House Stats
H446-03 NEA

Candidate Name - Morgan Rhys Thomas
Centre Name - Christleton High School
Candidate Number - 1185
Centre Number - 40329

June 2022

Contents

Chapter 1

Analysis

1.1 Problem Identification

Housing markets are a complex and convoluted system with hidden patterns which are hard to visualise and discover without using complex and expensive software. This means that investing in property is arduous for the average person as compared to large property developers who have the funds for this software and the economic know-how to make informed decisions.

Although house sales data is public there are few too many solutions that let you visualise the data and analyse it. The best solution that exists as of right now is provided by the Land Registry and is formatted in a monthly report. These reports have very little analysis and only show how the current month performed to the previous year's month with basic statistical operations performed and only focus on England & Wales.

1.2 Computational Methods

The main objective of my application can be broken down into lots of smaller problems to be solved independently. This will make the application a lot more feasible and efficient to programme rather than tackling it as one monolithic problem. These smaller tasks will be:

- Store house data in a database
- Query that data via a programming language

- Process the data and apply statistical operations to it
- Present this data in a web interface for the user

The data required for this project is published by the Land Registry as one large file and a monthly file containing amendments and additions to be made to the original data. This will need to be queried so that statistical operations can be performed on specific areas or time frames and amended so that new data can be added to keep it as up-to-date as possible to provide the best user experience. These queries will also need to be performed in a sub-second time to provide the best experience which would not be possible without a database running on a computer.

One problem that will be appropriate to be solved by computational methods will be performing statistical operations on the data. A data processing platform will be required as it will often be processing 100,000s if not millions of data points. These calculations would not be able to be performed manually as the sheer time it would take and would often be impossible to be done via analogue methods. This will need to be performed in a minimal amount of time to produce the best user experience which will in turn allow the most up-to-date data to be used as it won't need to be preprocessed and instead can be processed upon request.

A data ingest system will also need to be done computationally as each month 100,000s of new data points are released and will need to be added efficiently to allow people to perform analysis on the most up-to-date data. This will require parallel processing to optimise the process as otherwise, it will take an inordinate amount of time to update the data. It will also require validation of the data and will be most suited to a computational solution as it requires a great deal of accuracy. If this were to be carried out by a human there would be room for lots of error when inserting large amounts of data.

This will also require a graphical user interface to abstract the complex calculations so that the user can view and understand the statistics derived from the dataset. These will be large numbers and a lot of data that will require graphics to show patterns and correlations. These would not suit a text interface or outputting to a file as the users would not be able to make full use of the results due to the unintuitive interface.

1.3 Stakeholders

The demographic this programme would be aimed at is real estate agents, property developers, surveyors & solicitors. Another essential demographic will be the average citizen who is looking into buying or selling a home. For the average person, the programme will be able to provide a simple tool which will be easy to navigate with limited functionality. Professionals will have access to more advanced tools to allow them to perform more advanced analyses. This will require the website to be intuitive to use as people in these industries may not be as computer literate. It will also need to provide highly accurate data as it will often be used in applications like house valuations, legal proceedings & investment decisions.

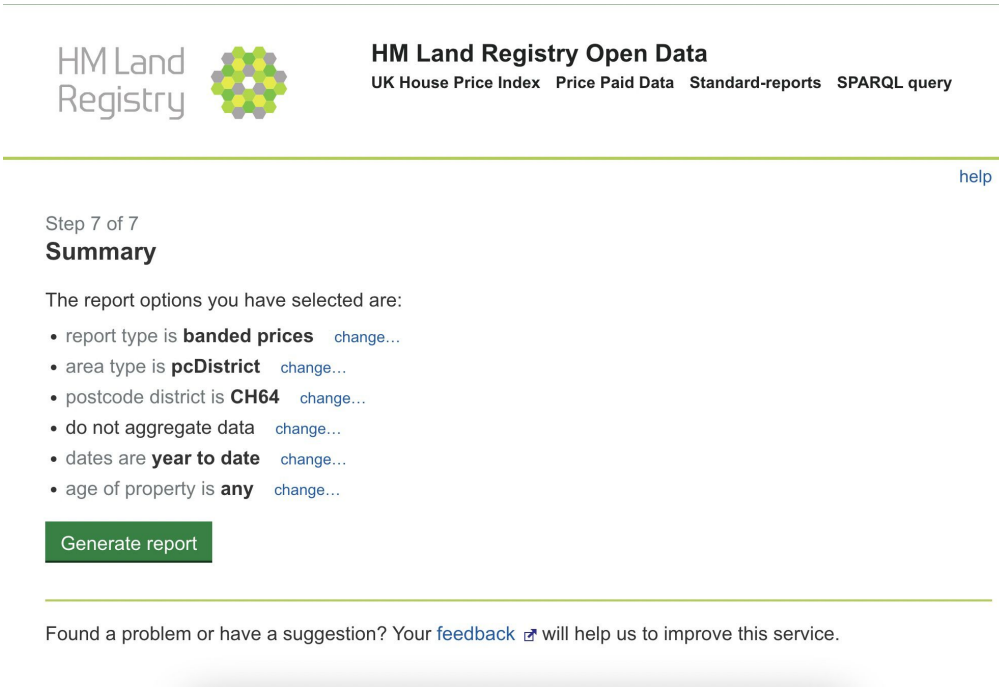
Often individual property investors looking to flip houses will end up looking on sites like Rightmove trying to find houses in their budget and not taking into account any of the economic backgrounds of that area. This can lead to poor investment decisions and end up with them losing a lot of money. When compared to commercial investors these individuals do simply not have the resources to put into analysing data themselves and investing money into tools to view that data. My application will allow them to analyse data and make decisions via intuitive graphs and a simple interface for free. These scenarios will lend themselves to my application as they will allow the average Joe to analyse residential areas simply and effectively. The stakeholder for this demographic is Liam Carter who is experienced with property investment to either flip or rent out.

Customers of real estate agents often just want an idea of their house value to decide whether they want to sell it or not. Nowadays when valuing a house you need to have surveyors come around to gather data and come back after a prolonged period with a price. At this point, the customer could have changed their mind or moved to a different realtor. With my application, they'll be able to get back to the customer within a few minutes with a rough estimate of a price and save the arduous surveying for another time not leaving the customer to have second thoughts. Allowing them to retain customers and optimise their pricing workflow as it can be used as another data point in their final calculation of the price. The stakeholder for this demographic is Sophia Bennett who has been in the real estate business for 25 years and currently works for a letting agency.

1.4 Research

1.4.1 Land Registry Price Paid Data App

https://landregistry.data.gov.uk/app/ppd



The Land Registry has created a basic application that allows users to search the data and generate reports. The Land Registry are a government agency that provides and maintains the price-paid data. This site aims to allow users to search for previous sales and to analyse historical data via reports that are exported as an excel sheet. These reports are very limited in what they can show and are very slow to process often taking at least an hour to process.

You can create 'Standard Reports' which allow you to see the aggregated prices and sales volume of an area grouped by house type or overall. These are created by a web interface where the user can pick where they want the data to be localised and the date range for the data to be gathered from. It also gives the user the option to display banded prices or the average prices of that area. The data can be localised further by grouping it via a different

sub-category of the address for the chosen region. These reports are quite basic in the data they provide and leave a lot of the data to be manipulated by the end user. They also often take a long time to be created when taking into account their simplicity. When testing the reports took from 30-120 minutes.

Another feature of this application is the ability to search historical house sales going back to 1995 when the Housing Regulations were introduced requiring the price of property sales to be logged with the Land Registry. This can be searched using several parameters including:

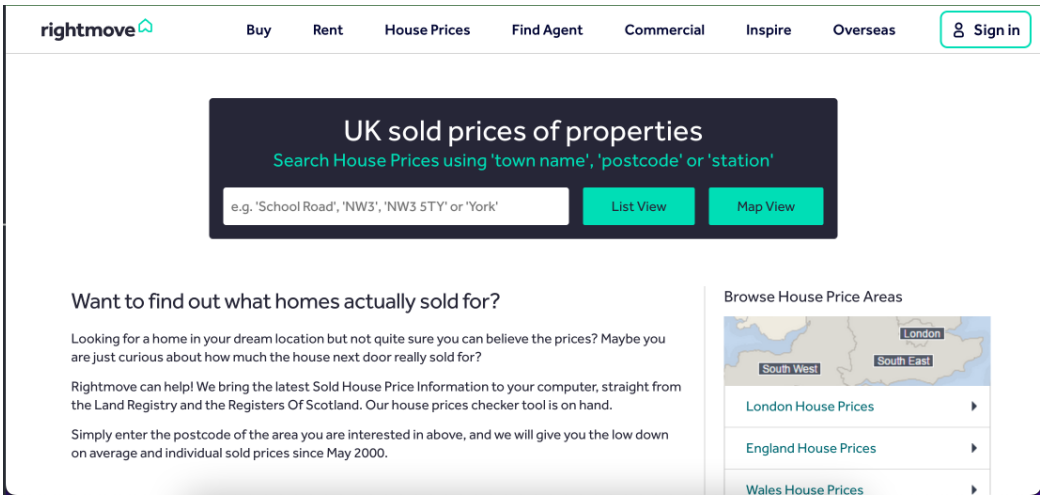
- Building Number
- Street
- Town or City
- District
- County
- Locality
- Postcode
- Property Type
- New Build
- Estate Type
- Price
- Date

Parts I Can Apply to My Solution

The PPD(Price Paid Data) applications have a lot of the features I would like in my application like the ability to create reports and search historical data but these features lack the speed and efficiency I would like from my application. It also uses an effective user interface which allows for simple usage and interaction which is what I'm aiming for with my applications as I want it to be used by people ranging from inexperienced members of the public to real estate agents with years of experience.

1.4.2 Rightmove

https://www.rightmove.co.uk/house-prices.html



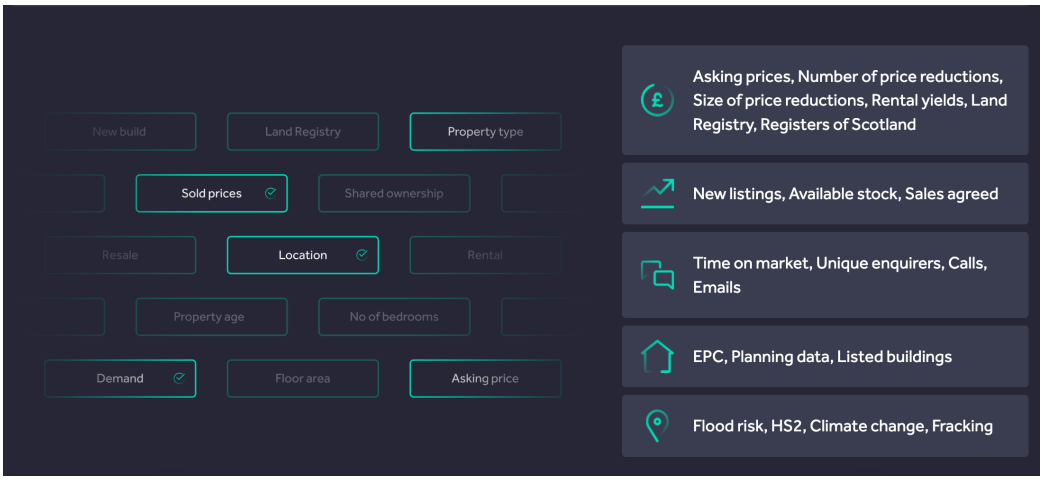
Rightmove is the UK's number one property portal and has been trading since 2000 and is now an FTSE 100 company hosting 90% of all property listings on their site. This gives them a wealth of data on current property sales which they can combine with previous sales data from the PPD dataset along with many other datasets they have access to due to their corporate stature. Combined these make up their subsidiary Rightmove Data Services. Here they provide numerous services:

- Market Intelligence Centre
- Bespoke Data Analysis
- Development Insights Reports
- Surveyors Comparable Tool
- Automated Valuation Model
- Property Risk Alerts

Market Intelligence Centre

The market intelligence centre is an application that visualises and shows the user's current market data, current property prices, supply & demand data. Combined this provides a bespoke suite of data visualisations and figures which cannot be found anywhere else due to their monopolistic-like company. All this data can be derived from a specific area and then filtered via dates and property types to then be exported via CSVs or PDF reports.

Bespoke Data Analystics



For this, they work closely with their client to design a detailed report for their requirements giving the client control of what they want out of it like the date ranges, asking prices, number of price reductions, size of reductions, rental yields and data from the Land Registry & Registers of Scotland. All this data is combined and aggregated into a bespoke report that can be published on a one-time basis or periodically over some time.

Automated Valuation Model

Rightmove's AMV is used by over 400,000 properties each month to value properties using its proprietary data and data provided by the Land Registry and Registers of Scotland which is accompanied by a confidence score. This is done via an interactive web interface or over an API allowing companies to

create their interface for in-house software or integrate it with their current infrastructure.

1.4.3 Parts I Can Apply to My Solution

Rightmove has a lot of services with very similar features to what I aim for my solution to have but are even more advanced as they have access to lots more data. However, their use of GUI and an API to allow clients to interact with their systems is exactly what I plan to do with my software allowing users with basic skills to use it and for people to integrate it with their tools. They also have high-speed data aggregation which is also what I aim for my platform to have as a slow programme would result in a bad user experience. Unfortunately, their system is locked behind a paywall and is only accessible by incorporated businesses which is what I want to avoid with my software as I want everyone to be able to use it.

1.4.4 Questionaire

Liam Carter's Q&A

- **How much research do you do before investing in a property?**
  "I tend to invest in the area around me as I know that best so I only have to do limited research."
- **What do you use to look at properties to invest in?**
  "The main platform I use is Rightmove and I occasionally use Purple Bricks when I can't find what I want on Rightmove."
- **What type of apps do you prefer web, desktop or mobile?**
  "I'm often on the move looking at houses so I tend to be using either my phone or a laptop so a website would be most suited to allowing me to access it on the move with all my devices."
- **What features would you like to see?**
  "I only tend to buy houses near me as they are easier to manage and maintain so a function that would allow me to localise the analyses. I would also like to see the historical prices for areas and see how they compare so I can see what areas have prices on the rise."

- **What issues do you have with current solutions?**
  "A lot of the services are aimed at companies so often have very expensive licences or simply won't give the time of day to an individual investor leaving me high and dry as there are no other options except the Land Registries PPD app but that's very limited in its functionality and takes a while to complete the aggregation."

- **Would you be willing to pay for a service to analyse the housing market?**
  "Yes most certainly as long as it was within budget obviously unlike those other commercially aimed services but it would have to meet all my requirements which are a simple interface, lots of customizability when it comes to filtering and aggregating the data, a quick interface so that I'm not stuck there waiting for my results wasting time and accessible wherever I am with an internet connection."

Sophia Bennett's Q&A

- **What process do you use for the valuation of a property?**
  "Well first our agents will have background knowledge of the area the property is in and will do some more research before even entering the property like local amenities, schools and shops nearby. Once that is all complete they'll come and see your property. They'll walk through the property and take extensive notes along with measurements and look for key selling points like original fittings and new heating systems. After this, they'll ask the buyer a lot of questions ranging from whether they need the house to be sold quickly or any other information they could offer up to help give us an idea of the price. Once all of this is done they'll go back to the office and combine their notes with their other research and come back to the owner with a valuation."

- **How long does it take to value a house?**
  "For a firm with our expertise and experience the actual valuation process is quite quick and can be done in a few hours overall but the main issue is fitting people in with appointments."

- **Do you know of any solutions for analysing the housing market?**
  "Yes we use some ourselves for doing research on a property before going to the in-person part of the valuation. The system we use is provided by Rightmove but is very costly as it has a yearly licence fee."

- **What would be your preferred method of accessing software?**
  "A lot of our staff are often out and about on properties or travelling between them so they will often be using either laptops or iPads and in the worst-case scenario their phone so all the software we use is web-based allowing them to access it from anywhere."

1.5 Features

My solution will consist of a web-based application accessible via any internet-connected device with a web browser allowing most people to access it. It will have a page for searching historical sales and viewing data about a house whilst also comparing the prices to the local average and allowing analyses to be done on individual houses. There will also be a section for analysing larger areas which could be defined by a postcode or a county and anything in between. These areas can then be compared with one another providing greater insight. This data will be shown via graphs and interactive maps to allow anyone to understand the data. It will also have an admin panel accessible via authentication to allow for new sales data to be uploaded and see common queries along with other analytical statistics.

1.6 Limitations

The major limitation of my solution is that I will not have access to individual data on the houses like the number of rooms and the acreage of the plot it lies on. This could lead to incorrect predictions as my algorithm will be ignorant to the fact that a house was sold for £50,000 as it had a pool rather than the value of that area going up. This can be combated by analysing areas with a greater volume of houses as they would be ruled out as anomalous but this would not work in all scenarios. Gaining access to this data would also pose a great challenge as it would require scraping data from other sites like Rightmove which would be computationally and time intensive.

Another issue would be getting up-to-date and accurate data. Although the PPD dataset is perfectly acceptable it only gets updated once a month and is always a month or two behind what is happening right now. This could cause discrepancies in the data shown to the user and would also not

allow them to get the full insight as there could've been a drastic shift in the past few months. Luckily the housing market is not very volatile most of the time but there have been times when it changed overnight and would make the statistics provided completely irrelevant. Unfortunately, there is no way to mitigate this issue as the Land Registry is the only provider of this dataset and doesn't seem to be changing its upload schedule any time soon.

