selected upon loading of a filter. This can be found documented at **Appendix J.1**

5.4.2 Unit testing

Two-unit tests were written in the form of a test class. The functionality being tested was the method which returned the 'n' closest nodes to a set of geo-coordinates. This was a crucial method was it was referenced in many locations in various apps. This method was tested with data consisting of six nodes, with a fixed geo-location each time. The first test would ensure that the method was returning the correct distance each time. The second would ensure that the method would return the correct distances in ascending order, for n of the nearest points. Other methods that would have been ideal to would have been API responses but was not attempted due to lack of time. The definition can be found in **Appendix J.2**

5.5 Deployment

The final project database had a size of approximately 180 megabytes when dumped, thus required a database with the appropriate capacity and with enough headroom for additionally generated data during the survey stage. The project code was implemented in JavaScript and Python, and so required appropriate compilers and execution packages to run on the back-end. Having looked at a few hosting services, such as Heroku and AWS, their free tiers did not provide enough resources to host the application. The requirement of the database was not met on either, as Heroku only provided ten thousand database rows, not enough to cover the 200,00+ rows of the application. AWS did not provide any free database options, however did issue credit, which would not have been enough to keep the system online for the duration of the project. The only viable solution was to self-host the server.

The web application was hosted on an Ubuntu based server with a domain name (http://project.addilafzal.com) that users would be able to access over the internet. This was achieved having followed a tutorial online at DigitalOcean. Once the environment was configured, the only necessary changes were related to the source code. Each time the code was updated, it had to be re-deployed on the server. This was a painful process, which solutions such as Heroku and AWS would have simplified through their use of continuous deployment tools. A deployment script was written in bash shell to automate this process (**Appendix R**). To accommodate the need to allow multiple user to use the application, the server It was configured to use all 4 cores of the server machine.

5.6 Survey

The survey was undertaken by 6 participants. The full set of results of the survey can be found in **Appendix O**. Prior to starting the investigation, each participant was made aware that the property data used in the system was outdated and was only in place to help evaluate the usefulness of filters and categorical information, as well as functionality.

5.6.1 Background information

The results of this sub-section reviled that 5 out of the 6 participants had been looking for a home recently, with 4 participants having searched within the last year. The other two selected 'More than a year ago'. This suggested that most people were familiar with pre-existing methods of finding a home. The one exception that selected 'No' for the first question also selected 'Never' for the second, suggesting that they have had the least experience in the housing market.

5.6.2 Filters

The feedback given for the filters was mostly positive, with two neutral responses, both of which were given for the school filter. This was expected since the audience selected were undergraduate students, due to convenience sampling, and did not have the need to test the functionality. The most positively rated filters were the area and price filters. This may have been due to the fact the they were implemented significantly differently from existing solutions, with a more intuitive design. The area filter was expressed as an interactive map and opposed to a text field, and the price filter was implemented as a slider instead of drop-down options. When asked about improvements that could be made to the filters in general, two users asked for more information to be displayed, and another requested that the design for the commute filter be updated. This showed that there were missing details that that individuals wanted to see and that the representation of the functionality still requires some improvement. All participants agreed that the compulsory filters were relevant. When asked about the need for additional filters, half the participants had suggestions, with answers such as 'Supermarket filter' and 'Council tax'. Proximity from supermarket was a factor that was considered earlier, however, was not feasible to implement due to lack of data. Council tax was found during the research stage but was not chosen for implementation due to the range of information that was already being displayed.

5.6.3 Map

The feedback given for the information shown on the results page again was mostly positive. All participants strongly agreed with the usefulness of the ‘Summary’ tab. This may have been since it contained the information that they were looking for or found interesting. ‘Demographics’ and ‘Restaurants’ both received a single neutral response. Looking at the overall ratings for each showed that they had the lowest values amongst all other tabs, however, significantly lower for demographics. This was expected given that not a lot of information was being shown to represent the local area, as only a single statistic was being shown. When asked whether the categories of information could be improved, everyone replied no, except for one. The feedback given was to do with the design for the ‘commute’ tab button, as it would only be enabled when the commute filter was selected on the previous page. Arguably, this was not clear as it was simply greyed out. Given that this was the second feedback given in relation to the commute functionality, it showed that many had interest in this area. Surprisingly, the demographics functionality did not receive criticism, despite it being the least useful, illustrating that maybe demographics was not a factor that many people considered. When asked whether the information shown on the map could be improved, 50% answered yes. Two participants requested to see functionality such as an option to save a property and another to be able to go straight to the listing page on Zoopla. The third comment was regarding transportation links. The participant wanted to see transportation options such as “nearest bus stop” and “nearest train station”. Given that this information was not added intentionally as it was no longer seen necessary during the research stage, the results showed that there was still a need. The commute functionality may have not cover certain aspects of the commute process. When asked whether participants would use the system again upon their concerns being addressed, all agreed that they would. This indicated that there was a potential for such a solution in the market that is not currently available. When asked for opinions of the solution, half the people chose to provide a comment. The comments overall were very positive, with many people praising the UI design and the ease of use as being aspects which stuck out.

Chapter 6: Conclusion and Discussion

6.1 Assessing objects

The primary objective of this project was “*to simplify the home-finding process*”. To make this objective easier to understand and handle, it was split into a list of four sub-objectives, as defined in section 1.2. Each of these had to be attempted effectively for the end-solution to be successful and delivered within the constrained time. The first objective addressed was to examine existing publications of home finders. This objective was met, as two pre-existing home finder publications were reviewed and some of their findings were applied. For example, upon reviewing the original home finder, the project incorporated the use of animations through the form of loading icons. This was done to engage the user as it was something that subjects of the study rated positively. The idea to display route paths on the map, between properties and their key points of interest was obtained from having viewed HomeFinder Revisited. The extent till which the publications were used was very little. Even though the goal behind the publications was very similar, one was published many years ago and the other focused solely on reachability.

The second objective was “To analyse existing property finders”, with the aim of finding qualities that could be reused. This objective was attempted effectively with two solutions being reviewed. Both of which were practical examples of home-finders that were popular online. The flow of the search process used in both systems was examined and used to influence the final solution, with the search and display process being separated. The approach to contain different categories of data was heavily influenced from the design present on both examples. With the implementation of the filters, this project attempted to request as little free-text from the user as possible, which was an idea that was implemented based on the findings from having analysed the inputs interfaces of the systems.

The third objective was “*to find factors that are considered during the home-finding process, and to associate them with publicly available data*”, which was a task that was attempted mostly effectively. The task yielded many factors, however not all were chosen for implementation due to either data not being available in some cases or not being feasible in the constrained time. Out of the 10 factors that were selected, there was only one for which data could not be found, being grocery stores. The remaining factors were then identified with uses, as either a filter or information tab, or both. This approach was successful, as it simplified the requirements establishment stage, identifying how exactly the data could be used.

The first part of the fourth objective was to develop the functionality of the proposed system by using the appropriate technologies to achieve an intuitive design and structured code. This was attempted effectively given the complexity of the application. The system made use of ‘apps’ during the implementation stage, which helped contain functionality in segregated packages allowing for a structure approach to development. The front-end was designed intuitively by identifying the purpose of each functionality and how it could be best represented, as illustrated with the development of the filters. Much of the thought process was to improve upon the interactivity of existing implementations, by engaging the user with visuals and limited use of free-text fields. The testing aspect of the final objective was achieved, however, not till a reasonable extent. Given that the system incorporated a lot of functionality, not all of it was tested before the surveys were conducted, thus some bugs were encountered.

6.2 Issues and improvements

6.2.1 Property data

Given the issues faced with the API earlier on, the project was not able to be of practical use as initially intended. The solution was constrained by the availability of property data, which was not easy to retrieve. A better solution to getting property data would have been to scape listings directly from a website that allowed web-crawlers, as there are a few that exist, however they lack in data quantity. Even then, the data provider could change their terms of use preventing such a method from being implemented, as has been the case with RightMove in the past.

6.2.2 Quality of functionality

The data shown for the demographics tab was very limited, due to the lack of time spent on developing the category. In fact, there were a few situations in which functionality could have been implemented better but wasn’t, and the re-occurring answer to this was ‘lack of time’. For this reason, the project could have focused more on quality of information and functionality, rather than quantity. The initial scope of the project was very large too, meaning research had to be performed into several different aspects. The scope needed to target a specific group of people rather than to target as many as possible.

6.2.3 Comparing properties

Instead of allowing the user to find a property using the dataset of the system, a better approach may have been to allow the user to provide the approximate location of a few properties and use the system to fetch and compare their qualities. This would eliminate the need to import property data but would still require the user to access an external site.

6.3 Conclusion

From having embarked on this project, it is evident that there is a need for such a solution that simplifies the home finding process. The results of the survey showed that all six participants would use the solution if their concerns were addressed, which upon review, were entirely feasible. The only real hurdle in making such a solution a reality is the availability of property data, which as discovered, is not easy to obtain since many online property listers do not want to share. It will not be too long before RightMove or Zoopla attempt to do something similar, given that they have already started making use of open data. With this thought in mind, the author is looking forward to seeing where the home finder is headed.

To conclude an end to this report, the author believes that this project offered a great opportunity to develop many new skills, both personal and professional, and hopes that the findings of this project were valuable to others.

Glossary

Django	A python programming framework, based on the MVC architecture.
React	A JavaScript programming framework.
JSON	Stands for JavaScript Object Notation and is used as a method of encoding data for transmission.
API	Application Program Interface: An end-point from which data is requested.
Viewport	Portion of a browser window dedicated towards rendering a web-page.
Heroku	A web application hosting provider.
AWS	Amazon web services is a web application hosting provider.
Nullable	A field is considered nullable if it does not require a value.
Instance	An object derived from a class.
Filter back-end	A class which is initiated with a queryset of instances and a request object which contains attributes. The attributes are then used to perform filters on the query, and return a sub-set meeting the filter criteria.
Asynchronous	To perform operations in parallel.
POI	Point of Interest
Web-crawler	Software application designed to systematically browse a website.

References

Dawson, C. (2011). Projects in Computing and Information Systems. Old Tappan: Pearson Education UK.

HomeOwners Alliance. (2018). How To Choose A New Area To Live - HomeOwners Alliance. [online] Available at: https://hoa.org.uk/advice/guides-for-homeowners/i-am-buying/how-do-i-choose-a-new-area-to-live-in/ [Accessed 17 Mar. 2019].

Garber, S. (2019). Finding the best places to live. [online] Which? Money. Available at: https://www.which.co.uk/money/mortgages-and-property/first-time-buyers/buying-a-home/finding-the-best-places-to-live-akz9t5h0svcv [Accessed 17 Mar. 2019].

Wikipediaorg. 2019. London postal district. [Online]. [17 March 2019]. Available from: https://en.wikipedia.org/wiki/London_postal_district

Postcodesio. 2019. Postcode & Geolocation API for the UK. [Online]. [18 March 2019]. Available from: http://api.postcodes.io/

Data.gov.uk. (2019). Find open data - data.gov.uk. [online] Available at: https://data.gov.uk/ [Accessed 18 March. 2019].

Doogalcouk. 2019. Doogalcouk. [Online]. [13 March 2019]. Available from: https://www.doogal.co.uk/london_postcodes.php

Django, R. and Pla, J. (2019). Returning nearby locations in Django. [online] Stack Overflow. Available at: https://stackoverflow.com/questions/25623829/returning-nearby-locations-in-django?lq=1 [Accessed 6 Mar. 2019].

What is the fastest way to send 100, 0., Kálmán, T., Thompson, G. and Singh, A. (2019). What is the fastest way to send 100,000 HTTP requests in Python?. [online] Stack Overflow. Available at: https://stackoverflow.com/questions/2632520/what-is-the-fastest-way-to-send-100-000-http-requests-in-python [Accessed 1 Apr. 2019].

Moneycrashers.com. (2019). Where Should I Live? 14 Factors When Deciding the Best Place to Live. [online] Available at: https://www.moneycrashers.com/where-should-i-live-decide-best-places/ [Accessed 1 Apr. 2019].

kdTree, n. and Punnen, A. (2019). nearest neighbour search kdTree. [online] Stack Overflow. Available at: https://stackoverflow.com/questions/48126771/nearest-neighbour-search-kdtree [Accessed 1 Apr. 2019].

Docs.djangoproject.com. (2019). Django documentation | Django documentation | Django. [online] Available at: https://docs.djangoproject.com/en/2.2/ [Accessed 1 Apr. 2019].

Reactjs.org. (2019). Components and Props – React. [online] Available at: https://reactjs.org/docs/components-and-props.html [Accessed 2 Apr. 2019].

React.semantic-ui.com. (2019). Introduction - Semantic UI React. [online] Available at: https://react.semantic-ui.com/ [Accessed 2 Apr. 2019].

GitHub. (2019). Build software better, together. [online] Available at: https://github.com/ [Accessed 2 Apr. 2019].

Docs.djangoproject.com. (2019). Performance and optimization | Django documentation | Django. [online] Available at: https://docs.djangoproject.com/en/2.1/topics/performance/ [Accessed 2 Apr. 2019].

Zoopla.co.uk. (2019). *Contact Zoopla - Zoopla*. [online] Available at: https://www.zoopla.co.uk/contact/ [Accessed 2 Apr. 2019].

Developer.zoopla.co.uk. (2019). Zoopla Property API - Welcome to the Zoopla Developer Network. [online] Available at: https://developer.zoopla.co.uk/ [Accessed 2 Apr. 2019].

Rightmove.co.uk. (2019). Rightmove.co.uk. [online] Available at: https://www.rightmove.co.uk/ [Accessed 2 Apr. 2019].

Leafletjs.com. (2019). Documentation - Leaflet - a JavaScript library for interactive maps. [online] Available at: https://leafletjs.com/reference-1.4.0.html [Accessed 2 Apr. 2019].

Weng, D., Zhu, H., Bao, J., Zheng, Y. and Wu, Y., 2018, April. Homefinder revisited: finding ideal homes with reachability-centric multi-criteria decision making. In Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (p. 247). ACM.

Williamson, C. and Shneiderman, B., 1992, June. The Dynamic HomeFinder: Evaluating dynamic queries in a real-estate information exploration system. In Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval (pp. 338-346). ACM.

Medium. (2019). How not to run an API (looking at you, Zoopla). [online] Available at: https://medium.com/@mariomenti/how-not-to-run-an-api-looking-at-you-zoopla-bda247e27d15 [Accessed 4 Apr. 2019].

Rightmove.co.uk. (2019). Terms of use. [online] Available at: https://www.rightmove.co.uk/this-site/terms-of-use.html [Accessed 15 Apr. 2019].

Zoopla.co.uk. (2019). Zoopla Terms of Use - Zoopla. [online] Available at: https://www.zoopla.co.uk/terms/ [Accessed 15 Apr. 2019].

Localeyes. (2019). *Comparing Google Maps and HERE Maps Pricing | Localeyes*. [online] Available at: https://local-eyes.nl/page/comparing-google-maps-and-here-maps-pricing [Accessed 16 Apr. 2019].

Anon, (2019). [online] Available at: https://www.quora.com/How-can-I-host-my-Django-website [Accessed 16 Apr. 2019].

Soldo, M. (2019). Heroku Postgres Basic Plan and Row Limits. [online] Blog.heroku.com. Available at: https://blog.heroku.com/heroku_postgres_basic_plan_and_row_limits [Accessed 16 Apr. 2019].

Digitalocean.com. (2019). How To Set Up Django with Postgres, Nginx, and Gunicorn on Ubuntu 16.04 | DigitalOcean. [online] Available at: https://www.digitalocean.com/community/tutorials/how-to-set-up-django-with-postgres-nginx-and-gunicorn-on-ubuntu-16-04 [Accessed 16 Apr. 2019].

HERE Developer. (2019). Build apps with HERE Maps API and SDK Platform Access - HERE Developer. [online] Available at: https://developer.here.com/ [Accessed 16 Apr. 2019].

Django-filter.readthedocs.io. (2019). django-filter — django-filter 2.1.0 documentation. [online] Available at: https://django-filter.readthedocs.io/en/master/ [Accessed 16 Apr. 2019].

Appendix A – Project definition document

OPTIMAL LOCATION FINDER

Project by Addil Afzal

Student

Addil Afzal
07406 15059
addil.afzal@city.ac.uk

Supervisor

Jason Dykes
j.dykes@city.ac.uk

Project was proposed by Addil Afzal and does not have any propriety interests.

Word count: 1,489

City, University of London

Contents

Problem to be solved

When searching for a place to live, we are given basic filters to search by. The most common filters are: area, price, number of bed rooms and property type, as is the case with both Rightmove and Zoopla, two of the main websites used across the UK that have been around for several years.

However, in many cases, there are other factors that are also considered: distance from work/school, crime rates, local facilities, etc... With the availability of different types of data being accessible to the public, making such a solution to incorporate this data has become increasingly feasible. With optimal location finder (OLF), the aim will be to combine the functionality offered by existing property finding services and add additional filters and advanced searching parameters for a more user tailored result which I believe will introduce many new opportunities for many people of different backgrounds.

The first instance where advanced filtering functionality was integrated with a visual home finder was with the 'The Dynamic HomeFinder' developed by Christopher Williamson and Ben Shneiderman at the University of Maryland. This solution featured distance-based filtering from points of interest such as work, and filtering based on the services that a home should offer.

The reason why I have chosen to undertake this project is because I believe that I can produce an effective solution having worked with data visualisation in the past (e.g., on work placement) where I have worked with technologies to visualise data through the means of charts and interactive heat maps, having pre-processed the data in advance.

Project objectives

This project shall:

- display an interactive map with filtered locations pin-pointed.
- allow for the user to switch between map and list/table view, with sorting functionality.
- allow locations to be filtered by standard options such as area, min/max price, number of rooms, listing type (sale or rent).
- allow locations to be filtered by distance/commute time to a location such as work or university.
- allow filtering by distance to the nearest nursery, primary or high school. This option will be extended to allow for Ofsted ratings to be taken into consideration, as well as religion, gender and whether the school is selective.

- allow for locations to be filtered by distance from religious establishments such as mosques, churches and other establishments.
- display information for the closest healthcare facilities available at each location (dentist, optometrist, GP, hospital) and their CQC ratings.
- display average broadband speeds at the location.
- display crime rates in the area, broken down by frequency for each category of crime.

Project beneficiaries

This project will have many beneficiaries due to the vast number of filter options that will be offered. One group of people to benefit from my solution will be students, as the system will help find suitable locations for them to rent that are within their budget range. They will be able to insert the address of their place of study and be presented with the option to select price range and min-max commute times as well as potential travel costs via different means of transportation.

Another group of people that this solution will be of use to is parents. OLF will allow parents to filter for homes by the Ofsted ratings (academic standards) of school located nearby and will display the level of different types of crime that have occurred in the local area. There will also be the functionality to find selective (grammar), gender specific or religion-based schools. This will allow parents to determine whether the area surrounding each location is suitable for their children.

Finally, another group of people to whom my solution will be of great use are those looking to live at a location where they will fit in amongst others with similar religious beliefs or of similar wealth and age.

In general, the beneficiary of this system can be anyone looking to 'optimise' their daily routine by moving to a location that provides them with more of the amenities that they require. Further beneficiaries will arise during the research stage.

Work Plan

When attempting this project, I will follow a waterfall/traditional methodology where I am not necessarily finishing each stage completely before moving to the next; there will be over-lapping stages as there will be non-dependant tasks which can be started early.

Requirements analysis and research

10/10/18 - 14/11/18

The first stage of my project will involve going over the defined requirements and expanding them into small tasks that I can then prioritise. Each task will be numbered based on its order of implementation and then given a priority.

As the requirements have been based entirely on what I think will be useful, I also plan to conduct a survey to further understand the different aspects that people consider when looking for a location to live. I will assess the results of the survey and then decide whether there are any ideas that are feasible to implement.

Finally, I will check for recent literature on the subject to get further ideas.

Design

15/11/18 – 16/12/18

The system will then be designed, both graphically (GUI) and in terms of code structure. When developing graphical designs, I will begin coding dummy interfaces that can later have functional code added to them. To get feedback for my graphical data representations, I will see my supervisor Jason Dykes, the Professor of Visualization here at City and I will also consult with the book that they have suggested by Isabel Meirelles.

I will also design the architecture of the backend using a design class diagram showing the API end points and how the browser/client will communicate with them, as well as the relationships between the data classes and controllers.

A use case diagram will be developed to illustrate how the front-end and back-end will combine to fulfil user functionality.

Implementation

17/12/18 – 16/02/19

At this stage of the project, I will already have a lot of the front-end code written.

Before developing APIs, I will create scripts to import data from the various files that I have managed to source, after they have been cleaned (list can be found in the references section).

The next step will be to develop functions which perform computations on data before they can be sent to the front-end. Once that is done I can begin developing method to serve data using REST APIs which will return data in JSON form, ready to decode in java script by the front-end.

The front-end will then be updated to make calls to the newly declared APIs and perform functional behaviour.

Whilst developing my back-end code I will follow a test-driven development processes to ensure that my code produces the required results when changed at a later stage.

Testing and Validation

11/02/19 – 03/03/19

For this project, the types of tests that I will perform are functional tests, user tests and unit tests.

Functional testing will be performed to ensure that each of the requirements has been met. User testing will be used to establish whether the system has been designed in a way that is easy for the end-user to operate and an opportunity to get feedback from users. I will provide a digital survey for each of the candidates to fill. Unit tests will ensure that my code is working as intended and will be written throughout the implementation stage.

When performing functional tests, I will take screenshots and store them systematically so that I can easily reference them when needed for the documentation stage.

Project report

18/02/19 – 17/04/19

I will start writing the project document shortly after starting the testing stage.

Risk Analysis

[Medium] OLF will be a data heavy application, gathering data from several different sources, much of which will need to be kept up to date for the information produced to be of any value to the end user.

When starting the implementation, I will write data import methods for each of the data files, therefore all I must do to update my database is download the most recent copy of each set from the source and perform the import.

[High] One of the possible risks to my project is that Zoopla, the housing information provider, could withdraw their API at any time as it is a service that they are offering for free, meaning that I would need to get my data from elsewhere. This would be an issue as there are not many online housing sites which provide data for free via an API. I could scrape data from sites using a self-built tool however, it requires time to develop and will not always produce accurate results if the design of the website was to change.

[Medium] Another potential risk to the project is that my code may not perform optimally when faced with on-demand requests, effecting usability. The system will contain data from various sources, I will need to ensure that I am querying the database effectively, making use of local caching where possible.

References

Williamson, C. and Shneiderman, B. (1992). The Dynamic HomeFinder: Evaluating Dynamic Queries in a Real-Estate Information Exploration System. University of Maryland.

Dawson, C. (2015). *Projects in Computing and Information Systems*. 3rd ed. Pearson Education, pp.140-142.

Rightmove.co.uk. (2018). Rightmove.co.uk. [online] Available at: https://www.rightmove.co.uk/ [Accessed 23 Oct. 2018].

Zoopla.co.uk. (2018). Zoopla > Search Property to Buy, Rent, House Prices, Estate Agents. [online] Available at: https://www.zoopla.co.uk/ [Accessed 23 Oct. 2018].

Find and compare schools in England. (2018). All schools and colleges in England - GOV.UK - Find and compare schools in England. [online] Available at: https://www.compare-school-performance.service.gov.uk/schools-by-type?step=default&table=schools&region=all-england&for=secondary&orderby=ks4.0.P8MEA&orderdir=asc&datatype=integer&sortpolicy=inversepolicy [Accessed 25 Oct. 2018].

GOV.UK. (2018). School inspections and outcomes: management information. [online] Available at: https://www.gov.uk/government/statistical-data-sets/monthly-management-information-ofsteds-school-inspections-outcomes [Accessed 25 Oct. 2018].

Data.police.uk. (2018). Data downloads | data.police.uk. [online] Available at: https://data.police.uk/data/ [Accessed 25 Oct. 2018].

Cqc.org.uk. (2018). CQC care directory - csv format | Care Quality Commission. [online] Available at: https://www.cqc.org.uk/file/179940 [Accessed 26 Oct. 2018].

Naqshbandi, M. (2018). UK Mosque/Masjid Directory - via Google Maps, your mobile or your navigator (POI). [online] Mosques.muslimsinbritain.org. Available at: http://mosques.muslimsinbritain.org/gps.php [Accessed 27 Oct. 2018].