1.7 Requirements

1.7.1 Software

- **Linux operating system** - This will be required to run the SQL server, Dask and the web server
- **Python interpreter** - The project will be written in mostly python so this will be required to run it
- **Web browser** - This will be required by the users to access the programme and use it
- **PostgreSQL server** - This will be used as the database to store the sales data and query It
- **Dask** - This will be used for performing statistical operations on large amounts of data
- **NGINX** - This will be used to host the website and serve the pages to the users

1.7.2 Hardware

- **Server** - This will be required to run the database, web server and data aggregator. It will require at least an 8-core processor to run all the services and 16GB of RAM to hold the dataset in memory for performing the statistical operations and an SSD as an HDD would not be capable of the IOPS required to run a database

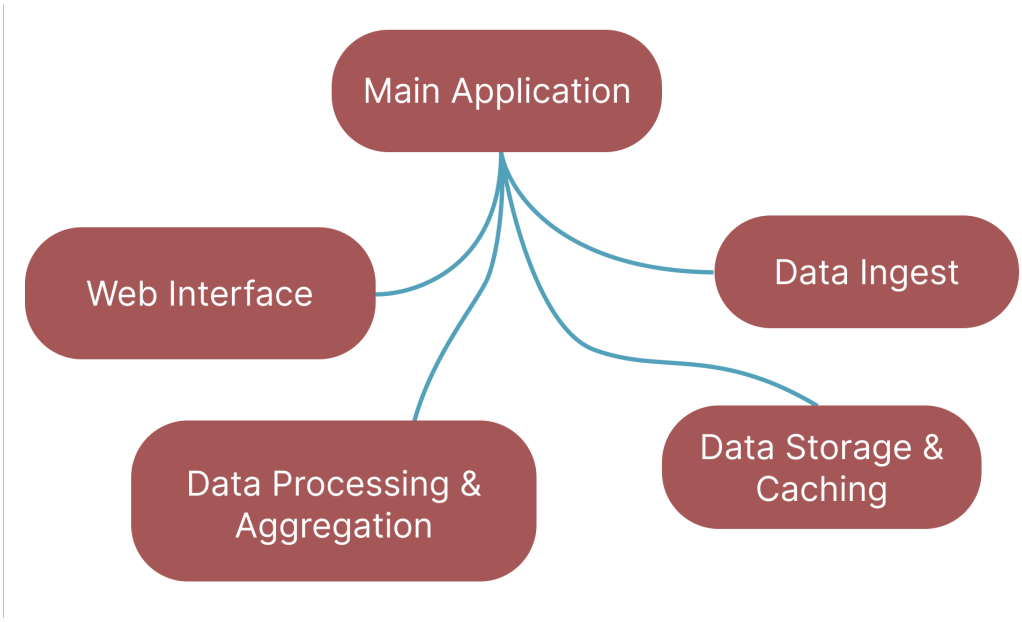
1.8 Success Criteria

Criteria	**Evidence**
Insert the PPD data into an SQL database in 3NF	Screenshot of a query from the database showing sales data
Query data from the database using Python	Showing the code and the terminal output showing data from Python
Performing statistical operations on data using Python	Screenshots of results from said operations
Selecting data from specific areas and aggregating it	Screenshot of result from aggregation with area
Creating an API to interface with and get data	Screenshots of successful web requests showing data
Creating a user interface to show data from the API via graphs and figures	Screenshots of the user interface
Having searched for historical data taking ¡500ms	Video of searching for historical data with HTTP response time
Generating statistics for an area and displaying taking ¡2000ms	Video of analysing a specific area
Upload new data to the website for analysing and searching	Screenshot of new data
Setting time frames for analyses of the data	Video showing the data changing based on the time frame

Chapter 2

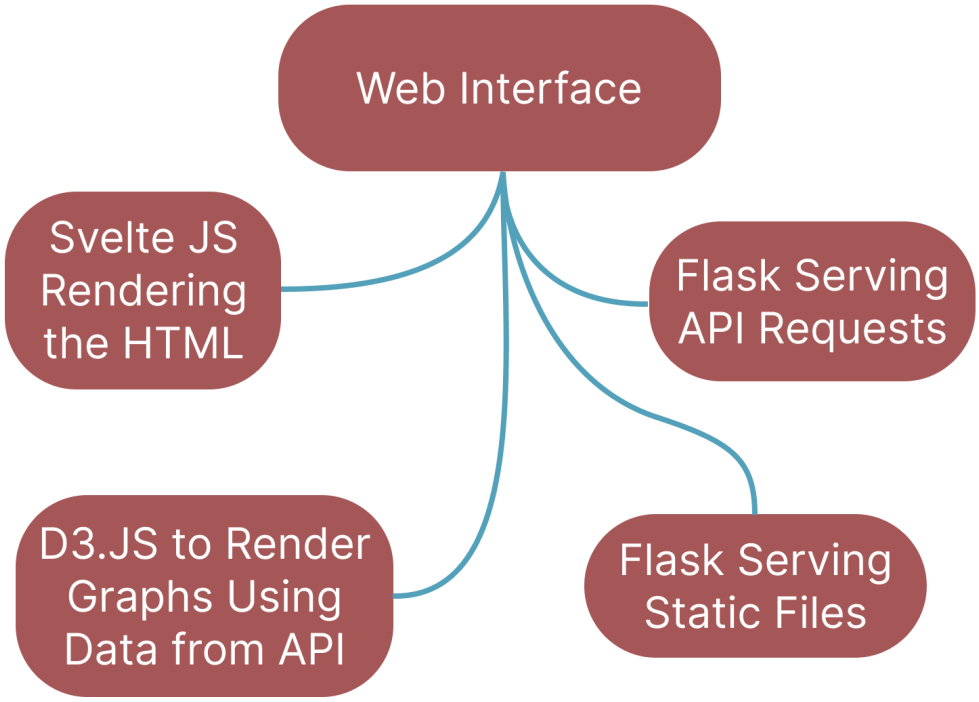
Design

2.1 Decomposition



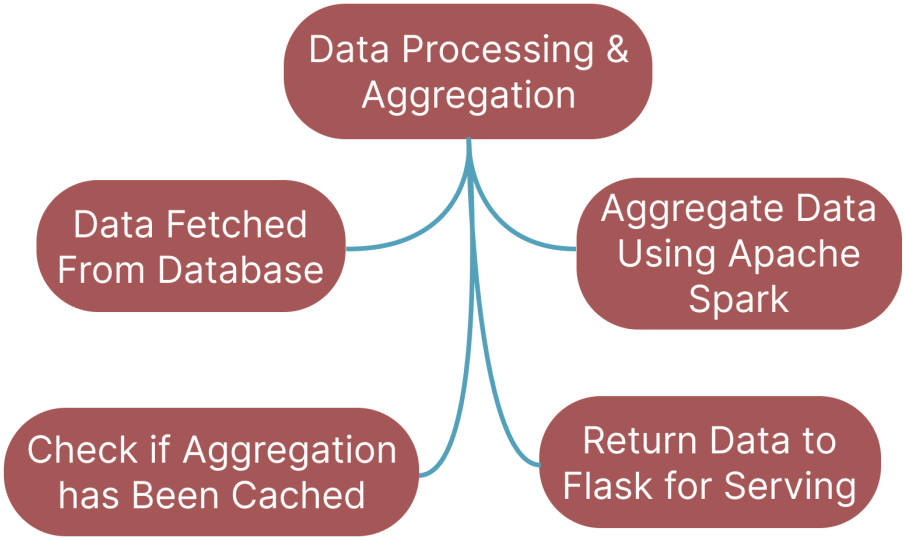
The application will be broken down into four smaller more manageable sections. These will allow me to develop the application more easily and allow for easier testing and refactoring. These sections will be the web interface, data processor & aggregator, data ingest and data storage & caching.

The web interface is how the user would interact with the application and where the data will be presented to them. This will need to be responsive and easy to use. This will further be broken down into the data visualisation aspects and the data selection part allowing me to develop them individually and get stakeholder feedback on each. This will be the only way the user interacts with the programme so stakeholder feedback will be essential for creating a good user experience. The visualisation will be done using D3.js and the web interface will be done using svelte.js allowing for interactivity and dynamic interfaces. It will receive the data for visualisation from the backend which will be a flask server running a web API and server to serve the static files.



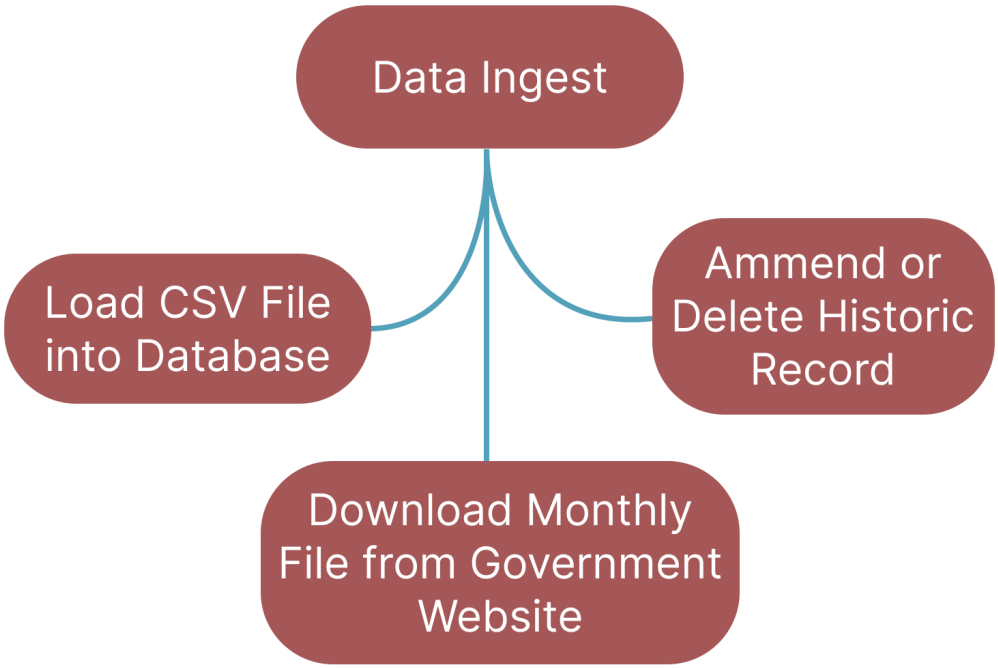
Data processing & aggregation will allow me to perform statistical operations on large datasets in a very little amount of time improving the user experience. This will work seamlessly with the data storage component allowing for even greater efficiency. The main sub-component will be aggregating

the data using Dask which allows data to be processed in parallel across multiple processors over multiple machines meaning it can aggregate hundreds of thousands to millions of rows in sub-second time again adding to the user experience. To again improve efficiency a caching layer will be added so that commonly executed queries need not be repeated saving computing time and costs. After processing this data will need to be served to the user which will be done via a Flask API which sends the data in JSON format over HTTP. All of this works as one big organism creating a coherent user experience.



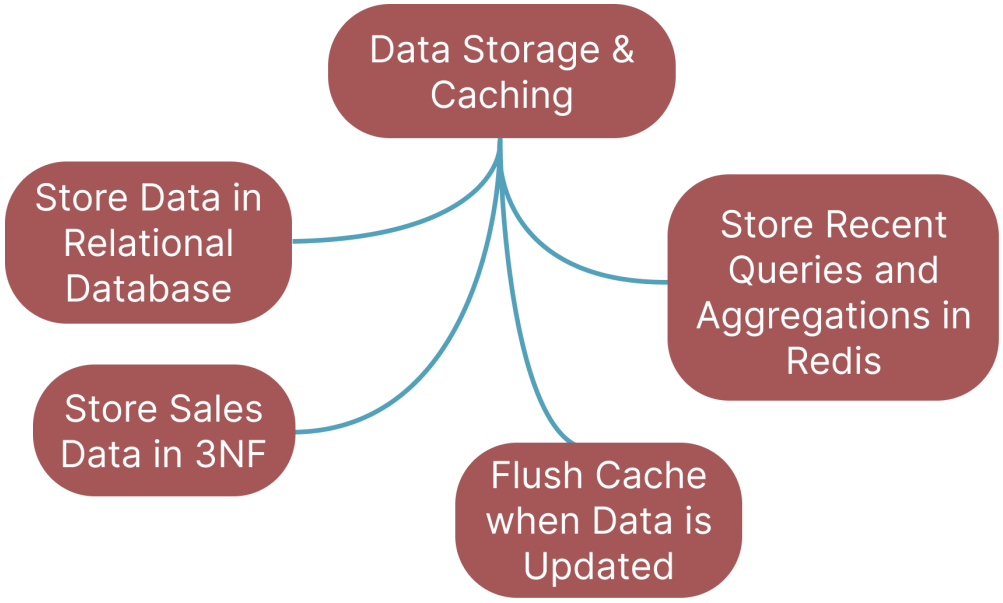
The data ingest component will be key for keeping the data accurate and up to date as update files are released every month so the ability to ingest and process this data promptly will result in a better representation of the housing market at that point in time. This will be done by three different components. Firstly the file will need to be downloaded from the government website every month as soon as it's available so a script will constantly be checking the website to see if a file has been changed. Secondly, the file will then need to be processed as it's in the CSV format so a script will convert it into a 2D array ready for insertion into the database. Thirdly it will be inserted into the database but some rows are insert rows, addition

rows and deletion rows so those will need to be handled accordingly so that the database remains up to date.



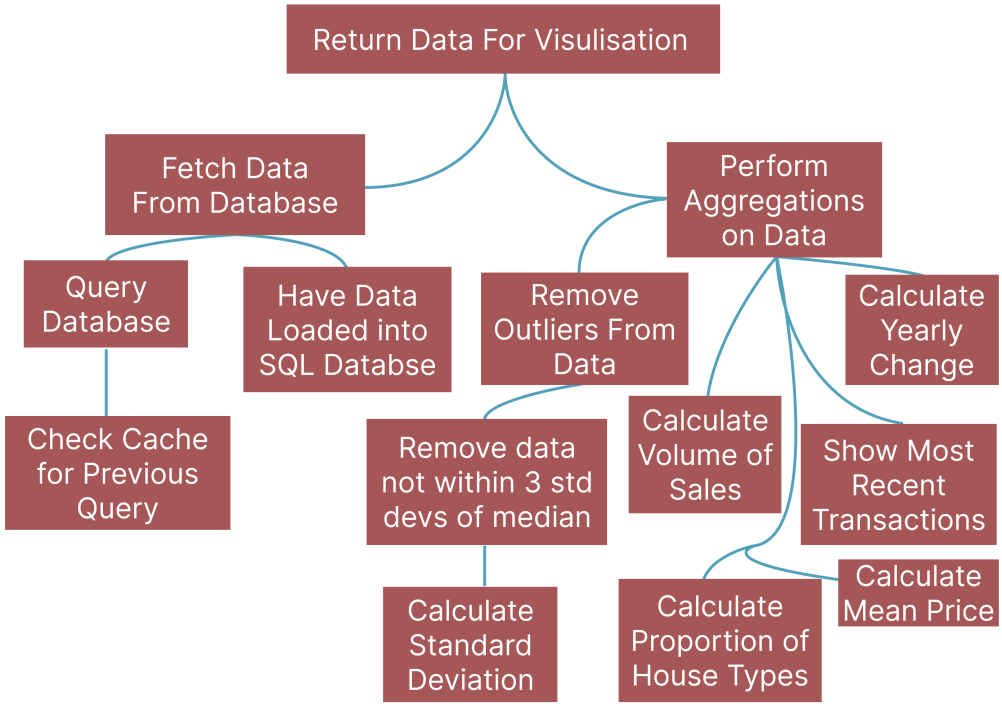
Data storage will play a large role in the overall efficacy of the application as it will need to be able to sort through millions of rows in a few hundred milliseconds and be able to join multiple tables each containing potentially hundreds of thousands of rows. This will require a relational database and the database of choice will be PostgreSQL. This will allow me to perform all of the above and more as I can then use a full-text index to search partial postcodes allowing for great granularity when analysing an area. All of the data will be stored in a third normal form allowing for no unnecessary data duplication and maintaining referential integrity. As mentioned before there will also be a caching layer allowing for even greater speed. This will be done using a Redis database which is an in-memory database which can store key-value pairs (query-hash, result). This cache will act between the web interface and data processing & aggregation components so that when a user requests data it can be fetched from the cache instead of being recalculated.

It will be flushed every time an update is made to the database so that the highest order of accuracy can be maintained.



2.2 Algorithms

The main part of my programme will be the statistical analysis & aggregation module as this will require the most attention and optimisation as all the other components have been done before and have lots of resources online making them a lot simpler to develop. The main purpose for this module will be to fetch data -> process it -> return it.



This will be broken down into two sections data acquisition & data aggregation. These will then be further broken down into smaller sections as you can see in the figure above. Once all these functions have been performed this will be sent to the web interface where it will be visualised for the user using various charts and tables.

2.2.1 Removing Outliers

This will be the function that is run the most out of all of these as every bit of data fetched will need to have all the outliers removed so that it does not make the results skewed in any way. This will be done using the standard deviation of the data and then removing any data that is greater than 3 times the standard deviation away from the mean. To speed the process up this will be run on Dask so the dataset can be processed in parallel. The reason I'm using 3 standard deviations as all the data I am using is continuous and can be displayed as a normal distribution $X \sim N(\bar{x}, \sigma^2)$. Where $\bar{x}$ is the mean and σ^2 is the variation. With normal distributions, the data can

be split up into standard deviations from the mean with a probability of it being within that range. For example 68% of values lie within one standard deviation, 95% lie within two standard deviations and 99.9% lie within 3 standard deviations of the mean.

```
function remove_outlier(data, sd){
    rdd = pyspark.rdd(data)
    mean = rdd.mean()
    filtered_data = rdd.filter(lambda x: mean-(3*sd) < x < mean+(3*sd))
    return filtered_data
}
```

2.2.2 Calculating the Standard Deviation

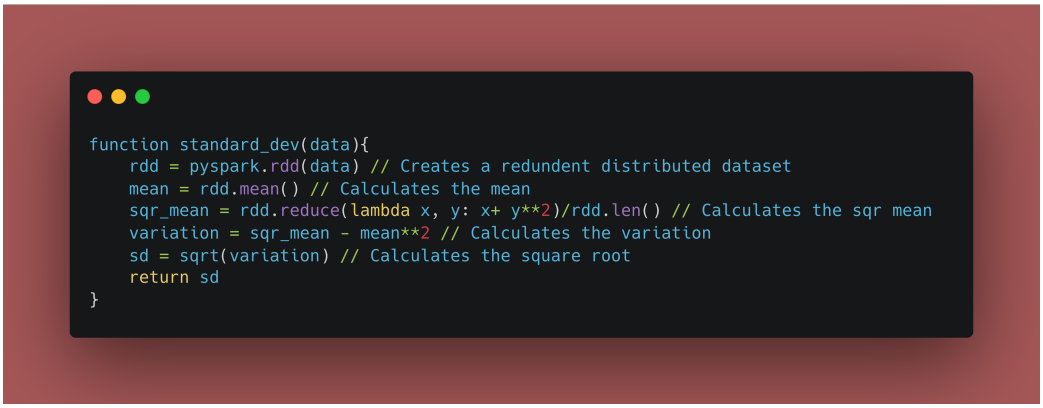
This will be run as much as removing the outliers as it will be required to remove the outliers when showing the volatility of the house prices over the month. This will be another metric in showing the user the state of the housing market as they can see when three is the most potential to make money and when the market is acting rather stale. So it will be used on a wide variety of datasets like sales transactions, monthly percentage changes and yearly percentage changes so will need to handle them accordingly.

This is the equation for standard deviation where n is the length of the dataset and x is all the items in the dataset.

$$\sigma = \sqrt[2]{\frac{\Sigma x^2}{n} - \left(\frac{\Sigma x}{n}\right)^2} \tag{2.1}$$

x is 4 x is 4

I imagine in the future I will need to further optimise this function as it is pretty basic right now and only used the tip of the iceberg when it comes to Dask functions.



```
function standard_dev(data){
    rdd = pyspark.rdd(data) // Creates a redundent distributed dataset
    mean = rdd.mean() // Calculates the mean
    sqr_mean = rdd.reduce(lambda x, y: x+ y**2)/rdd.len() // Calculates the sqr mean
    variation = sqr_mean - mean**2 // Calculates the variation
    sd = sqrt(variation) // Calculates the square root
    return sd
}
```

2.2.3 Calculating the Percentage Change

Percentage change will be the key metric for comparing different areas and their performance as house prices between areas so representing the changes as percentages instead of absolute values allows for them to be easily understood since they will be normalised. It is also one of the more complex functions as it requires a lot of manipulation of the data since the transactions will need to be grouped by houses and then calculate the percentage change between sales. This will then be extrapolated over the period between sales so 10% over two years will result in a 5% APY. These will then need to have the outliers removed and then averaged for an area. On a small scale, this function will run in a reasonable amount of time but as you increase the dataset the time increase exponentially so $O(n) \approx 2^n$. Further optimisation will be required at a later date but for now, it is acceptable enough to proceed. This function will also be able to perform this aggregation on a more granular scale for example monthly percentage change allowing the user a better insight over a shorter period.

```
function yearly_change(data){
    df = pyspark.df(data)
    df[prev_price] = df.groupby(['houseID']).shift(1)
    percents = df.groupby(['houseID']).apply(interpolate)
    avg_chng = percents.groupby(['date']).mean()
    return avg_chng
}

function interpolate(df){
    results = [0 for i in range(1995*12, df[0][date].year*12)]
    for idx, i in enumerate(df):
        if idx != 0 and len(df) > 1:
            change = ((i[prev_price]/i[price])-1)*100
            start = df[idx-1][date]
            end = i[date]
            dif = end.year*12-start.year*12
            month_chng = change/dif
            results.append(month_chng)
    return results
}
```

2.2.4 Parsing the CSV

Parsing the CSV file which contains the monthly sales data or the yearly sales data will need to be a highly optimised process but luckily it is a rather simple task. This will be written in Golang instead of python as it is significantly quicker for these kinds of tasks that require a lot of iterations. As of right now, the function will be single threaded but later down the line if required it can be converted into a multi-threaded function allowing for it to be parsed at a speed n time faster where n is the number of threads. Of course, this will be limited by the number of cores in the CPU it is running on. The file will be parsed line by line and each line is split into cells using a delimiter, in this instance, it is a ’,’. These lines will then be saved as a list with each item in it representing a cell from that row which will be stored in an array where each item is a sales transaction. This array is then passed onto another function to insert each transaction.

```
function parse_csv(csv_file) {
    file_obj = open(csv_file)
    output = []
    for line in file_obj:
        sale_info = line.split(",")
        output.append(sale_info)
    return output
}
```

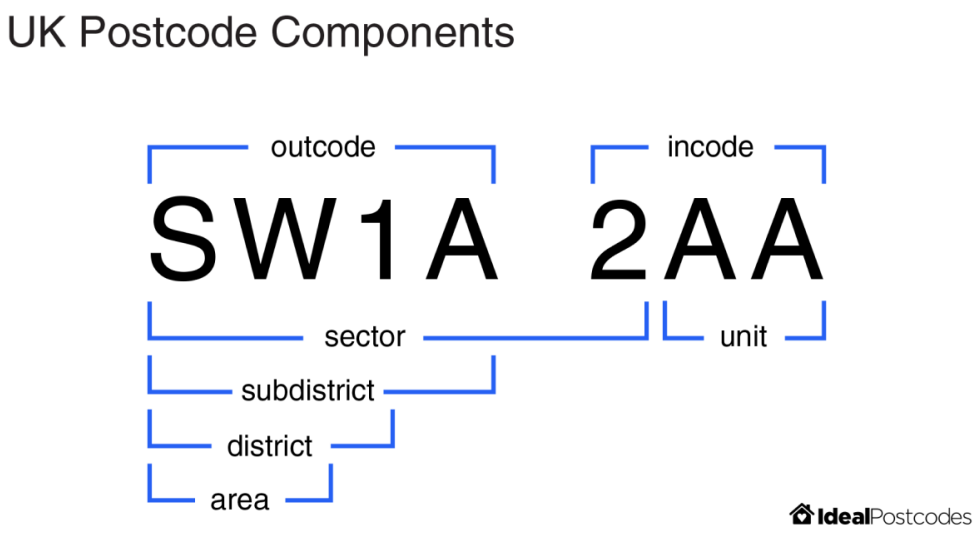
2.2.5 Inserting or Rectifying Transactions

Inserting a transaction is a fairly simple process and only requires a few lines of SQL though it is a bit more complicated as a transaction can result in different operations being performed which is denoted by the last cell and the letter stored within. The letter A means that this transaction is to be added to the dataset. The letter C means to edit the transaction with this transaction id to have this information. The letter D means delete this transaction with this transaction id. All these operations have to be performed whilst maintaining referential integrity so that it maintains the third normal form. Furthermore having the transactions inserted sequentially will take forever as there are hundreds of thousands of transactions each month so a master-slave architecture will be used. This is where the master sends out jobs and the slaves receive those jobs and then complete them. This means that the transactions can be inserted n times quicker where n is the number of slaves but this will be limited by the throughput of the database I am using.