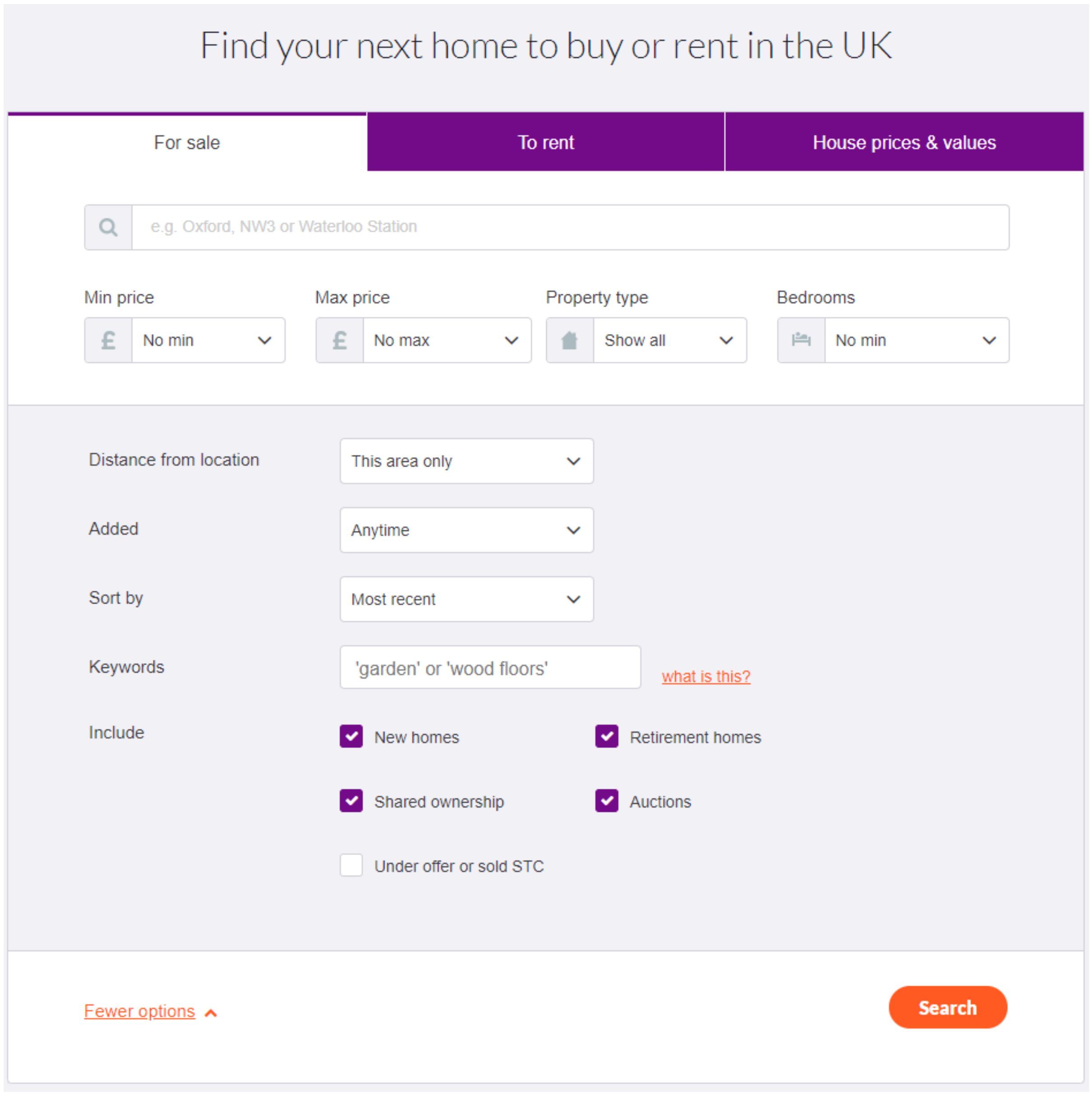
Genuki.org.uk. (2018). GENUKI: Maintaining the Church Records Files, Non-geographic. [online] Available at: https://www.genuki.org.uk/big/churchdb/ChurchRecs [Accessed 27 Oct. 2018].

Meirelles, I. (2006). *Design for Information : An Introduction to the Histories, Theories, and Best Practices Behind Effective Information Visualizations*. Quayside Publishing Group.

Appendix B – Existing system interfaces

Appendix B.1 – Showing Zoopla's filter options

https://www.zoopla.co.uk/



Find your next home to buy or rent in the UK

For sale | To rent | House prices & values

e.g. Oxford, NW3 or Waterloo Station

Min price: No min

Max price: No max

Property type: Show all

Bedrooms: No min

Distance from location: This area only

Added: Anytime

Sort by: Most recent

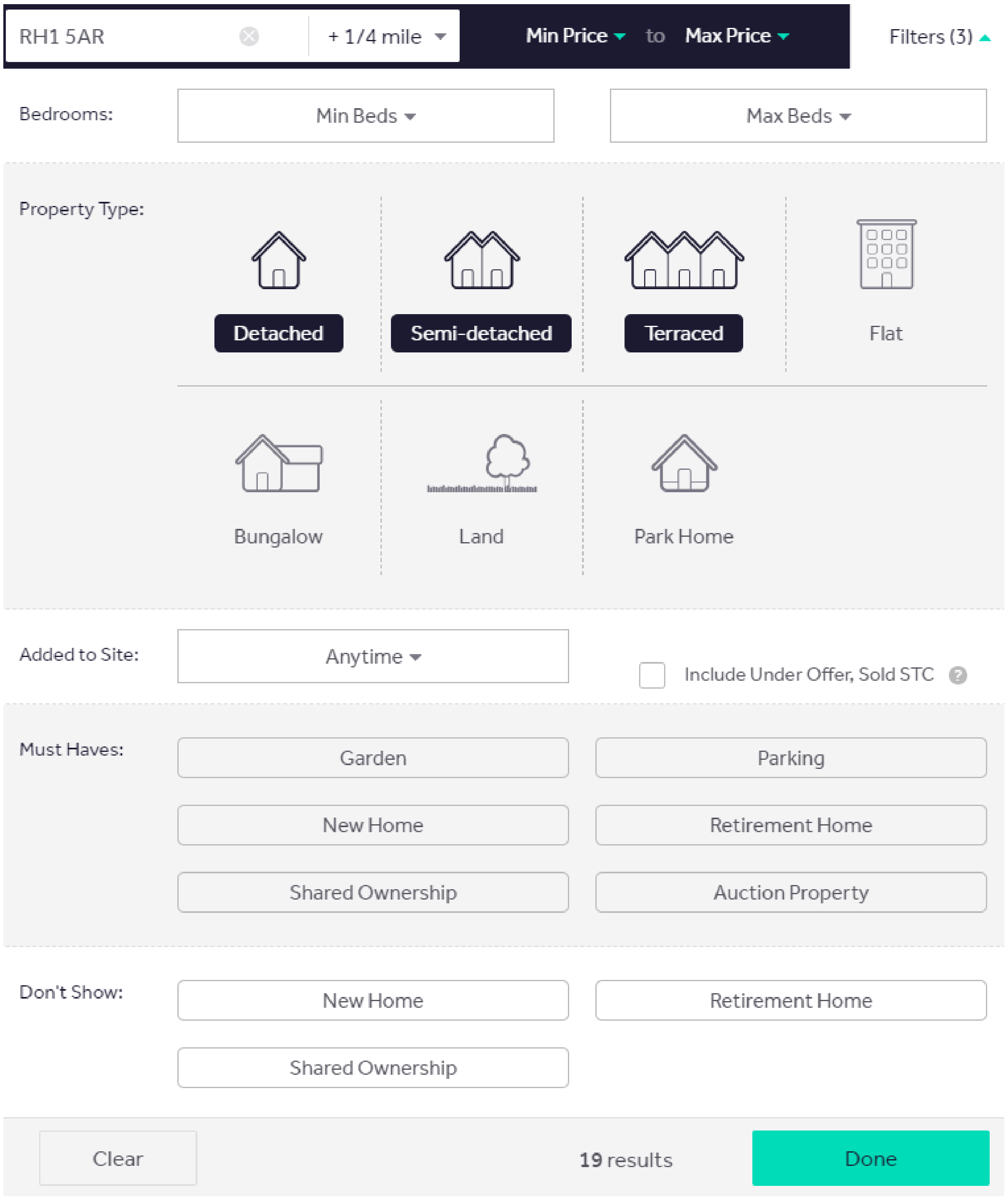
Keywords: 'garden' or 'wood floors' what is this?

Include:
- ✓ New homes
- ✓ Retirement homes
- ✓ Shared ownership
- ✓ Auctions
- ☐ Under offer or sold STC

Fewer options

Search

Appendix B.2 – Showing Rightmove's filtering capabilities



RH1 5AR | + 1/4 mile | Min Price to Max Price | Filters (3)

Bedrooms: Min Beds | Max Beds

Property Type: [✓ Detached] [✓ Semi-detached] [✓ Terraced] Flat Bungalow Land Park Home

Added to Site: Anytime

☐ Include Under Offer, Sold STC

Must Haves: Garden | Parking | New Home | Retirement Home | Shared Ownership | Auction Property

Don't Show: New Home | Retirement Home | Shared Ownership

Clear | 19 results | Done

Ethics Review Form: BSc, MSci, MSc and MA Projects

Computer Science Research Ethics Committee (CSREC)

Undergraduate and postgraduate students undertaking their final project in the Department of Computer Science are required to consider the ethics of their project work and to ensure that it complies with research ethics guidelines. In some cases, a project will need approval from an ethics committee before it can proceed. Usually, but not always, this will be because the student is involving other people ("participants") in the project.

In order to ensure that appropriate consideration is given to ethical issues, all students must complete this form and attach it to their project proposal document. There are two parts:

Part A: Ethics Checklist. All students must complete this part. The checklist identifies whether the project requires ethical approval and, if so, where to apply for approval.

Part B: Ethics Proportionate Review Form. Students who have answered "no" to questions 1 – 18 and "yes" to question 19 in the ethics checklist must complete this part. The project supervisor has delegated authority to provide approval in this case. The approval may be provisional: the student may need to seek additional approval from the supervisor as the project progresses.

A.1 If your answer to any of the following questions (1 – 3) is YES, you must apply to an appropriate external ethics committee for approval.		*Delete as appropriate*
1.	Does your project require approval from the National Research Ethics Service (NRES)? For example, because you are recruiting current NHS patients or staff? If you are unsure, please check at http://www.hra.nhs.uk/research-community/before-you-apply/determine-which-review-body-approvals-are-required/.	**No**
2.	Does your project involve participants who are covered by the Mental Capacity Act? If so, you will need approval from an external ethics committee such as NRES or the Social Care Research Ethics Committee http://www.scie.org.uk/research/ethics-committee/.	**No**
3.	Does your project involve participants who are currently under the auspices of the Criminal Justice System? For example, but not limited to, people on remand, prisoners and those on probation? If so, you will need approval from the ethics approval system of the National Offender Management Service.	**No**

A.2 If your answer to any of the following questions (4 – 11) is YES, you must apply to the City University Senate Research Ethics Committee (SREC) for approval (unless you are applying to an external ethics committee).		*Delete as appropriate*
4.	Does your project involve participants who are unable to give informed consent? For example, but not limited to, people who may have a degree of learning disability or mental health problem, that means they are unable to make an informed decision on their own behalf?	**No**

5.	Is there a risk that your project might lead to disclosures from participants concerning their involvement in illegal activities?	**No**
6.	Is there a risk that obscene and or illegal material may need to be accessed for your project (including online content and other material)?	**No**
7.	Does your project involve participants disclosing information about sensitive subjects? For example, but not limited to, health status, sexual behaviour, political behaviour, domestic violence.	**No**
8.	Does your project involve you travelling to another country outside of the UK, where the Foreign & Commonwealth Office has issued a travel warning? (See http://www.fco.gov.uk/en/)	**No**
9.	Does your project involve physically invasive or intrusive procedures? For example, these may include, but are not limited to, electrical stimulation, heat, cold or bruising.	**No**
10.	Does your project involve animals?	**No**
11.	Does your project involve the administration of drugs, placebos or other substances to study participants?	**No**

A.3 If your answer to any of the following questions (12 – 18) is YES, you must submit a full application to the Computer Science Research Ethics Committee (CSREC) for approval (unless you are applying to an external ethics committee or the Senate Research Ethics Committee). Your application may be referred to the Senate Research Ethics Committee.		*Delete as appropriate*
12.	Does your project involve participants who are under the age of 18?	**No**
13.	Does your project involve adults who are vulnerable because of their social, psychological or medical circumstances (vulnerable adults)? This includes adults with cognitive and / or learning disabilities, adults with physical disabilities and older people.	**No**
14.	Does your project involve participants who are recruited because they are staff or students of City University London? For example, students studying on a specific course or module. (If yes, approval is also required from the Head of Department or Programme Director.)	**No**
15.	Does your project involve intentional deception of participants?	**No**
16.	Does your project involve participants taking part without their informed consent?	**No**
17.	Does your project pose a risk to participants or other individuals greater than that in normal working life?	**No**

18.	Does your project pose a risk to you, the researcher, greater than that in normal working life?	**No**

A.4 If your answer to the following question (19) is YES and your answer to all questions 1 – 18 is NO, you must complete part B of this form.		
19.	Does your project involve human participants or their identifiable personal data? For example, as interviewees, respondents to a survey or participants in testing.	**Yes**

Part B: Ethics Proportionate Review Form

If you answered YES to question 19 and NO to all questions 1 – 18, you may use this part of the form to submit an application for a proportionate ethics review of your project. Your project supervisor has delegated authority to review and approve this application.

However, if you cannot provide all the required attachments (see B.3) with your project proposal (e.g. because you have not yet written the consent forms, interview schedules etc), the approval from your supervisor will be provisional. You **must** submit the missing items to your supervisor for approval prior to commencing these parts of your project. Failure to do so may result in you failing the project module.

There may also be circumstances in which your supervisor will ask you to submit a full ethics application to the CSREC, e.g. if your supervisor feels unable to approve your application or if you need an approval letter from the CSREC for an external organisation.

B.1 The following questions (20 – 24) must be answered fully.		*Delete as appropriate*
20.	Will you ensure that participants taking part in your project are fully informed about the purpose of the research?	**Yes**
21.	Will you ensure that participants taking part in your project are fully informed about the procedures affecting them or affecting any information collected about them, including information about how the data will be used, to whom it will be disclosed, and how long it will be kept?	**Yes**
22.	When people agree to participate in your project, will it be made clear to them that they may withdraw (i.e. not participate) at any time without any penalty?	**Yes**
23.	Will consent be obtained from the participants in your project? Consent from participants will be necessary if you plan to involve them in your project or if you plan to use identifiable personal data from existing records. "Identifiable personal data" means data relating to a living person who might be identifiable if the record includes their name, username, student id, DNA, fingerprint, address, etc. *If YES, you must attach drafts of the participant information sheet(s) and consent form(s) that you will use in section B.3 or, in the case of an existing dataset, provide details of how consent has been obtained.* *You must also retain the completed forms for subsequent inspection. Failure to provide the completed consent request forms will result in withdrawal of any earlier ethical approval of your project.*	**No**
24.	Have you made arrangements to ensure that material and/or private information obtained from or about the participating individuals will remain confidential? Provide details: Will not collect sensitive information.	**Yes**

B.2 If the answer to the following question (25) is YES, you must provide details		*Delete as appropriate*
25.	Will the research be conducted in the participant's home or other non-University location? *If **YES**, provide details of how your safety will be ensured:*	**No**

B.3 Attachments (these should be provided if applicable):	*Delete as appropriate*
Participant information sheet(s)**	**Yes**
Consent form(s)**	**Yes**
Questionnaire(s)**	**Yes**
Topic guide(s) for interviews and focus groups**	**Not applicable**
Permission from external organisations (e.g. for recruitment of participants)**	**Not applicable**

If these items are not available at the time of submitting your project proposal, provisional approval through proportionate review can still be given, under the condition that you must submit the final versions of all items to your supervisor for approval at a later date. **All such items **must** be seen and approved by your supervisor before the activity for which they are needed starts.

Templates

You must use the templates provided by the University as the basis for your participant information sheets and consent forms. These are available from the links below but you **must** adapt them according to the needs of your project before you submit them for consideration.

Adult information sheet:

http://www.city.ac.uk/__data/assets/word_doc/0018/153441/TEMPLATE-FOR-PARTICIAPNT-INFORMATION-SHEET.doc

Adult consent form:

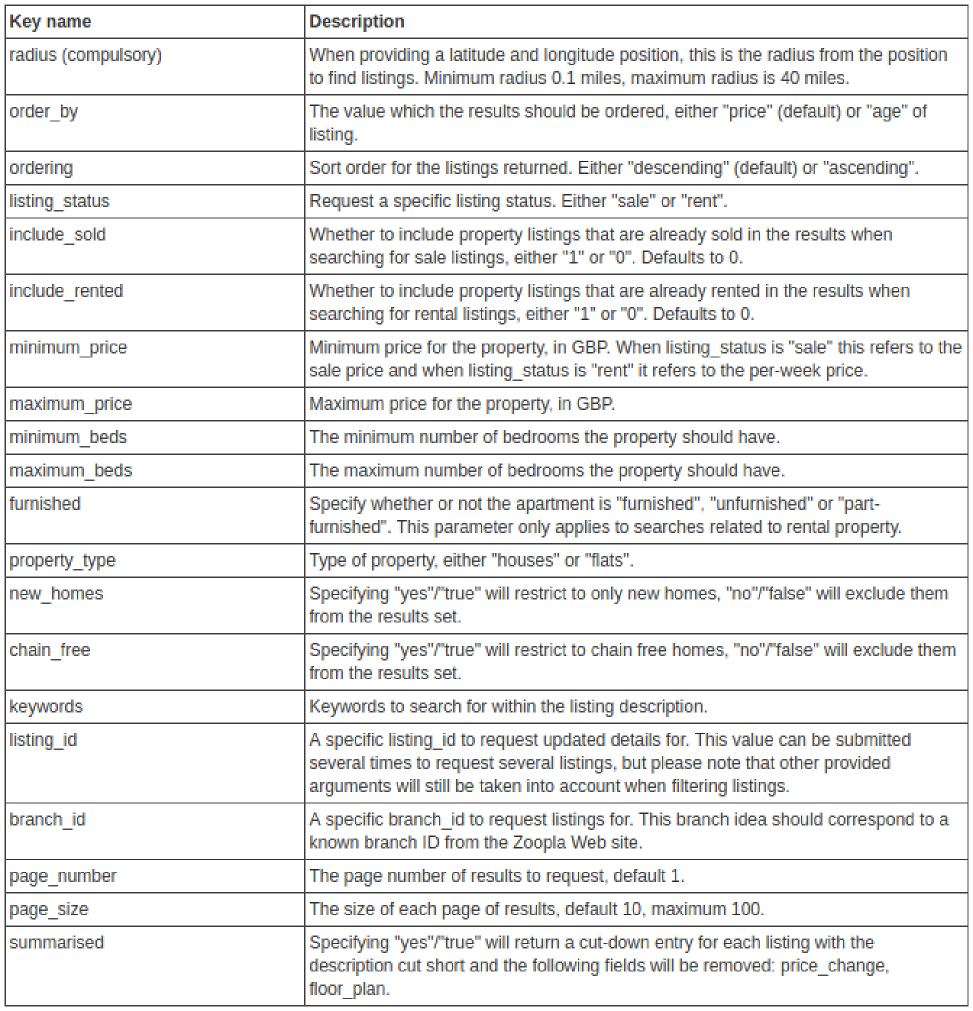
http://www.city.ac.uk/__data/assets/word_doc/0004/153418/TEMPLATE-FOR-CONSENT-FORM.doc

Further Information

Information about the Computer Science Research Ethics Committee (CSREC) is available at: http://www.city.ac.uk/department-computer-science/research-ethics

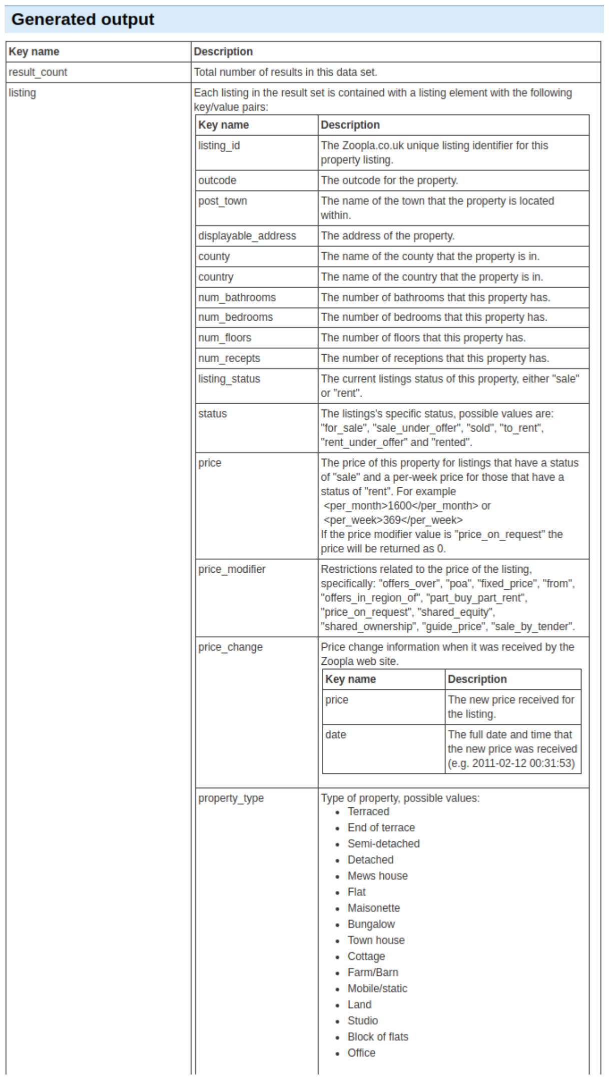
Information about the City University Senate Research Ethics Committee is available at:
http://www.city.ac.uk/research/research-and-enterprise/research-ethics

Appendix C - Zoopla API inputs



Key name	Description
radius (compulsory)	When providing a latitude and longitude position, this is the radius from the position to find listings. Minimum radius 0.1 miles, maximum radius is 40 miles.
order_by	The value which the results should be ordered, either "price" (default) or "age" of listing.
ordering	Sort order for the listings returned. Either "descending" (default) or "ascending".
listing_status	Request a specific listing status. Either "sale" or "rent".
include_sold	Whether to include property listings that are already sold in the results when searching for sale listings, either "1" or "0". Defaults to 0.
include_rented	Whether to include property listings that are already rented in the results when searching for rental listings, either "1" or "0". Defaults to 0.
minimum_price	Minimum price for the property, in GBP. When listing_status is "sale" this refers to the sale price and when listing_status is "rent" it refers to the per-week price.
maximum_price	Maximum price for the property, in GBP.
minimum_beds	The minimum number of bedrooms the property should have.
maximum_beds	The maximum number of bedrooms the property should have.
furnished	Specify whether or not the apartment is "furnished", "unfurnished" or "part-furnished". This parameter only applies to searches related to rental property.
property_type	Type of property, either "houses" or "flats".
new_homes	Specifying "yes"/"true" will restrict to only new homes, "no"/"false" will exclude them from the results set.
chain_free	Specifying "yes"/"true" will restrict to chain free homes, "no"/"false" will exclude them from the results set.
keywords	Keywords to search for within the listing description.
listing_id	A specific listing_id to request updated details for. This value can be submitted several times to request several listings, but please note that other provided arguments will still be taken into account when filtering listings.
branch_id	A specific branch_id to request listings for. This branch idea should correspond to a known branch ID from the Zoopla Web site.
page_number	The page number of results to request, default 1.
page_size	The size of each page of results, default 10, maximum 100.
summarised	Specifying "yes"/"true" will return a cut-down entry for each listing with the description cut short and the following fields will be removed: price_change, floor_plan.

Appendix D - Zoopla API outputs

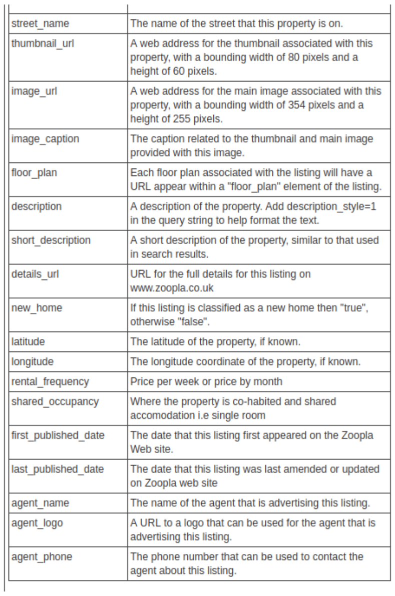


Generated output

Key name	Description
result_count	Total number of results in this data set.
listing	Each listing in the result set is contained with a listing element with the following key/value pairs:

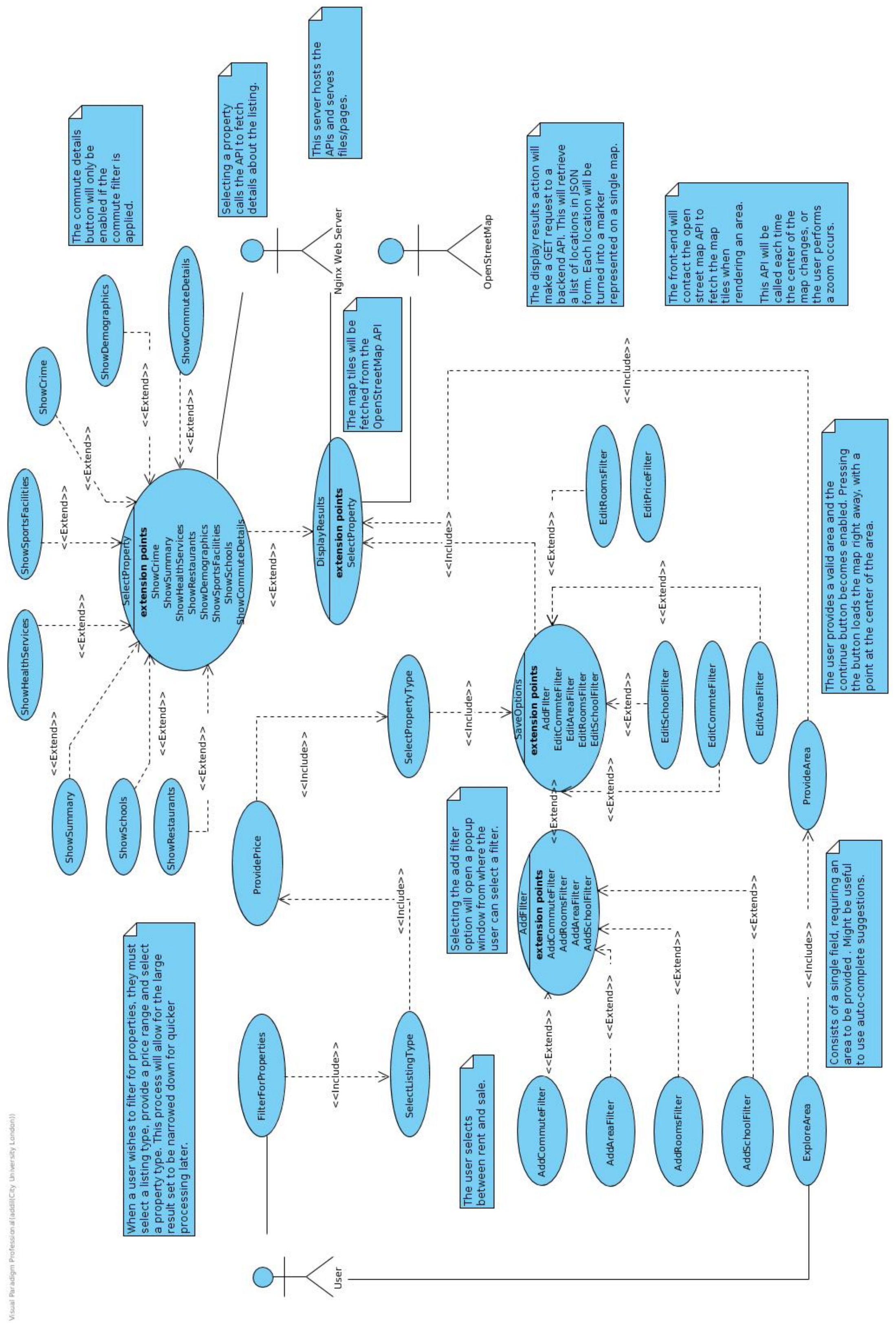
Key name	Description
listing_id	The Zoopla.co.uk unique listing identifier for this property listing.
outcode	The outcode for the property.
post_town	The name of the town that the property is located within.
displayable_address	The address of the property.
county	The name of the county that the property is in.
country	The name of the country that the property is in.
num_bathrooms	The number of bathrooms that this property has.
num_bedrooms	The number of bedrooms that this property has.
num_floors	The number of floors that this property has.
num_recepts	The number of receptions that this property has.
listing_status	The current listings status of this property, either "sale" or "rent".
status	The listings's specific status, possible values are: "for_sale", "sale_under_offer", "sold", "to_rent", "rent_under_offer" and "rented".
price	The price of this property for listings that have a status of "sale" and a per-week price for those that have a status of "rent". For example <per_month>1600</per_month> or <per_week>369</per_week> If the price modifier value is "price_on_request" the price will be returned as 0.
price_modifier	Restrictions related to the price of the listing, specifically: "offers_over", "poa", "fixed_price", "from", "offers_in_region_of", "part_buy_part_rent", "price_on_request", "shared_equity", "shared_ownership", "guide_price", "sale_by_tender".
price_change	Price change information when it was received by the Zoopla web site.
property_type	Type of property, possible values: • Terraced • End of terrace • Semi-detached • Detached • Mews house • Flat • Maisonette • Bungalow • Town house • Cottage • Farm/Barn • Mobile/static • Land • Studio • Block of flats • Office

Key name	Description
price	The new price received for the listing.
date	The full date and time that the new price was received (e.g. 2011-02-12 00:31:53)



Key name	Description
street_name	The name of the street that this property is on.
thumbnail_url	A web address for the thumbnail associated with this property, with a bounding width of 80 pixels and a height of 60 pixels.
image_url	A web address for the main image associated with this property, with a bounding width of 354 pixels and a height of 255 pixels.
image_caption	The caption related to the thumbnail and main image provided with this image.
floor_plan	Each floor plan associated with the listing will have a URL appear within a "floor_plan" element of the listing.
description	A description of the property. Add description_style=1 in the query string to help format the text.
short_description	A short description of the property, similar to that used in search results.
details_url	URL for the full details for this listing on www.zoopla.co.uk
new_home	If this listing is classified as a new home then "true", otherwise "false".
latitude	The latitude of the property, if known.
longitude	The longitude coordinate of the property, if known.
rental_frequency	Price per week or price by month
shared_occupancy	Where the property is co-habited and shared accomodation i.e single room
first_published_date	The date that this listing first appeared on the Zoopla Web site.
last_published_date	The date that this listing was last amended or updated on Zoopla web site
agent_name	The name of the agent that is advertising this listing.
agent_logo	A URL to a logo that can be used for the agent that is advertising this listing.
agent_phone	The phone number that can be used to contact the agent about this listing.