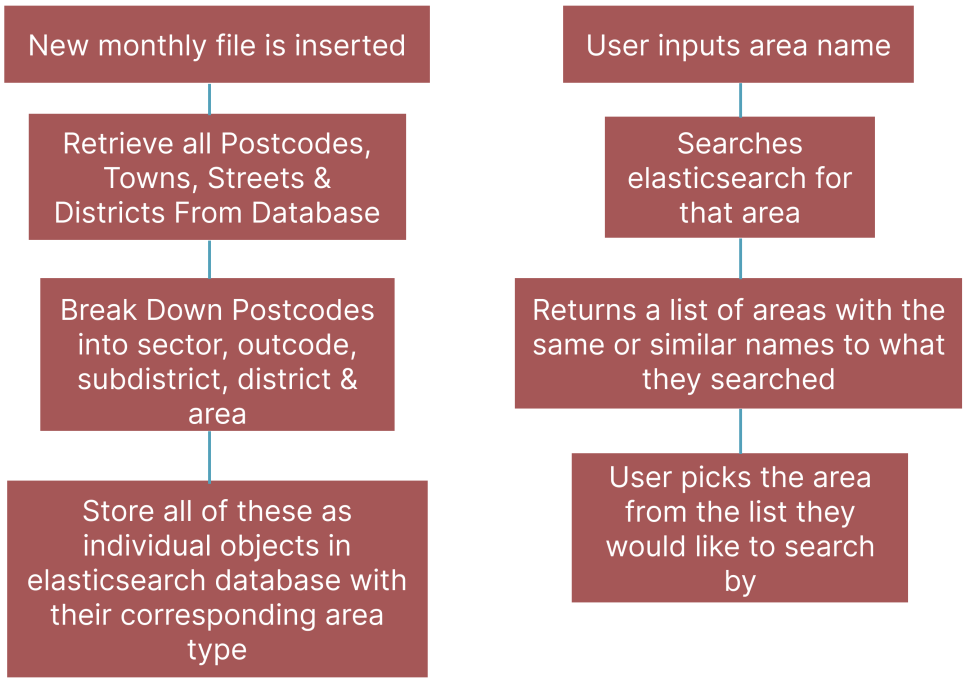
```
function insert_row(sale_info){
    type = sale_info[-1]
    if type == "A":
        db.QueryRow("INSERT INTO postcodes (postcode, street, town, district, county)
                    VALUES ($1,$2,$3,$4,$5) ON CONFLICT (postcode) DO NOTHING;;",
                       sale_info[3], sale_info[9], sale_info[11], sale_info[12], sale_info[13])
        db.QueryRow("INSERT INTO houses (houseID, PAON, SAON, postcode)
                    VALUES ($1,$2,$3,$4) ON CONFLICT (houseID) DO NOTHING;;",
                       houseID, sale_info[7], sale_info[8], sale_info[3])
        db.QueryRow("INSERT INTO sales (tui, price, date, new, freehold, type, ppd_cat, houseID)
                    VALUES ($1,$2,$3,$4,$5,$6,$7,$8) ON CONFLICT (tui) DO NOTHING;;",
                       sale_info[0][1:37], sale_info[1], sale_info[2],
                       new_build, freehold, sale_info[4], sale_info[14], houseID)
    elif type == "C":
        db.QueryRow("DELETE FROM sales WHERE tui = $1;", sale_info[3])
        db.QueryRow("INSERT INTO postcodes (postcode, street, town, district, county)
                    VALUES ($1,$2,$3,$4,$5) ON CONFLICT (postcode) DO NOTHING;;",
                       sale_info[3], sale_info[9], sale_info[11], sale_info[12], sale_info[13])
    elif type == "D":
        db.QueryRow("DELETE FROM sales WHERE tui = $1;", sale_info[3])
}
```

2.2.6 Searching for Areas Flow Charts

When a user searches for an area they might not know the exact name or what type of area it is so by storing all of the areas and their corresponding types in an ElasticSearch database I can create a search index. Postcodes will also be broken down into 5 different types Of areas allowing for greater granularity.

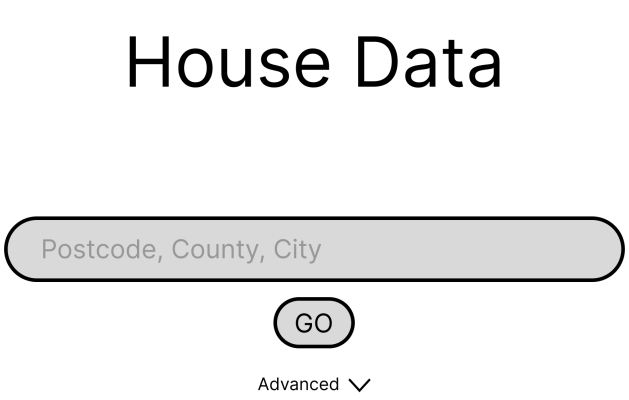


This will allow me to send queries to the database and have it return all areas and their types that match or are similar to what the user searched. These will then be presented in a list for the user to select from allowing for a greater user experience and making the programme easier to use for people unfamiliar with the geography of England & Wales.

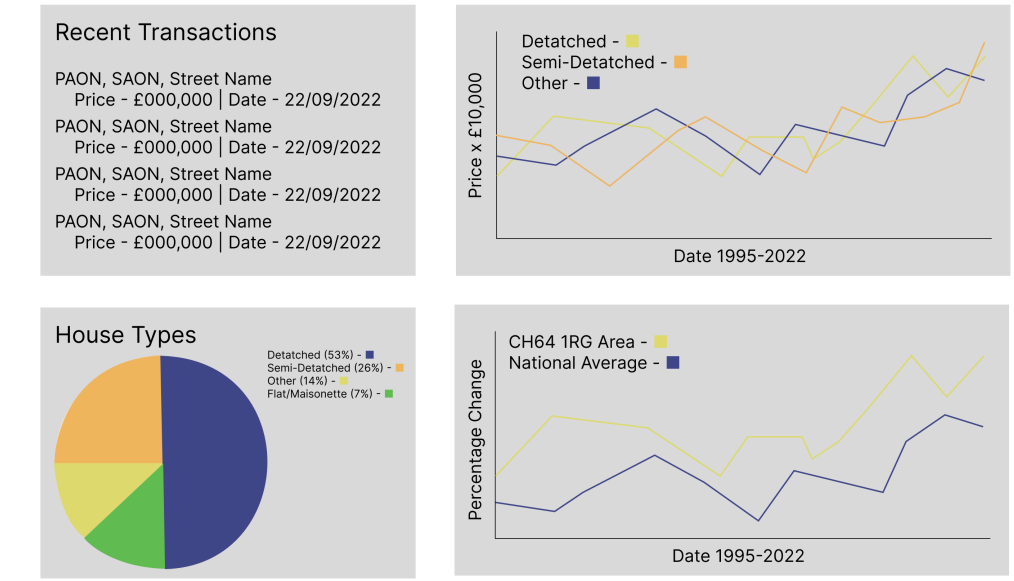


(FYI this is to be extended but I cannot think of any other major functions at this point as IDK what other statistical operations I will perform until I start development.)

2.3 Usability Features



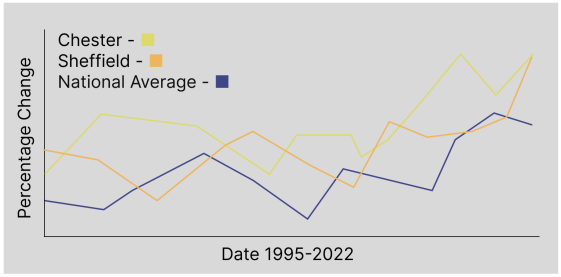
This will be the first page the user lands on and aims to be as simple as possible whilst still maintaining all the functionality required. The interface is focused around this search bar as the user should be able to type in a place or area and have the data come up without any extra work. I chose this as I want this to be useable by any average person that has no extra knowledge about computers or houses other than the basics. Whilst the user is searching it will come up with suggestions below the search bar in a list with the name of the area and what type it is whether it's a postcode or a district allowing them to more easily find places. Below the search bar is a button which allows the user to enable more advanced options like excluding certain building types or limiting the date range. These options are still all accessible but are hidden unless needed so as not to distract the user.

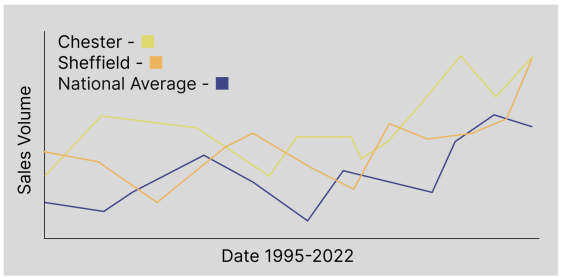


After the user searches for the area and selects the one they want to view from the list they will be sent to this page. In-between this page and the search bar there will be a loading screen to indicate that something is happening as the aggregations could take an extended period depending on the area but for the time being, this will only be a nicety and not a required feature. On this page will be a range of graphs and visualisations all rendered using D3.js as it allows for a high level of customizability for each plot so that they can be understood by all the users. There is also a minimal amount of text as having a wall of text containing all the statistics would be daunting for almost any user so using visualisations and only having the bare minimum amount of text creates a lot better user experience. The figure will also be made using contrasting pastel colours to not dazzle the user and make viewing and reading easier on the eyes. The colours used are also all colour-blind friendly.

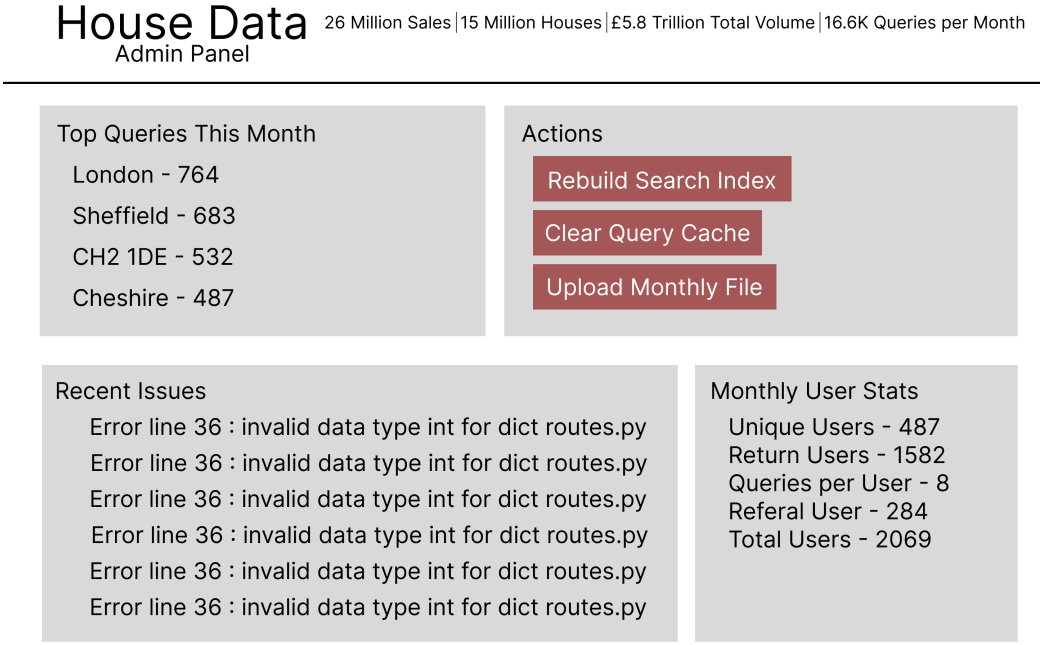
Chester & Sheffield

	Chester	Sheffield
Mean APY	3.7%	2.8%
Houses	38K	145K
Mean Price	245K	186K





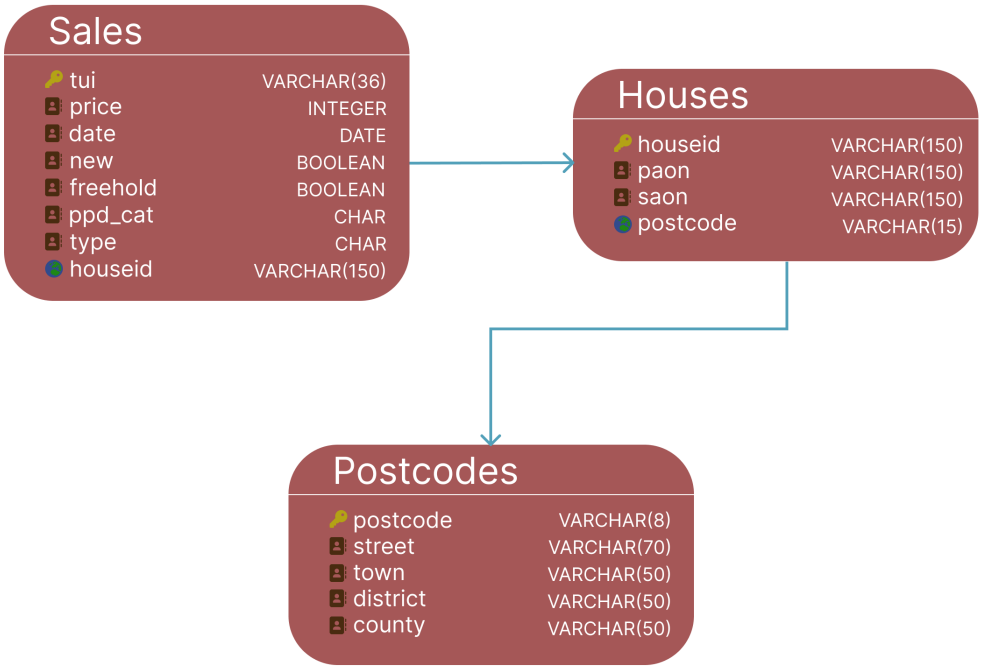
Looking at one specific area is useful but only provides a micro view of the housing market but comparing areas with each other and seeing their performance alongside the national average provides a macro view of the housing market. This will be useful for users who use this platform to judge investment decisions as it allows them to look back through historic data and judge which area has the most potential or which is the most stable. Combining all these statistics gives the user greater insight into house prices but requires the user to be able to interpret the figures correctly so it is more aimed at the advanced user base. This will allow me to display more complex statistics allowing an even greater insight. These more complex statistics will also take longer to process so the page will use lazy loading to only load them when the user can see them speeding up the overall load time and creating a better user experience.



Being able to manage the site from a single page will be essential to find bugs and seeing trends in what users are doing on the platform. From this page, you will be able to upload the new monthly file manually if it hasn't detected them automatically, flush the query cache forcefully if some incorrect data is being stored there and rebuild the search index. All of these operations would normally be performed automatically but having a manual override allows me to fix any issues that may occur. It will also allow me to look at popular queries and precache them to improve user experience. Alongside them will be all the most recent errors that have been logged so that they can be analysed and debugged improving the overall user experience. This page will be locked behind a username and password so that no one can access this panel and use it for malicious purposes.

2.4 Variables & Data Structures

2.4.1 Database Architecture



The database structure has been designed to be in the third normal form. This means that there is little to no duplicate data, referential integrity will be maintained and will simplify data management. This will improve efficiency when fetching, inserting and updating data which will improve the overall flow of the entire application. All of the columns will have individual indexes as they will be sorted pr searched via making these quicker so that it can do an index search vs a collection scan which is highly inefficient. All the columns are defined below with their corresponding data type and purpose.

Name	Data Type	Table	Purpose
price	int	sales	The sale price of the house
date	Date	sales	The date of the sale

new	Boolean	sales	The sale price of the house
freehold	Boolean	sales	Whether or not it is freehold or leasehold
type	char	sales	The house type whether it is detached, semi-detached or a terrace
ppd_cat	char	sales	Whether or not it was sold at a standard price A or had additional costs like repossessions B
tui	varchar(36)	sales	String used to uniquely identify a transaction
houseid	varchar(150)	sales/houses	A string used to uniquely identify a house
paon	varchar(150)	houses	Primary addressable object name like house number or name
saon	varchar(150)	houses	Secondary addressable object name e.g. the flat number in the building 3b
postcode	varchar(15)	houses/postcodes	The postcode of the house
street	varchar(70)	postcodes	The street of the house
town	varchar(50)	postcodes	The town of the house
district	varchar(50)	postcodes	The district of the house
county	varchar(50)	postcodes	The county of the house

2.4.2 Key Variables

Name	**Data Type**	**Purpose**
Requested Data	list	Contains the list of tuples for the house data (id, price, houseID, date)
Time frame	tuple	Contains the two dates to request data between
Location	string	The location to filter the data by
Session	dict	Contains all the session data for the website

Database host	str	IP address of the database server
Database user	str	Username for the database
Database password	str	Password for the database
Standard Deviation	float	The standard deviation of a given dataset
Dataset	python object	This will be a resilient distributed dataset which allows aggregations to be performed on it in parallel. This will contain the sales data for a given area
Dask Address	str	The IP address for the Dask cluster
Job ID	str	The id of a Dask job so that the status can be displayed to the user
Dark Theme	boolean	Whether or not to display the website in a light or dark theme
Exclude	dict	Contains information on what types of sales to include or exclude from the aggregations. Can be set by the user via the advanced section

2.4.3 Validation

Area Search Input

The user will be able to type whatever they like in this input so a few validations will need to take place. The first one that will take place is client-side validation. This is where code running on the web browser validates the input but this type of validation can be easily bypassed so is not used for security and will be used for improving the user experience. It will work by not allowing the user to type in special characters or characters from other languages as these cannot be used in addresses or the names of areas in England & Wales. The other type of validation that will take place is server-side which will aim to prevent SQL injection as the data inputted by the user will be used in an SQL query to find the relevant places to suggest to the user. This will be done by iterating through the string and looking for any special characters and removing them as they are not necessary and are mostly used with malicious intent.

Monthly Update CSV File

When inserting the monthly update file into the database it needs to comply with the CSV format which by standard uses a comma to separate cells and a new line to indicate a new row. This is all checked when it is being loaded into memory and a 2d array. This validation is paramount as incorrect formatting could result in cells getting mixed up or rows combined which then would make any statistics derived from that data inaccurate and unusable.

2.5 Testing

2.5.1 Unit Testing

Testing will be done throughout development making debugging easier as any issue can then be narrowed down to a single function or group of functions. The applications will be broken into four distinct units (Web Interface, Data Ingest, Data Storage, Data Processing). These will each have numerous tests as they will be made up of tens of functions so the tests will be abstracted into these 4 units. The tests will be carried out automatically on each push request & merge to the GitHub repository. GitHub has a built-in DevOps framework which will allow me to create a YAML file defining all the tests it needs to run and then will return the results. If it fails any of the tests it will not allow any of the code to be merged therefore not creating any issues. By doing this I can be sure that the entire programme is functional and that breaking changes are not implemented. Below is a list of all the tests I will carry out. If all these tests are successful I can be sure that my programme is fully functional.

Function	**Working**
Check standard deviation function produces correct values	
Check removes_outliers removes all data outside the 99.99 percentile	
Check CSV parser detects bad files	
Check SQL escape function removes all special characters	
Check monthly update files are parsed correctly into amend, delete & change	
Check monthly file downloader downloads latest file available	

Check D3.js accepts correct data only for graphs	
Check Redis cache is hit before running an aggregation	
Check the correct data is fetched from the database for an SQL query	
Check connection to Dask is working	
Check relational integrity of database to maintain third normal form	
Check cache is flushed after new data is added	
Check recent queries are properly stored in Redis cache	

Post Development Testing

Once all of the above tests have been successful my programme will be passed on to my stakeholders who can use the programme and find any issues that were not detected during the development phase. These could be misaligned text in the GUI, graphs not loading or images not showing up. There could also be programmatic issues that I missed that they found. These can be logged using a piece of software that will implement in my code called sentry that logs code errors and crashes which are then logged and aggregated to show repeated issues and what caused them allowing me to more easily debug them. All of these combined will allow me to eliminate as many bugs or issues as possible resulting in the best user experience.

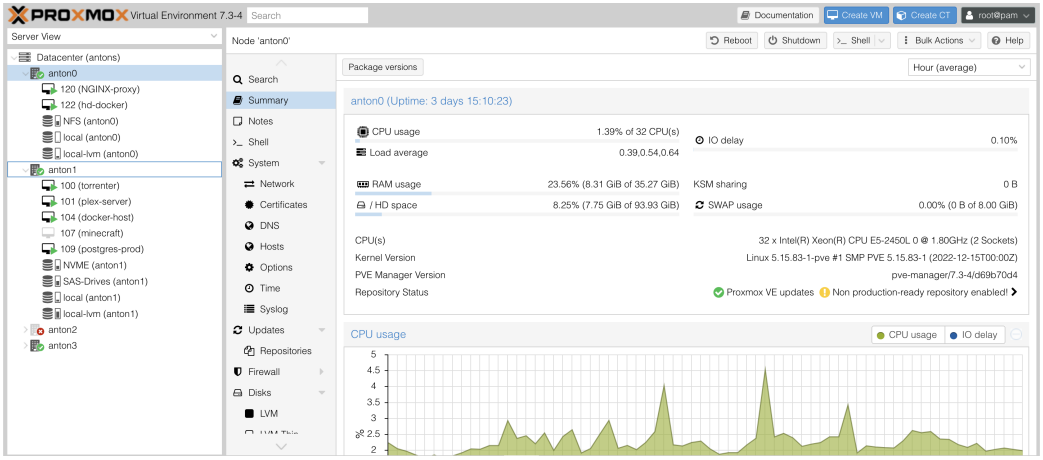
Chapter 3

Development & Testing

3.1 Ingesting Sales Data

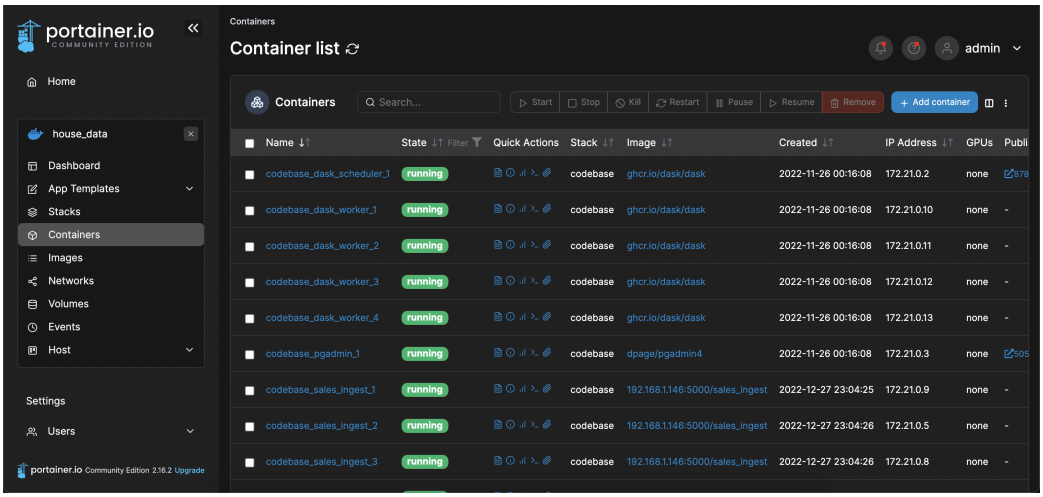
3.1.1 Initial Setup of Service

My application will rely on a fair few services for data storage, data processing and communication between the modules. This will all be hosted on servers at my home using virtual machines running on a hypervisor called Proxmox. Proxmox allows me to manage virtual machines over multiple servers from one web UI. By using VMs I can provision a set amount of computing resources for each service.



I will need to access these VMs remotely so I can develop the application

when I'm at school or away from home. To accomplish this I will use a VPN which allows me to tunnel into my home network from anywhere with an internet connection and access all devices on the network as if I was there. The VPN is run on a containerisation platform called Docker (running in a VM) that allows me to virtualise an application with very little overhead as opposed to a VM to run services independently of each other as if they were running on independent machines. By doing this I remove all dependency conflicts and give myself the ability to scale my platform by simply running more instances of a service if it supports that. Once I was able to access my network remotely I could start setting up other services on Docker. To do this I used a container management tool called Portainer which allows me to create, stop, start, restart and edit containers using a web UI. In the image below you can see all the containers I am running.



These services are all deployed using a 'docker-compose.yml' file where I can define what service I want to run and any variables that it will need. This makes adding new services and updating existing ones super easy. Here below you can see an example of how a YAML file is formatted and how a service is defined. The services I will be running on docker will include Dask for data processing, a container for inserting sales into the database, the webserver for the web interface, database management UI, and MongoDB for caching aggregations. The service being defined here is a web UI used to manage a PostgreSQL database.

```
version: "3.9"
    services:
        pgadmin:
            image: dpage/pgadmin4
            restart: always
            environment:
                PGADMIN_DEFAULT_EMAIL: user@eg.com
                PGADMIN_DEFAULT_PASSWORD: root
            volumes:
                - pgadmin-data:/var/lib/pgadmin
            ports:
                - "5050:80"
            networks:
                - net
```

The database this will be managing is not deployed using a container and instead runs in its virtual machine as it provides better performance than a Docker container. The VM is running Ubuntu Server 22.04 with Postgres 15 running on it as a standalone service. This VM has been allocated 6 cores and 16GB of RAM so that it can perform numerous queries at once and have enough RAM to store indexes in. Below are the SQL statements used to create the tables required.

```
        create table postcodes
        (
            postcode varchar(15) not null
                constraint postcode_key
                    primary key,
            street   varchar(70),
            town     varchar(50),
            district varchar(50),
            county   varchar(50),
            outcode  varchar(4),
            area     varchar(2),
            sector   varchar(6)
        );
        create unique index postcode_idx on postcodes
        (postcode);
```

```
create index area_idx on postcodes (area);
create index county_idx on postcodes (county);
create index district_idx on postcodes
(district);
create index outcode_idx  on postcodes
(outcode);
create index sector_idx on postcodes (sector);
create index street_idx on postcodes (street);
create index town_idx on postcodes (town);

create table houses
(
    houseid  varchar(150) not null
        constraint houseid_key
            primary key,
    paon     varchar(150),
    saon     varchar(150),
    postcode varchar(15)
        constraint postcodes_keys
            references postcodes,
    type     char
);
create index postcodes_index on houses
(postcode);
create index type_idx on houses (type);

create table sales
(
    tui      varchar(36) not null
        constraint tui_key
            primary key,
    price    integer,
    date     date,
    new      boolean,
    freehold boolean,
    ppd_cat  char,
    houseid  varchar(150)
        constraint houseid_fk
```

```
                references houses
);
create index date_idx on sales
(date);
create index freehold_idx on sales
(freehold desc);
create index ppd_cat_idx on sales (ppd_cat);
```

Once all the tables are created I will need to set up Apache Kafka which allows the software modules to communicate with each other. This is run on a VM as I had trouble running it on Docker as it has its own networking and I wasn't able to access the container from outside the Docker environment, so it is instead run as a standalone service. By running it on a VM I can get around this issue allowing me to access Kafka whilst running the application on my laptop so I can test it whilst developing.

3.1.2 Sales Ingest & Upload

Once all of the services are running I can start development. The first module that needs developing is the data ingest programme which will allow sales to be read from a text file and then inserted into the database. It will consist of 3 scripts one for reading sales from the text file and sending them to Kafka, another for checking if a new file has been released and then sending the sales to Kafka and finally one to receive sales from Kafka and then insert them into the database.

Sales Upload Programme

First I will start with the script to read sales from a text file as data is needed to test the other scripts. The Land Registry has a text file which contains all sales from 1995 to the most recently published month. This can be downloaded from their website. Inside the file are millions of rows each representing a sale with comma-separated columns storing the corresponding data for that sale. Below is an example of a row from the file.

To send this on Kafka and stored in a database it will need to be parsed into tuple format each representing a sale. Below is the code which opens the text file and then parses it. If the file opened is not a valid CSV file an error is thrown and the programme stops so that no further errors are created.

“{75520218-C2D9-4926-95E3-00061A7A3784}”,”248000”,=”2013-10-04 00:00”,”CR2 6PB”,”T”,”N”,”F”,”36”,”2 - 7”,”SELSDON ROAD”,””,”SOUTH CROYDON”,”CROYDON”,”GREATER LONDON”,”A”

```
from csv import reader

with open("./pp-complete.txt", "r") as f:
    # Opens the file
    try:
        csv_file = reader(f) # Parses the file into a
            list
    except:
        raise ValueError("Invalid CSV file")
```

Once the list of sales has been generated the sales need to be sent as messages in a Kafka topic. This is done by iterating through the list and then sending each row one after the other once it has been converted to the correct data type and encoded as bytes. Below is the code to do this.

```
for sale in csv_file:
    sale = [sale[0][1:-1]] + [i for i in sale[1:]]
    # Removes brackets from the transaction ID
    sale_bytes = str(sale).encode("UTF-8")
    # Converts the list into a string and
    #then encodes it into bytes
    self._producer.send("new_sales", sale_bytes)
    # Send the sale to kafka
```

After running this code I found it to be very slow. To find the reason for this I used CProfiler which allowed me to see the execution time of each line and how many times it was run. I concluded that although the operations by themselves are very quick when run 26,000,000 times it quickly adds up as a 1ms difference can take off 7 hours. To combat this I used a while loop as they are quicker than for loops and I then used a generator to apply the operations en-masse to the list which is quicker than doing each sale individually. Generators are quicker as they don’t load all the results into

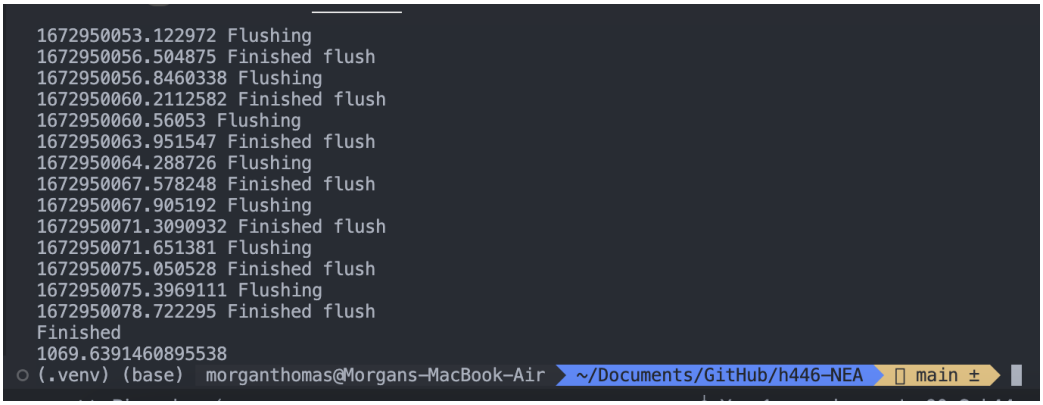
memory at once and effectively lazy-load the results as you iterate through. Then apply the operations en-masse using the map function as it is built into python and designed for this exact purpose so is a lot quicker as is written in native C. As opposed to converting the list to a string and then encoding it as bytes it instead encodes it into bytes directly using the pickle library which reduced the time further. Another issue I found more to do with efficacy rather than efficiency is the Kafka official library for Python would often have issues connecting to my Kafka instance but would act as if it was connected but not send messages. Eventually, I concluded that I should use a better-supported library and switched to the confluent_kafka library which not only eliminates this issue but is more efficient at sending messages and has a much larger group of maintainers.

After solving all of these issues the final code ended up looking like this.

```
with open("./pp-complete.txt", "r") as f:
    csv_file = reader(f)
    csv_file = map(lambda x: dumps([x[0][1:-1]] +
                                   [i for i in x[1:]]),
                   csv_file)
    # Applies operations to all sales
while True:
# While is quicker than a for loop
    try:
        list_bytes = next(csv_file)
        # Converts list to byte array
        while True: # Retries if message fails
            try:
                self._producer.produce("new_sales",
                                       list_bytes)
                # Send each sale as bytes
                self._producer.poll(0)
                break
            except BufferError:
            # Flushes message buffer if full
                print(time.time(), "Flushing")
                self._producer.flush()
                # Sends any unsent messages
                # that are clogging up the
```

```
                # buffer
                print(time.time(),
                      "Finished flush")
    except StopIteration:
        self._producer.flush()
        break
```

All of this is then put into a class and broken up into functions for opening the file, connecting to Kafka and uploading the sale. I then tested it to see how long it would take to upload the entire sales history in one go. Below is the output once the programme has finished running.

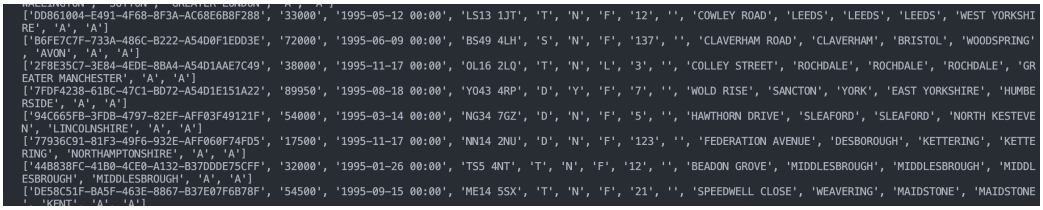


You can see it took 1069 seconds. This works out to about 24,300 inserts per second making it perfectly adequate for my use case as this only needs to happen once and then be updated each month. With this kind of speed, the update should only take 5 seconds. I also had To test to see if Kafka was receiving the messages so I made a quick script to receive and print out messages from Kafka.

```
consumer.subscribe(["new_sales"])
while True:
    msg = self._consumer.poll(1.0)
     # Fetches the latest message from kafka
    if msg is None:  #Checks the message isnt empty
        continue
    if msg.error():  # Checks there are no errors
        print("Consumer error: {}".format(msg.error()))
```

```
            continue
        sale: List = loads(msg.value())
        # Converts the bytes into a python list
        # and asserts the correct data type validating
        # the message
        print(sale)
```

I then successfully ran this code and received back all the sales that had been sent to Kafka. Below you can see the output for this.



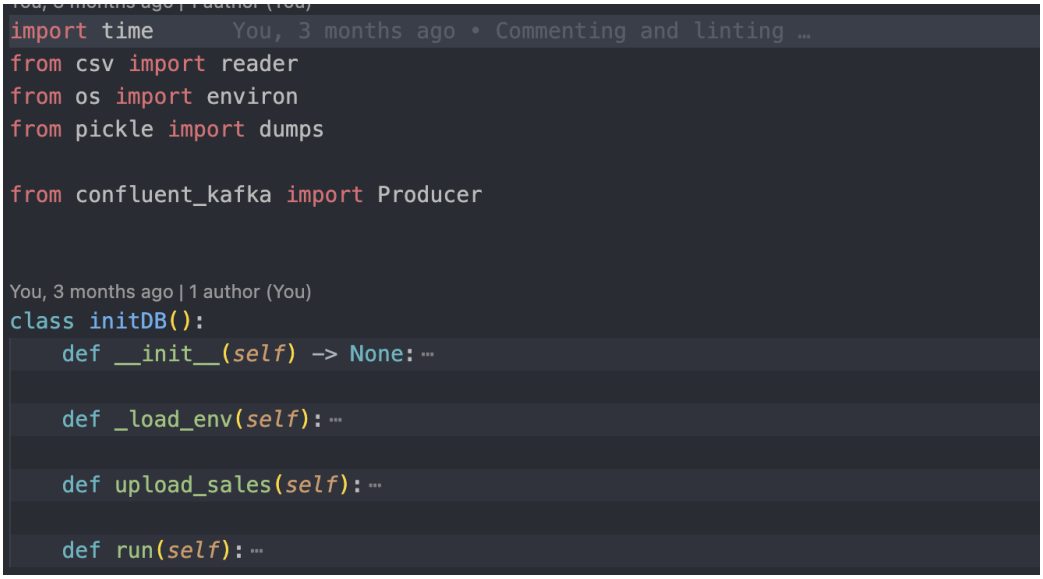
Unfortunately due to the nature of this module, the stakeholders weren't able to provide any meaningful input as the user has nothing to do with the insertion of the sales and only deals with the aftermath of it. But it does meet the success criteria of being able to upload new data to the website for analysis.