Appendix E - Use-case diagram



The commute details button will only be enabled if the commute filter is applied.
Selecting a property calls the API to fetch details about the listing.
This server hosts the APIs and serves files/pages.
Nginx Web Server
OpenStreetMap
The display results action will make a GET request to a backend API. This will retrieve a list of locations in JSON form. Each location will be turned into a marker represented on a single map.
The front-end will contact the open street map API to fetch the map tiles when rendering an area.
This API will be called each time the center of the map changes, or the user performs a zoom occurs.
The map tiles will be fetched from the OpenStreetMap API
ShowCrime
ShowDemographics
ShowCommuteDetails
ShowSportsFacilities
ShowHealthServices
ShowSummary
ShowSchools
ShowRestaurants
SelectProperty
extension points
ShowCrime
ShowSummary
ShowHealthServices
ShowRestaurants
ShowDemographics
ShowSportsFacilities
ShowSchools
ShowCommuteDetails
DisplayResults
extension points
SelectProperty
EditRoomsFilter
EditPriceFilter
SaveOptions
extension points
AddFilter
EditCommteFilter
EditAreaFilter
EditRoomsFilter
EditSchoolFilter
EditSchoolFilter
EditCommteFilter
EditAreaFilter
SelectPropertyType
ProvidePrice
ProvideArea
The user provides a valid area and the continue button becomes enabled. Pressing the button loads the map right away, with a point at the center of the area.
Selecting the add filter option will open a popup window from where the user can select a filter.
AddFilter
extension points
AddCommuteFilter
AddRoomsFilter
AddAreaFilter
AddSchoolFilter
Consists of a single field, requiring an area to be provided . Might be useful to use auto-complete suggestions.
When a user wishes to filter for properties, they must select a listing type, provide a price range and select a property type. This process will allow for the large result set to be narrowed down for quicker processing later.
FilterForProperties
SelectListingType
The user selects between rent and sale.
AddCommuteFilter
AddAreaFilter
AddRoomsFilter
AddSchoolFilter
ExploreArea
User
<<Extend>>
<<Include>>
Visual Paradigm Professional(addil(City University London))

Appendix F - Django Models

These models were defined entirely by the Author.

Appendix F.1 - ActivePlaces/models.py

```python
from django.db import models

from Core.models import Location


class Equipment(models.Model):
    tableTennisTables = models.IntegerField(default=0)
    poolHoist = models.IntegerField(default=0)
    bowlingMachine = models.IntegerField(default=0)
    trampolines = models.IntegerField(default=0)
    parallelBars = models.IntegerField(default=0)
    highBars = models.IntegerField(default=0)
    stillRings = models.IntegerField(default=0)
    unevenBars = models.IntegerField(default=0)
    balanceBeam = models.IntegerField(default=0)
    vault = models.IntegerField(default=0)
    pommelHorse = models.IntegerField(default=0)


class Activity(models.Model):
    name = models.CharField(max_length=30)

    def __str__(self):
        return self.name


class Contacts(models.Model):
    contactType = models.CharField(max_length=30)
    email = models.EmailField(null=True)
    telephone = models.CharField(max_length=60, null=True)
    website = models.URLField(null=True, max_length=400)


class Disability(models.Model):
    access = models.BooleanField(null=True)
    notes = models.CharField(max_length=1000, null=True)
    parking = models.BooleanField()
    findingReachingEntrance = models.BooleanField()
    receptionArea = models.BooleanField()
    doorways = models.BooleanField()
    changingFacilities = models.BooleanField()
    activityAreas = models.BooleanField()
    toilets = models.BooleanField()
    socialAreas = models.BooleanField()
    spectatorAreas = models.BooleanField()
    emergencyExits = models.BooleanField()


class ActivePlace(Location):
    active_place_id = models.IntegerField(primary_key=True, db_index=True) # AKA id
    state = models.CharField(max_length=30)
    kind = models.CharField(max_length=30)
    outputAreaCode = models.CharField(max_length=30)
    lowerSuperOutputArea = models.CharField(max_length=30)
    middleSuperOutputArea = models.CharField(max_length=30)
    parliamentaryConstituencyCode = models.CharField(max_length=30)
    parliamentaryConstituencyName = models.CharField(max_length=60)
    wardCode = models.CharField(max_length=30)
    wardName = models.CharField(max_length=60)
    localAuthorityCode = models.CharField(max_length=30)
    localAuthorityName = models.CharField(max_length=60)
    buildingName = models.CharField(max_length=100, null=True)
    buildingNumber = models.CharField(max_length=20)
    hasCarPark = models.BooleanField(default=False)
    carParkCapacity = models.IntegerField()
    dedicatedFootballFacility = models.BooleanField(default=False)
    cyclePark = models.BooleanField(default=False)
    cycleHire = models.BooleanField(default=False)
    cycleRepairWorkshop = models.BooleanField(default=False)
    nursery = models.BooleanField(default=False)
    ownerType = models.CharField(max_length=60)
    equipment = models.OneToOneField(Equipment, on_delete=models.CASCADE)
    disability = models.OneToOneField(Disability, on_delete=models.CASCADE)
    contact = models.OneToOneField(Contacts, on_delete=models.CASCADE, null=True)
    activities = models.ManyToManyField(Activity)

    def __str__(self):
        return self.name


class Facility(models.Model):
    active_place = models.ForeignKey(ActivePlace, on_delete=models.CASCADE)
    facilityType =  models.CharField(max_length=30)
    yearBuilt = models.IntegerField(null=True)
    yearBuiltEstimated = models.BooleanField(default=False)
    isRefurbished = models.BooleanField(default=False)
    yearRefurbished = models.IntegerField(null=True)
    hasChangingRooms = models.BooleanField(null=True)
    areChangingRoomsRefurbished = models.BooleanField(null=True)
    yearChangingRoomsRefurbished = models.IntegerField(null=True)
    # Opening times - implemented
    # facilitySpecifics - disability -- not added
    seasonalityType = models.CharField(max_length=30)
    seasonalityStart = models.CharField(max_length=30)
    seasonalityEnd = models.CharField(max_length=30)


class OpeningTimes(models.Model):
    facility = models.ForeignKey(Facility, on_delete=models.CASCADE)
    accessDescription = models.CharField(max_length=100)
    openingTime = models.CharField(max_length=20)
    closingTime = models.CharField(max_length=20)
    periodOpenFor = models.CharField(max_length=50)

    def __str__(self):
        return "%s - (%s - %s)" % (self.periodOpenFor, self.openingTime, self.closingTime)
```

Appendix F.2 - Core/models.py

```
from django.db import models


class Location(models.Model):
    name = models.CharField(max_length=100)
    postcode = models.CharField(max_length=10)
    street = models.CharField(max_length=100, null=True)
    locality = models.CharField(max_length=60, null=True)
    town = models.CharField(max_length=60, null=True)

    lng = models.DecimalField(max_digits=9, decimal_places=6)
    lat = models.DecimalField(max_digits=9, decimal_places=6)

    class Meta:
        abstract = True

    def __str__(self):
        return self.name


class Postcode(models.Model):
    postal_code = models.CharField(max_length=9)
    latitude = models.DecimalField(max_digits=9, decimal_places=6)
    longitude = models.DecimalField(max_digits=9, decimal_places=6)
    local_authority_name = models.CharField(max_length=50, null=True)
    local_authority_code = models.CharField(max_length=50, null=True)

    def __str__(self):
        return self.postal_code


class Demographic(models.Model):
    local_authority_name = models.CharField(max_length=50)
    local_authority_code = models.CharField(max_length=50)
    age = models.IntegerField(null=True) # null represents the total count
    ethnic_group = models.CharField(max_length=60)
    population_2018 = models.IntegerField()
    population_2019 = models.IntegerField()
    def __str__(self):
        return "%s - %s - %s" % (self.local_authority_name, self.ethnic_group, self.age)
```

Appendix F.3 - CQC/models.py

```
from django.db import models
from Core.models import Location


class CQCLocation(Location):
    cqc_id  = models.CharField(null=True, max_length=20)
    website = models.URLField(null=True)
    location_type = models.CharField(max_length=600)
    last_inspection_date = models.DateField(null=True)
    last_rating = models.CharField(max_length=30, null=True)

    def __str__(self):
        return self.name
```

Appendix F.4 - HereMaps/models.py

```
from django.db import models
from django_extensions.db.models import TimeStampedModel


class ReverseGeoCodeCache(models.Model):
    latitude = models.DecimalField(max_digits=9, decimal_places=6)
    longitude = models.DecimalField(max_digits=9, decimal_places=6)
    label = models.CharField(max_length=100)


class RouteCache(TimeStampedModel):
    start_latitude = models.DecimalField(max_digits=9, decimal_places=6,)
    start_longitude = models.DecimalField(max_digits=9, decimal_places=6)
    des_latitude = models.DecimalField(max_digits=9, decimal_places=6)
    des_longitude = models.DecimalField(max_digits=9, decimal_places=6)
    commute_time = models.IntegerField()
    data = models.TextField()
```

Appendix F.5 - Schools/models.py

```
from django.db import models
from Core.models import Location


class School(Location):
    BOYS = 'B'
    GIRLS = 'G'
    MIXED = 'M'

    GENDER_CHOICES = (
        (BOYS, 'Boys'),
        (GIRLS, 'Girls'),
        (MIXED, 'Mixed'),
    )

    urn = models.IntegerField()
    other_name = models.CharField(max_length=100)
    phone = models.CharField(max_length=20)
    is_new = models.BooleanField(default=False)
    is_primary = models.BooleanField(default=False)
    is_secondary = models.BooleanField(default=False)
    is_post16 = models.BooleanField(default=False)
    age_from = models.IntegerField()
    age_to = models.IntegerField()
    gender = models.CharField(max_length=2,
                              choices=GENDER_CHOICES,
                              default=MIXED)
    sixth_form_gender = models.CharField(max_length=2,
                                         choices=GENDER_CHOICES,
                                         default=MIXED)
    religion = models.CharField(max_length=100, null=True)
    selective = models.BooleanField(null=True)

    def __str__(self):
        return self.name
```

```python
class OfstedInspection(models.Model):
    OUTSTANDING = 1
    GOOD = 2
    REQUIRES_IMPROVEMENT = 3
    INADEQUATE = 4

    OVERALL_EFFECTIVENESS_CHOICES = (
        (OUTSTANDING, 'Outstanding'),
        (GOOD, 'Good'),
        (REQUIRES_IMPROVEMENT, 'Requires Improvement'),
        (INADEQUATE, 'Inadequate'),
    )

    school = models.ForeignKey(School, on_delete=models.CASCADE)
    inspection_date = models.DateTimeField(null=True)
    publication_date = models.DateTimeField(null=True)
    overall_effectiveness = models.IntegerField(choices=OVERALL_EFFECTIVENESS_CHOICES)

    def __str__(self):
        return "%s %s" % (self.school.name, self.overall_effectiveness)

    @property
    def link(self):
        return "http://www.ofsted.gov.uk/oxedu_providers/full/%s/" % self.school.urn

    def get_overall_effectiveness(self):
        return [i for i in self.OVERALL_EFFECTIVENESS_CHOICES if i[0] == self.overall_effectiveness][0][1]
```

Appendix F.6 - Zoopla/models.py

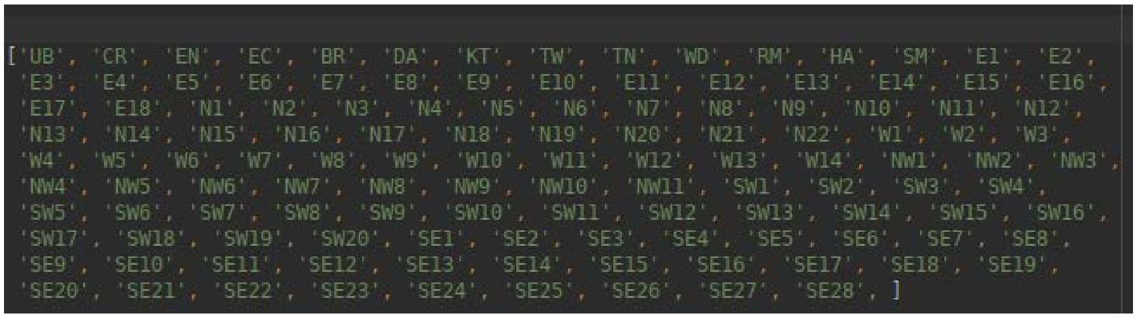
```python
from django.db import models

# Create your models here.
from model_utils.models import TimeStampedModel


class ZooplaQuery(TimeStampedModel):
    area = models.CharField(max_length=100)
    listing_status = models.CharField(max_length=10)
    radius = models.FloatField()
    minimum_price = models.FloatField(null=True)
    maximum_price = models.FloatField(null=True)
    # furnished = models. : TODO: Implement
    property_type = models.CharField(max_length=20)
    page = models.IntegerField(default=1)
    number_of_results = models.IntegerField()
    results = models.TextField()


class Agent(models.Model):
    agent_address = models.CharField(max_length=200)
    agent_logo = models.URLField(null=True)
    agent_name = models.CharField(max_length=200, null=True)
    agent_phone = models.CharField(max_length=20, null=True)


class Property(TimeStampedModel):
    zoopla_query = models.ManyToManyField(to=ZooplaQuery)
    listing_id = models.IntegerField(primary_key=True)
    displayable_address = models.CharField(max_length=200)
    num_bathrooms = models.IntegerField()
    num_bedrooms = models.IntegerField()
    num_floors = models.IntegerField()
    num_recepts    = models.IntegerField()
    listing_status = models.CharField(max_length=10)
    status = models.CharField(max_length=10)
    price = models.FloatField()
    first_published = models.DateTimeField()
    last_published = models.DateTimeField()
    agent = models.ForeignKey(Agent, on_delete=models.DO_NOTHING)
    category = models.CharField(max_length=30)
    county = models.CharField(max_length=30, null=True)
    description = models.TextField(null=True)
    details_url = models.URLField(null=True)
    furnished_state = models.CharField(max_length=100)
    latitude = models.DecimalField(max_digits=9, decimal_places=6)
    longitude = models.DecimalField(max_digits=9, decimal_places=6)
    outcode = models.CharField(max_length=5)
    post_town = models.CharField(max_length=30, null=True)
    property_type = models.CharField(max_length=50, null=True)
    short_description = models.TextField(null=True)
    street_name = models.CharField(max_length=100, null=True)
    thumbnail_url = models.URLField(null=True)


class RentalPrice(models.Model):
    accurate = models.CharField(max_length=30, null=True)
    per_month = models.FloatField()
    per_week = models.FloatField()
    shared_occupancy = models.CharField(max_length=30)
    zoopla_property = models.ForeignKey(Property, on_delete=models.CASCADE, null=True)


class PriceHistory(models.Model):
    zoopla_property = models.ForeignKey(Property, on_delete=models.CASCADE)
    date_changed = models.DateTimeField()
    direction = models.CharField(max_length=20, null=True)
    percent = models.FloatField()
    price = models.FloatField()


class PropertyImage(models.Model):
    zoopla_property = models.ForeignKey(Property, on_delete=models.CASCADE)
    url = models.URLField()
```

Appendix G - Import data

This section of the appendix show the code that was written during the data import stage.

Appendix G.1 - Zoopla

Appendix G.1.1 - List of postal areas



```
['UB', 'CR', 'EN', 'EC', 'BR', 'DA', 'KT', 'TW', 'TN', 'WD', 'RM', 'HA', 'SM', 'E1', 'E2',
 'E3', 'E4', 'E5', 'E6', 'E7', 'E8', 'E9', 'E10', 'E11', 'E12', 'E13', 'E14', 'E15', 'E16',
 'E17', 'E18', 'N1', 'N2', 'N3', 'N4', 'N5', 'N6', 'N7', 'N8', 'N9', 'N10', 'N11', 'N12',
 'N13', 'N14', 'N15', 'N16', 'N17', 'N18', 'N19', 'N20', 'N21', 'N22', 'W1', 'W2', 'W3',
 'W4', 'W5', 'W6', 'W7', 'W8', 'W9', 'W10', 'W11', 'W12', 'W13', 'W14', 'NW1', 'NW2', 'NW3',
 'NW4', 'NW5', 'NW6', 'NW7', 'NW8', 'NW9', 'NW10', 'NW11', 'SW1', 'SW2', 'SW3', 'SW4',
 'SW5', 'SW6', 'SW7', 'SW8', 'SW9', 'SW10', 'SW11', 'SW12', 'SW13', 'SW14', 'SW15', 'SW16',
 'SW17', 'SW18', 'SW19', 'SW20', 'SE1', 'SE2', 'SE3', 'SE4', 'SE5', 'SE6', 'SE7', 'SE8',
 'SE9', 'SE10', 'SE11', 'SE12', 'SE13', 'SE14', 'SE15', 'SE16', 'SE17', 'SE18', 'SE19',
 'SE20', 'SE21', 'SE22', 'SE23', 'SE24', 'SE25', 'SE26', 'SE27', 'SE28', ]
```