Code Layout

Sales Ingest Programme

Now that the upload programme has worked I can develop the ingest server as it needs the data from the upload script to test it. Firstly I need to be able to receive the data from Kafka. This was done using the confluent_kafka module like the uploader. I realised as well that I don't want to be storing the login credentials for Kafka and the database in the programme. To solve this I used a .env file where I can store key-value pairs containing all the data. This can then be loaded into the programme using the os module.

```
def _load_env(self):
    # Loads the environment variables
    # where 1st arg is the name and
    # 2nd is the default value
```




```
        self._DB = environ.get("DBNAME", "house_data")
        self._USERNAME = environ \
                         .get("POSTGRES_USER", "house_data")
        self._PASSWORD = environ.get("POSTGRES_PASSWORD",
                                     "password")
        self._HOST = environ.get("POSTGRES_HOST",
                                 "192.168.1.133")
        self._KAFKA = environ.get("KAFKA",
                                  "192.168.1.146:9092")
```

Once the environment variables have been loaded they can be accessed to connect to the database and Kafka. Now that the programme has connected to Kafka it can start receiving messages. The code for this is the same as the test for the sales upload script above. Once I confirmed it could receive messages I then needed to process the data to be inserted. This meant breaking the data up into its corresponding data and changing a few of the data types. For example, the new_build and sale_type columns were stored as characters despite only having two possible values so they were changed to boolean. The dates were also represented in a string format so they needed to be converted into a Python Datetime object so that they can be inserted correctly. All of these operations also validate the data as if any were incorrect an error

would get thrown and they wouldn't be inserted.

```
new = True if sale[5] == "Y" else False
# Convets to boolean type
freehold = True if sale[6] == "F" else False
# Converts to boolean type
date = datetime.strptime(sale[2], "%Y-%m-%d %H:%M")
# Converts string to datetime object
```

This allows the columns to be more easily searchable as indexes can be applied to them dramatically decreasing the search time. Without an index, the database has to do a linear search which takes a lot longer than a binary search which is used by the database when a column is indexed. A unique identifier also had to be created for each house so that they could be referenced by each sale. Initially, I created a sha256 hash of the attributes which would then be set as the houseid. After some testing, I realised that this would take too long and was completely necessary as I could simply append the attributes to each other making a unique string.

```
houseID = str(sale[7]) + str(sale[8]) + str(sale[3])
```

Once I had completed all of these operations the data could be inserted into the database. I originally did this using the official library for PostgreSQL. This worked fine and allowed me to insert all the data into the database. Below is the code for this. When I tested this though I was getting about 50-60 inserts per second which are acceptable but when doing 26,000,000 sales it would take about 120 hours or 5 days which is an unnecessary amount of time. I used CProfiler again to see where its main source of time loss was and it would often get stuck waiting for the insert to finish. To combat this issue I switched to an asynchronous model which allowed me to run other code whilst it was waiting for the query to finish. Luckily there is another library for PostgreSQL which is asynchronous and uses the same interface as the standard one so it was as easy as putting an await keyword in front of the function and then it would run asynchronously. This means I was able to get about 500 transactions per second meaning I could get the insertion done in about 14 hours which is a lot quicker than the other method. Due to how I structured the insertion system the ingest script could be horizontally scaled by simply running more instances of it and Kafka

```
cur.execute("INSERT INTO postcodes \
            (postcode, street, town, district, county, outcode, area, sector) \
            VALUES ($1,$2,$3,$4,$5,$6,$7,$8) ON CONFLICT (postcode) DO NOTHING;",
            sale[3], sale[9], sale[11], sale[12], sale[13], postcode_parts[0],
            postcode_parts[1], postcode_parts[2])  # Insert into postcode table

cur.execute("INSERT INTO houses (houseID, PAON, SAON, postcode) \
            VALUES ($1,$2,$3,$4,$5) ON CONFLICT (houseID) DO NOTHING;",
            houseID, sale[7], sale[8], sale[4], sale[3])  # Insert into house table
cur.execute("INSERT INTO sales (tui, price, date, new, freehold, ppd_cat, houseID) \
            VALUES ($1,$2,$3,$4,$5,$6,$7) ON CONFLICT (tui) DO NOTHING;",
            sale[0], int(sale[1]), date, new, freehold, sale[14], houseID) # Insert into sales
table
```

would split the messages up between all the instances. This would make it x times quicker where x is the number of instances running as long as the database could keep up. I found a good number was about 4 instances as it would be quick enough and wouldn't completely bog down the database.

After testing my ability to search the types of areas I realised that it was very slow as I would often have to do multiple joins and linear searches as I could use the correct index as the data wasn't in the correct format. This lead me to create another table called area which had two columns the area_type and the name of the area. A compound key was made of these to stop duplicates from being inserted and a full-text index was applied to the area name column allowing for searching using partial bits of the name like "Che" for "Chester". To accomplish this I needed to insert every area along with its corresponding type. These area types consisted of "postcode", "street", "town", "district", "county", "outcode", "area", and "sector". The last three are sub-parts of a postcode allowing for great greater granularity between town and street. For this I needed to break the postcode down into three parts. This was done you RegEx which allows you to search a string for parts which match a certain pattern. This pattern is defined as a string which is then passed to the function along with the string to search. Below is the code which breaks the postcode down into its fundamental parts and then reassembles it as each area.

```
        postcode_re = "^(?:(?P<a1>[Gg][Ii][Rr])(?P<d1>)
--------(?P<s1>0)(?P<u1>[Aa]{2}))|(?:(?:(?:(?P<a2>
--------[A-Za-z])(?P<d2>[0-9]{1,2}))|(?:(?:(?P<a3>
```

```
--------[A-Za-z][A-Ha-hJ-Yj-y])(?P<d3>[0-9]{1,2}))|
--------(?:(?:(?P<a4>[A-Za-z])(?P<d4>[0-9][A-Za-z]))|
--------(?:(?P<a5>[A-Za-z][A-Ha-hJ-Yj-y])(?P<d5>[0-9]
--------?[A-Za-z]))))-(?P<s2>[0-9])(?P<u2>[A-Za-z]{2}
--------))$" # RegEx for extracting postcode parts
        parts = re.findall(postcode, postcode_re)[0]
        outcode = parts[0] + parts[1]
        area = parts[0]
        sector = parts[0] + parts[1] + "-" + parts[2]
```

Once all the area types have their corresponding value they can be inserted into the database. I did this using a simple for loop which iterates through the area types and their values and then passes them into an SQL statement to be executed one by one.

```
area_types = ["postcode", "street", "town",
              "district", "county", "outcode",
              "area", "sector"]
areas = [sale[3], sale[9], sale[11], sale[12],
         sale[13], postcode_parts[0],
         postcode_parts[1], postcode_parts[2]]
# Extracts area values from sale
values = []
for idx, area_type in enumerate(area_types):
    area_data = (area_type, areas[idx])
    await self._conn \
        .execute("""INSERT INTO areas (area_type, area)
        VALUES ($1, $2) ON CONFLICT (area_type, area)
        DO NOTHING;""", area_data)
```

This code worked and was able to successfully insert the areas into the table. When I ran it with the rest of the code to insert all the sales into the database it would only get about 10-20 transactions per second. I then found that each insert into the areas table was taking 50ms so when that is multiplied by 8 it's about a second per sale which would take forever. Instead, I researched ways to batch-insert data using the PostgreSQL library. I came across an 'executemany' function where you pass it a list of data and a single SQL statement and it will batch insert that data. This is more efficient as each transaction has a set amount of overhead so by doing them all at once I

remove a lot of that overhead. This got the transaction time down to about 200ms which is a lot better but still a relatively slow time. The code below is the final implementation of the batch insert.

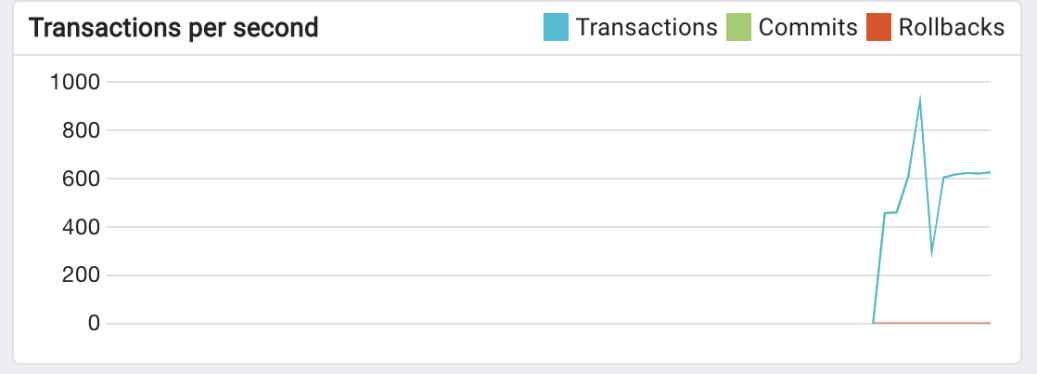
```
areas = [sale[3], sale[9], sale[11], sale[12],
         sale[13], postcode_parts[0],
         postcode_parts[1], postcode_parts[2]]
         # Extracts areas values from sale
values = []
for idx, area_type in enumerate(self._areas):
    # Iterates area_types where idx is a counter
    area_data = (area_type, areas[idx])
    values.append(area_data)
    # Create list of areas and their types
    await self._conn \
    .executemany("""
                 INSERT INTO areas (area_type, area)
                 VALUES ($1,$2) ON CONFLICT
                 (area_type, area)
                 DO NOTHING;""", values)
    # Batch inserts areas
```

I further researched how to improve the performance of a database when inserting lots of data and found indexes are the main cause of slowdown as they take a lot of CPU time to calculate. This is especially prevalent with a full-text index as they require a lot of preprocessing. To combat this issue I manually disabled the index whilst it is loading large amounts of data into the database and then reenable it to process the index all at once. This would only really work when the database is being initialised as there won't be any other queries being executed. When updating the database with monthly sales the index will have to be enabled but this should work out fine as there are a lot fewer sales to be inserted so the issue is not as amplified.

Now that the code is complete and working I need to be able to run it on my servers. I did this using a Docker image and defined the parameters using a 'Dockerfile'. In this file, I select what images I want to base the container on which would be python in my case and then install any dependencies and set the run command. This can then be pushed to a docker image repository on my server where it can be pulled and have multiple instances run.

```
FROM python:3.9.7 # Defines python version

WORKDIR /app

COPY . .

RUN wget https://github.com/librdkafka/v1.9.0.tar.gz
&& tar xvzf v1.9.0.tar.gz
&& cd librdkafka-1.9.0/
&& ./configure
&& make
&& make install
&& ldconfig

# Downloads and compiles kafka library

RUN python3 -m
 pip install --upgrade pip setuptools wheel
RUN python3 -m pip install confluent_kafka asyncpg

CMD ["python3", "__init__.py"]
# Defines the command to run the script
```

To test this I ran the programme and gave it the complete file containing all of the sales from 1995 to the present day. To confirm that it was inserting sales I was able to use the database management web UI which showed this.



Here you can see the sales being inserted and the rate at that they are being inserted. I left it to run and when I came back it had finished. It had successfully inserted all of the sales as you can see in the image below.

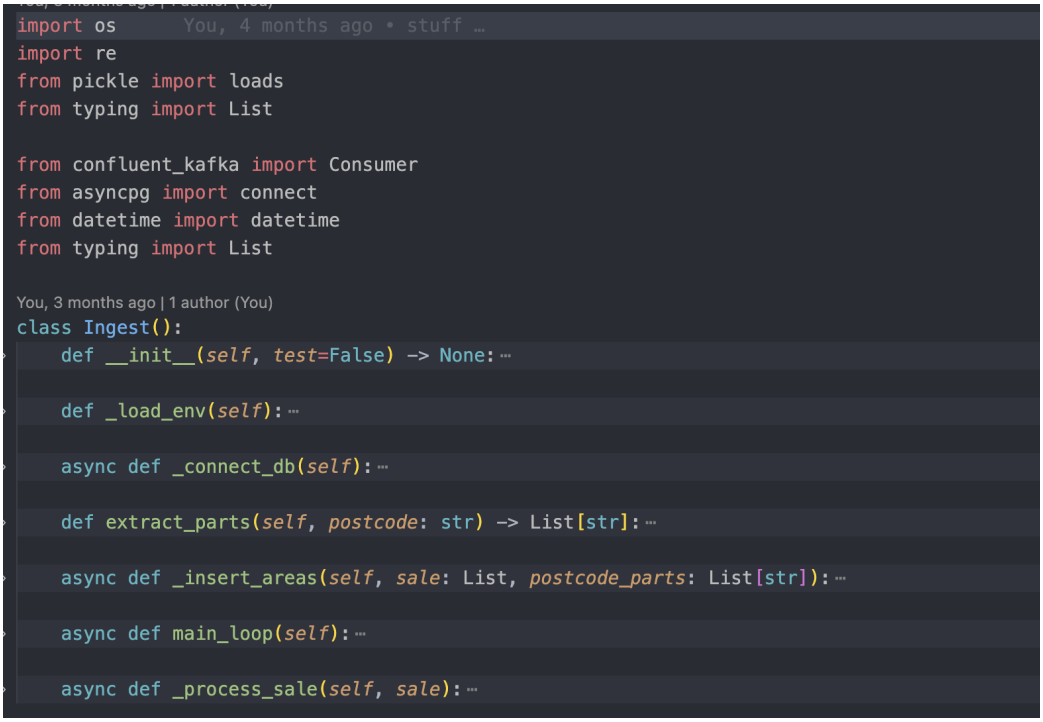
The little feedback I did get from my stakeholders for this module was about the ability to update the database with new sales in a very little amount of time meaning that they get the most up-to-date statistic and the ability to quickly search for a specific area with auto-complete making their workflow more efficient. It also hit the success criteria of inserting data into a database in a third normal form where there is no duplication, avoids data anomalies, and maintains referential integrity.

Code Layout

3.1.3 Sales Updated Checker & Uploader

Now that I can read sales from a file and then upload those sales to the database I need to be able to keep that data up to date. This will be done using a separate script that will pull the updated file from the land registry to see if it has changed since the last month. If it has it will then be sent to Kafka to be uploaded to the database. Luckily this script will use a lot of the code from the script used to upload the complete file. The main functionality will be getting the hash of the file and then comparing it to the previous one. This will be done using the SHA256 hash of the file which will be compared to the previous hash stored in a settings table in PostgreSQL. The code for downloading the file will use the Python library called requests.

```
file_link = "http://prod.publicdata.landregistry.gov.uk
    .
----s3-website-eu-west-1.amazonaws.com/
```




```
----pp-monthly-update.txt"
file = get(file_link).content
# Downloads file using HTTP get request and then stores
# the content
```

Once the file is downloaded it needs to be hashed so that it can be compared to the previous update file to see if it has changed or not. This is done using the Hashlib library which is a standard library for python so it is highly optimised and tested. Once the hash has been calculated it then needs to be compared to the hash of the previous update is fetched from the database using SQL the code for this is below.

```
file_hash = sha256(file).hexdigest()
# Calculate hash of file
self._cur.execute("SELECT data FROM settings
------------------WHERE name='update_hash';")
# Fetched previous hash from the database
prev_hash = self._cur.fetchone()
```

```
# Check to see if file has been inserted already
if prev_hash is not None:
    prev_hash = prev_hash[0]
if file_hash != prev_hash:
    # Compare hash to hash of old file
    self._update_database(file, file_hash)
    # Updates database with new hash
else:
    print("No new file yet")
```

Once the hashes have been compared if they are the same the file needs to be inserted into the database. This is done using the same code as the sales upload programme so I won't show that but there is a slight difference as a file downloaded needs to be converted so that it can be iterated through as a list since its current form is a byte list.

```
csv_file = reader(StringIO(file.decode("UTF-8")))
# Converts the bytes into a string and then into an
# in memory buffer to be read by the CSV reader
```

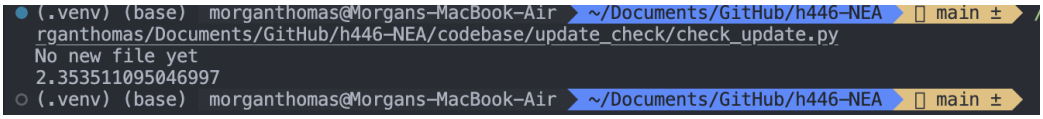
This is another module that doesn't get any feedback from the stakeholders but it does again hit one of the success criteria of being able to upload new data to be analysed. It was also tested by setting the hash in the database to an empty value so it would insert the update file. The output can be seen below.

```
(.venv) (base)  morganthomas@Morgans-MacBook-Air  ~/Documents/GitHub/h446-NEA  main ±
rganthomas/Documents/GitHub/h446-NEA/codebase/update_check/check_update.py
New file being uploaded
1673909378.859071 Flushing
1673909382.795545 Finished flush
7.780152797698975
(.venv) (base)  morganthomas@Morgans-MacBook-Air  ~/Documents/GitHub/h446-NEA  main ±
```

You can see it uploaded the new file in a matter of seconds and added the new hash to the database shown in the images below.

	name	data
1	last_updated	1673909383.554881
2	update_hash	eb3f3d378982bc3a43ab209f624647ce31f6993ca78df21b0415543eb08883c0

If I run the programme again it will say that there is no new file and will then stop the programme.



This module has successfully passed all of the tests and works as expected.

Code Layout

3.2 Data Processor Programme

Now that the data is stored in a database and is being kept up to date it can be processed and sent to the users. This needs to be done in an acceptable amount of time so that the user isn't sitting and waiting for the data to load and be processed. To accomplish this I'm using a few technologies. The first one is multiprocessing which gives me the ability to run aggregations on the data in parallel making them run quicker by as many cores as I allocate to that aggregation. I will also use Kafka to give me the ability to run multiple instances of the data Processor and share the queries between them increasing the number of queries I can run at once by the number of instances I have running. On top of this will be a query cache where the results will be stored along with the date when they were calculated. This means that when a query comes in it can be compared to the cache and if there is one in the cache it will be returned. This will reduce the number of calculations dramatically as once one area has been aggregated it won't need to be done again for another month or so when more data is released. The sub-components of this module will be the data loader to fetch the sales from the database; the aggregator which produces the stats from the sales; the processor which coordinates the data loader and aggregator along with

```python
from hashlib import sha256
from io import StringIO
from os import environ
from pickle import dumps

from confluent_kafka import Producer
from psycopg2 import connect
from requests import get
from csv import reader

You, 2 months ago | 1 author (You)
class checkForUpdate():
    def __init__(self) -> None: ⋯

    def _load_env(self): ⋯

    def _fetch_file(self): ⋯

    def _update_database(self, file, file_hash): ⋯

    def _send_file_db(self, file):      You, 4 months ago • stuff … ⋯

    def run(self):
        self._fetch_file()
```

checking the cache and receiving new queries and finally the web API which allows users to submit and monitor queries along with searching for an area.

3.2.1 Data Loader

To produce accurate statistics the programme needs to be able to access the data reliably and make sure it maintains integrity. This is done using PostgreSQL which is an ACID-compliant database. ACID stands for atomicity, consistency, isolation, and durability. Atomicity means that a transaction will either succeed or fail so you won't get part of it working and some of it not. This means that only the full data is returned and not part of it so the statistics will always be accurate. Consistency means that all the data will comply with the rules of the database so , for example, a house must have a postcode and a house number so therefore you can be sure that all the data is there. Durability means that once a transaction is committed it will always be there unless changed so even if the server crashes as long as

the transaction has been committed it will be there.

When accessing the data it will need to be joined from the 3 tables to get all the appropriate data. The most efficient way to do this is by reducing the number of rows it needs to search and then joining them as opposed to joining all the tables and then filtering. This is done by getting the area you want to search and then getting all the postcodes in that area and then getting all the houses which aren't of the 'other' type and then getting all the sales that have a ppd_cat of 'A'. The query for this is below.

```
SELECT s.price, s.date, h.type, h.paon, h.saon,
        h.postcode, p.street, p.town, h.houseid
FROM postcodes AS p
INNER JOIN houses AS h ON p.postcode = h.postcode
    AND p.outcode = 'CH64'
INNER JOIN sales AS s ON h.houseid = s.houseid
    AND h.type != 'O'
WHERE s.ppd_cat = 'A';
```

The line where it says "p.outcode = 'CH64'" can be swapped to work with any area type like a county or town. This will be done dynamically depending on what arguments are passed to the data loader. The area_type will be inserted using a python f-string as it has a discrete value so there is no need to escape it. The area name will be passed as an argument to the execute function along with the query string. The code for this is below.

```
query = f"""SELECT s.price, s.date, h.type, h.paon,
    h.saon, h.postcode, p.street, p.town, h.houseid
    FROM postcodes AS p
    INNER JOIN houses AS h ON p.postcode = h.postcode
    AND p.{self.area_type} = %s
    INNER JOIN sales AS s ON h.houseid = s.houseid
    AND h.type != 'O'
    WHERE s.ppd_cat = 'A';
    """
self._cur.execute(query, (self.area,))
# Executes query and inserts area
```

Before running this query though it needs to verify that the area and area_type are valid so that time is wasted running an expensive query. To verify the area_type the value is compared to a list of valid types. Once it has

passed that it runs a query to check that there are corresponding postcodes for that area. Once it passes all of these tests only then is the main query run. This reduces the number of unnecessary queries that are executed so that the database isn't bogged down.

```
self._areas = ["postcode", "street", "town",
               "district", "county", "outcode",
               "area", "sector"]
if self.area_type not in self._areas:
    # Checks if area type is in list
    # and raises error if not
    raise ValueError("Invalid area type")
self._cur.execute(f"SELECT postcode
--------------------FROM postcodes
--------------------WHERE {self.area_type} = %s
--------------------LIMIT 1;",
                      (self.area,))
# Searches postcode table for area
if self._cur.fetchall() is not []:
    return True
    # Returns true if result isn't empty
else:
    raise ValueError(f"Invalid
------------------------{self.area_type} entered")
    # Raises error if no postcodes returned
```

Below is a table with testing inputs their expected output and actual output and as you can see it worked perfectly.

Input	**Expected**	**Actual**
postcode, CH2 1DE	True	True
jwnjdbf, CH2 1DE	Inavlid Area Type	Invalid Area Type
postcode CH7384 342	Invalid postcode entered	Invalid postcode entered

Now that the area and area type has been verified the data is fetched from the database using the query from before. This is done using the PostgreSQL Python library but the synchronous version as the packages I plan to use to do aggregations are not asynchronous compatible. This part is fairly simple as it uses a lot of similar code from the previous parts it is simply executing

an SQL query and then storing the results in a variable and then raises an exception if there are no results. To store the data I'm using a Pandas data frame as it allows me to easily convert it into a Dask data frame later on for parallel processing. You can see this process in the code below.

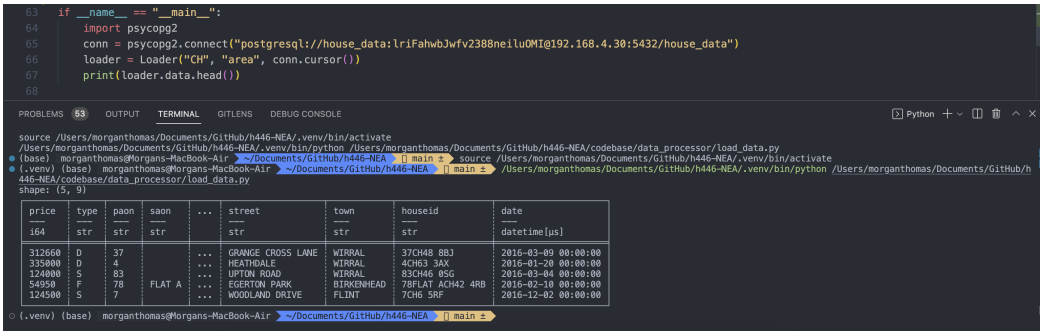
```
query = f"""SELECT s.price, s.date, h.type, h.paon,
        h.saon, h.postcode, p.street, p.town,
        h.houseid
    FROM postcodes AS p
    INNER JOIN houses AS h ON p.postcode = h.postcode
        AND p.{self.area_type} = %s
    INNER JOIN sales AS s ON h.houseid = s.houseid
        AND h.type != 'O'
    WHERE s.ppd_cat = 'A';
    """
self._cur.execute(query, (self.area,))
# Executes SQL query
data = self._cur.fetchall()
# Returns results from query
if data == []:
    # Raises error if no results returned
    raise ValueError(f"No sales for area {self.area}")
else:
    self._data = pd.DataFrame(data)
    # Stores results in pandas dataframe
```

Once this has been run the rest is handed off to the aggregator to process and reduce the data into statistics to give meaningful insight. Below you can see the output for the data loader when asking for sales from the 'CH' area.

Code overview.

3.2.2 Data Aggregator

The data aggregator is the main and most important part of the entire application. This is because it has to be highly performant to reduce the amount of time it takes for a query to run and highly accurate to provide the correct statistics. To achieve this I'm going to use a service called Dask which is made up of 3 components. A scheduler distributes tasks between workers, workers who then execute those tasks, and then a python library to interact




```python
from datetime import datetime, timedelta

import polars as pl
from typing import List

You, 1 second ago | 1 author (You)
class Loader():
    def __init__(self, area: str, area_type: str, db_cur) -> None:…

    def verify_area(self):…

    def fetch_area_sales(self) -> List:…

    def _format_df(self, data):…

    @property
    def data(self) -> pl.DataFrame:…

    @property
    def latest_date(self):…
```

with the scheduler and workers. Dask is built on top of the most popular data processing library for Python which is Pandas. It works by taking the data you give it and then dividing that up into smaller Pandas data frames running on the workers so the operations can be applied in parallel and then collected at the end.

The first part needed is a Dask cluster which can be set up using the docker-compose which is as simple as writing a few lines and then uploading them to the server. Below are the settings required to run a Dask cluster.

```
# Data Processer
dask_scheduler:
```

```
image: "ghcr.io/dask/dask"
restart: "always"
ports:
    - "8787:8787"
    - "8786:8786"
networks:
    - net
command: ["dask-scheduler"]

dask_worker:
image: "ghcr.io/dask/dask"
deploy:
    mode: replicated
    replicas: 4 # Sets amount of workers
command: ["dask-worker", "tcp://dask_scheduler:8786"]
networks:
    - net
```

Now that I have a desk cluster running I need to be able to communicate with it using python. This is done using the Dask library as seen below.

```
def __init__(self, data):
    self._load_env() # Loads enviroment variables
    self._dask_client = Client(self._DASK_SCHEDULER)
    # Connects to dask cluster
    print(self._dask_client.ncores())
    # Prints info about cluster to validate connection
```

The aggregator uses a similar structure to the data ingest module where it loads environment variables for all of the connections like the Dask cluster and the PostgreSQL database. Now that it is connected I need to upload the data fetched using the data loader. This can be done in one line but after some testing, I realised it's not quite plug-and-play and needed a bit of tuning. As stated before the data is broken down into lots of little data frames but the amount of data frames depends on the size of the data you are using so the general rule of thumb is about 10 for every 100MB of data. To calculate the number of partitions required I had to calculate the size of the data frame and then use that to say how many partitions. Luckily pandas have a function which returns the size in memory of the data frame. Below

you can see the implementation of this which is a continuation of the code above.

```
    self._data = data.data
    self._df_mem = data.mem_consump
    # Gets size of dataframe
    partitions = 10 if self._df_mem < 100
            else int(10 * (self._df_mem % 100))
    # Calculates amount of partitions needed
    self._dd_df = dd.from_pandas(self._data,
                    npartitions=partitions).persist()
    # Uploads data frame to the cluster and
    # persists it in memory
```

Now that the data is uploaded to the cluster I can crack on with the aggregation functions. To start with I created a function to remove the outliers from the dataset as I would need to do this before all aggregations. This also meant that it had to be as efficient as possible as it would get run a lot. The function needed to calculate the mean of the data than the standard deviation and then filter the data removing all data greater than the mean plus 3 times the standard deviation.

```
def remove_outliers(self):
    std = self._dd_df["price"].std()
    # Calculates standard deviation
    mean = self._dd_df['price'].mean()
    # Calculates mean
    filtered_df = self._dd_df \
                    .loc[(self._dd_df["price"]
                            < mean+(sd_thresh*std))
                        & (self._dd_df["price"]
                            > mean-(sd_thresh*std))]
    # Filters database
return filtered_df
```

After testing this function on a range of datasets varying in size from a few thousand sales to a quarter million I realised that it was unbelievably slow. It would take anywhere from 1 minute for the smaller dataset up to 25 minutes for the larger ones. After checking over my code numerous times and asking people online in forums I realised my mistake. I had misinterpreted

Dask, instead of it just being a multiprocessing library for python to speed up data aggregations it was aimed at processing humongous datasets ranging anywhere from a couple of hundred gigabytes to terabytes. With those sorts of datasets you're not expecting it to be done anytime soon so you can afford more overhead but when that overhead is applied to datasets of a much smaller size it is completely disproportionate which results in queries taking an unnecessary amount of time.

After realising my issue I started researching python libraries for parallelised data processing and came across a library called Polars. Polars is a library written in Rust so it is as performant as C or C++ but is much safer when it comes to memory management and garbage collection. This makes it a highly attractive systems programming language. Polars was designed as a competitor to pandas which is written in pure Python so suffers from poor performance. Below is a graph comparing the time taken to group data and perform aggregations on it. You can see Polars is almost 2x quicker and as the dataset scales Polars only gets quicker relative to Pandas.

Implementing Polars

Fortunately, Polars has a very similar interface to Pandas as it wanted to be used by people who use Pandas in their everyday workflow. This meant that implementation was a rather trivial task and it allowed me to use the pseudo code from my design to help me. Unlike Dask Polars is designed to run locally on your machine so a cluster is not needed. This allowed me to remove a lot of code that was required to connect to the cluster and initialize the dataset on it. I also had to make a slight change to the data loader as that returned the data as a Pandas data frame since that was compatible with Dask which was as simple as swapping 'pd' for 'pl' and using 'import polars as pl' instead of 'import pandas as pd'. Now that I was using Polars instead of Dask I could move onto developing the code which will produce the statistics.

Property Type Proportions

This statistic tells the user how many of each type of property is in an area along with its corresponding proportion. To accomplish this it will have to

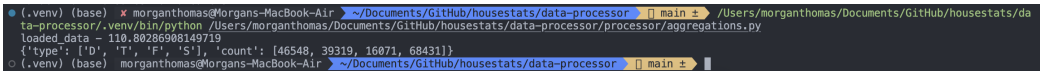
group all the sales by their property type and then return a count of each one along with the total number of properties to calculate the proportions. Due to the possibility of there being multiple sales for one property, it will also have to filter out all duplicates. Once this is all done the data will be returned as a dictionary. The code for this is below.

```
def calc_type_proportions(self) -> Dict:
    df = self._data # Copies dataframe to process
    df = df.unique(subset=["houseid"]) # Filters out
        duplicates
    df = df.groupby("type").count() # Gets count of
        each property type
    data = df.to_dict(as_series=False) # Converts
        results to a dict
    return data
```

It was then tested by fetching the data for the 'CH' area and then passing it to be aggregated. Below is the code for this.

```
import psycopg2
import time

start = time.time()
conn = psycopg2.connect("<DATABASE_URL>") # Connects to
    database
data_loader = Loader("CH", "area", conn.cursor()) #
    Fetches sales for area
print("loaded_data_=_" + time.time()-start)

agg = Aggregator(data_loader) # Passes sales to
    aggregator
res = agg.calc_type_proportions() # Calculates
    proportions
print(res) # Prints results
```

Below is a screenshot showing the output after running this code. It shows it was able to successfully output the proportions for this area along with the total amount of properties.



Monthly Average Price

This is one of my key statistics as it gave a view of the property market on a month-by-month basis. It would calculate the mean price of house sales each month for all property types and then for each individually. By doing this I allow the users to get a more tailored look at the data for their needs. I'll accomplish this by first splitting the data into their respective types and then sorting the sales by date. These sales will then be grouped by month and the mean will then be calculated. This will be repeated for each house type and then the same will be done for all house types at once. Below is the code for this.

```
def _calc_average_price(self) -> Dict:
    df = self._data.partition_by("type", as_dict=True)
    # Splits data into separate property types

    house_types_means = {}
    for house_type in df:
        temp_df = df[house_type]
        house_types_means[house_type] = temp_df \
            .sort("date") \
            # Sorts sales
            .groupby_dynamic("date", every="1mo") \
            # Groups sales into months
            .agg(pl.col("price").mean()) \
            # Calculates the mean
            .to_dict(as_series=False)
            # Converts it to dict

    all_sales = self._data
    house_types_means["all"] = all_sales \
        .sort("date") \
        .groupby_dynamic("date", every="1mo") \
        .agg(pl.col("price").mean()) \
        .to_dict(as_series=False)
```

```
    data = {
        "type": [key for key in sorted(
            house_types_means)],
        # List all property types alphabeticly
        "prices": [house_types_means[key]["price"] for
            key in sorted(house_types_means)],
        # List all average prices corresponding order
            to property types
        "dates": house_types_means["all"]["date"]
        # List all of the dates
    }
    return data
```

I tested this using the same area as the proportions function so the testing code was the same except instead of calling 'calc_type_proportions' I called 'calc_average_price'. Unfortunately, the output for this is rather long so I can't show it below as it shows the average sale price for every month since 1995 for every property type. To see if my results were correct I compared the most recent points against the Land Registry Price Index which can be found here https://landregistry.data.gov.uk/app/ukhpi. When I compared the numbers I realised my results were all quite a bit higher than the Land Registry's.

My initial conclusion was I hadn't removed the outliers from the dataset. I had implemented this before with Dask so switching to Polars was a breeze due to how similar they are.

```
        std = all_sales.select(pl.col("price")).std().
            collect()[0, 0]
        mean = all_sales.select(pl.col("price")).mean()
            .collect()[0, 0]
        temp_df = all_sales.filter((pl.col("price") <
            mean+(2*std)))
```

After implementing this I again tested it but the results were still off from the House Price Index. I then did some reading on the Office for National Statistics website as they publish how they calculate the HPI. There they stated that instead of using the arithmetic mean they use the harmonic mean which helps reduce the effect large sales have on the average. The

harmonic mean can be calculated by multiplying all of the items together and then finding the nth route where n is equal to the number of items in the list. After doing some further reading on it I realised that the numbers can quite easily get out of hand, for example, house sales are in the hundreds of thousands so each time I multiply them together I am increasing the product by 100000 each time which when done 10,000 times produces a very large number potentially causing it to overflow which would then crash the programme this. To combat this people recommend using log rules to find the mean as it can be done more efficiently.
$x * x = logx + logx$
$x^{1/y} = logx/logy$

Using these two rules above I can calculate the harmonic mean by getting the log of all the values and calculating the arithmetic mean of them and then doing e^x where x is the result of the arithmetic mean. This is implemented below in Polars.

```
house_types_means[house_type] = temp_df \
    .sort("date") \ # Sort sales by date
    .groupby_dynamic("date", every="1mo") \ # Group by
       month
    .agg(
        pl.col("price") \
            .log() \ # Log of all values
            .mean() \ # Arithmetic mean
            .exp() # e^x of result
        ) \
    .to_dict(as_series=False) # Convert to dict
```

I then tested this and got results that were within a few thousand of the Land Registries. I put the error down to the Land Regsitry including Scottish property sales which aren't included in my data.

Quantity & Volume

These two statistics are very similar and are hence calculated in a very similar manner. The structure of the code will also look very similar to the monthly average price as these are calculated for each property type and then for all types and it is done on a month-by-month basis. Polars had an inbuilt method for count and sum so I can copy the code for the average and simply

change the aggregation function for each one. The function will be the same for each one minus a few variable name changes so for brevity I'll just show what the line will be for volume and quantity.

```
        # For sale quantity
        .agg(pl.col("price").count())

        # For price volume
        .agg(pl.col("price").sum())
```

Since the Land Registry doesn't publish these statistics and I couldn't find anywhere that did I will just have to assume that the code works based on the output I received since I have nothing to compare it to. Also since the average code works and these share a lot of the same code it would be logical to assume they also work. Similarly to the average function the output of this is a long list of numbers so I don't see much point in showing the output for each one.

Quick Stats

To give the user a brief and up-to-date view of an area at the top of the page will be the most recent figure for each statistic along with how it compares to the previous month. To do this I will need to get the most recent number off the top of each list and the second most recent. The percentage change will then be calculated by dividing the most recent by the oldest. Once I have the percentage change and the most recent figure they will be added to a dictionary. The dictionary will be laid out as shown below.

```
        {
            "current_month": 0,
            "average_price": 0,
            "average_change": 0,
            "current_sales_volume": 0,
            "sales_volume_change": 0,
            "current_price_volume": 0,
            "price_volume_change": 0,
            "expensive_sale": 0
        }
```

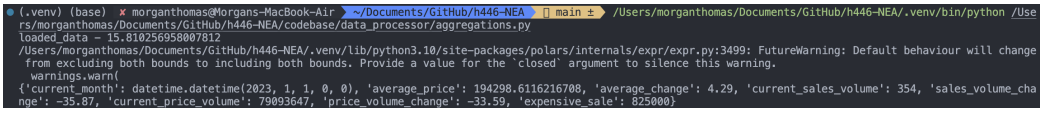
To get all of these values I will need to have already run all of the other

aggregations. Once they have been run I will then go through each one doing the aforementioned. The code for this is below. It will also find the most expensive sale for the current month. To do this will has to filter the sales to only ones from this month. Then it searches the sales to find which one has a value equal to the greatest value in the table. This will be the most expensive sale of the month

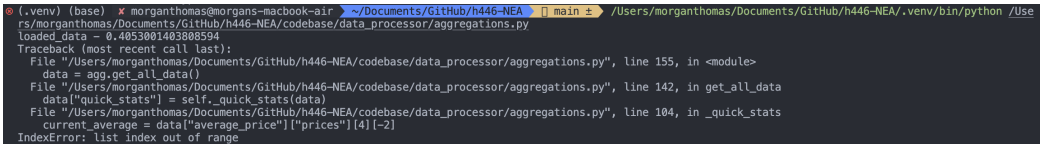
```
def _quick_stats(self, data) -> Dict:
    current_month = data["average_price"]["dates"][-2]
    # Gets the current month
    current_average = data["average_price"]["prices"
        ][4][-2]
    # Gets the most recent average
    prev_average = data["average_price"]["prices"
        ][4][-3]
    # Gets second most recent average
    current_average_change = round(100*(current_average
        -prev_average)/prev_average,2)
    # Calculates the percentage change to 2 decimal
        places


    current_sales_vol = data["monthly_sales_volume"]["
        volume"][4][-2]
    # Gets most recent sale volume
    prev_sales_vol = data["monthly_sales_volume"]["
        volume"][4][-3]
    # Gets second most recent sales volume
    current_sales_vol_change = round(100*(
        current_sales_vol-prev_sales_vol)/prev_sales_vol
        ,2)
    # Calculates the percentage change to 2 decimal
        places

    current_price_vol =  data["monthly_price_volume"]["
        volume"][4][-2]
    # Gets most recent price volume
    prev_price_vol = data["monthly_price_volume"]["
```

```
        volume"][4][-3]
    # Gets second most recent price volume
    current_price_vol_change = round(100*(
        current_price_vol-prev_price_vol)/prev_price_vol
        ,2)
    # Calculates the percentage change to 2 decimal
        places

    expensive_sale = (self._data
        .filter(pl.col("date").is_between(current_month
            , current_month + timedelta(days=31)))
        # Filters sales to this month
        .filter(pl.col("price") == pl.col("price").max
            ())
        # Searches for sale with maximum price
        )[0,0]

    # Puts results into a dictionary to be returned
    quick_stats = {
        "current_month": current_month,
        "average_price": current_average,
        "average_change": current_average_change,
        "current_sales_volume": current_sales_vol,
        "sales_volume_change": current_sales_vol_change
            ,
        "current_price_volume": current_price_vol,
        "price_volume_change": current_price_vol_change
            ,
        "expensive_sale": expensive_sale
    }
    return quick_stats
```

I tested this code using the area I've been using for all the other tests and it worked perfectly. Below are the results of this test, the error message is simply a warning that can be ignored.



After this success, I then decided to test it on some other areas so I put my postcode in to get some results. I encountered an IndexError when I ran this test. After doing some investigating I realised it was because areas with fewer sales might not have any sales for this month so when it goes to fetch the most recent average it isn't there as there have been no sales in that month. Below you can see the error message for this.



To remediate this error I added try statements around the bits of code where it fetches the most recent stats. These allowed me to catch the IndexErrors and then set the values to zero as a default. The code for this is below.

```
try:
    current_sales_vol = data["monthly_sales_volume"]["
        volume"][4][-2]
    prev_sales_vol = data["monthly_sales_volume"]["
        volume"][4][-3]
    current_sales_vol_change = round(100*(
        current_sales_vol-prev_sales_vol)/prev_sales_vol
        ,2)
except IndexError:
    current_sales_vol = 0
    current_sales_vol_change = 0

try:
    current_price_vol =  data["monthly_price_volume"]["
        volume"][4][-2]
```

```
        prev_price_vol = data["monthly_price_volume"]["
            volume"][4][-3]
        current_price_vol_change = round(100*(
            current_price_vol-prev_price_vol)/prev_price_vol
            ,2)
    except IndexError:
        current_price_vol = 0
        current_price_vol_change = 0
```

Now the quick stats function will work for any area I give it and if there is any missing data it will return zero as the value by default.