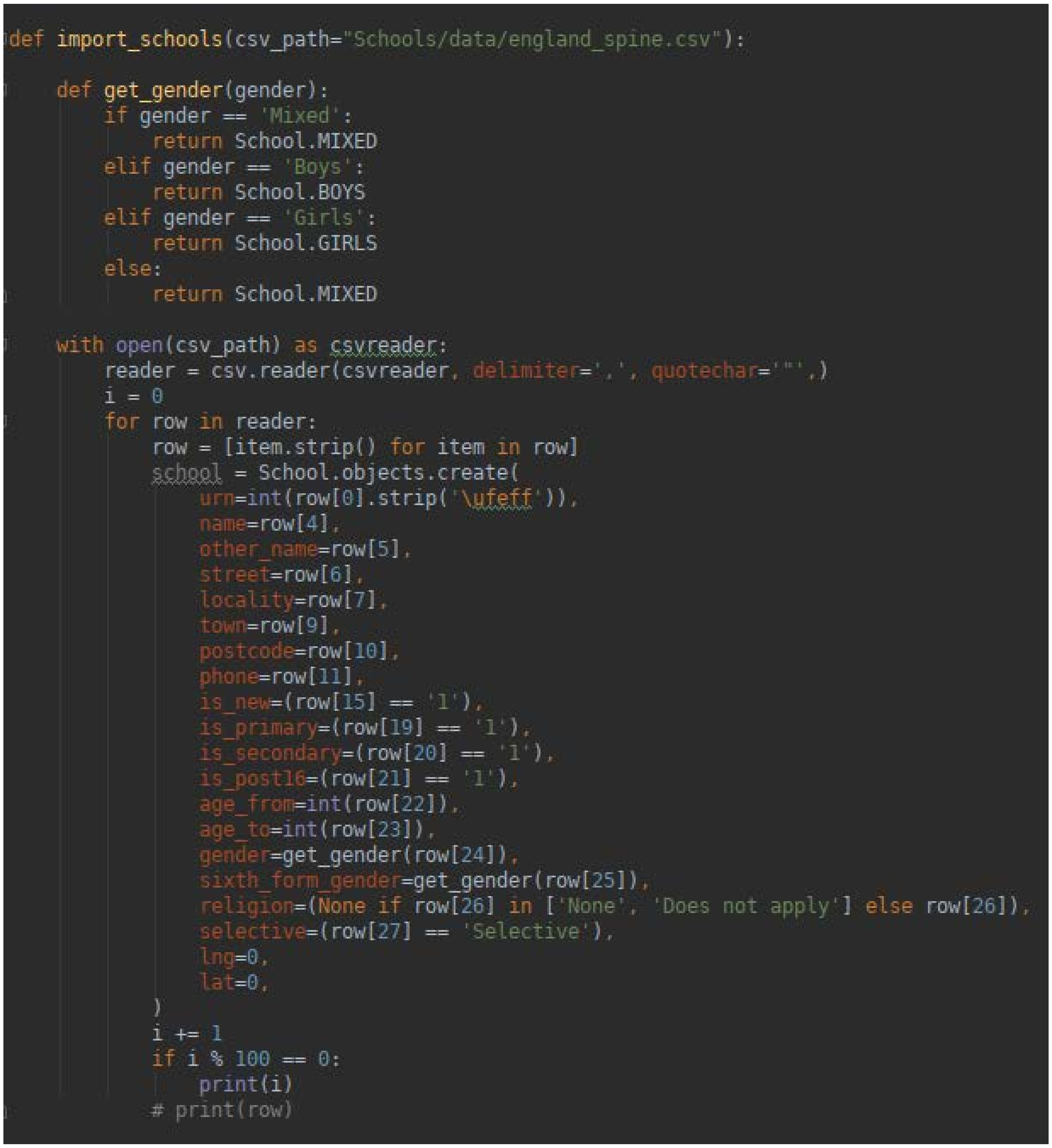
Appendix G.1.2 - Import method definition

```python
from LocationFinder.settings import ZOOPLA_API_KEY
from Zoopla.models import ZooplaQuery, Property, Agent, PropertyImage, PriceHistory, RentalPrice

tz_london = pytz.timezone('Europe/London')


def search_properties(area, listing_status, radius=1, min_price=None, max_price=None, furnished=None,
                      property_type=None):
    """
    Search for a property listing
    :param area:
    :param listing_status: 'sale' or 'rent'
    :param radius: in miles
    :param min_price: Sale -> Total price / Rent -> per week cost
    :param max_price: Sale -> Total price / Rent -> per week cost
    :param furnished:
    :param property_type: house or flat
    :return:
    """

    price_filters = Q()

    if max_price:
        price_filters &= Q(maximum_price__lte=max_price)

    if min_price:
        price_filters &= Q(minimum_price__gte=min_price)

    query = ZooplaQuery.objects.filter(
        price_filters,
        area=area,
        listing_status=listing_status,
        radius=radius, created__gte=timezone.now() - relativedelta(days=2))

    if not query:
        url = "https://api.zoopla.co.uk/api/v1/property_listings?" \
              "api_key=%s" \
              "&area=%s" \
              "&radius=%s" \
              "&page_size=100" \
              "&listing_status=%s" \
              % (ZOOPLA_API_KEY, area, radius, listing_status)

        url += "&minimum_price=%s" % min_price if min_price else ''
        url += "&maximum_price=%s" % max_price if max_price else ''
        # url += "&furnished=%s" % furnished if furnished else '' // only applies to rental
        url += "&property_type=%s" % property_type if property_type else ''

        r = requests.get(url)

        response_dict = xmltodict.parse(r.content)
        response_json = json.dumps(response_dict)

        # Insert the query into the database
        zoopla_query = ZooplaQuery.objects.create(
            minimum_price=min_price,
            maximum_price=max_price,
            area=area,
            listing_status=listing_status,
            radius=radius,
            results=response_json,
            # number_of_results=response_dict['response']['listing'].__len__()
            number_of_results=response_dict['response']['result_count'],
        )

        listing_id_timestamps = [(val['listing_id'], val['last_published_date']) for val in
                                 response_dict['response']['listing']]

        condition = Q()

        for (id, timestamp) in listing_id_timestamps:
            condition |= Q(listing__id=id) & Q(last_published=datetime.strptime(timestamp, "%Y-%m-%d %H:%M:%S"))

        listing_ids, a = zip(*listing_id_timestamps)
        existing_listing_ids = list(
            Property.objects.filter(condition).values_list('listing_id', flat=True))
        new_listing_ids = set(listing_ids) - set(existing_listing_ids)
        print(new_listing_ids)

        for p in response_dict['response']['listing']:
            if p['listing_id'] in new_listing_ids:
                property_instance, updated = Property.objects.update_or_create(

                )

        return zoopla_query

    else:
        return query.first()


london_postcode_districts = ['UB', 'CR', 'EN', 'EC', 'BR', 'DA', 'KT', 'TW', 'TN', 'WD', 'RM', 'HA', 'SM', 'E1', 'E2',
                             'E3', 'E4', 'E5', 'E6', 'E7', 'E8', 'E9', 'E10', 'E11', 'E12', 'E13', 'E14', 'E15', 'E16',
                             'E17', 'E18', 'N1', 'N2', 'N3', 'N4', 'N5', 'N6', 'N7', 'N8', 'N9', 'N10', 'N11', 'N12',
                             'N13', 'N14', 'N15', 'N16', 'N17', 'N18', 'N19', 'N20', 'N21', 'N22', 'W1', 'W2', 'W3',
                             'W4', 'W5', 'W6', 'W7', 'W8', 'W9', 'W10', 'W11', 'W12', 'W13', 'W14', 'NW1', 'NW2', 'NW3',
                             'NW4', 'NW5', 'NW6', 'NW7', 'NW8', 'NW9', 'NW10', 'NW11', 'SW1', 'SW2', 'SW3', 'SW4',
                             'SW5', 'SW6', 'SW7', 'SW8', 'SW9', 'SW10', 'SW11', 'SW12', 'SW13', 'SW14', 'SW15', 'SW16',
                             'SW17', 'SW18', 'SW19', 'SW20', 'SE1', 'SE2', 'SE3', 'SE4', 'SE5', 'SE6', 'SE7', 'SE8',
                             'SE9', 'SE10', 'SE11', 'SE12', 'SE13', 'SE14', 'SE15', 'SE16', 'SE17', 'SE18', 'SE19',
                             'SE20', 'SE21', 'SE22', 'SE23', 'SE24', 'SE25', 'SE26', 'SE27', 'SE28', ]


def results_counter():
    base_page_url =
"https://api.zoopla.co.uk/api/v1/property_listings?&api_key=e9gkxstnn2jq4d7njqz2mnw4&page_size=100&order_by=age&ordering=
ascending&area="
    i = 0

    for p in london_postcode_districts:
        try:
            api_url = base_page_url + p
            r = requests.get(api_url)
            response_dict = xmltodict.parse(r.content)

            i += int(response_dict['response']['result_count'])
            print("total: %s, %s: %s  " % (i, p, response_dict['response']['result_count']))
        except Exception as e:
            print(e)
            continue


def data_grabber():
    """
    Method to extract data from the Zoopla API
    :return:
    """
    test_mode = True

    api_key = 'TO PASTE KEY HERE' # To replace by user

    if test_mode:
        # Will limit the system to the defined area
        sub_areas = ["Clerkenwell", "marylebone"]
    else:
        sub_areas = london_postcode_districts

    base_page_url =
"https://api.zoopla.co.uk/api/v1/property_listings?api_key=%s&page_size=100&order_by=age&ordering=ascending" % (api_key)

    page = 1

    # Boolean to control when to stop the loop - Will change to true once the hourly API call limit has been reached
    limitted = False

    try:
        for sub_area in sub_areas:
            while True:
                # Form the new page URL and fetch the data.
                page_url = base_page_url + ("&page_number=%s&area=%s" % (page, sub_area))
                r = requests.get(page_url)
                print(page_url)

                # If a response code other than 200 is returned, there was an error
                if not r.status_code == 200:
                    print("Over rate...")
                    print(page_url)
                    print('Postcode: %s  I: %s' % (sub_area, page) )
                    limitted = True
                    break
                    # raise Exception('Postcode: %s  I: %s' % (postcode, page) )

                try:
                    response_dict = xmltodict.parse(r.content)
                except Exception as e:
                    print("Parse error on page %s" % page)
                    page += 1
                    continue

                response_json = json.dumps(response_dict)

                zoopla_query = ZooplaQuery.objects.create(
                    minimum_price=0,
                    maximum_price=0,
                    area="london",
                    listing_status="-",
                    radius=0,
                    results=response_json,
                    # number_of_results=response_dict['response']['listing'].__len__()
                    number_of_results=response_dict['response']['result_count'],
                )

                x = 0
                try:
                    for p in response_dict['response']['listing']:

                        try:
                            agent, created = Agent.objects.get_or_create(
                                agent_address__iexact=p['agent_address'],
                                defaults={
                                    'agent_logo': p['agent_logo'],
                                    'agent_name': p['agent_name'],
                                    'agent_phone': p['agent_phone'],
                                }
                            )

                            property_instance, property_created = Property.objects.update_or_create(
                                listing_id=p['listing_id'],
                                defaults={
                                    'category': p['category'],
                                    'county': p['county'],
                                    'description': p['description'],
                                    'details_url': p['details_url'],
                                    'first_published': timezone.datetime.fromtimestamp(
                                        timezone.datetime.strptime(p['first_published_date'], '%Y-%m-%d
%H:%M:%S').timestamp(),

                                        tz=tz_london),
                                    'last_published': timezone.datetime.fromtimestamp(
                                        timezone.datetime.strptime(p['last_published_date'], '%Y-%m-%d
%H:%M:%S').timestamp(),

                                        tz=tz_london),
                                    'furnished_state': "Unfurnished" if p['furnished_state'] == None else
p['furnished_state'],
                                    'latitude': p['latitude'],
                                    'longitude': p['longitude'],
                                    'listing_status': p['listing_status'],
                                    'num_bathrooms': p['num_bathrooms'],
                                    'num_bedrooms': p['num_bedrooms'],
                                    'num_floors': p['num_floors'],
                                    'num_recepts': p['num_recepts'],
                                    'outcode': p['outcode'],
                                    'post_town': p['post_town'],
                                    'price': p['price'],
                                    'property_type': p['property_type'],
                                    'short_description': p['short_description'],
                                    'status': p['status'],
                                    'street_name': p['street_name'],
                                    'thumbnail_url': p['thumbnail_url'],
                                    'agent': agent,
                                }
                            )

                            property_instance.zoopla_query.add(zoopla_query)
                            p_dict = dict(p)

                            if 'rental_prices' in p_dict:
                                rp = RentalPrice.objects.create(
                                    accurate=p_dict['rental_prices']['accurate'],
                                    per_month=p_dict['rental_prices']['per_month'],
                                    per_week=p_dict['rental_prices']['per_week'],
                                    shared_occupancy=p_dict['rental_prices']['shared_occupancy'],
                                    zoopla_property=property_instance,
                                )

                            if 'price_change' in p_dict:
                                for price_change in p_dict['price_change']:
                                    try:
                                        ph = PriceHistory.objects.create(
                                            zoopla_property=property_instance,
                                            date_changed=timezone.datetime.fromtimestamp(
                                                timezone.datetime.strptime(price_change['date'], '%Y-%m-%d
%H:%M:%S').timestamp(),

                                                tz=tz_london),
                                            price=price_change['price'],
                                            percent=price_change['percent'][:-1]
                                        )
                                    except TypeError as e:
                                        pass

                            if 'image_url' in p_dict and p_dict['image_url']:
                                image, created = PropertyImage.objects.get_or_create(
                                    url=p['image_url'],
                                    zoopla_property=property_instance
                                )

                            x += 1
                            print(x)
                        except Exception as e:
                            print(e)
                            print(p)
                            #print(p_dict)
                            break

                    # Increment the page number if successful
                    print("Page %s complete" % page)
                    page += 1
                except KeyError:
                    break

            # Reset the page counter to one and start query the next postcode.
            page = 1
            if limitted:
                break
```

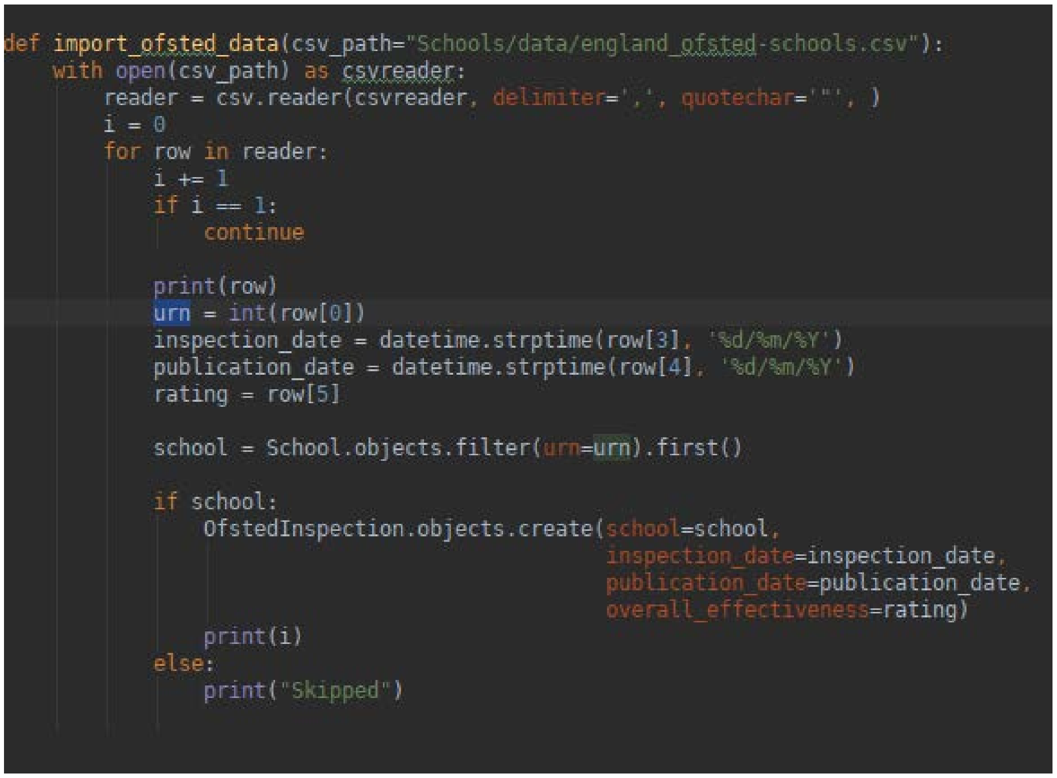
```
    except Exception as e:
        print(e)
```

Appendix G.2 - Schools and Ofsted inspections

Appendix G.2.1 - Import method definition

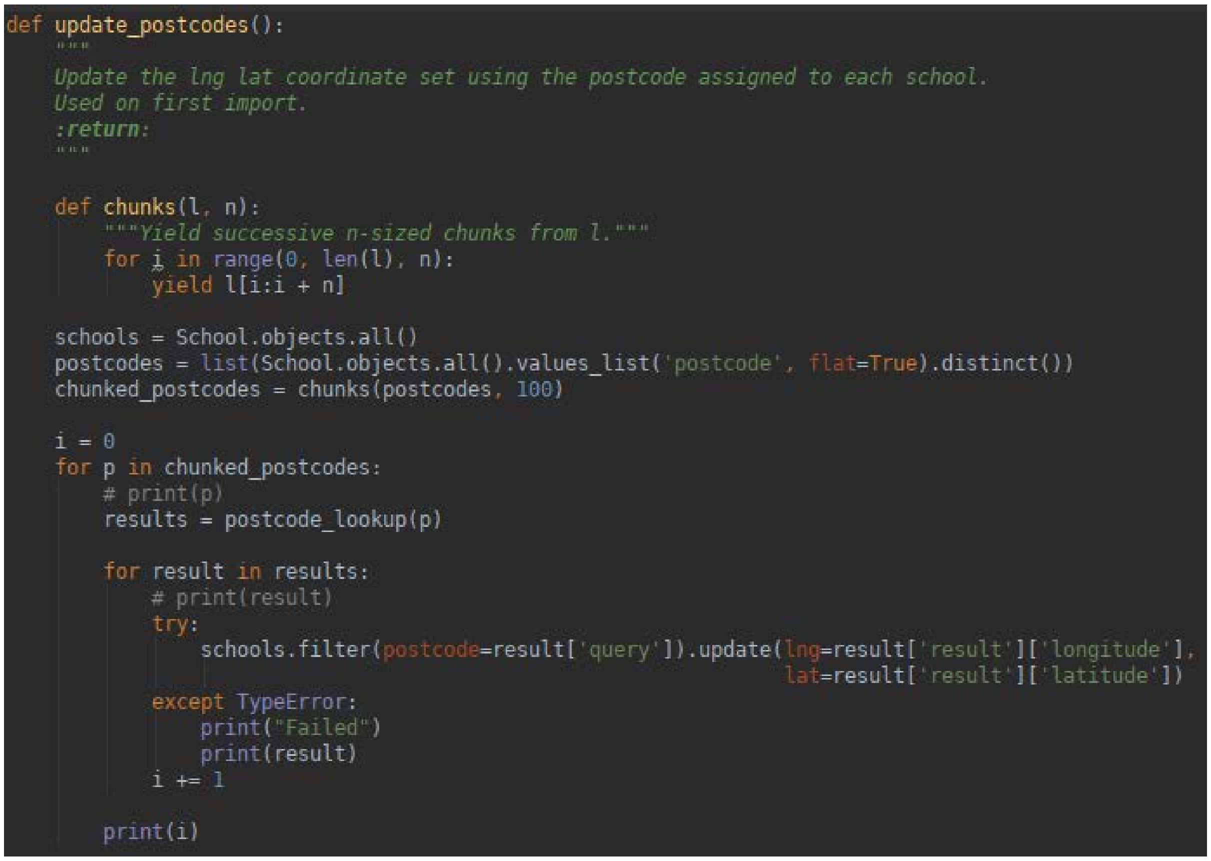


```
def import_schools(csv_path="Schools/data/england_spine.csv"):

    def get_gender(gender):
        if gender == 'Mixed':
            return School.MIXED
        elif gender == 'Boys':
            return School.BOYS
        elif gender == 'Girls':
            return School.GIRLS
        else:
            return School.MIXED

    with open(csv_path) as csvreader:
        reader = csv.reader(csvreader, delimiter=',', quotechar='"',)
        i = 0
        for row in reader:
            row = [item.strip() for item in row]
            school = School.objects.create(
                urn=int(row[0].strip('\ufeff')),
                name=row[4],
                other_name=row[5],
                street=row[6],
                locality=row[7],
                town=row[9],
                postcode=row[10],
                phone=row[11],
                is_new=(row[15] == '1'),
                is_primary=(row[19] == '1'),
                is_secondary=(row[20] == '1'),
                is_post16=(row[21] == '1'),
                age_from=int(row[22]),
                age_to=int(row[23]),
                gender=get_gender(row[24]),
                sixth_form_gender=get_gender(row[25]),
                religion=(None if row[26] in ['None', 'Does not apply'] else row[26]),
                selective=(row[27] == 'Selective'),
                lng=0,
                lat=0,
            )
            i += 1
            if i % 100 == 0:
                print(i)
            # print(row)
```



```python
def import_ofsted_data(csv_path="Schools/data/england_ofsted-schools.csv"):
    with open(csv_path) as csvreader:
        reader = csv.reader(csvreader, delimiter=',', quotechar='"', )
        i = 0
        for row in reader:
            i += 1
            if i == 1:
                continue

            print(row)
            urn = int(row[0])
            inspection_date = datetime.strptime(row[3], '%d/%m/%Y')
            publication_date = datetime.strptime(row[4], '%d/%m/%Y')
            rating = row[5]

            school = School.objects.filter(urn=urn).first()

            if school:
                OfstedInspection.objects.create(school=school,
                                                inspection_date=inspection_date,
                                                publication_date=publication_date,
                                                overall_effectiveness=rating)

                print(i)
            else:
                print("Skipped")
```

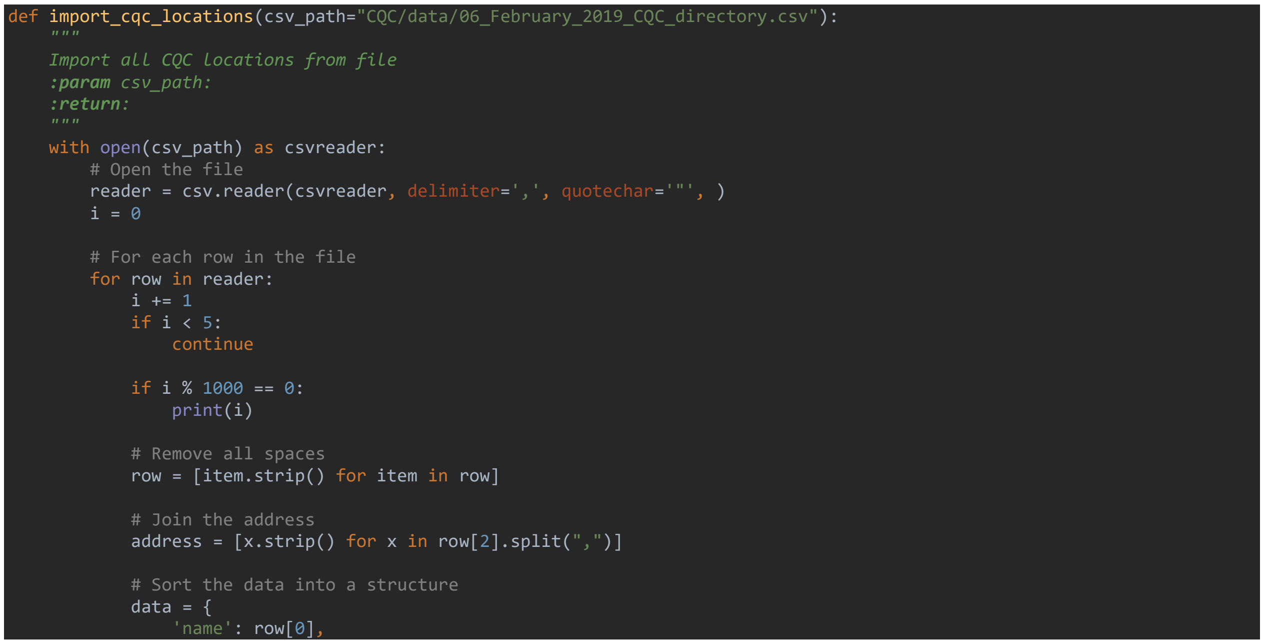
Appendix G.2.2 - Retrieving Geo-coordinates



```python
def update_postcodes():
    """
    Update the lng lat coordinate set using the postcode assigned to each school.
    Used on first import.
    :return:
    """

    def chunks(l, n):
        """Yield successive n-sized chunks from l."""
        for i in range(0, len(l), n):
            yield l[i:i + n]

    schools = School.objects.all()
    postcodes = list(School.objects.all().values_list('postcode', flat=True).distinct())
    chunked_postcodes = chunks(postcodes, 100)

    i = 0
    for p in chunked_postcodes:
        # print(p)
        results = postcode_lookup(p)

        for result in results:
            # print(result)
            try:
                schools.filter(postcode=result['query']).update(lng=result['result']['longitude'],
                                                                lat=result['result']['latitude'])
            except TypeError:
                print("Failed")
                print(result)
            i += 1

        print(i)
```

Appendix G.3 - CQC Locations

Appendix G.3.1 - Import method definition



```python
def import_cqc_locations(csv_path="CQC/data/06_February_2019_CQC_directory.csv"):
    """
    Import all CQC locations from file
    :param csv_path:
    :return:
    """
    with open(csv_path) as csvreader:
        # Open the file
        reader = csv.reader(csvreader, delimiter=',', quotechar='"', )
        i = 0

        # For each row in the file
        for row in reader:
            i += 1
            if i < 5:
                continue

            if i % 1000 == 0:
                print(i)

            # Remove all spaces
            row = [item.strip() for item in row]

            # Join the address
            address = [x.strip() for x in row[2].split(",")]

            # Sort the data into a structure
            data = {
                'name': row[0],
                'street': address[0],
                'postcode': row[3],
                'website': row[5],
                'location_type': row[6],
                'cqc_id': row[13],
                'lat': 0,
                'lng': 0
            }

            # If the address consists of three elements, fetch the second and use it as the town name.
            if address.__len__() == 3:
                data['town'] = address[1]

            # Create the object in the database.
            l = CQCLocation.objects.create(**data)
```

Appendix G.3.2 - Generic postcode to coordinates method

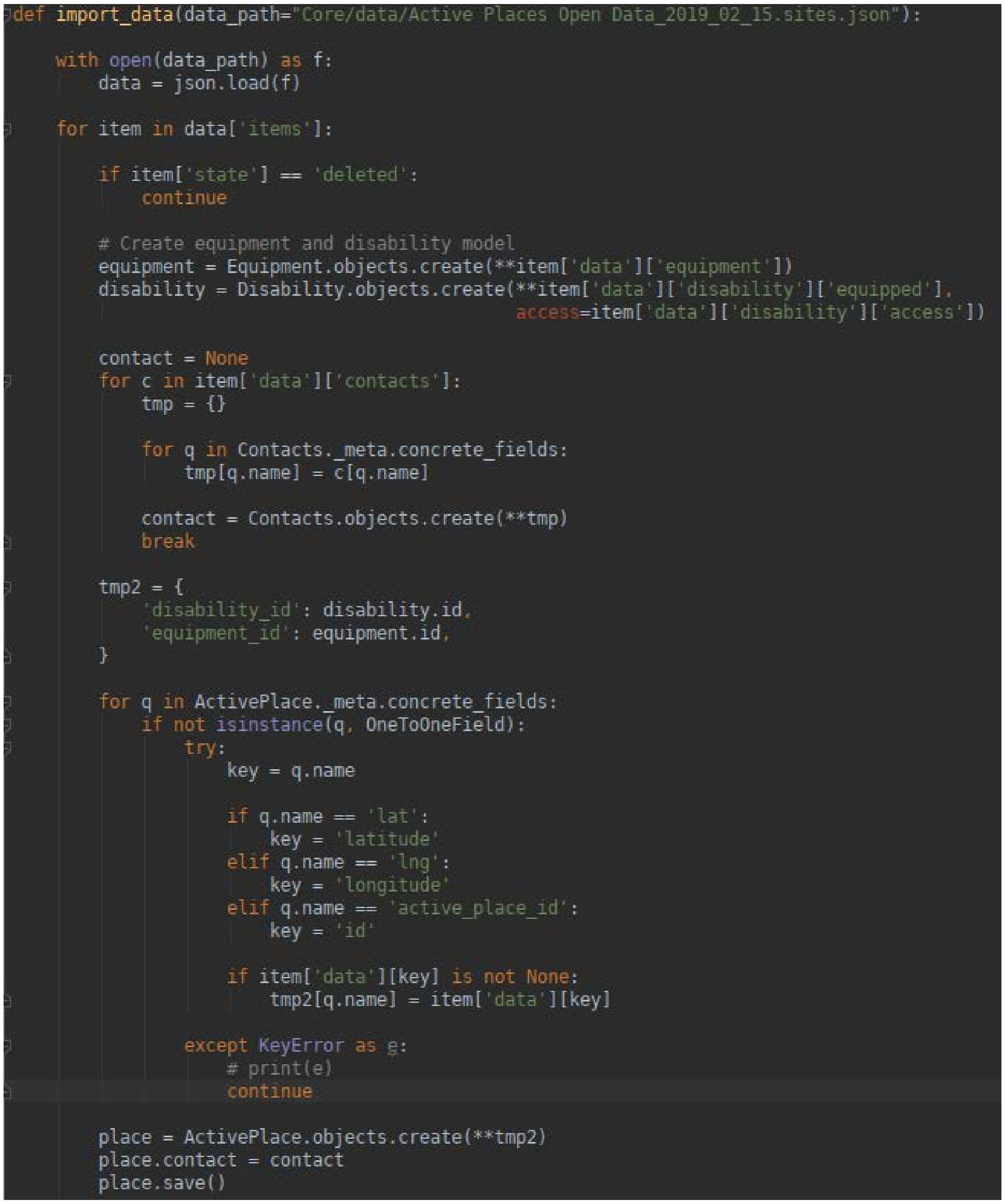
```
def postcode_lookup(postcodes):
    """
    Convert a list of postcodes to geo-coordinates.
    :param postcodes:
    :return:
    """

    # Structure the data for conversion to JSON.
    obj = { 'postcodes': postcodes}
    url = "http://api.postcodes.io/postcodes"

    # Create the request object
    req = urllib.request.Request(url)
    req.add_header('Content-Type', 'application/json')

    # Convert the request data to JSON and encode as UTF-8
    jsondata = json.dumps(obj)
    jsondataasbytes = jsondata.encode('utf-8')

    # Add content length as header.
    req.add_header('Content-Length', len(jsondataasbytes))

    # Make the request and receive response.
    response = urllib.request.urlopen(req, jsondataasbytes)

    # Decode the response from JSON to a dictionary.
    resp_data = json.loads(response.read())

    return resp_data['result']
```