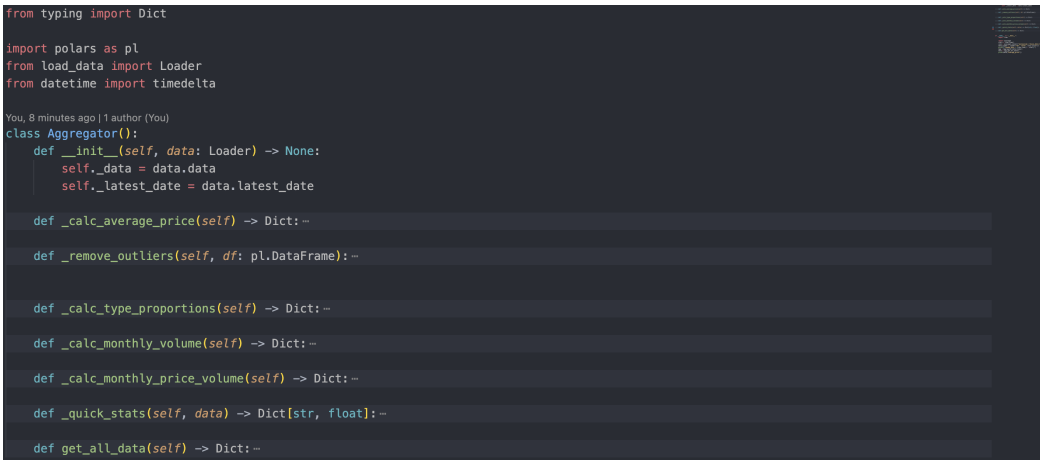
Run Function

This function is what ties all the other functions together and then returns the data. The data will be returned as a dictionary. There isn't much to this function as it simply calls the functions and then stores the returned values in a dictionary.

```
def get_all_data(self) -> Dict:
    data = {
        "average_price": self._calc_average_price(),
        "type_proportions": self._calc_type_proportions
            (),
        "monthly_sales_volume": self.
            _calc_monthly_volume(),
        "monthly_price_volume": self.
            _calc_monthly_price_volume(),
    }
    data["quick_stats"] = self._quick_stats(data)
    # Passes aggregations to quick stats functions

    return data
```

There isn't much need to test this function as long as all the other functions have been tested and are in working order this one will work fine. Now that all of the aggregations have been programmed I can move on to integrating it with Kafka to receive jobs.

Code Layout



3.2.3 Intergrating With Kafka

To analyse areas effectively the processing needs to be detached from the webserver. This will prevent the web server from getting clogged up with requests and stop anyone else from accessing the website. Instead, the user sends a request to analyse an area and then they are given a link to check if it has been processed or not. To implement this I need to create a Kafka client which can check the database to see if an area has been cached or not; check if that cache is out of date; load data using the 'loader'; a process that data using the 'aggregator' and then update the cache.

Firstly I need to create the __init__ function to connect to the database and Kafka. This will be very similar to the other modules so I won't go into detail about it. It will also need to load the credentials for the databases from environment variables which will be a function. Below is the code for both of these.

```
def __init__(self):
    self._load_env()
    self._sql_conn = psycopg2.connect(f"postgresql://{
        self._SQL_USERNAME}:{self._SQL_PASSWORD}@{self.
        _SQL_HOST}:5432/house_data")
    # Connect to the SQL database
```

```
        self._cur = self._sql_conn.cursor()

        self._mongo_conn = MongoClient(f"mongodb://{self.
            _MONGO_USERNAME}:{self._MONGO_PASSWORD}@{self.
            _MONGO_HOST}:27017/?authSource=house_data")
        # Connect to MongoDB for caching
        self._mongo_db = self._mongo_conn["house_data"]

        self._consumer = Consumer({
                'bootstrap.servers': self._KAFKA,
                'group.id': 'PROCESSOR',
                'auto.offset.reset': 'earliest'
            })
        # Connect to Kafka

    def _load_env(self):
        # Loads the enviroment variables
        self._DB = os.environ.get("DBNAME")
        self._SQL_USERNAME = os.environ.get("POSTGRES_USER"
            )
        self._SQL_PASSWORD = os.environ.get("
            POSTGRES_PASSWORD")
        self._SQL_HOST = os.environ.get("POSTGRES_HOST")
        self._KAFKA = os.environ.get("KAFKA")
        self._MONGO_HOST = os.environ.get("MONGO_HOST")
        self._MONGO_USERNAME = os.environ.get("
            MONGO_USERNAME")
        self._MONGO_PASSWORD = os.environ.get("
            MONGO_PASSWORD")
```

Now that the boilerplate code has been set up I can start developing the first of the key functions which is the cache checker. This will work by generating the query ID based on the area and area type. These will then be used to search mongoDB. If a result is found it will then check if it is out of date or not. If it is out of date it will return False and if it isn't it will return True. If no result is found at all it will return False.

To generate the query ID the area and area type will simply be concatenated together and have any spaces removed. I'll create a function for this

as I will be performing it a lot throughout the process. The code for this is below

```
def _calc_query_id(self, area: str, area_type: str) ->
    str:
      query_id = (area + area_type).replace("-", "")
      # concatenate strings and remove spaces
      return query_id # Return result
```

Input	Expected	Actual
CHESTER, TOWN	CHESTERTOWN	CHESTERTOWN
CH2 1DE, POSTCODE	CH21DEPOSTCODE	CH21DEPOSTCODE
CH, AREA	CHAREA	CHAREA

Now that I can generate the query id to search and have tested it using the table above I need to be able to get the date the dataset was last updated. This will be done using SQL as the update checker module writes the date and time it is updated to the database. If there is no date recorded in the database or an incorrect date provided it will return 01/01/1970 00:00:00 so that it will be forced to update the cache no matter what. To fetch the date from the database I will use a fairly basic SELECT SQL query which is below.

```
"SELECT * FROM settings WHERE name ='last_updated'"
```

The SQL query will return a UNIX timestamp which is an integer representing the number of seconds since 01/01/1970 00:00:00. This will then need to be converted into a python datetime object so that I can perform comparisons with it. Putting this together with the SQL the code will look like this.

```
def _get_last_updated(self):
    self._cur.execute("SELECT * FROM settings WHERE
       name = 'last_updated'")
    last_updated = self._cur.fetchone()
    # Fetch timestamp from database

    if last_updated == None:
        return datetime.fromtimestamp(0)
        # Return default time if no timestamp found
```

```
    else:
        if last_updated[1] is not None:
            return datetime.fromtimestamp(float(
                last_updated[1]))
            # convert timestamp to datetime object
        else:
            return datetime.fromtimestamp(0)
            # Return default time if invalid timestamp
```

I tested this code too by changing the value in the database and checking to see if it corresponded with the date returned and as you can see in the table below it worked.

Database Value	Expected	Actual
1680284312	31 03 2023 17:38:32 GMT+0000	31 03 2023 17:38:32 GMT+0000
1230454412	28 12 2008 08:53:32 GMT+0000	28 12 2008 08:53:32 GMT+0000
wu3gbk.qwjf	01 01 1970 00:00:00 GMT+0000	01 01 1970 00:00:00 GMT+0000
Nothing	01 01 1970 00:00:00 GMT+0000	01 01 1970 00:00:00 GMT+0000

Now that I have both of these functions working I can start programming the function to check the cache. It's quite a simple function and only consists of a few lines, generating id; searching the database and compare dates. Below is the code for this.

```
def _check_cache(self, area, area_type) -> bool:
    query_id = self._calc_query_id(area, area_type)
    # Generate query id
    query = self._mongo_db.cache.find_one({"_id":
        query_id})
    # Search database for cached query
    if query is not None:
        last_updated = self._get_last_updated()
        # Get date of last updated
        if query["last_updated"] < last_updated:
            # Compare dates
            return False # Isn't in cache
        return True # Is in cache
```

```
    else:
        return False # Isn't in cache
```

Unfortunately, I won't be able to properly test this code till there are queries that have been cached so I will have to come back and test this later once I have developed more of this. Now I can develop the function to load the data and then aggregate it so it can then be cached. This function will simply be calling the loader module and then pass that data into the aggregator module. Each part will be timed so that I can evaluate it against my success criteria. Firstly it will load the data using the loader module so the function for this is below.

```
def _get_area_data(self, area: str, area_type: str):
    lodr = Loader(area, area_type, self._cur)
    # Pass area details and database cursor to loader
    return lodr
```

Once the data is loaded it will then need to be aggregated using the function below.

```
def _get_aggregation(self, loader: Loader) -> Dict:
    agg = Aggregator(loader) # Init aggregator
    data = agg.get_all_data() # Run aggregations
    return data
```

Now both of these functions will be tied together along with some logic to time each part and save it to a variable to be written to the database later. The code for this function is below.

```
def _get_stats(self, area: str, area_type: str) -> bool
    :
    load_start = time.time() # Start timer
    data = self._get_area_data(area, area_type)
    # Load data
    load_time = time.time()-load_start # End timer

    if data is not None:
        start = time.time() # Start timer
        stats = self._get_aggregation(data) # Aggregate
        time_taken = time.time()-start # End timer
```

```
            query_id = self._calc_query_id(area, area_type)
                # Generate ID
            self._cache_query(stats, query_id, area,
               area_type, time_taken, load_time)
            # Save query to cache
            return True
        else:
            return False
```

This function won't be able to be tested since it uses the '_cache_query()' function which hasn't been written yet so it will all be tested together.

Now moving on to the query caching function. This function will need to take all of the data and statistics and then store them in the database. Data is stored in MongoDB in the BSON format which is essentially the same as JSON except it is slightly more performant and better for databases. Each entry is called a document and will be laid out like this.

```
{
    "_id": query_id,
    "area": area,
    "area_type": area_type,
    "data": stats,
    "last_updated": datetime.now(),
    "exec_time": exe_time,
    "load_time": load_time
}
```

If a query has already been cached but is out of date the document will need to be updated instead of simply inserting it. This is done by using the $set function which allows you to modify the properties of an existing document. The code for this is quite long but is only one of two functions being called depending on the scenario.

```
def _cache_query(self, stats: Dict, query_id: str, area
    : str, area_type: str, exe_time: float, load_time:
    float):
     query = self._mongo_db.cache.find_one({"_id":
        query_id})
     # Check cache for exisiting document
     if query is not None:
```

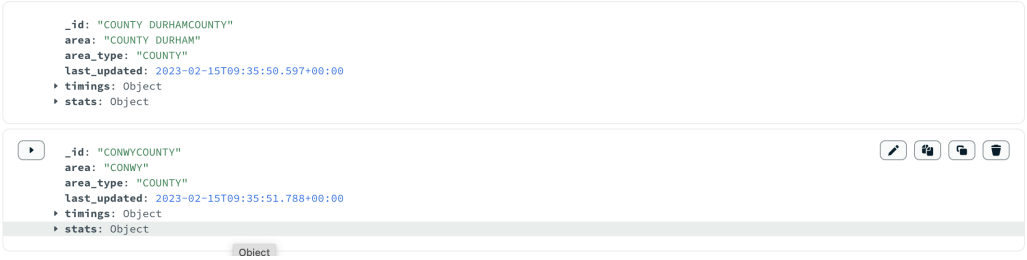
```
            self._mongo_db.cache.update_one(
                {"_id": query_id},
                {"$set": {
                    "data": stats,
                    "last_updated": datetime.now(),
                    "exec_time": exe_time,
                    "load_time": load_time
                    }
                }
                )
                # Updated exisiting document
        else:
            document = {
                "_id": query_id,
                "area": area,
                "area_type": area_type,
                "data": stats,
                "last_updated": datetime.now(),
                "exec_time": exe_time,
                "load_time": load_time
            }
            self._mongo_db.cache.insert_one(document)
            # Insert new document
```

Now that I have the caching function written I can test all the other functions. This will be done by giving it different areas and then seeing if they are added to the MongoDB database and then trying to analyse them and seeing it not process them as they are cached. Afer that I will then modify the date of one and get it to update them.

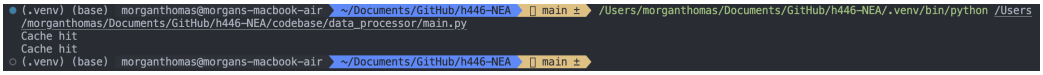
```
processor = Processor()
if not self._check_cache(area, area_type):
    print(query, "- Aggregating data")
    self._get_stats(area, area_type)
else:
    print("Cache hit)
````

The images below are from after running the test initially and as you can

see it successfully inserted them into the database.

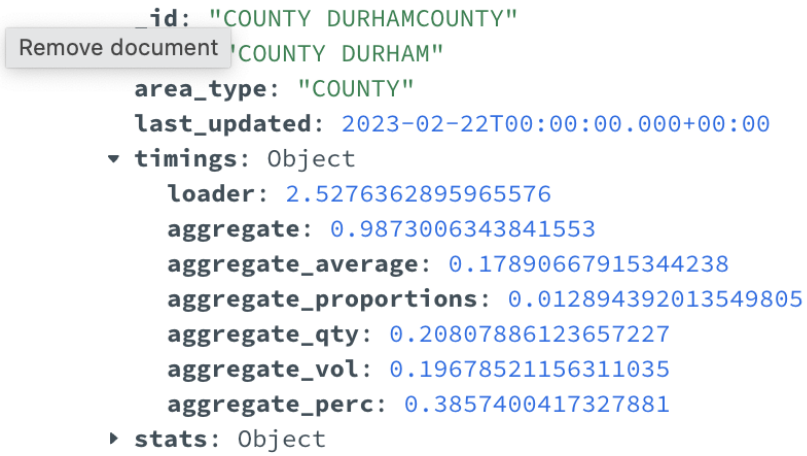


I then ran the same test again and got his printout in the terminal showing that it was able to detect the documents in the cache so didn't process them again.



I then manually edited the date on the first one to be before the last update.

The image below is the result of running the script again and as you can see it updated and changed the date.



All of these tests conclude that the caching and processing part is fully functional. Now all I need to do is programme the Kafka client so it can receive jobs and execute them. Fortunately, I have already done this for the sales ingest module so I can borrow some of the code from there to make it easier. It will need to recieve the message; decode it to the area and area type and then pass these onto the other functions to analyse and then cache if needed. The code for this is below.

```
def main_loop(self) -> None:
    self._consumer.subscribe(["query_queue"])
    # Subscribe to job queue
    print("Waiting for queries")
    while True:
        msg = self._consumer.poll(1.0)  # Fetches the
            latest message from kafka
        if msg is None:  # Checks the message is empty
            continue
        if msg.error():  # Checks there are no errors
            print("Consumer error: {}".format(msg.error
                ()))
            continue
        query: tuple = loads(msg.value()) # (area,
            area_type)
```

```
            query = tuple(map(lambda x: x.upper(), query))
                # Makes all items upper case

            print(f"{time.time()} - {query[0]}({query[1]})"
                )
            if not self._check_cache(*query): # Checks
                cache
                    print(query, " - Aggregating data")
                    self._get_stats(*query) # Gets stats
            else:
                    print(query, " - Cache hit")
                    continue
```

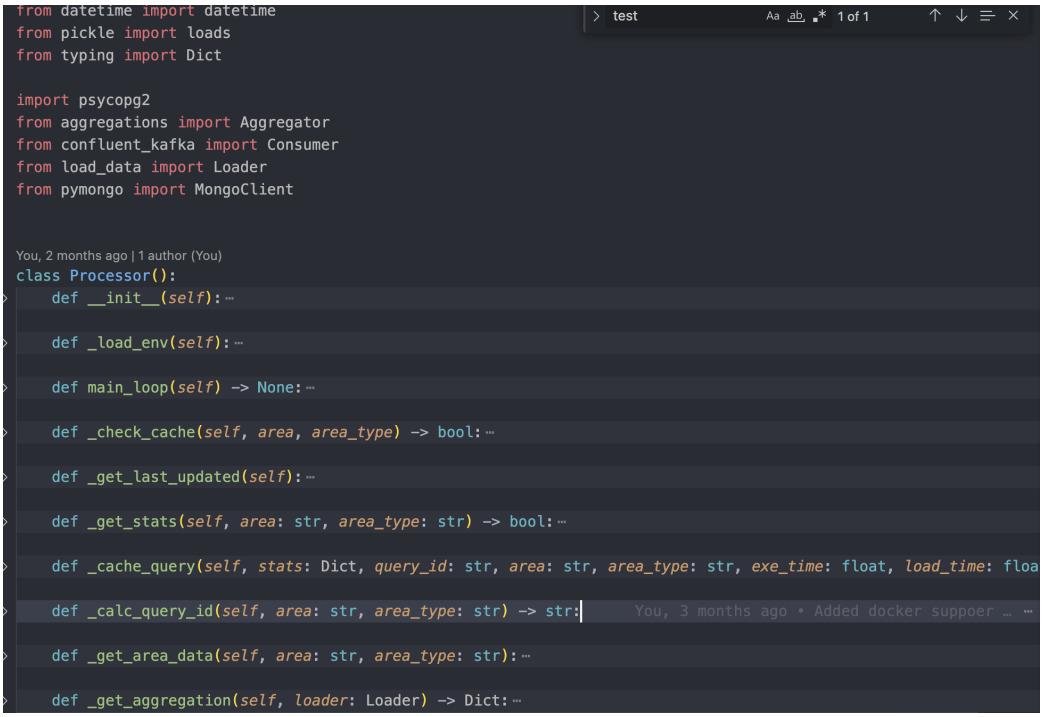
As you can see this is very similar to the sales ingest module with its layout. One of the features I added is to make all of the letters in the query uppercase to make sure that they are all stored in uppercase so that there are no duplicates for validity. To test this I will have to make a short script to send jobs along this Kafka queue.

```
            list_bytes = dumps(("CH", "AREA")) # Converts
                list to byte array
            self._producer.produce("new_sales", list_bytes)
                 # Send each sale as string to kafka
            self._producer.poll(0) # Wait for message to
                send
```

I made sure to run the aggregator and then run this script. The result of this script is below.

As you can see it was able to successfully receive the job and then aggre-

gated it and store the results in the cache. The aggregator completely works now and can be integrated into the web API once that is built. Fortunately, due to the design of the aggregator, new statistics can be added by simply adding another aggregation function to the aggregator class and editing the dictionary to include it. This will allow me to improve upon it in the future and add features requested by my stakeholders. Below is the code for the Kafka client and caching functions.



3.3 Web API

To allow for the retrieval of data and the ability to search it I will be using an HTTP API. This will be developed using the Flask library for python which is a lightweight webserver framework. Flask has a do-it-yourself methodology where it comes with the essential features and you build everything else you need. This allows it to be super compact and simple but can grow to be used in large applications if done correctly. To do this you need to plan out what

API routes your going to have and have a well-thought-out file structure so the application is modular and can be easily expanded in the future.

3.3.1 Planning

For my Flask application, I will be using a file structure which is quite common among flask applications and one that I have used before. It will look like the diagram below.

```
flask_app
    app
        __init__.py
        main
            __init__.py
            routes.py
        posts
            __init__.py
            routes.py
        questions
            __init__.py
            routes.py
        config.py
```

This structure will allow me to use a flask feature called blueprints which allows for greater modularity. They work by having their separate file which contains all the routes for that section. This file can then be imported into the __init__.py where it is added to the main flask instance and can be customised. For example, you can set a URL prefix for all routes in a blueprint or set authentication for all of those routes. In my case, I will only use one blueprint as I don't have many routes and they all come under the same category.

3.3.2 Boilerplate Code

There is some code that all Flask projects will have like the configuration file where all environment variables are stored and the __init__.py file where the flask app is initiated and the blueprints are imported into. When passing config to Flask it requires it in an object format where each environment variable is an attribute. Below is this.

```
import os

class Config:
    SECRET_KEY = os.environ.get('SECRET_KEY')
    SQL_USER = os.environ.get("DB_USER")
    SQL_PASSWORD = os.environ.get("DB_PASSWORD")
    SQL_HOST = os.environ.get("DB_HOST")
    KAFKA = os.environ.get("KAFKA")
    MONGO_HOST =os.environ.get("MONGO_HOST")
    MONGO_USER = os.environ.get("MONGO_USER")
    MONGO_PASSWORD = os.environ.get("MONGO_PASSWORD")
```

Now that I have a configuration file I can write the __init__.py file. This will contain the flask instance and is where all of the database or Kafka connections will be handled.

```
import psycopg2
from config import Config
from confluent_kafka import Producer
from flask import Flask, current_app
from pymongo import MongoClient


def create_app(config_class=Config) -> Flask:
    app = Flask(__name__) # Initializes flask app
    app.config.from_object(config_class) # Imports
        config to flask app



    kafka_producer = Producer({"bootstrap.servers": app
        .config["KAFKA"]})
    # Connects to kafka
    mongo_db = MongoClient(f"mongodb://{app.config['
        MONGO_USER']}:{app.config['MONGO_PASSWORD']}@{
        app.config['MONGO_HOST']}:27017/?authSource=
        house_data")
    # Connects to mongoDB
    sql_db = psycopg2.connect(f"postgresql://{app.
```

```
        config['SQL_USER']}:{app.config['SQL_PASSWORD']}
        @{app.config['SQL_HOST']}:5432/house_data")
    # Connects to PostgreSQL

    with app.app_context():
        # Stores connections in app context
        current_app.kafka_producer = kafka_producer
        current_app.mongo_db = mongo_db.house_data
        current_app.sql_db = sql_db


    from app.api import bp as api_bp # Imports
        blueprint
    app.register_blueprint(api_bp, url_prefix="/api/v1"
        )
    # Adds blueprint to flask app

    return app
```

Reading the code above you may be wondering what the app context is. This allows the database connections to be accessed from anywhere in the app even if they're in a completely different file. This means that each blueprint doesn't need its connections and they can all be managed centrally here.