Appendix G.3.3 - Importing CQC ratings

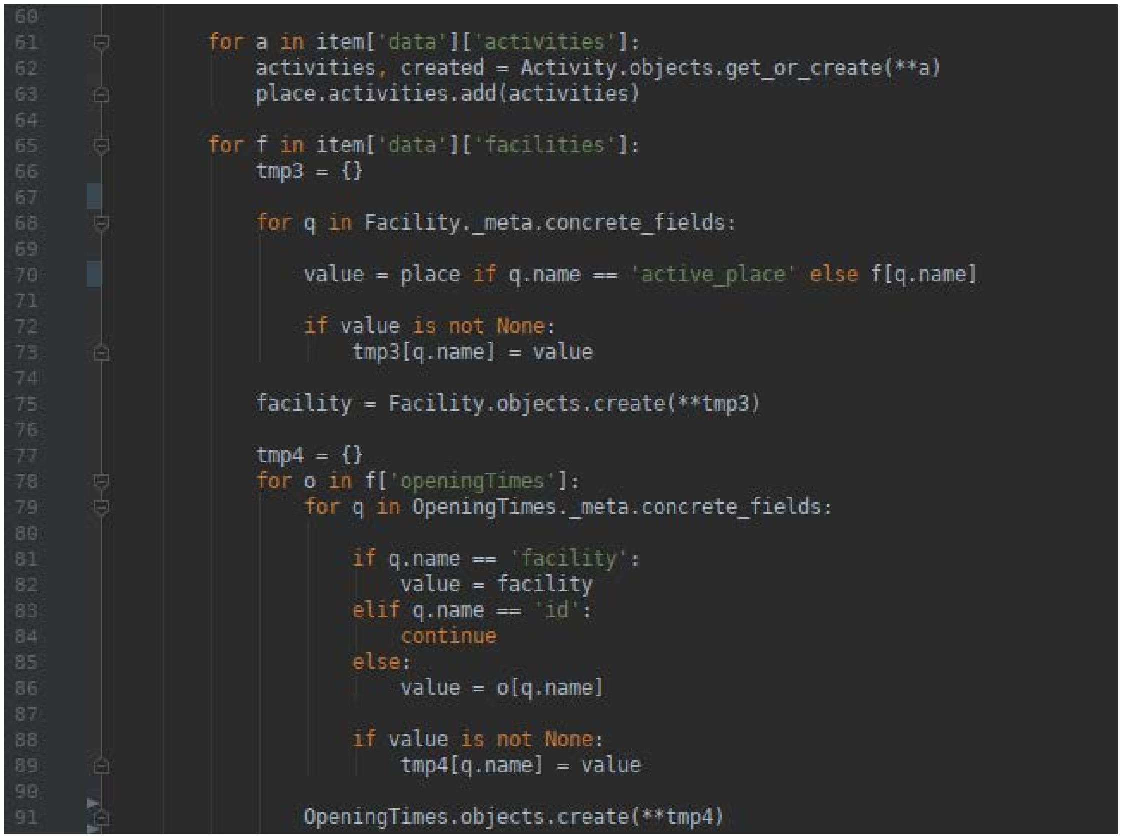
```
def update_ratings(csv_path="CQC/data/ratings February 2019.csv"):
    """
    Method to import CQC location ratings from file.
    :param csv_path:
    :return:
    """

    # Read the file into memory
    with open(csv_path) as csvreader:
        reader = csv.reader(csvreader, delimiter=',', quotechar='"', )
        i = 0

        # For each row in the file.
        for row in reader:
            i += 1

            # Skip the first two rows.
            if i < 2:
                continue

            if i % 1000 == 0:
                print(i)

            # Remove spaces.
            row = [item.strip() for item in row]

            # Extract the needed data.
            cqc_id = row[0]
            last_inspection_date = datetime.strptime(row[8], "%d/%m/%Y").date()
            last_rating = row[7]

            # Try to find the CQC locations in the database.
            try:
                if row[5] == 'Overall' and row[6] == 'Overall':
                    l = CQCLocation.objects.get(cqc_id=cqc_id)

                    if not l.last_inspection_date or l.last_inspection_date <
last_inspection_date:
                        # If the inspection data in the database is older, or does not exist,
then overwrite.
                        l.last_inspection_date = last_inspection_date
                        l.last_rating = last_rating
                        l.save()
                continue

            except ObjectDoesNotExist:
                continue
```

Appendix G.4 - Active places

Appendix G.4.1 - Import method definition



```
def import_data(data_path="Core/data/Active Places Open Data_2019_02_15.sites.json"):

    with open(data_path) as f:
        data = json.load(f)

    for item in data['items']:

        if item['state'] == 'deleted':
            continue

        # Create equipment and disability model
        equipment = Equipment.objects.create(**item['data']['equipment'])
        disability = Disability.objects.create(**item['data']['disability']['equipped'],
                                               access=item['data']['disability']['access'])

        contact = None
        for c in item['data']['contacts']:
            tmp = {}

            for q in Contacts._meta.concrete_fields:
                tmp[q.name] = c[q.name]

            contact = Contacts.objects.create(**tmp)
            break

        tmp2 = {
            'disability_id': disability.id,
            'equipment_id': equipment.id,
        }

        for q in ActivePlace._meta.concrete_fields:
            if not isinstance(q, OneToOneField):
                try:
                    key = q.name

                    if q.name == 'lat':
                        key = 'latitude'
                    elif q.name == 'lng':
                        key = 'longitude'
                    elif q.name == 'active_place_id':
                        key = 'id'

                    if item['data'][key] is not None:
                        tmp2[q.name] = item['data'][key]

                except KeyError as e:
                    # print(e)
                    continue

        place = ActivePlace.objects.create(**tmp2)
        place.contact = contact
        place.save()
```



```
        for a in item['data']['activities']:
            activities, created = Activity.objects.get_or_create(**a)
            place.activities.add(activities)

        for f in item['data']['facilities']:
            tmp3 = {}

            for q in Facility._meta.concrete_fields:

                value = place if q.name == 'active_place' else f[q.name]

                if value is not None:
                    tmp3[q.name] = value

            facility = Facility.objects.create(**tmp3)

            tmp4 = {}
            for o in f['openingTimes']:
                for q in OpeningTimes._meta.concrete_fields:

                    if q.name == 'facility':
                        value = facility
                    elif q.name == 'id':
                        continue
                    else:
                        value = o[q.name]

                    if value is not None:
                        tmp4[q.name] = value

                OpeningTimes.objects.create(**tmp4)
```

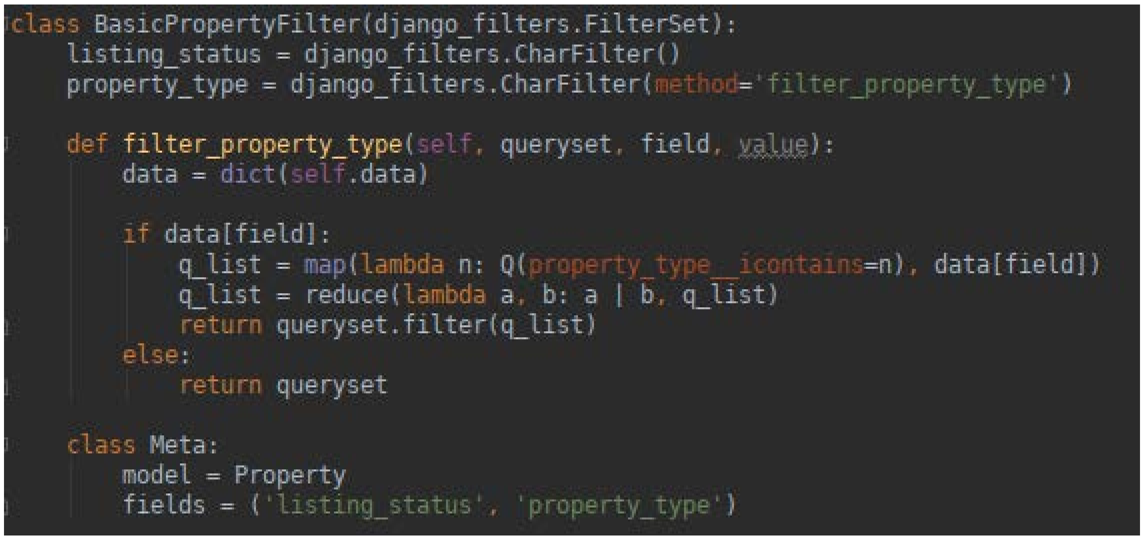
Appendix G.5 – Core

Appendix G.5.1 – Demographics Import method definition

```
def import_demographics_data(csv_path="Core/data/ethnic_group_data.csv"):
    with open(csv_path) as csvreader:
        reader = csv.reader(csvreader, delimiter=',', quotechar='"',)
        i = 0
        for row in reader:
            if i is not 0:
                if i % 1000 == 0:
                    print(row)

                row = [item.strip() for item in row]
                Demographic.objects.create(
                    local_authority_code=row[0],
                    local_authority_name=row[1],
                    age=int(row[3]) if not row[3] == 'All ages' else None,
                    ethnic_group=row[4],
                    population_2018=int(row[12]),
                    population_2019=int(row[13]),
                )

            i += 1
```

Appendix H - Filters

Appendix H.1 Property Filter



```python
class BasicPropertyFilter(django_filters.FilterSet):
    listing_status = django_filters.CharFilter()
    property_type = django_filters.CharFilter(method='filter_property_type')

    def filter_property_type(self, queryset, field, value):
        data = dict(self.data)

        if data[field]:
            q_list = map(lambda n: Q(property_type__icontains=n), data[field])
            q_list = reduce(lambda a, b: a | b, q_list)
            return queryset.filter(q_list)
        else:
            return queryset

    class Meta:
        model = Property
        fields = ('listing_status', 'property_type')
```

Appendix H.2 Room Filter

```python
class RoomFilter(django_filters.FilterSet):
    num_bathrooms = django_filters.RangeFilter()
    num_bedrooms = django_filters.RangeFilter()
    num_floors = django_filters.RangeFilter()
    num_recepts    = django_filters.RangeFilter()

    class Meta:
        model = Property
        fields = ('num_bathrooms', 'num_bedrooms', 'num_floors', 'num_recepts')
```

Appendix H.3 Price Filter

```python
class PriceFilter(django_filters.FilterSet):
    ACCURATE_CHOICES = (
        (0, 'per_month'),
        (1, 'per_week'),
    )

    rentalprice__per_month = django_filters.RangeFilter(method='filter_price') # price_min and
price_max
    rentalprice__per_week = django_filters.RangeFilter(method='filter_price') # price_min and
price_max
    price = django_filters.RangeFilter(method='filter_price')

    accurate = django_filters.ChoiceFilter(choices=ACCURATE_CHOICES, required=False)

    def filter_price(self, queryset, field, value):
        data = dict(self.data)
        if 'listing_status' in data:
            listing_status = data['listing_status']
            filters = {}
            filters['listing_status'] = listing_status
            if value.stop:
                filters[field + '__lte'] = value.stop

            if value.start:
                filters[field + '__gte'] = value.start

            return queryset.filter(**filters)
        else:
            return queryset

    class Meta:
        model = Property
        fields = ['price']
```

Appendix H.4 Area Filter

```
class AreaFilter(django_filters.FilterSet):
    area = django_filters.CharFilter(method="filter_area")

    def filter_area(self, queryset, field, value):
        data = dict(self.data)
        if 'radius' in data:
            lat, lng = value.split(",")
            return queryset.extra(where=["(6367*acos(cos(radians(%s))*"
                                         "cos(radians(latitude))*"
                                         "cos(radians(longitude)-radians(%s))+"
                                         "sin(radians(%s))*sin(radians(latitude)))) < %s" %
(lat, lng, lat, data['radius'])])
        return queryset

    class Meta:
        model = Property
        fields = ('area',)
```

Appendix H.5 School Filter

```
def filter_properties_for_schools(data, qs):

    if 'school' in data:
        property_list = list()
        school_radius = 0.5

        # Schools that match our conditions
        schools = SchoolFilter(data['school'], School.objects.all()).qs.values_list('lat',
'lng')

        ecef_schools = [geodetic2ecef(lat, lon) for lat, lon in schools]
        tree = KDTree(numpy.array(ecef_schools))

        for p in qs:
            l = geodetic2ecef(p.latitude, p.longitude)
            if tree.query_ball_point([l], r=euclidean_distance(school_radius))[0].__len__() >
0:
                property_list.append(p)
                # print(i)

        return property_list
    else:
        return qs
```

Appendix H.6 Commute Filter

```python
def extract_duration(raw_duration="PT24M"):
    """
    Given a duration in TimeDelta format, convert it into minutes.
    :param raw_duration:
    :return:
    """

    # Define the patern
    pattern = re.compile('-?P(\d+Y)?(\d+M)?(\d+D)?T((\d+H)?(\d+M)?(\d+S)?)?')

    # Perform a pattern match on the duration
    matches = pattern.match(raw_duration)

    # Extract hours and minutes.
    hours = matches.groups()[4]
    minutes = matches.groups()[5]

    total = 0

    # Accumulate duration as minutes
    if hours:
        total += 60 * int(hours[:-1])

    if minutes:
        total += int(minutes[:-1])

    return total


def get_reverse_geo_code(lat, lng):
    """
    Fetch the name of an area based on geo-coordinates.
    :param lat:
    :param lng:
    :return:
    """

    # Make the request to the HereMaps API
    r = requests.get(
        'https://reverse.geocoder.api.here.com/6.2/reversegeocode.json?'
        'app_id=%s&app_code=%s&'
        'mode=retrieveAreas&'
        'prox=%s,%s,5' % (HERE_MAPS_APP_ID, HERE_MAPS_APP_CODE, lat, lng))

    # Extract the address.
    address_data = r.json()['Response']['View'][0]['Result'][0]['Location']['Address']

    # Extract the label and postcode.
    label = address_data['Label']
    postcode = address_data['PostalCode']

    # Format the text as one line and return.
    return "%s, %s" % (label, postcode)


def get_route(a, b, mode="publicTransport"):
    """
    Get the fastest route traveling from point a to b
    :return:
    """

    # If the transportation method is public transport
    if mode == 'publicTransport':

        # Substitute the URL parameters
        api_url = ('https://route.api.here.com/routing/7.2/calculateroute.json'
                   '?app_id=%s'
                   '&app_code=%s'
                   '&waypoint0=geo!%s'
                   '&waypoint1=geo!%s'
                   '&departure=now'
                   '&mode=fastest;publicTransport'
                   '&traffic:disabled' % (
                       HERE_MAPS_APP_ID, HERE_MAPS_APP_CODE, "%s,%s" % (a[0], a[1]),
"%s,%s" % (b[0], b[1])))

        # Make the API call.
        print(api_url)
        r = requests.get(api_url)
        print("Performing query")

        # Try and decode the response and create a database cache object.
        try:
            j = r.json()
            commute_time = j['response']["route"][0]['summary']['baseTime']
            RouteCache.objects.create(start_latitude=a[0], start_longitude=a[1],
                                      des_latitude=b[0], des_longitude=b[1],
                                      data=r.text,
                                      commute_time=commute_time)
            return j, commute_time
        except Exception as e:
            print(e)
            return None, None
        # Otherwise return an empty response.


def filter_properties_by_commute(data, qs):
    """
    Take the queryset of locations and return those that meet the commute time requirements.
    :param data:
    :param qs:
    :return:
    """
    if 'commute' in data:
        # properties = random.sample(list(qs), 100)
        # properties = qs[:300]

        # Get a list of all property coordinates from the database
        property_coordinates = qs.values('latitude', 'longitude')
        print("ECEF")

        # Convert each set of coordinates into ecef
        ecef_properties = [geodetic2ecef(x['latitude'], x['longitude']) for x in
property_coordinates]
        print("Commute points")

        # Convert each POI into ecef
        commute_points = [geodetic2ecef(x['position'][0], x['position'][1]) for x in
data['commute']]
```

```
        # Supply the ecef distances into the KDTree as a numpy array (for performance reasons)
        print("Tree")
        tree = KDTree(numpy.array(ecef_properties))

        # Fetch the 300 closest points to the commute points, as they will most likely have
the quickest commute times.
        distance_ar, index_ar = tree.query(commute_points, k=300)
        # print(index_ar)

        # Map over the list of indexes, converting back to int
        print("Extracting points to list")
        index_ar = list((map(int, index_ar[0].tolist() )))

        # Convert the queryset into a numpy array and fetch elements at given index. (quicker
with numpy array)
        print("Index to properties")
        numpy_qs = numpy.array(qs)
        properties = [numpy_qs[i] for i in index_ar]

        # For each POI as defined in the commute filter.
        for l in data['commute']:
            properties_filtered = []
            properties_left = properties

            # Define variables for substitution
            text = l['text']
            position = l['position']
            required_commute_time = l['time'] * 60

            route_requests = []

            conditions = Q()

            # Accumulate conditions
            print("Accumulating")
            for p in properties:
                conditions |= (Q(start_latitude=p.latitude, start_longitude=p.longitude,
                                 des_latitude=position[0], des_longitude=position[1]))

            # Find previously checked properties meeting conditions in the database.
            cache_query = RouteCache.objects.filter(conditions).values()
            for route_cache in cache_query:
                # Find the property that corresponds with the cache object.
                p = list(filter(lambda p: p.latitude == route_cache['start_latitude'] and
                                 p.longitude == route_cache['start_longitude'],
properties_left))

                if p:
                    # If found, remove from the list of properties to check
                    p = p[0]
                    properties_left.remove(p)

                    # Check whether the commute time meets the users needs.
                    if route_cache['commute_time'] <= required_commute_time:
                        # If it does, add the route data to the property object for sending to
the front-end later.
                        data = json.loads(route_cache['data'])
                        if hasattr(p, 'route_data'):
                            p.route_data.append(data)
                        else:
                            p.route_data = [data]

                        properties_filtered.append(p)

            # For each of the properties which do not have a cache route object, a API call is 
performed.
            for p in properties_left:
                # Append each property to check into a list.
                route_requests.append([
                    p,
                    position,
                    required_commute_time,
                ])

            # Call the HereMaps API, performing the task on multiple threads, asynchronously
            with ThreadPoolExecutor(max_workers=50) as pool:
                # Remove None values. None value is returned by properties that don't meet the 
commute time required.
                properties = list(filter(None, list(pool.map(get_route_data, 
route_requests))))
                # Append to list of filtered properties.
                properties += properties_filtered

        return properties

    return qs


def get_route_data(data):
    """
    Given a property (location), call the HereMaps API to retrieve an estimated commute to a 
point (position).
    If the time falls below the threshold (required_commute_time) then append the route data 
to the location and return.
    :param data:
    :return:
    """

    # Extract the parameters
    location, position, required_commute_time = data

    # Perform API call.
    route, expected_commute_time = get_route([location.latitude, location.longitude], 
position)

    # If a commute time is returned, and not None.
    if expected_commute_time:
        # Check whether the estimated commute time meets the commute time needs.
        if expected_commute_time <= required_commute_time:
            # If so, append route data to property object.
            if hasattr(location, 'route_data'):
                location.route_data.append(route)
            else:
                location.route_data = [route]

            return location
```

Appendix I - APIs

Appendix I.1.1 Main API for searching properties.

```python
@csrf_exempt
def property_api(request):
    """
    The main API definition.
    This method calls a series of filters. Each filter decides whether it should be performed or not
    depending on the contents of the request (data).
    :return:
    """

    if request.method == 'POST':

        # Fetch a queryset of all properties.
        queryset = Property.objects.filter(latitude__isnull=False, longitude__isnull=False)

        start = time.time()

        # Load the contents of the request
        data = json.loads(request.body)

        # Perform each of the filters on the set of properties, given the contents of the request.
        print("Basic filter")
        qs = BasicPropertyFilter(data, queryset).qs
        qs = RoomFilter(data, qs).qs
        print("Price filter")
        qs = PriceFilter(data, qs).qs
        print("Area filter")
        qs = AreaFilter(data, qs).qs.distinct()
        print("School filter")
        qs = filter_properties_for_schools(data, qs)
        print("Commute filter")
        qs = filter_properties_by_commute(data, qs)

        # Serialize the data into a dictionary format.
        print("Serializing")
        serialized_data = BasicPropertySerializer(qs, many=True).data

        # Convert the response contents into JSON.
        print("Dumping")
        response = json.dumps(
            {
                'count': qs.__len__(),
                'results': serialized_data
            }
        )

        # Print the number of queries performed to establish whether optimistions can happen.
        print(len(connection.queries))

        # Print the time taken to perform the filters, again for optimisation purposes.
        end = time.time()
        print("time: ", end - start)

        return HttpResponse(response, content_type="json")

    return Http404('Error')
```

Appendix I.1.2 API to fetch an individual property.

```python
def get_property(request, listing_id):
    """
    Return details for a property with a given listing_id.
    :param request:
    :param listing_id:
    :return:
    """

    # Find the property in the database or return a 404 response.
    p = get_object_or_404(Property, listing_id=listing_id)

    # Serialize the property into
    rooms_serialized = PropertyInformationSerializer(p).data

    response = json.dumps(
        rooms_serialized
    )

    return HttpResponse(response, content_type="json")
```

Appendix I.1.3 API for the price histograms

```python
def sale_price_histogram(request):
    """
    Generate a price histogram to be displayed above the price range slider.
    :param request:
    :return:
    """
    data = get_price_histogram()

    response = json.dumps(list(data))

    return HttpResponse(response, content_type="json")


def rental_price_histogram(request):
    """
    Generate a price histogram to be displayed above the price range slider.
    :param request:
    :return:
    """
    data = get_price_histogram(listing_status="rent")

    response = json.dumps(list(data))

    return HttpResponse(response, content_type="json")
```

Appendix I.1.4 Get active places API

```python
def get_active_places(request, latitude, longitude):
    """
    Using the URL parameters, return the 6 nearest locations to the coordinates.
    :param request:
    :param latitude:
    :param longitude:
    :return:
    """
    latitude, longitude = float(latitude), float(longitude)
    knn = 6
    active_places = ActivePlace.objects\
        .exclude(nursery=True)\
        .exclude(name__icontains="school")\
        .exclude(ownerType="Community School")\
        .exclude(name__icontains="(CLOSED)")\
        .values('active_place_id', 'lat', 'lng')

    # Call the generic method to find the nearest locations. A list of indexes will be
returned, with distances.
    locations = get_closest_locations(latitude, longitude, active_places, k=knn)

    # Extract the indexes
    index_list = list(locations[1][0])

    # Convert index to active_place_id
    nearest_locations = [active_places[int(x)]['active_place_id'] for x in index_list]

    # Combine ids with distances, and order based on id.
    loc_dist = sorted(list(zip(nearest_locations, locations[0][0])), key=lambda x: x[0])

    # Fetch all the data for only the 6 nearest locations, will be displayed on front-end
    locations_with_associations = ActivePlace.objects.filter(pk__in=nearest_locations)\
        .prefetch_related('activities', 'facility_set', 'facility_set__openingtimes_set',
'disability')

    # Associate indexes with their corresponding data objects and distances. Sort by distance
ascending.
    sorted_locations = sorted(list(zip(loc_dist, locations_with_associations)), key=lambda  x:
x[0][1])

    # Re-structure that data for serialization
    for ((a, b), obj) in sorted_locations:
        obj.distance = b

    # Serialize and convert to JSON.
    response_data = json.dumps(ActivePlaceSerializer([x[1] for x in sorted_locations],
many=True).data)

    return HttpResponse(
        response_data,
        content_type="json"
    )
```

Appendix I.1.5 API for getting nearest health-care services

```
def get_closest_health_services(request, latitude, longitude):
    latitude, longitude = float(latitude), float(longitude)

    dentists = CQCLocation.objects.filter(location_type__icontains="dentist").values()
    gps = CQCLocation.objects.filter(location_type__icontains="GP").values()
    hospital = CQCLocation.objects.filter(location_type__icontains="hospital").values()

    distance_den, dentist_index = get_closest_location(latitude, longitude, dentists)
    distance_gp, gp_index = get_closest_location(latitude, longitude, gps)
    distance_hos, hospital_index = get_closest_location(latitude, longitude, hospital)

    response = json.dumps(
        {
            'dentist': {
                **CQCLocationSerializer(dentists[dentist_index]).data,
                'distance': round(distance_den, 3)
            },
            'hospital': {
                **CQCLocationSerializer(hospital[hospital_index]).data,
                'distance': round(distance_hos, 3)
            },
            'gp': {
                **CQCLocationSerializer(gps[gp_index]).data,
                'distance': round(distance_gp, 3)
            }
        }
    )
    return HttpResponse(response, content_type="json")
```

Appendix I.1.6 APIs for the HereMaps sub-system

```
def distance_api(request):
    """
    Distance filter API
    :return:
    """
    try:
        data = json.loads(request.body)
        # print(data)
        # {"by_distance": [{"lat": 0, "lng": 0, "max_distance": 20}]}

        # TODO: Create a formset to validate the data...

        # Looping over the distance restrictions as set by the user.
        to_locations = [(l['latitude'], l['longitude']) for l in data['by_distance']]
        print(to_locations)

        properties = Property.objects.filter(pk__in=data['queryset']).values('latitude', 'longitude')
        # print(properties)

        for p in properties:
            from_location = [ (p['latitude'], p['longitude']) ]

            routes = get_routes(from_location, to_locations)
            # Check that the routes satisfy the user time requirements

    except (JSONDecodeError, KeyError):
        return HttpResponse(content="Failed", status=500)

    return HttpResponse(content="success")


def reverse_geo_code(request, lat, lng):
    o = ReverseGeoCodeCache.objects.filter(latitude=lat, longitude=lng)
    if o.count() == 1:
        o = o.get()
    else:
        s = get_reverse_geo_code(lat, lng)
        o = ReverseGeoCodeCache.objects.create(latitude=lat, longitude=lng, label=s)

    response = {'label': o.label}
    return HttpResponse(json.dumps(response))


def search_input(request, search):
    """
    Call the here maps API to return auto complete sugggestions for user input.
    :return:
    """
    # r = requests.get(
    #     'http://autocomplete.geocoder.api.here.com/6.2/suggest.json'
    #     '?app_id=%s'
    #     '&app_code=%s'
    #     '&query=%s' % (HERE_MAPS_APP_ID, HERE_MAPS_APP_CODE, search))
    r = requests.get(
        'https://places.cit.api.here.com/places/v1/autosuggest'
        '?at=51.49,-0.14'
        '&app_id=%s'
        '&app_code=%s'
        '&q=%s' % (HERE_MAPS_APP_ID, HERE_MAPS_APP_CODE, search))
    response = r.json()
    print(response)

    return HttpResponse(json.dumps(response), content_type="json")
```

Appendix I.1.7 API to get near by schools

```
def get_closest_schools(request, latitude, longitude):
    """
    Using the URL parameters, return the 10 nearest locations to the coordinates.
    :param request:
    :param latitude:
    :param longitude:
    :return:
    """
    latitude, longitude = float(latitude), float(longitude)
    knn = 10

    schools = School.objects.values('urn', 'lat', 'lng')

    # Call the generic method to find the nearest locations. A list of indexes will be
returned, with distances.
    locations = get_closest_locations(latitude, longitude, schools, k=knn)

    # Extract the indexes
    index_list = list(locations[1][0])

    # Convert index to urn
    nearest_locations = [schools[int(x)]['urn'] for x in index_list]

    # Combine urns with distances, and order based on urn.
    loc_dist = sorted(list(zip(nearest_locations, locations[0][0])), key=lambda x: x[0])

    # Fetch all the data for only the 10 nearest locations, will be displayed on front-end
    locations_with_associations =
School.objects.filter(urn__in=nearest_locations).order_by('urn')

    # Associate indexes with their corresponding data objects and distances. Sort by distance
ascending.
    sorted_schools = sorted(list(zip(loc_dist, locations_with_associations)), key=lambda  x:
x[0][1])

    # Re-structure that data for serialization
    for ((a, b), obj) in sorted_schools:
        obj.distance = b

    # Serialize and convert to JSON.
    response_data = json.dumps(SchoolSerializer([x[1] for x in sorted_schools],
many=True).data)

    return HttpResponse(
        response_data,
        content_type="json"
    )
```

Appendix I.2 - API Methods

Appendix I.2.1.- get_closest_locations

Formed based on method seen on StackOverFlow, however not coppied.

https://stackoverflow.com/questions/48126771/nearest-neighbour-search-kdtree

```python
def get_closest_locations(latitude, longitude, locations, k=3):
    """
    Given coordinates, find the nearest locations.
    :param latitude:
    :param longitude:
    :param locations:
    :param k:
    :return:
    """

    # Convert to ecef
    ecef_locations = [geodetic2ecef(d['lat'], d['lng']) for d in locations]

    # Create tree
    tree = KDTree(numpy.array(ecef_locations))

    # Convert location coordinates to ecef
    l = geodetic2ecef(latitude, longitude)

    # Retrieve the k nearest nodes/locations
    result = tree.query([l], k=k)

    return result


def get_closest_location(*args):
    result = get_closest_locations(*args, k=1)
    return float(result[0][0]), int(result[1][0])
```

Appendix I.2.2

These methods were copied from StackOverflow

```python
A = 6378.137
B = 6356.7523142
ESQ = 6.69437999014 * 0.001


def geodetic2ecef(lat, lon, alt=0):
    """Convert geodetic coordinates to ECEF."""
    lat, lon = radians(lat), radians(lon)
    xi = sqrt(1 - ESQ * sin(lat))
    x = (A / xi + alt) * cos(lat) * cos(lon)
    y = (A / xi + alt) * cos(lat) * sin(lon)
    z = (A / xi * (1 - ESQ) + alt) * sin(lat)
    return x, y, z


def euclidean_distance(distance):
    """Return the approximate Euclidean distance corresponding to the
    given great circle distance (in km).
    """
    return 2 * A * sin(distance / (2 * B))
```

Appendix I.2.3

```
def coordinates_to_postcode(lat, lon):
    """
    Call an external API to convert a set of coordinates to a postcode.
    :param lat:
    :param lon:
    :return:
    """

    # Substitute coordinate.
    url = "http://api.postcodes.io/postcodes?lon=%s&lat=%s" % (lon, lat)

    # Make the request
    req = requests.get(url)

    # Decode the request as a dictionary from JSON
    data = req.json()

    # Return the first result
    return data['result'][0]['postcode']
```

Appendix J – Tests

Appendix J.1 - Functional Tests

The purpose of this was to ensure that the validation functionality was working correctly and that the default options were being set correctly.

Round 1

Purpose	To test the default appearance and validation functionality of the ‘Listing type’ filter.
Expected behaviour	The default option ‘Rent’ should be preselected and the ‘done’ button should remain enabled regardless of which option is selected.
Actual behaviour	As expected.

Purpose	To test the default appearance and validation functionality of the ‘Price’ filter.
Expected behaviour	The slider should extend all the way by default. The done button should remain enabled regardless of the slider range.
Actual behaviour	As expected.

Purpose	To test the default appearance and validation functionality of the ‘property type’ filter.
Expected behaviour	By default, all options should be selected. At least one item must be selected, otherwise the user shall not be able to proceed.
Actual behaviour	As expected.

Purpose	To test the default appearance and validation functionality of the ‘area’ filter.

Expected behaviour	By default, the 'done' button will remain disabled, preventing the user from saving an invalid option. A map will be shown being center at London, however no points will be plotted. The radius slider will be set to 4KM.
Actual behaviour	As expected.

Purpose	To test the default appearance and validation functionality of the 'room' filter.
Expected behaviour	By default, each of the three sliders should have the full range selected. The 'done' button will always remain activated to allow any combination on room configurations to be searched.
Actual behaviour	As expected.

Purpose	To test the default appearance and validation functionality of the 'commute' filter.
Expected behaviour	By default, the 'done' button should remain disabled until at least one POI is defined and added to the table.
Actual behaviour	Failed. The 'done' button is always enabled. Resolved upon review.

Purpose	To test the default appearance and validation functionality of the 'school' filter.
Expected behaviour	By default, the 'done' button should remain disabled until at least one checkbox for school type is selected. Gender should be selected as 'mixed' and 'Administration type' should be selected as 'Any'.
Actual behaviour	Failed. The 'done' button is always enabled. Resolved upon review.

Purpose	To test the condition associated with the 'Submit' button.
Expected behaviour	By default, 'Submit' button should remain disabled. This button will become enabled once all three of the compulsory filters have been completed.

Actual behaviour	As expected.

Purpose	To test the visibility condition for the ‘additional filter’ button.
Expected behaviour	By default, the button should remain hidden. This button will become visible once all three of the compulsory filters have been completed.
Actual behaviour	As expected.

Appendix J.2 – Unit tests

```python
from _decimal import Decimal

from django.test import TestCase

from Core.methods import get_closest_locations, get_closest_location


class ClosestLocationTestCase(TestCase):
    def setUp(self):
        self.locations = [{'lat': Decimal('51.504807'), 'lng': Decimal('-0.470537')},
                          {'lat': Decimal('51.509476'), 'lng': Decimal('0.005619')},
                          {'lat': Decimal('51.509476'), 'lng': Decimal('0.005619')},
                          {'lat': Decimal('51.509476'), 'lng': Decimal('0.005619')},
                          {'lat': Decimal('51.509476'), 'lng': Decimal('0.005619')},
                          {'lat': Decimal('51.509476'), 'lng': Decimal('0.005619')}]

    def test_get_nearest_one(self):
        """Test that the closest point to a location is returned, with the correct distance in KM"""
        location = (51.608116, -0.163689)
        distance, index = get_closest_location(location[0], location[1], self.locations)

        self.assertEqual(round(distance,2), 16.08)
        self.assertEqual(index, 1)

    def test_get_nearest_two(self):
        """Test that the closest 2 points to a location are returned, with the correct distance in KM"""

        location = (51.608116, -0.163689)
        distance_index_array = get_closest_locations(location[0], location[1], self.locations, 2)

        points = list(zip(distance_index_array[0][0], distance_index_array[1][0]))

        self.assertEqual(round(points[0][0], 2), 16.08)
        self.assertEqual(points[0][1], 1)

        self.assertEqual(round(points[0][0], 2), 16.08)
        self.assertEqual(points[0][1], 1)

        self.assertEqual(points.__len__(), 2)
```

Appendix K – API definitions

The definitions were used to aid the integration process between the back-end and the front-end, when placeholder data was being overwritten.

Request/URL/APP: GET /api/properties (Zoopla)	
Purpose: To filter and then return a list of property locations (latitude, longitude). These locations would then be plotted onto a map.	
Definition: Zoopla/views/api.py	
Input: Filtering criteria as URL parameters.	**Output:** JSON array of properties. Each property only has geo-coordinates and listing_id.

Request/URL/APP: GET **/**api/properties/<int:listing_id>/ (Zoopla)	
Purpose: To return data associated with a specific property. Quicker to fetch details for individual properties than to return all the details in the list of properties in the previous query.	
Definition: Zoopla/views/api.py	
Input: listing_id as a URL parameter.	**Output:** JSON object of property specific data.

Request/URL/APP: GET **/**api/get_sales_histogram/ (Zoopla)	
Purpose: Return data to display a sales-price histogram.	
Definition: Zoopla/views/api.py	
Input: N/A	**Output:** JSON array of object.

Request/URL/APP: GET **/**api/get_rental_histogram/ (Zoopla)	
Purpose: Return data to display a rental-price histogram.	
Definition: Zoopla/views/api.py	
Input: N/A	**Output:** JSON array of object.

Request/URL/APP: GET /api/get_closest_schools/<str:latitude>/<str:longitude>/ (Schools)
Purpose: Return a list of the closest schools to display in a table.
Definition: Schools/views/api.py

Input: Latitude and longitude of area, as URL parameters.	**Output:** A JSON array of 10 objects, corresponding to schools.

Request/URL/APP: GET /api/auto_complete/<str:search>/ (HereMaps)	
Purpose: Return a list of auto complete suggestions. The suggestions are fetched from the HereMaps API.	
Definition: HereMaps/views/api.py	
Input: Search text, as URL parameter.	**Output:** A JSON array of objects. Each object represents a suggestion, containing text and geo-coordinates.

Request/URL/APP: GET /api/reverse-geo-code/<str:lat>/<str:lng>/ (HereMaps)	
Purpose: Return a string representation of the area at a given set of geo-coordinates.	
Definition: HereMaps/views/api.py	
Input: Latitude and longitude, as URL parameter.	**Output:** A JSON object with a string.

Request/URL/APP: GET /api/get_closest_active_places/<str:latitude>/<str:longitude>/ (ActivePlaces)	
Purpose: Return a list of the closest active places to display in a table.	
Definition: ActivePlaces/api.py	
Input: Latitude and longitude, as URL parameter.	**Output:** A JSON array of objects representing locations with information about the facilities.

Request/URL/APP: GET /api/get_closest_health_services/<str:latitude>/<str:longitude>/ (CQC)	
Purpose: Return a list of the closest healthcare facilities to display in a table.	
Definition: CQC/api.py	
Input: Latitude and longitude, as URL parameter.	**Output:** A JSON array of objects representing healthcare facilities with CQC ratings.

Request/URL/APP: GET /api/get_demographics/<str:lat>/<str:lon>/ (Core)

Purpose: Return a list of the closest healthcare facilities to display in a table.	
Definition: Core/api.py	
Input: Latitude and longitude, as URL parameter.	**Output:** JSON response containing values and labels.

Appendix L – Git commits

4f25afc	addil	14/04/2019	Comments in code
43b61d3	addil	11/04/2019	Bug fix
5b2e4b5	addil	09/04/2019	Sxhool filter collapsed view
c544ddd	addil	09/04/2019	Remove filter + further validation changes
5950b5a	addil	09/04/2019	Comments + validation fix
922ca0c	addil	08/04/2019	Clean code.
59f42e6	addil	08/04/2019	Commute filter text changes.
4290561	addil	08/04/2019	Range updated
8313d89	addil	07/04/2019	Updated
1a3dcea	addil	26/03/2019	Added item to menu
c443b0f	addil	26/03/2019	Schools information functionality updated.
d690193	addil	26/03/2019	Styling changes
1f6d687	addil	26/03/2019	Removed unused imports + console logs
56b54dc	addil	26/03/2019	Chart changes
396c265	addil	26/03/2019	Added data sources
ab88c09	addil	26/03/2019	Explore functionality updated
9902a35	addil	25/03/2019	Styling changes
36d617f	addil	23/03/2019	Line colour fix
7a1d593	addil	23/03/2019	Updated code
17abcff	addil	23/03/2019	Optimisations to the commute filter + condensed serializer + migrations
e09e60a	addil	20/03/2019	ActivePlace url hyperlink + finished route API call optimisation
15311f9	addil	20/03/2019	Attempted to refactor code.
d3aeb08	addil	19/03/2019	New commute component + only show polyline once clicked onto commute tab
957c4b4	addil	19/03/2019	Show loader for all information components on mount
aafdb6b	addil	19/03/2019	Added loading animation to information components.
4fb328f	addil	19/03/2019	Disabled tooltip
ef801ae	addil	18/03/2019	process optimisation
cde797a	addil	18/03/2019	Assign more threads
470caaa	addil	18/03/2019	uwsgi config
069ea1d	addil	16/03/2019	Plot locations with tooltips
3e8a796	addil	15/03/2019	Add markers to map on healthcare tab selection
fd41e88	addil	13/03/2019	Added icon + directions
ddd6f9b	addil	13/03/2019	Show all polylines
b9a4372	addil	13/03/2019	Remove points
64823a0	addil	10/03/2019	Re-center map on property select.
43d2d6a	addil	10/03/2019	Map resize
c55c928	addil	09/03/2019	Added back button + split contents and house into individual layers
2fd1ba8	addil	09/03/2019	New layer to allow for fit bounds of custom markers.

9817499	addil	09/03/2019	New home icon + show crime locations on map.
9369a4c	addil	09/03/2019	Created files for commute information
df24809	addil	08/03/2019	Added a static name attribute to each filter class.
e772648	addil	08/03/2019	Added static root
aaa3c38	addil	08/03/2019	Fixed bug with school filter + added uwsgi file.
a5b4988	addil	25/02/2019	Updated requirements file + added information text to crime data.
1184f25	addil	25/02/2019	Crime stats visual changes
23be841	addil	25/02/2019	Added facilities + disabled access
7d184da	addil	25/02/2019	Basic sports facility table
026001d	addil	25/02/2019	Added API for active places + nearest locations method refactor
1f036be	addil	24/02/2019	Data import method.
8a0134c	addil	23/02/2019	Updated property API + added new sub-system.
d015d49	addil	21/02/2019	Demographics API updated.
4b15171	addil	17/02/2019	Get demographics API
1738cdb	addil	17/02/2019	Added restaurant information
49ac24d	addil	17/02/2019	import method + new model.
034ea41	addil	17/02/2019	Added navigation to map component
f6cfe38	addil	14/02/2019	Commute filter updated.
4d6ea02	addil	13/02/2019	Default value updated for price slider.
29cc899	addil	13/02/2019	Health care facilities component updated.
86ca095	addil	13/02/2019	Import script for CQC ratings.
21faaf8	addil	13/02/2019	Basic API to fetch closest health care services.
314c913	addil	13/02/2019	Refactor + added CQC app with data import method + updated postcodes
89cffa5	addil	12/02/2019	Updated figures for rental prices histogram in the price filter
7dbbcff	addil	12/02/2019	Added area chart for sales.
11924ca	addil	12/02/2019	Reposition Zoopla button
5a8fef1	addil	12/02/2019	Added basic chart to crime information card.
d1e45ec	addil	11/02/2019	Crime information component
4d923aa	addil	10/02/2019	Show polylines when clicked onto a marker
bd88dd9	addil	10/02/2019	Property image bug fix + show first 100 properties.
99feac3	addil	10/02/2019	Update commute time filter.
2c33512	addil	10/02/2019	Added route filtering with basic query caching
b44b35e	addil	10/02/2019	Added distinct to filter + updated dropdown to fix bug
287a2d2	addil	09/02/2019	Formatted search box styling + defined getData method in CommuteFilter.
070785e	addil	09/02/2019	Added search API + started creating table for listing markers.
1.41E+08	addil	07/02/2019	Changed distance calculation method to use KD trees instead of nested loops
def64f7	addil	06/02/2019	Started adding school filter backend
6a60240	addil	06/02/2019	Added hot reloading support
67b53d9	addil	05/02/2019	package changes + developed school filter front-end
c3f07be	addil	05/02/2019	Added control layer to show property summary

ec5a8bf	addil	05/02/2019	Price filter fix
2458ad0	addil	04/02/2019	Duplicate property fix + basic popup
607a0d7	addil	03/02/2019	File import + started creating school filter
212116d	addil	03/02/2019	price filter fix
82de76f	addil	03/02/2019	Fixed price filter.
d4950f1	addil	02/02/2019	Area filter changes
95e2782	addil	02/02/2019	Added radius filtering
a0bc55a	addil	02/02/2019	Edit functionality from map.
4024262	addil	02/02/2019	Area filter integration.
a71b1bc	addil	02/02/2019	Styling changes + new serializer.
8ff5a04	addil	01/02/2019	Number format + loading icon.
fb9f911	addil	01/02/2019	Marker clustering + fixed room filter.
af78e57	addil	01/02/2019	React router integration.
dc39cf4	addil	01/02/2019	Filter backend change + added API call.
3.22E+06	addil	01/02/2019	Added on save.
320ec45	addil	31/01/2019	Added getLabel method to area filter.
c8a723e	addil	31/01/2019	Added functionality to return area name for a give coordinate.
6910c6d	addil	30/01/2019	Component clone to refresh props.
f9b07c2	addil	30/01/2019	Added slider + validation.
c669984	addil	30/01/2019	Remove marker + london geojson
cc3c765	addil	30/01/2019	Added map to Area filter with basic click functionality.
0ea9edc	addil	29/01/2019	Fix lock bug
2032f23	addil	29/01/2019	Refactor filter class
ec59dae	addil	29/01/2019	Added loader + fixed bug with data not reloading correctly
ef774fa	addil	29/01/2019	Refactor of the filters class
16478d8	addil	29/01/2019	Enable/disable editing of filters depending on whether a filter is current uncollapsed.
f423495	addil	29/01/2019	Added new price filter class
02a6589	addil	29/01/2019	Added property type filter backend
54d4925	addil	29/01/2019	Django toolbar + combined filter
449ac5b	addil	28/01/2019	Fixed remove filter method.
fc99657	addil	27/01/2019	Requires updating notice + fixed rerender issue.
859e45a	addil	27/01/2019	Store data in parent on save
86a11d7	addil	27/01/2019	Show price and property type filter by default + fixed on select for price term
ca887e6	addil	26/01/2019	Clean up
488d1dc	addil	26/01/2019	Remove filter
71c78e9	addil	26/01/2019	Updated rooms filter
e8bd1ab	addil	26/01/2019	Display the listing status value when the property type filter is collapsed
5fbdfbf	addil	26/01/2019	Allow for filter values to be changed
32cb2a3	addil	24/01/2019	CHanged the filter styling + added action to change button and save
d41280d	addil	24/01/2019	Dynamically add a filter

77b6e22	addil	23/01/2019	Control filters to open from parent class.
16a8c17	addil	23/01/2019	Started moving show attribute to parent
84945a2	addil	22/01/2019	Collapsable filters + generic BaseFilter class.
122e1c4	addil	22/01/2019	Improved filters + updated fontawesome URL
a7b9ec5	addil	17/01/2019	Filter options
b1eb9b7	addil	20/12/2018	Added area filter option.
19a9757	addil	20/12/2018	Developed method further.
6219dce	addil	19/12/2018	Updated price filter + added here maps api
458fa91	addil	16/12/2018	Created map + displayed basic results set.
ab0f201	addil	14/12/2018	Added serializers + filters
8423bec	addil	14/12/2018	Created basic API
8e17dc7	addil	13/12/2018	Refactor + added additional checks
e0058f9	addil	13/12/2018	Finished method to grab data and store into model
5f23869	addil	20/11/2018	Starting to cache
299880c	addil	07/11/2018	Get property method started + front-end changes
00a7fd3	addil	03/11/2018	Zoopla
bc6d288	addil	01/11/2018	Integration with API
1f6a53c	addil	30/10/2018	Basic methods.
d9303e0	addil	08/10/2018	New model to store ofsted data.
9501329	addil	08/10/2018	Added model + extend
7758f02	addil	07/10/2018	Basic react setup
9e07ca8	addil	07/10/2018	Basic react setup
d04a6c9	addil	06/10/2018	Database connection
f976438	addil	06/10/2018	Setup
cd94eb3	addil	04/10/2018	Setup

Appendix M – Requirements documents

Requirements

This document contains requirements that will be implemented into the proposed system. The requirements were devised based on the research gather during the earlier stages.

Functional requirements

The proposed solution:

1. Must allow the user to filter for homes by area.

   The purpose is to help users narrow their search to an area that they may have in mind. Could be implemented in one of two ways:

   - A search box where the user inserts the name of an area, and an autocomplete drop down is shown from which the user can clarify the search.
   - A pin is dropped, around which a circle is drawn. The radius of the circle can be changed by the user. The area below the circle represents the search area.
2. Must allow the user to filter for homes by commute times via public transport.

   The purpose is to allow the user to narrow their search based on desired travel time.

   - Provide the user with a list of max commute times – avoiding free text
   - Allow the user to define multiple points of interest.
   - Use the here maps API to retrieve commute time estimations.
3. Must allow the user to filter for homes by property specific attributes.

   The purpose is to allow the user to eliminate properties that don't meet facility needs.

   - Rooms – by type since the API provides the count of each type of room. Will not be made compulsory since the room-based data provided by the API is not entirely accurate when searching for rooms in a shared property; enforcing the filter may remove such entries.
   - Listing type - compulsory as users don't typically want to look for both types of listings at the same time. Ideally positioned first.
   - Price – compulsory since everybody has a budget in mind.
   - Property type – compulsory, however, select all property types by default as this won't perform the filter.
4. Must display properties on a map, pin pointing each location.

   The purpose is to show the user the exact location of the property so that they can explore nearby facilities when selecting each.

   - Each location might not actually be represented by a pin, but rather a different icon, which is to be determined at the design stage. (Should)
   - Must show contrast from background.
5. Must display crime-based statistics for the area surrounding a property.

   The purpose is to allow the user to evaluate the safeness of the local area.

   - Since police.uk provide an exact location of each crime, they must be plotted on the map.
   - Must display crime type when hovering over each location.
   - Could display a comparison with nearby areas but will require data to be imported.
6. Must display distances from *closest* health-care services such as GP, hospital and dentist.

   The purpose is to allow the user to evaluate whether the property was suitable for them, based on whether the health-care facilities meets their distance and quality needs.

   - Plot on map

- Represent as a table.

7. Must display the *nearest* sporting facilities to a property.

   The purpose was to help those looking to practice a particular type of activity find relevant facilities.
   - Plot on map
   - Represent as a table.
   - The number of locations to display/logic is yet to be determined.
8. Must display commute times for up to 3 points of interest.

   The purpose is to show route specific information for properties that meet the user's commute time needs.
   - Must display commute time and distance.
   - Should display route specific information on select.
9. Must display information regarding the demographics of the borough in which the property is located.

   The purpose is to build an image of the type of people that are living within the local area.
   - Must Implement at least one:
     - Age groups
     - Ethnicities
   - Must use an appropriate graph to display the data.
10. Should display information relating to the closest restaurants to a property.

    The purpose is to show the distance from local places to eat and their ratings.
    - Plot on map
    - Represent as a table.
    - Color code ratings
11. Must allow the user to filter by proximity from schools meeting conditions.

    The purpose is to allow parents/guardians to filter for schools meeting education needs such as education type, gender and admission type.
12. Must display information for schools nearby to property.

    The purpose of this information is to allow parents/guardians to evaluate whether the educational resources are suitable for their child/children.
    - Display as a table.
    - Pin point on map.
13. Must notify the user when data is being loaded using animations and feedback.

    The purpose is to make the system engaging to use.
    - Display a loading icon or text.

Non-functional requirements

14. The UI must be easy and intuitive to use.

    This requirement can be accomplished by:
    - ensuring that inputs are expressed appropriately.
    - avoiding the use of free text fields where possible, as they take longer to fill.
    - spacing information and functionality appropriately, so that the user does not get overloaded with too much information.
    - focusing the user's attention towards actions using colour.

15. The map must show contrast between points of interest and the map (layer) in the background.
16. The system must display results in a reasonable amount of time.

Appendix N – Factor analysis document

Can be downloaded on Moodle from Other Appendices 1

Appendix O – Survey questions + results

Can be downloaded on Moodle from Other Appendices 2

Appendix P – Software documentation

The documentation to access / locally compile the system is provided in Other Appendices 3 on Moodle.

Appendix Q – Project log

03/04/2019
Started conducting the survey.

07/03/2019
Survey approved

02/03/2019
Began testing, ensuring run-errors aren't being raised will filtering data or displaying categorical information, and checking that the filters are being applied correctly with each other.

28/02/2019
Sent back an updated version of the survey. Rephrased the each of the matrices to ask the user about how much they agree with the statement.

26/02/2019
Received feedback for my survey via email. Minor grammatical mistakes and had a concern with the matrices I was using to get users to describe the usefulness of each type of data filter and information category.

24/02/2019
Sent an initial draft of my survey to Jason for feedback.

10/02/2019
Finished implementing the commute filter's basic functionality, to display properties that are located within a certain travel time away from a point of interest. (only by public transport currently) The filter is only applied to the first 100 properties returned, otherwise it can take very long to perform all API calls.

08/02/2019
Renamed distance filter to commute filter and added a search box to it.

07/02/2019
Met with Jason and got some feedback. Suggested I get the questionnaire prepared and approved soon. Also suggested I use hashing to solve my comparison problem.

06/02/2019
Encountering a difficulty whilst trying to calculate distances from schools. Planning to look into the use of numpy or pandas to perform the calculations as the libraries are written in C which have less overhead.

28/01/2019
Got the map showing and is populated with basic marker.

19/01/2019
Started developing use case and class diagrams.
Imported python code to create classes with their attributes.
The code may change over time therefore it will be important for me to keep the diagrams up-to-date.

17/12/2018
Implemented a basic map to display a pin at the location of each property, currently displaying at random locations, limited to 500.

16/12/2018
Finished fetching data, around 130,000 results found.

15/12/2018
Finished defining the Django models in python. Started fetching the data from the API for all 120,000+ listings located in London. Limited to 100 API calls an hour, and there are just over 120,000 listings therefore it will take at least 12 hours, unless I register for a second account to retrieve another API key, halving the fetching time.

14/12/2018
Another issue with the Zoopla API is that it can only provide upto 10,000 results for a query. If I search for the term ‘London’, there are 120,000 properties listed on Zoopla however I am only able to retrieve the first 10,000 listings via the API. Solution to this is to filter the London area by the postcodes that fall under it. E.g UB1, UB2, W7, WC1, etc...

04/12/2018
Will stick with Zoopla and try to fetch as much data as possible and store it.
Rightmove does not provide coordinates for its locations which are important for the solution to calculate distances.

03/12/2018
Discovered that the Zoopla API has been abandoned
https://medium.com/@mariomenti/how-not-to-run-an-api-looking-at-you-zoopla-bda247e27d15
Re-considering to use an alternative method or simply fetch as much data as possible in advanced, limiting searches to a confined area.

01/12/2018
Created trello board to manage tasks

30/11/2018
Booked project progress review for 5pm on 7th December

30/11/2018

Had a group update meeting with Jason and other students that he supervises. Discussed what to expect for the project progress review next week.

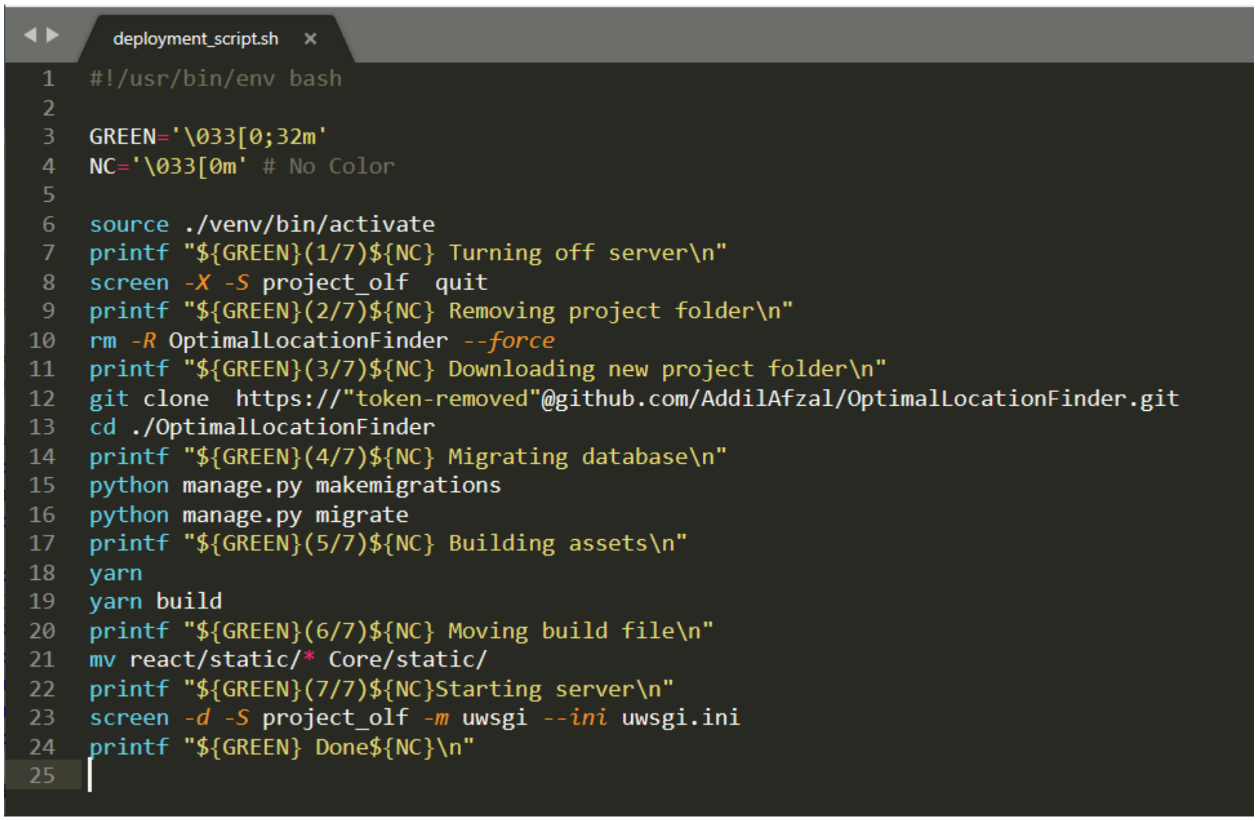
30/10/2018
Began modeling classes and associations in Django.

07/10/2018
Zoopla API issue seems to have reappeared.
The issue is that the API key gets disabled. A problem because I don’t want to have to get a new one each time I’m testing my solution.

05/10/2018
Started experiences issues with the Zoopla API, created a new account and the issue seems to be resolved.

03/10/2018
Obtained Zoopla API key and began testing for possible uses.

Appendix R – Deployment script



deployment_script.sh ×

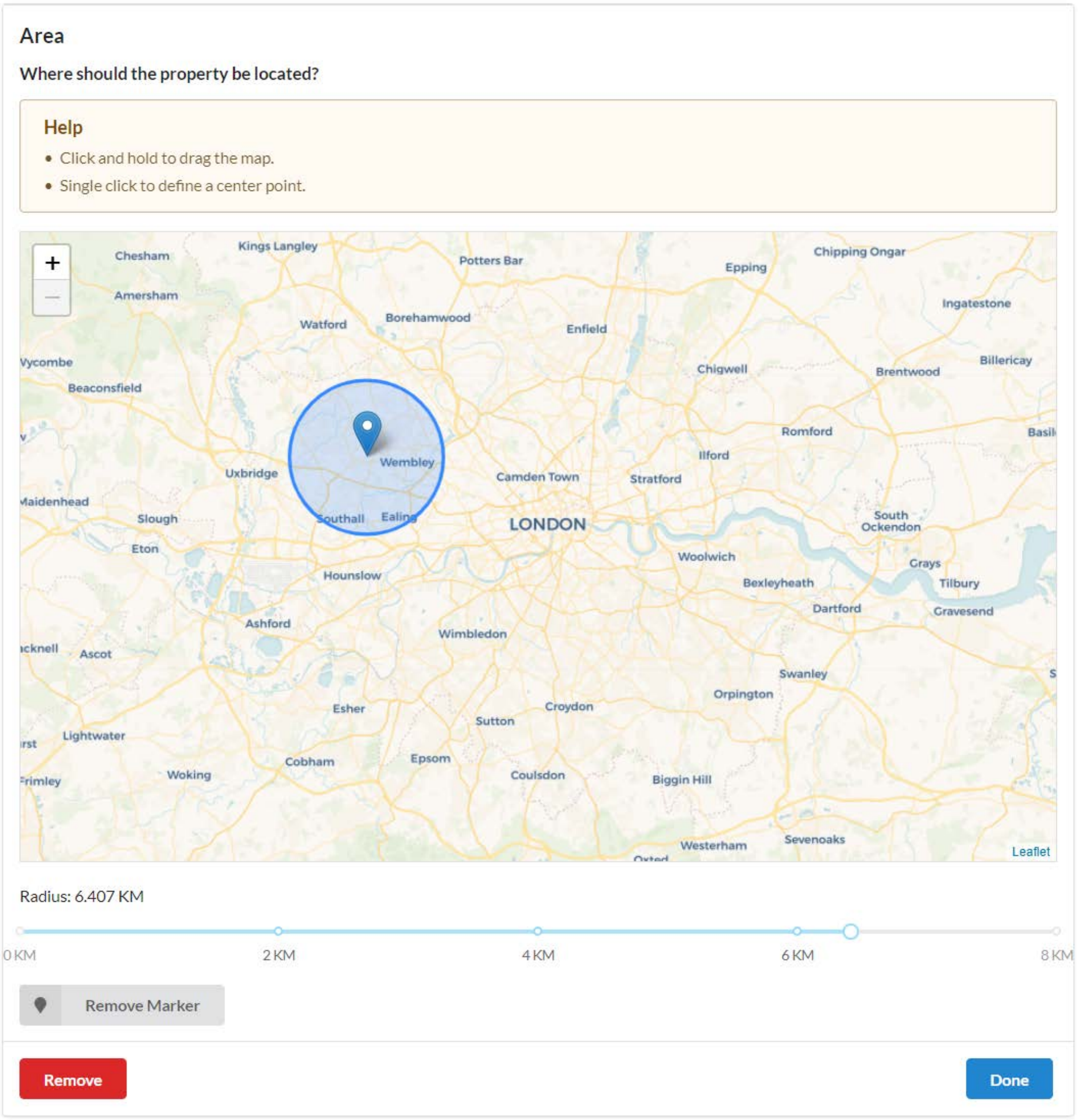
```bash
#!/usr/bin/env bash

GREEN='\033[0;32m'
NC='\033[0m' # No Color

source ./venv/bin/activate
printf "${GREEN}(1/7)${NC} Turning off server\n"
screen -X -S project_olf  quit
printf "${GREEN}(2/7)${NC} Removing project folder\n"
rm -R OptimalLocationFinder --force
printf "${GREEN}(3/7)${NC} Downloading new project folder\n"
git clone  https://"token-removed"@github.com/AddilAfzal/OptimalLocationFinder.git
cd ./OptimalLocationFinder
printf "${GREEN}(4/7)${NC} Migrating database\n"
python manage.py makemigrations
python manage.py migrate
printf "${GREEN}(5/7)${NC} Building assets\n"
yarn
yarn build
printf "${GREEN}(6/7)${NC} Moving build file\n"
mv react/static/* Core/static/
printf "${GREEN}(7/7)${NC}Starting server\n"
screen -d -S project_olf -m uwsgi --ini uwsgi.ini
printf "${GREEN} Done${NC}\n"

```

Appendix S – React Code and UI of filters

Appendix S.1 Area Filter

```
import React, {Component, Fragment} from "react";
import {
    Header, Form, Label, Button, Message
} from 'semantic-ui-react'

import BaseFilter from "./BaseFilter";
import {Circle, CircleMarker, Map, Marker, Popup, TileLayer} from 'react-leaflet'
import Slider from 'rc-slider';

export default class AreaFilter extends BaseFilter {
    constructor(props) {
        super(props);

        this.state = {
            ...this.state,
            area: null,
            londonBoundary: null,

            mapZoom: 10,
            mapCenterPosition: [51.49, -0.14],
            mapDragging: true,

            markerShow: false,
            markerPosition: null,
            markerRadius: 4000, // in M
            label: null,
        };

    }

    // componentWillMount() {
    //     fetch("/static/json/greaterlondon.json")
    //         .then(response => response.json())
    //         .then(response => this.setState({londonBoundary: response.features}));
    //
    // }

    static filter_name = "Area filter";
    static description = "Filter the list of homes to be located within a specific area.";

    componentDidMount() {
        super.componentDidMount();
        let area = this.props.data.area;
        if(area) {
            const tmp = area.area.split(",");
            this.state.markerPosition = {lat: tmp[0], lng: tmp[1]};
            this.state.markerRadius = area.radius*1000;
            this.state.markerShow = true;
            this.save();
        }
    }

    getData = () => {
        let {area, markerRadius, markerPosition} = this.state;
        return {
            'area': {
                area: markerPosition.lat + "," + markerPosition.lng,
                radius: markerRadius/1000
            },
        };
    };

    getCollapsedText = () => {
        const {markerRadius, label} = this.state;
        return (
            <Fragment>
                <h3>Area</h3>
                <p><i className="fas fa-map-marker-alt"/> Within a {markerRadius / 1000} KM
radius from {label}</p>
            </Fragment>
        )
    };

    getLabel = async (lat, lng) => {
        let x = await fetch(`/api/reverse-geo-code/${lat}/${lng}`)
            .then((r) => r.json())
            .then((x) => x.label)

        return x;
    };

    mapOnClick = (e) => {
        this.setState({markerPosition: e.latlng, markerShow: true});
        console.log(this.mapRef)
        console.log(e)
    };

    mapOnDrag = (e) => {
        console.log(e)
    };

    mapRef = React.createRef();

    onSave = async () => {
        if(this.state.label === null) {
            let label = await this.getLabel(this.state.markerPosition.lat,
this.state.markerPosition.lng);
            this.setState({label});
        }
    };

    renderBody() {
        const {
            area, mapCenterPosition, mapZoom,
            markerShow, markerPosition, markerRadius
        } = this.state;

        return (
            <Fragment>
                <h3>Area</h3>
                <Form.Group inline>
                    {/*<label>Where should the property be situated?</label>*/}
                    <Header size='small'>Where should the property be located?</Header>
                </Form.Group>
                <Message warning>
                    <Message.Header>Help</Message.Header>
                    <Message.List>
```

```
                        <Message.Item>Click and hold to drag the map.</Message.Item>
                        <Message.Item>Single click to define a center point.</Message.Item>
                    </Message.List>
                </Message>
                <Map ref={this.mapRef}
                    // maxBounds={null}
                     center={mapCenterPosition}
                     minZoom={10}
                     zoom={mapZoom}
                     onClick={this.mapOnClick}
                     dragging={true}
                     onDrag={this.mapOnDrag}
                     style={{height: 550, border: "1px solid #ddd", padding: 26, marginTop:
16}}>
                    <TileLayer

url="https://a.basemaps.cartocdn.com/rastertiles/voyager/{z}/{x}/{y}.png"
                    />
                    {(markerShow && markerPosition) && [<Marker position={markerPosition}
draggable={true}/>,
                        <Circle center={markerPosition} radius={markerRadius}/>]}
                </Map>
                <br/>
                <p>Radius: {this.state.markerRadius / 1000} KM</p>
                <Slider
                    min={1}
                    max={8000}
                    defaultValue={markerRadius}
                    marks={{0: "0 KM", 2000: "2 KM", 4000: "4 KM", 6000: "6 KM", 8000: "8
KM"}}
                    onChange={(e) => this.setState({markerRadius: e})}/>
                <br/>
                <br/>
                <Button.Group vertical labeled icon>
                    {markerPosition &&
                    <Button icon='marker' content='Remove Marker'
                            onClick={() => this.setState({markerPosition: null})}
                            disabled={!this.state.markerPosition}/>}
                </Button.Group>

            </Fragment>
        )
    }

    isValid = () => {
        return this.state.markerPosition && this.state.markerRadius;
    };
}
```



Area

Where should the property be located?

Help
- Click and hold to drag the map.
- Single click to define a center point.

Radius: 6.407 KM

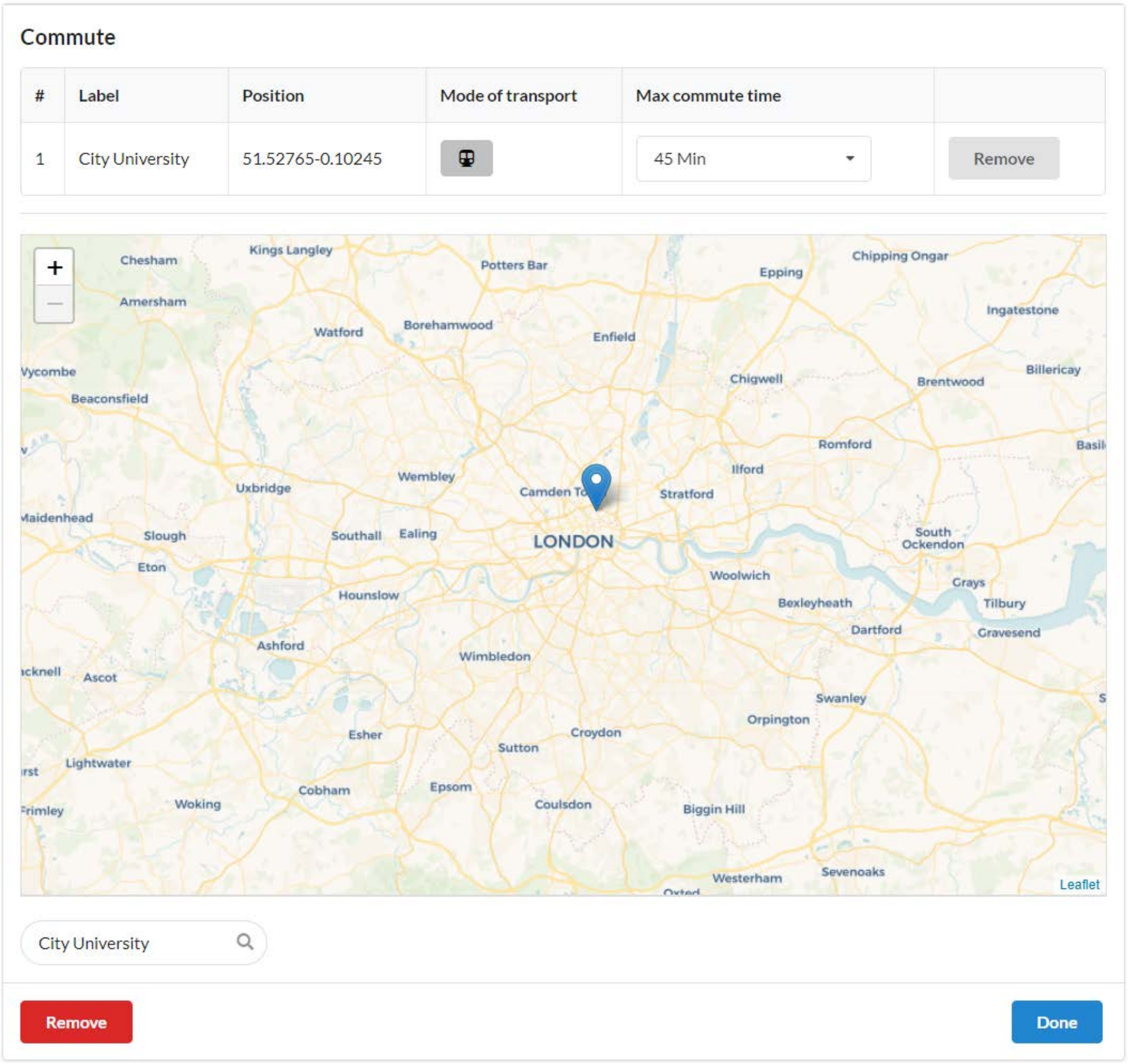
Appendix S.2 BaseFilter

```
import React, {Component, Fragment} from "react";
import {
    Button, Dimmer, Loader, Message, Segment
} from 'semantic-ui-react'


export default class BaseFilter extends Component {
    constructor(props) {
        super(props);

        this.state = {};

        if (new.target === BaseFilter) {
            throw new TypeError("Cannot construct BaseFilter instances directly");
        }

        this.state.canRemove = true;
        this.state.collapse = false;
        this.state.needsReview = false;
        this.state.showLoader = true;

        props.enableLock();
    }

    componentDidMount() {
        setTimeout(() => this.setState({showLoader: false}),500);
    }

    removeFilter = () => {
        this.props.removeFilter(this);
        this.props.disableLock();
    };

    isValid = () => {
        return true;
    };

    renderCollapsed() {
        let reviewMessage =
          ( <Message warning>
                <Message.Header>Requires updating</Message.Header>
                <p>
                    A change to another filter means this will need to be updated.
                </p>
            </Message>);
        return (
            <Fragment>
                <Segment secondary={this.props.lock}>
                    {this.state.needsReview && reviewMessage}
                    {this.getCollapsedText()}
                </Segment>
                <Segment secondary={this.props.lock}>
                    <Button.Group labeled icon>
                        <Button icon="edit" href="javascript:void(0)"
onClick={this.unCollapse} primary
                                disabled={this.props.lock} content={"Change"}  compact/>
                        {this.state.canRemove &&
                        <Button icon={"trash"} href="javascript:void(0)"
onClick={this.removeFilter} color={"red"}
```

```
                                disabled={this.props.lock} compact content={"Remove"}/>}
                    </Button.Group>
                </Segment>
            </Fragment>)
    }

    render() {
        return (
            <Segment.Group>
                <Dimmer inverted active={this.state.showLoader}>
                    <Loader inverted content='Loading'/>
                </Dimmer>
                {this.state.collapse ? this.renderCollapsed() :
                    [<Segment> {this.renderBody()}</Segment>,
                    <Segment> {this.renderSave()}</Segment>]}
            </Segment.Group>)
    }

    collapse = () => {
        this.setState({collapse: true})
    };

    unCollapse = () => {
        this.props.enableLock();
        this.setState({collapse: false, needsReview: false})
    };

    save = () => {
        this.collapse();
        this.props.reloadData();
        this.props.disableLock();
        if (this.props.hasOwnProperty("onFirstValid") &&
!this.hasOwnProperty('doneOnFirstValid')) {
            this.props.onFirstValid();
            this.doneOnFirstValid = true;
        }

        if(this.hasOwnProperty("onSave")) {
            this.onSave();
        }
    };

    renderSave() {
        return (
            <Fragment>
                { this.state.canRemove && <div style={{float: 'left', display: 'inline'}}>
                    <Button onClick={this.removeFilter} color="red">Remove</Button>
                </div> }
                <div style={{textAlign: 'right', display: 'block'}}>
                    <Button onClick={this.save} primary
disabled={!this.isValid()}>Done</Button>
                </div>

            </Fragment>
        )
    }
}
```

Appendix S.3 CommuteFilter

```
import React, {Component, Fragment} from "react";
import {
    Header, Form, Label, Button, Divider, Search
} from 'semantic-ui-react'

import BaseFilter from "./BaseFilter";
import {Circle, Map as LeafletMap, Map, Marker, TileLayer} from "react-leaflet";
import * as debounce from "debounce";
import Control from 'react-leaflet-control';
import ControlSearch from "./CommuteFilter/ControlSearch";
import MarkerClusterGroup from 'react-leaflet-markercluster';
import MarkerTable from "./CommuteFilter/MarkerTable";


export default class CommuteFilter extends BaseFilter {
    constructor(props) {
        super(props);

        this.state = {
            ...this.state,

            mapZoom: 10,
            mapCenterPosition: [51.49, -0.14],
            mapDragging: true,
            mapMarkers: [],

            markerShow: false,
            markerPosition: null,
            markerRadius: 4000, // in M
        };
    }

    static filter_name = "Commute filter";
    static description = "Filter for homes that match your ideal commute time.";

    getData = () => {
        const formatMarkerData = (x) => {
            const {text, position, time} = x.props;
            return {
                text,
                position,
                time,
            }
        };

        let markerData = this.state.mapMarkers.map(formatMarkerData);
        return {'commute': {'commute': markerData}};
    };


    componentDidMount() {
        super.componentDidMount();
        let commute = this.props.data.commute;

        if (commute) {
            console.log(commute)
            this.state.mapMarkers = commute.commute.map(x =>
                <Marker key={Math.random()} ref={React.createRef()} draggable={false} {...x}/>);
            this.save();
        }
    }

    getCollapsedText = () => {
        return (
            <Fragment>
                <h3>Commute</h3>
                <MarkerTable markers={this.state.mapMarkers} edit={false}/>
            </Fragment>
        )
    };

    setBounds = async () => {
        let markerClusterBounds = this.markerCluster.current.leafletElement.getBounds();
        this.mapRef.current.leafletElement.fitBounds(markerClusterBounds)
    };

    addMarker = (marker) => {
        const mapMarkers = [...this.state.mapMarkers, marker];
        this.setState({mapMarkers});
        // this.setBounds();
    };

    updateMarker = (key, new_props) => {
        let marker = this.state.mapMarkers.find(m => m.key === key);
        marker = React.cloneElement(marker, new_props);
        this.setState( {mapMarkers: [...this.state.mapMarkers.filter(x => x.key !== key), marker]});
    };

    removeMarker = (key) => {
        this.setState( {mapMarkers: [...this.state.mapMarkers.filter(x => x.key !== key)]});
    };

    isValid = () => {
        return this.state.mapMarkers.length > 0;
    };

    mapRef = React.createRef();
    markerCluster = React.createRef();

    renderBody() {
        const {mapCenterPosition, mapZoom, mapMarkers} = this.state;

        return (
            <Fragment>
                <h3>Commute</h3>
                <MarkerTable markers={mapMarkers} edit={true}
                             updateMarker={this.updateMarker} removeMarker={this.removeMarker}/>
                <Divider/>
                <Map ref={this.mapRef}
                     maxZoom={16}
                     center={mapCenterPosition}
                     minZoom={10}
                     zoom={mapZoom}
                     dragging={true}
                     style={{height: 550, border: "1px solid #ddd", padding: 26, marginTop: 16}}>
                    <TileLayer
                        url="https://a.basemaps.cartocdn.com/rastertiles/voyager/{z}/{x}/{y}.png"
                    />
                    {mapMarkers}
                </Map>
                <br/>
                <ControlSearch markers={mapMarkers} addMarker={this.addMarker}/>

            </Fragment>
        )
    }
}
```



Commute

#	Label	Position	Mode of transport	Max commute time	
1	City University	51.52765-0.10245		45 Min	Remove

City University

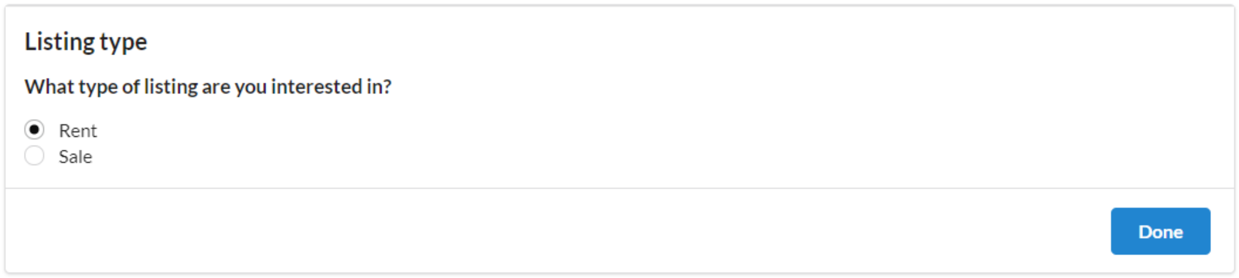
Remove

Done

Appendix S.4 ListingTypeFilter

```
import React, {Fragment} from "react";
import {
    Header, Form
} from 'semantic-ui-react'

import BaseFilter from "./BaseFilter";


export default class ListingTypeFilter extends BaseFilter {
    constructor(props) {
        super(props);

        this.state.canRemove = false;
        this.state.listing_status = "rent";

    }

    componentDidMount() {
        super.componentDidMount();
        let listingType = this.props.data.listingType;
        if(listingType) {
            this.state.listing_status = listingType.listing_status;
            this.save();
        }
    }

    static filter_name = "Listing type filter";
    static description = "Type of listing...";

    getCollapsedText = () => {
        let {listing_status} = this.state;
        return (
            <Fragment>
                <h3>Listing type</h3>
                <p>{listing_status && (listing_status.charAt(0).toUpperCase() +
listing_status.slice(1))}</p>
            </Fragment>
        )
    };

    handleChangeListingStatus = (e, {value}) => {
        this.setState({listing_status: value});
    };

    getData = () => {
        return {'listingType':
                {'listing_status': this.state.listing_status}
        };
    };

    renderBody() {
        const {listing_status} = this.state;

        return (
            <Fragment>
                <h3>Listing type</h3>
                <Header style={{marginTop: 0}} size='small'>What type of listing are you
interested in?</Header>
                <Form.Group inline>
```

```
                    <Form.Radio
                        label='Rent'
                        value='rent'
                        checked={listing_status === 'rent'}
                        onChange={this.handleChangeListingStatus}
                    />
                    <Form.Radio
                        label='Sale'
                        value='sale'
                        checked={listing_status === 'sale'}
                        onChange={this.handleChangeListingStatus}
                    />
                </Form.Group>
            </Fragment>

        )
    }

    isValid = () => {
        return !!(this.state.listing_status)
    };
}
```



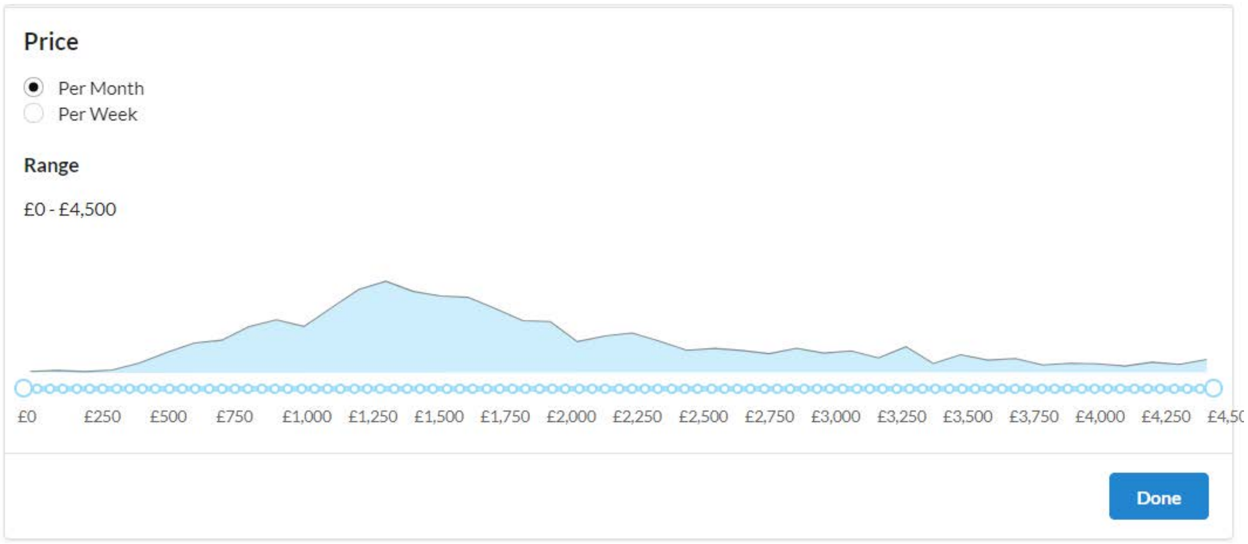
Listing type

What type of listing are you interested in?

● Rent
○ Sale

Appendix S.5 PriceFilter

```
import React, {Component, Fragment} from "react";
import {
    Header, Form, Label, Divider, Segment, Button
} from 'semantic-ui-react'

import BaseFilter from "./BaseFilter";
import {Range} from "rc-slider";
import {Area, AreaChart, ResponsiveContainer, Tooltip, XAxis, YAxis} from "recharts";

function formatCurrency(i) {
    let s = new Intl.NumberFormat('en-GB', {style: 'currency', currency: 'GBP'}).format((i));
    return s.substring(0, (s.length) - 3)
}


function salesPriceRange() {
    let values = {};

    for (let i = 0; i <= 60; i += 1) {
        if(i % 5 === 0) {
            values[i] = formatCurrency(i * 25000);
        } else {
            values[i] = "";
        }

    }
    return values;
}
function rentalPriceRange() {
    let values = {};

    for (let i = 0; i <= 90; i += 1) {
        if(i % 5 === 0) {
            values[i] = formatCurrency(i * 50);
        } else {
            values[i] = "";
        }

    }
    return values;
}

export default class PriceFilter extends BaseFilter {
    constructor(props) {
        super(props);

        this.state = {
            ...this.state,
            canRemove: false,
            term: 'month', // Week/Month
            listingType: props.data.listingType.listing_status,
            chartData: null,
        }

        this.state.price = this.state.listingType === 'sale' ? [0, 1500000] : [0, 4500];
    }

    static description = "price...";

    getData = () => {
        let { price, term, listingType } = this.state;

        let tmp = {};
        if(listingType === 'rent') {
            tmp[`rentalprice__per_${term}_min`] = price[0];
            tmp[`rentalprice__per_${term}_max`] = price[1];
        } else {
            tmp['price_min'] = price[0];
            tmp['price_max'] = price[1];
        }

        return {
            'price': tmp
        };
    };

    componentDidMount() {
        super.componentDidMount();
        let price = this.props.data.price;
        let listingType = this.props.data.listingType;

        if(price && listingType && listingType.listing_status === 'rent') {
            const {rentalprice__per_month_min,
                rentalprice__per_month_max,
                rentalprice__per_week_min,
                rentalprice__per_week_max,
                } = price;

            this.state.price = [rentalprice__per_month_min | rentalprice__per_week_min,
                rentalprice__per_month_max | rentalprice__per_week_max];
            this.state.term = rentalprice__per_week_max !== undefined ? 'week': 'month';
            this.save();
        } else if(price && listingType && listingType.listing_status === 'sale') {
            this.state.price = [price.price_min, price.price_max];
            this.state.listingType = listingType.listing_status;
            this.save();
        }
        this.fetchChartData()
    }

    componentWillReceiveProps(nextProps, nextContext) {
        // console.log(nextProps, this.state)
        if(nextProps.data.listingType.listing_status !== this.state.listingType) {
            this.setState({
                listingType: nextProps.data.listingType.listing_status,
                price: [0, 0],
                term: null,
                needsReview: true
            }, this.fetchChartData);
        }
    }

    getCollapsedText = () => {
        let { price, term, listingType } = this.state;
        return (
            <Fragment>
                <h3>Price</h3>
                <p>{formatCurrency(price[0])} - {formatCurrency(price[1])}{listingType === 'rent' && '/' +
term}</p>
            </Fragment>
        )
    };

    fetchChartData = async () => {
        const tmp = this.state.listingType === 'sale' ? 'get_sales_histogram' : 'get_rental_histogram';
        const chartData = await fetch(`/api/${tmp}/`).then(x => x.json());
        await this.setState({chartData});
    };

    handleChangePriceTerm = (elmm, a) => {
        this.setState({term: a.value})
    };

    renderBody() {
        const {term, price, listingType, chartData} = this.state;
        return (
            <Fragment>
                <h3>Price</h3>

                {listingType === 'rent' && [<Form.Group inline>
                    <Form.Radio
                        label='Per Month'
                        value='month'
                        checked={term === 'month'}
                        onChange={this.handleChangePriceTerm}
                    />
                    <Form.Radio
                        label='Per Week'
                        value='week'
                        checked={term === 'week'}
                        onChange={this.handleChangePriceTerm}
                    />
                </Form.Group>]}
                <br/>
                <Header style={{marginTop: 0}} size='small'>Range</Header>
                {formatCurrency(price[0])} - {formatCurrency(price[1])}
                <br/>
                <br/>

                {chartData &&
                <div style={{height: 100}}>
                    <ResponsiveContainer>
                        <AreaChart
                            hover={false}
                            label={false}
                            data={chartData}
                            // margin={{top: 20, right: 20, bottom: 20, left: 20,}}
                        >
                            {/*<XAxis dataKey="price"/>*/}
                            {/*<YAxis dataKey="value"/>*/}
                            <Area dataKey="value" label="Price" stroke="#838d92" fill="#abe2fb"/>
                            {/*<Tooltip/>*/}
                        </AreaChart>
                    </ResponsiveContainer>
                </div>}

                <Range
                    defaultValue={[0,0]}
                    value={price.map(x => x/(listingType === 'rent' ? 50 : 25000))}
                    step={1}
                    max={listingType === 'rent' ? 90 : 60}
                    onChange={(price) => this.setState({price: price.map(x => x * (listingType === 'rent'
? 50 : 25000))})}
                    marks={listingType === 'rent' ? rentalPriceRange() : salesPriceRange()}
                />
                <Divider horizontal/>
            </Fragment>
        )
    }
}
```



Price

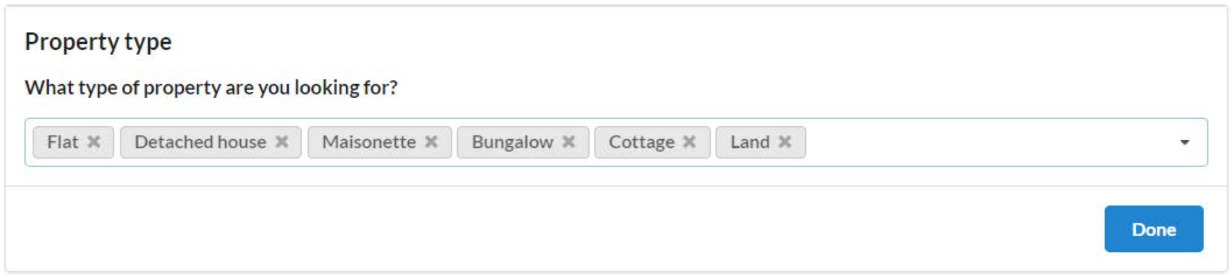
● Per Month
○ Per Week

Range

£0 - £4,500

Appendix S.6 PropertyTypeFilter

```
import React, {Component, Fragment} from "react";
import {
    Button, Header, Form, Dropdown, List
} from 'semantic-ui-react'

import {Segment} from 'semantic-ui-react'
import BaseFilter from "./BaseFilter";

const options = [
  { key: 'flat', text: 'Flat', value: 'Flat' },
  { key: 'terraced_house', text: 'Terraced house', value: 'Terraced house' },
  { key: 'semi-detached_house', text: 'Semi-detached house', value: 'Semi-detached house' },
  { key: 'detached_house', text: 'Detached house', value: 'Detached house' },
  { key: 'studio', text: 'Studio', value: 'Studio' },
  { key: 'maisonette', text: 'Maisonette', value: 'Maisonette' },
  { key: 'bungalow', text: 'Bungalow', value: 'Bungalow' },
  { key: 'cottage', text: 'Cottage', value: 'Cottage' },
  { key: 'land', text: 'Land', value: 'Land' },

];

export default class PropertyTypeFilter extends BaseFilter {
    constructor(props) {
        super(props);

        this.state.canRemove = false;
        this.state.propertyTypes = options.map(x => x.value);
    }

    componentDidMount() {
        super.componentDidMount();

        this.setState({options});

        let propertyTypes = this.props.data.propertyTypes;
        if(propertyTypes) {
            this.state.propertyTypes = propertyTypes.property_type;
            this.save();
        }
        // this.setState({propertyTypes: ['Flat']});
    }

    static filter_name = "Property type filter";
    static description = "Type of property...";


    getCollapsedText = () => {
        let {propertyTypes} = this.state;
        return (
            <Fragment>
                <h3>Property type</h3>
                  <List bulleted horizontal>
                      { propertyTypes.map((value, index) => <List.Item key={index}
as='span'>{value}</List.Item>)}
                  </List>
                {/*<p>{propertyTypes.reduce((x, i) => x + ", " + i)}</p>*/}
            </Fragment>
        )
    };

    getData = () => {
        return {'propertyTypes': {'property_type': this.state.propertyTypes}};
    };

    renderBody() {
        const {propertyTypes} = this.state;

        return (
            <Fragment>
                <h3>Property type</h3>
                <Header style={{marginTop: 0}} size='small'>What type of property are you
looking for?</Header>
                <Dropdown
                    placeholder='Property type'
                    options={options}
                    value={propertyTypes}
                    onChange={(a, b) => this.setState({propertyTypes: b.value})}
                    fluid multiple selection />
            </Fragment>
        )
    }

    isValid = () => {
        return (this.state.propertyTypes).length > 0
    };
}
```

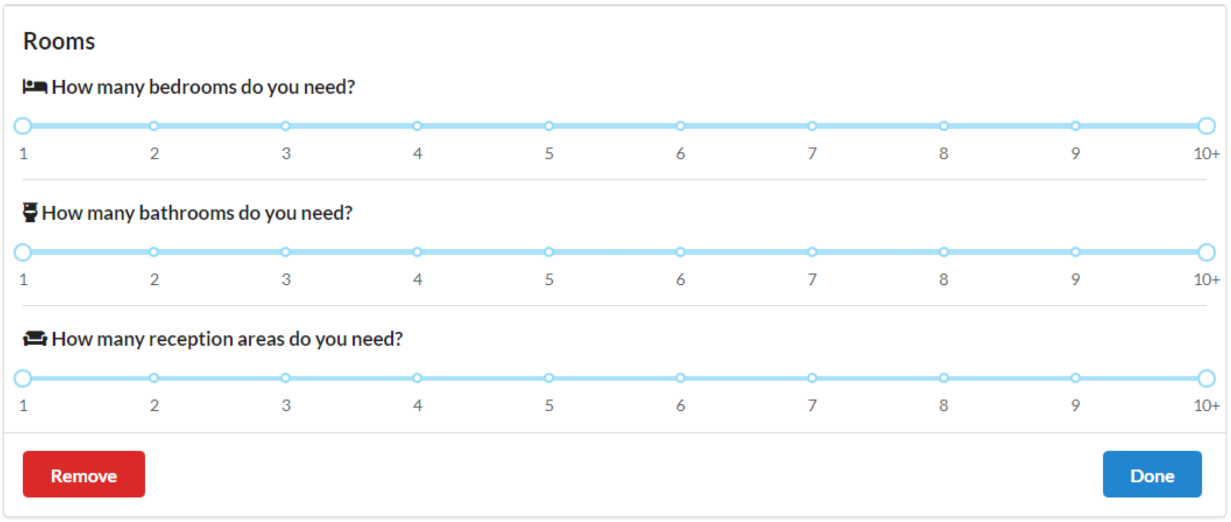

Property type

What type of property are you looking for?

Appendix S.7 RoomFilter

```
import React, {Component, Fragment} from "react";
import {
    Header, Divider
} from 'semantic-ui-react'

import BaseFilter from "./BaseFilter";
import {Range} from "rc-slider";

function bedroomsRange() {
    let values = {};

    for (let i = 1; i <= 10; i += 1) {
        values[i] = (i == 10) ? i + "+" : i;
    }

    return values;
}

export default class RoomsFilter extends BaseFilter {
    constructor(props) {
        super(props);

        this.state.bedrooms = {min: 0, max: 0};
        this.state.bathrooms = {min: 0, max: 0};
        this.state.receptions = {min: 0, max: 0};
    }

    static filter_name = "Rooms filter";
    static description = "Select the number of each type of room needed.";

    componentDidMount() {
        super.componentDidMount();

        let rooms = this.props.data.rooms;
        if(rooms) {
            console.log("updating rooms")
            this.state.bedrooms = {min: rooms.num_bedrooms_min, max: rooms.num_bedrooms_max};
            this.state.bathrooms = {min: rooms.num_bathrooms_min, max: rooms.num_bathrooms_max};
            this.state.receptions = {min: rooms.num_recepts_min, max: rooms.num_recepts_max};
            this.save();
        }
    }

    getData = () => {
        let { bedrooms, bathrooms, receptions} = this.state;
        return {
            'rooms': {
                num_bathrooms_min: bathrooms.min,
                num_bathrooms_max: bathrooms.max,
                num_recepts_min: receptions.min,
                num_recepts_max: receptions.max,
                num_bedrooms_min: bedrooms.min,
                num_bedrooms_max: bedrooms.max,
            }
        };
    };

    getCollapsedText = () => {
        const {bedrooms, bathrooms, receptions} = this.state;
        return (
            <Fragment>
                <h3>Rooms</h3>
                {bedrooms.min && <p><i className="fas fa-bed"/> Bedrooms {bedrooms.min}-
{bedrooms.max}</p>}
                {bathrooms.min && <p><i className="fas fa-toilet"/> Bathrooms {bathrooms.min}-
{bathrooms.max}</p>}
```

```
                {receptions.min && <p><i className="fas fa-couch"/> Receptions {receptions.min}-
{receptions.max}</p>}
            </Fragment>
        )
    };

    renderBody() {
        const {bedrooms, bathrooms, receptions} = this.state;

        return (
            <Fragment>
                <h3>Rooms</h3>
                <Header style={{marginTop: 0}} size='small'><i className="fas fa-bed"/> How many bedrooms
do you
                    need?</Header>
                <Range
                    defaultValue={bedrooms.min === 0 && bedrooms.max === 0 ? [1,10] : [bedrooms.min,
bedrooms.max]}
                    step={1}
                    min={1}
                    max={10}
                    marks={bedroomsRange()}
                    onChange={(a) => this.setState({bedrooms: {min: a[0], max: a[1]}})}
                />

                <br/>
                <Divider/>
                <Header style={{marginTop: 0}} size='small'><i className="fas fa-toilet"/> How many
bathrooms do
                    you need?</Header>
                <Range
                    defaultValue={bathrooms.min === 0 && bathrooms.max === 0 ? [1,10] : [bathrooms.min,
bathrooms.max]}
                    step={1}
                    min={1}
                    max={10}
                    marks={bedroomsRange()}
                    onChange={(a) => this.setState({bathrooms: {min: a[0], max: a[1]}})}
                />

                <br/>
                <Divider/>
                <Header style={{marginTop: 0}} size='small'><i className="fas fa-couch"/> How many
reception
                    areas do you need?</Header>
                <Range
                    defaultValue={receptions.min === 0 && receptions.max === 0 ? [1,10] : [receptions.min,
receptions.max]}
                    step={1}
                    min={1}
                    max={10}
                    marks={bedroomsRange()}
                    onChange={(a) => this.setState({receptions: {min: a[0], max: a[1]}})}
                />
                <br/>
            </Fragment>

        )
    }
}
```


Rooms

How many bedrooms do you need?

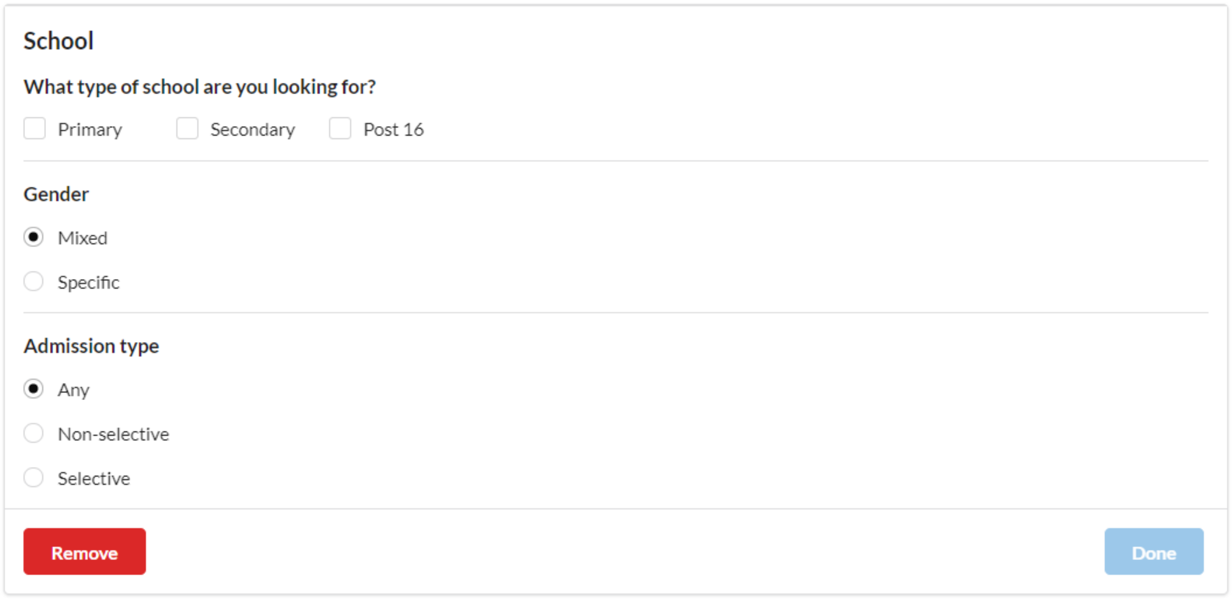
How many bathrooms do you need?

How many reception areas do you need?

Appendix S.8 School

```
import React, {Component, Fragment} from "react";
import {
    Header, Form, List, Checkbox, Grid, Divider
} from 'semantic-ui-react'

import {Segment} from 'semantic-ui-react'
import BaseFilter from "./BaseFilter";


export default class SchoolFilter extends BaseFilter {
    constructor(props) {
        super(props);
        this.state = {
            ...this.state,
            is_primary: false,
            is_secondary: false,
            is_post16: false,
            gender: "M",
            selective: 'any',
        }
    }

    componentDidMount() {
        super.componentDidMount();

        if(this.props.data.school && this.props.data.school.school) {
            const school = this.props.data.school.school;

            this.setState({
                ...this.state,
                ...school
            });

            this.save();
        }
    }

    static filter_name = "School filter";
    static description = "Distance from nearest school meeting specific requirements.";

    getCollapsedText = () => {
        // let {propertyTypes} = this.state;
        const {is_primary, is_secondary, is_post16} = this.state;

        return this.renderBody()
    };

    getData = () => {
        const {is_primary, is_secondary, is_post16, gender, selective} = this.state;

        return {
            'school': {
                'school': {
                    is_primary,
                    is_post16,
                    is_secondary,
                    gender,
                    selective,
                }

            }
        };
    };

    handleChangeAdmissionType = (_, {value}) => this.setState({selective: value});
    renderBody() {
        const {is_primary, is_secondary, is_post16, gender, selective, collapse} = this.state;

        return (
            <Fragment>
                <h3>School</h3>
                <Header style={{marginTop: 0}} size='small'>What type of school are you looking
for?</Header>
                <Grid>
                    <Grid.Row>
                        <Grid.Column width={2}>
                            <Checkbox
                                label={<label>Primary </label>}
                                checked={is_primary}
                                onChange={() => this.setState({is_primary: !is_primary})}
                                disabled={collapse}
                            />
                        </Grid.Column>
                        <Grid.Column width={2}>
                            <Checkbox
                                label={<label>Secondary </label>}
                                checked={is_secondary}
                                onChange={() => this.setState({is_secondary: !is_secondary})}
                                disabled={collapse}
                            />
                        </Grid.Column>
                        <Grid.Column width={2}>
                            <Checkbox
                                label={<label>Post 16 </label>}
                                checked={is_post16}
                                onChange={() => this.setState({is_post16: !is_post16})}
                                disabled={collapse}
                            />
                        </Grid.Column>
                    </Grid.Row>
                </Grid>
                <Divider/>
                <Header style={{marginTop: 0}} size='small'>Gender</Header>
                <Grid>
                    <Grid.Column width={6}>
                        <Form>
                            <Form.Field>
                                <Checkbox
                                    radio
                                    label='Mixed'
                                    name='checkboxRadioGroup'
                                    value='M'
                                    checked={gender === 'M'}
                                    onChange={() => this.setState({gender: 'M'})}
                                    disabled={collapse}
                                />
                            </Form.Field>
                            <Form.Field>
                                <Checkbox
                                    radio
                                    label='Specific'
                                    name='checkboxRadioGroup'
                                    value='that'
                                    onChange={() => this.setState({gender: null})}
                                    checked={gender === 'B' | gender === 'G' | gender === null}
                                    disabled={collapse}
                                    // onChange={this.handleChange}
                                />
                            </Form.Field>
                        </Form>
                    </Grid.Column>

                    {(gender === null | gender === 'B' | gender === 'G') ?
                    <Grid.Column width={6}>
```

```
                        <Grid>
                            <Grid.Row>
                                <Grid.Column width={4}>
                                    <Checkbox
                                        checked={gender === 'B'}
                                        label={<label>Boys </label>}
                                        onChange={() => this.setState({gender: 'B'})}
                                        disabled={collapse}
                                    />
                                </Grid.Column>
                                <Grid.Column width={4}>
                                    <Checkbox
                                        checked={gender === 'G'}
                                        label={<label>Girls</label>}
                                        onChange={() => this.setState({gender: 'G'})}
                                        disabled={collapse}
                                    />
                                </Grid.Column>
                            </Grid.Row>
                        </Grid>
                    </Grid.Column> : ''}

                </Grid>
                <Divider/>
                <Header style={{marginTop: 0}} size='small'>Admission type</Header>
                <Form>
                    <Form.Field>
                        <Checkbox
                            radio
                            label='Any'
                            name='checkboxRadioGroup'
                            value='any'
                            checked={selective === 'any'}
                            onChange={this.handleChangeAdmissionType}
                            disabled={collapse}
                        />
                    </Form.Field>
                    <Form.Field>
                        <Checkbox
                            radio
                            label='Non-selective'
                            name='checkboxRadioGroup'
                            value={false}
                            checked={selective === false}
                            onChange={this.handleChangeAdmissionType}
                            disabled={collapse}
                        />
                    </Form.Field>
                    <Form.Field>
                        <Checkbox
                            radio
                            label='Selective'
                            name='checkboxRadioGroup'
                            value={true}
                            checked={selective === true}
                            onChange={this.handleChangeAdmissionType}
                            disabled={collapse}
                        />
                    </Form.Field>
                </Form>
            </Fragment>
        )
    }

    isValid = () => {
        const {is_primary, is_secondary, is_post16} = this.state;
        return is_post16 | is_secondary | is_primary;
    };
}
```



School

What type of school are you looking for?

☐ Primary ☐ Secondary ☐ Post 16

Gender

[✓ Mixed]

Specific

Admission type

[✓ Any]

Non-selective

Selective