The last bit of boilerplate code is for the API blueprint __init__.py file which simply initializes the bluprint and imports the routes from the routes.py file like this

```
from flask import Blueprint

bp = Blueprint("api", __name__)
# Initializes blueprint

from app.api import routes
# Imports routes
```

3.3.3 Routes

Now that all the boilerplate code is done I can get started on the routes. These are the HTTP paths that the user will go to fetch the data. My app will consist of 6 routes. The first one I will work on is the route to analyse an area.

Analyse Area

To create an analysis job for the data processors it will need to send a message in the Kafka topic. Once this message has been sent it will then need to return a URL with the query id for the user to access the results. The route will take two arguments the area and area type. These will be passed in the url as the path e.g. /CH/AREA, /CHESTER/TOWN. These will then be pickled into bytes to be sent via Kafka. The code for this is below.

```
@bp.route("/analyse/<string:area_type>/<string:area>")
# Take area and area type as arguments from the path
def index(area_type, area):
    data = dumps((area.upper(), area_type.upper()))
    # Pickle the area and area type into bytes
```

The area and area type are set to upper case for validation. Now that the job is ready to send it needs to access the app context to send it via Kafka. This is done using a with statement. I also added the same logic from the update checker where it will retry the message until it sends waiting for validation. The code for this is below which is a continuation of the above.

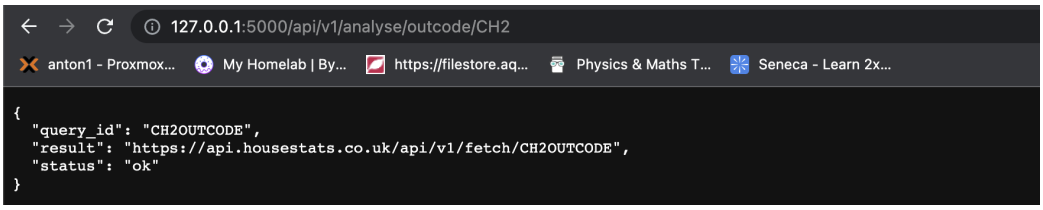
```
    with current_app.app_context(): # Access app
        context
        while True:
            try:
                print("Sending query")
                current_app.kafka_producer.produce("
                    query_queue", data)  # Send each
                    sale as string to kafka
                current_app.kafka_producer.poll(0) #
                    Wait for message to send
                break
            except BufferError:
```

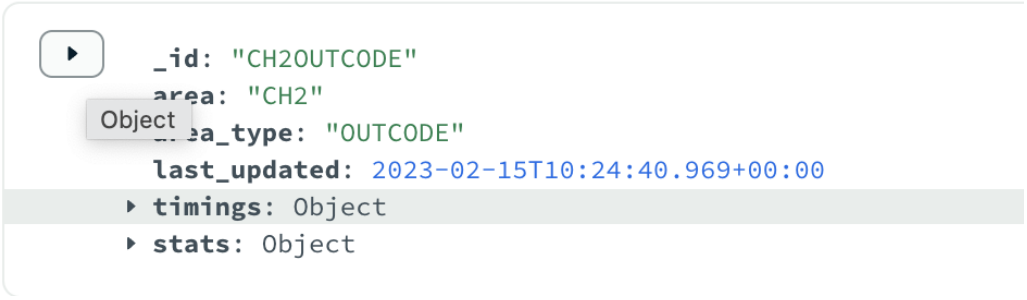
```
                current_app.kafka_producer.flush() #
                    Flush buffer if failed
```

Now that it has sent the job it needs to return a URL. This URL will contain the query id so when someone goes to it they will be able to see if the job is done and if it is they will get the results. It will be returned in JSON format. Below is the code for this.

```
    query_id = (area + area_type).replace("-", "").
        upper()
    return jsonify(
        status="ok",
        query_id=query_id,
        result=f"https://api.housestats.co.uk{url_for('
            api.fetch_results',query_id=query_id)}"
    )
```

The url_for function allows me to automatically link to the function I want even if I change the path for it. To test this I ran the data processor and the web server at the same time. I know if the results for my query appear in the database after my request it has worked.





You can see in the two images above that it was able to successfully send the job which then ran and stored the results in the database.

Now I need to write the route to check if the results are in the database and then fetch them if they are. This will simply consist of a mongoDB query to search the database. It will also need to get the date the data was last updated so it knows that even if there is data in the cache it will be overwritten once the job has been completed so don't return it. The code for this is the same as the data aggregator except it uses the app context.

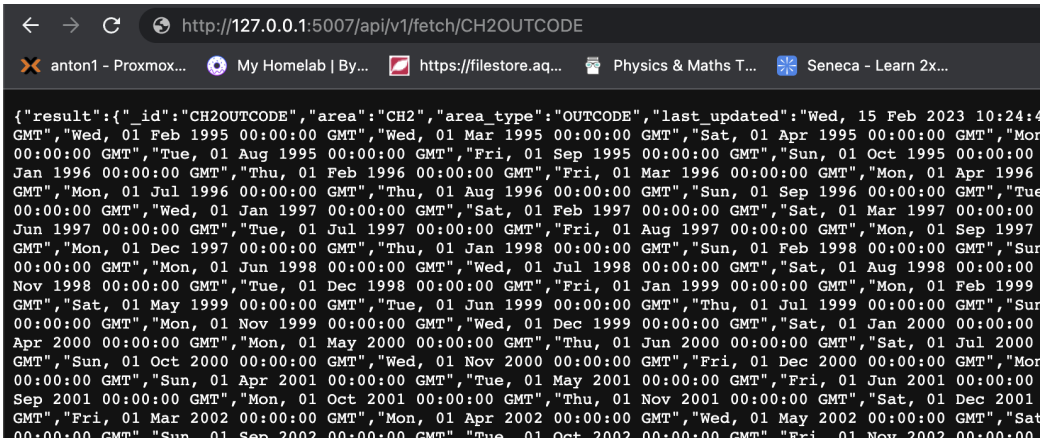
```
def get_last_updated():
    cur = current_app.sql_db.cursor()
    # Get SQL connection from app constext
    cur.execute("SELECT * FROM settings WHERE name = '
        last_updated';")
    # Get timestamp of latest update
    last_updated = cur.fetchone()
    if last_updated == None:
        return datetime.fromtimestamp(0)
    else:
        if last_updated[1] is not None:
            return datetime.fromtimestamp(float(
                last_updated[1]))
            # Convert timestamp to datetime object
        else:
            return datetime.fromtimestamp(0)
```

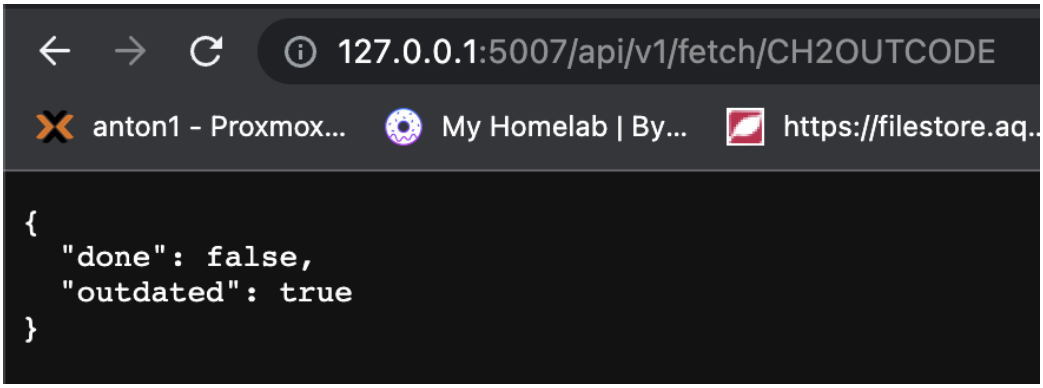
Now that I have this function copied over and edited I can write the route.

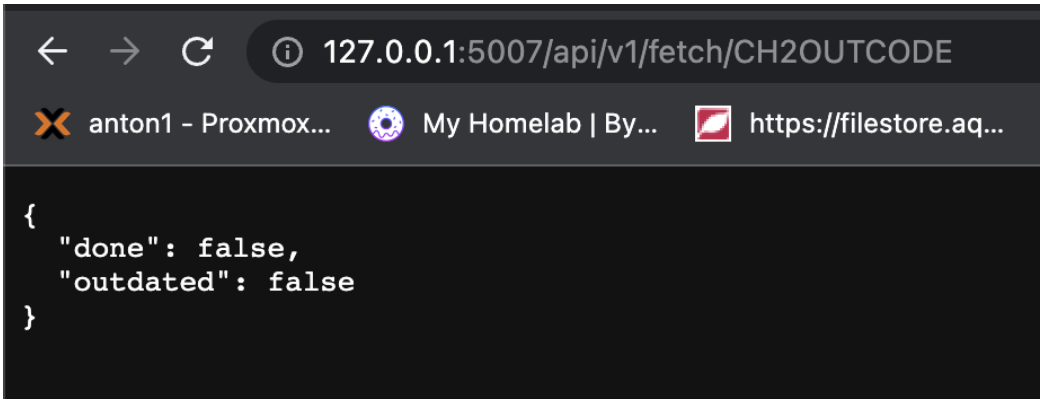
```
@bp.route("/fetch/<string:query_id>")
# Take query id as argument
def fetch_results(query_id):
    with current_app.app_context(): # Connect to app
        context
        query = current_app.mongo_db.cache.find_one({"
            _id": query_id.upper()})
        # Search cache for query
        if query is not None:
            if query["last_updated"] > get_last_updated
                ():
```

```
            # Check if cache is outdated
                return jsonify(
                    results=query,
                    outdated=False,
                    done=True
                    )
            else:
                # Return if query outdated
                return jsonify(
                    outdated=True,
                    done=False
                )
        else: # Return if query outdated
            return jsonify(
                outdated=False,
                done=False
            )
```

To test this I used the same area as before and went to that URL the results are below. To test the other scenarios I first changed the date manually to make it out of date and tested then I deleted it from the database to test it. The results are below.







As you can see from the results it performed as expected and passed all the tests.

3.3.4 Searching For Areas

This is quite a key feature for useability as it gives the user suggestions for areas they are searching for as they type which will making searching a lot easier as they don't have to keep attempting to type somewhere hoping it is correct. It will work by taking the query string in as an argument. It will then create an SQL query to search the area table for any matching ones. Once it gets the results these will then be returned to the user. The SQL query will use a full-text search index that has been created which will greatly improve the speed of searching. The query will look like this.

```
SELECT area, area_type
FROM areas WHERE substr(area, 1, 50)
LIKE '{query}%'
ORDER BY char_length(area)
LIMIT 10;
```

The 'LIKE' keyword is the key part of this and is what allows me to search using parts of an areas name. I also don't want to return more than 10 results as that would take up a lot of room on the screen and it would take significantly longer to load ruining the user experience. Implementing the SQL query with python would look like this below.

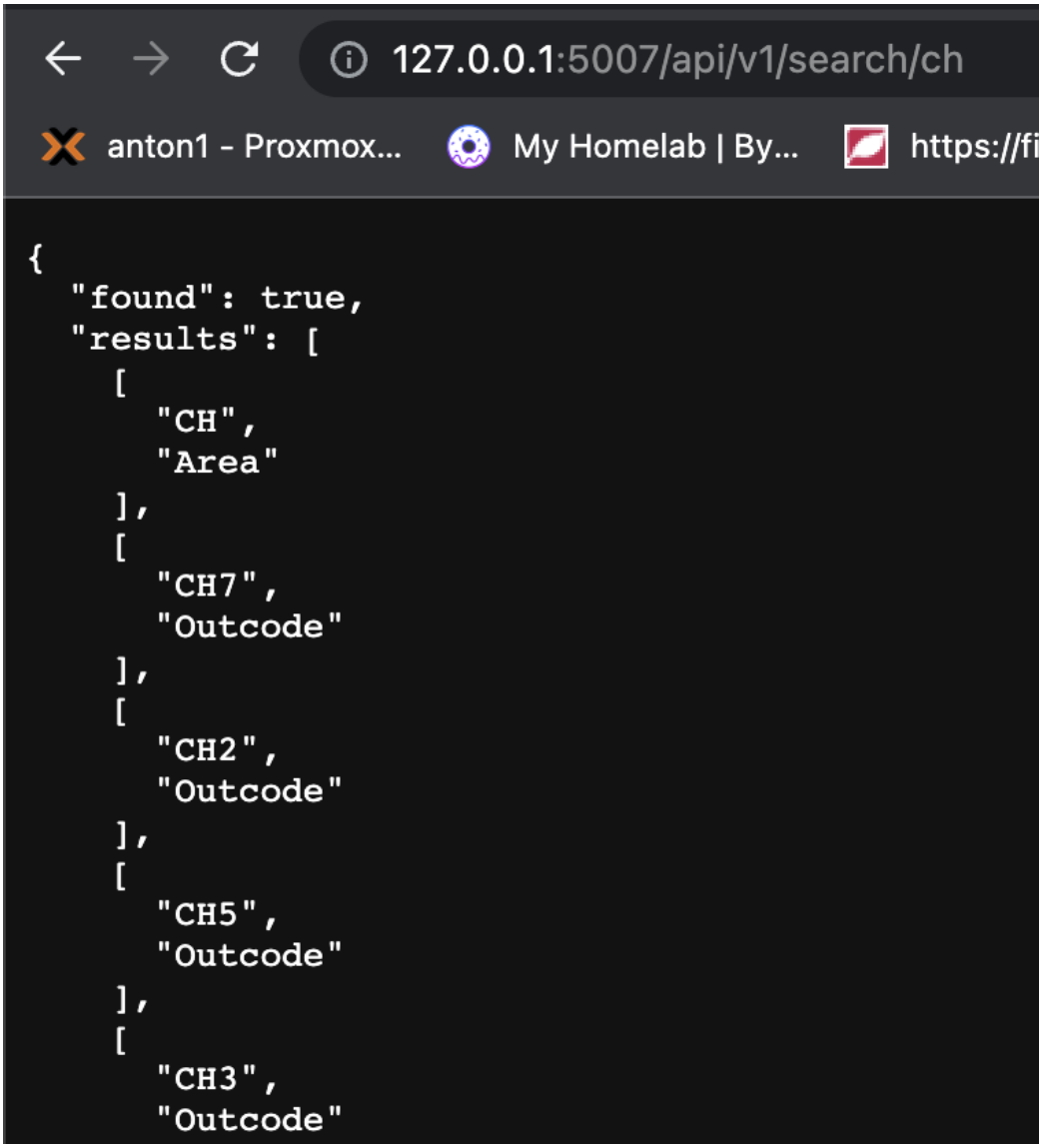
```
@bp.route("/search/<string:query>")
def search_area(query):
    query = urllib.parse.unquote(query).upper()
    # Decode URL safe characters like %20 for spaces

    sql_query = f"""SELECT area, area_type
                    FROM areas WHERE substr(area, 1, 50)
                    LIKE '{query}%'
                    ORDER BY char_length(area)
                    LIMIT 10;"""
    # Generate query with the query string

    with current_app.app_context():
        # Create app context
        cur = current_app.sql_db.cursor()
        cur.execute(sql_query)
        # Execute SQL query
        results: List[Tuple[str,str]] = cur.fetchall()
        # Store results
```

Now that the results have been fetched they need to be returned to the user. This will be done in JSON format like below.

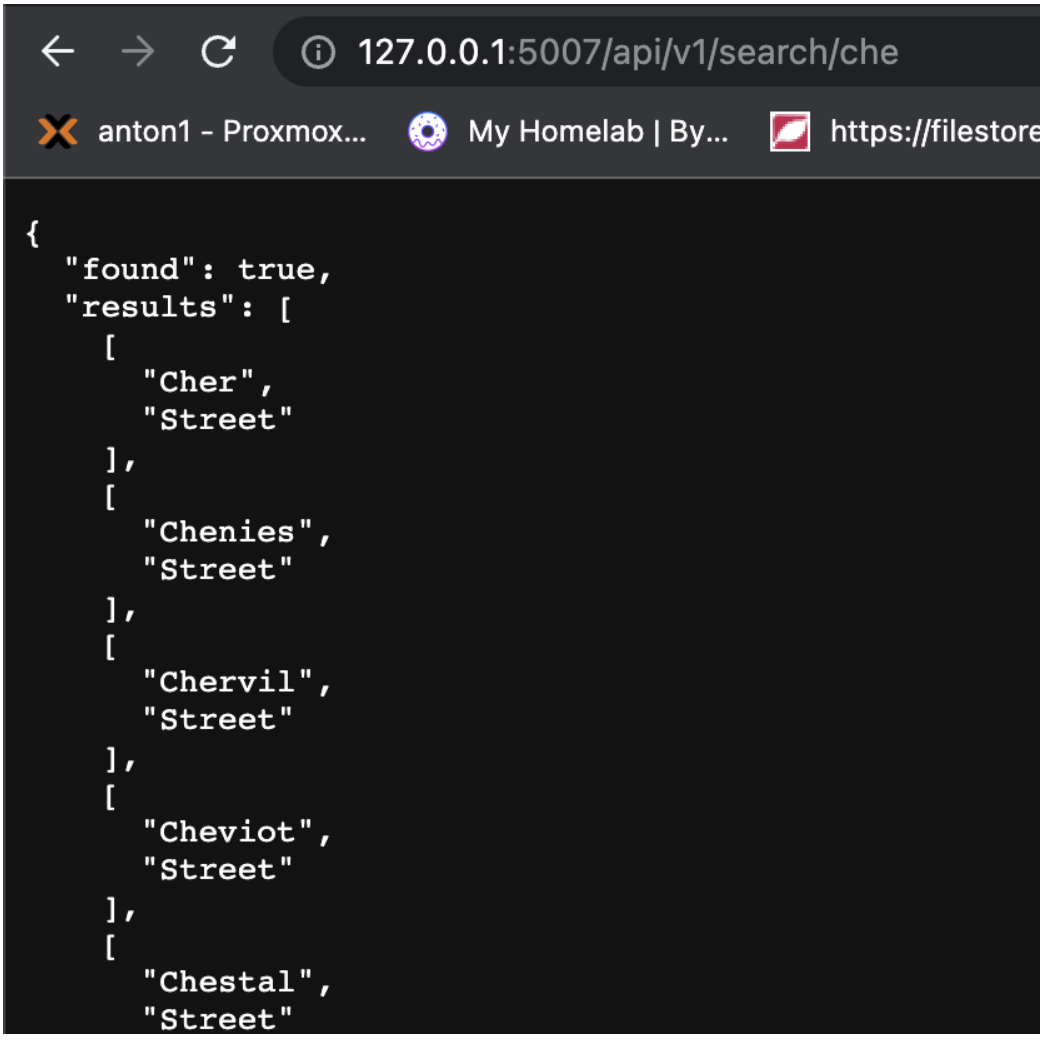
```
    if len(results) > 0:
    # Check if there are any results
        return jsonify(
            results=return_list,
```

```
            found=True
        ) # Return search results
    else:
        return jsonify(
            results=None,
            found=False
        ) # Return if no results
```

You can see in the image below after running it can successfully return suggestions for the search query 'CH'.



One issue I noticed when searching for 'Che' for Chester is I got street names first and then the city name. When people are searching it is unlikely they'll be searching for a street so I need to put the other area types above



To fix this I will sort the results by their area type in size decreasing order. I will do this using the inbuilt sort function in Python and use a key to get the order I want.

```
SORT_ORDER = {"area": 1, "outcode": 0, "sector": 2, "
    postcode": 3, "town": 4, "county": 5, "district": 6,
     "street": 7}
# Key to set sort order
return_list.sort(key=lambda val: SORT_ORDER[val[1].
    lower()], reverse=True)
```

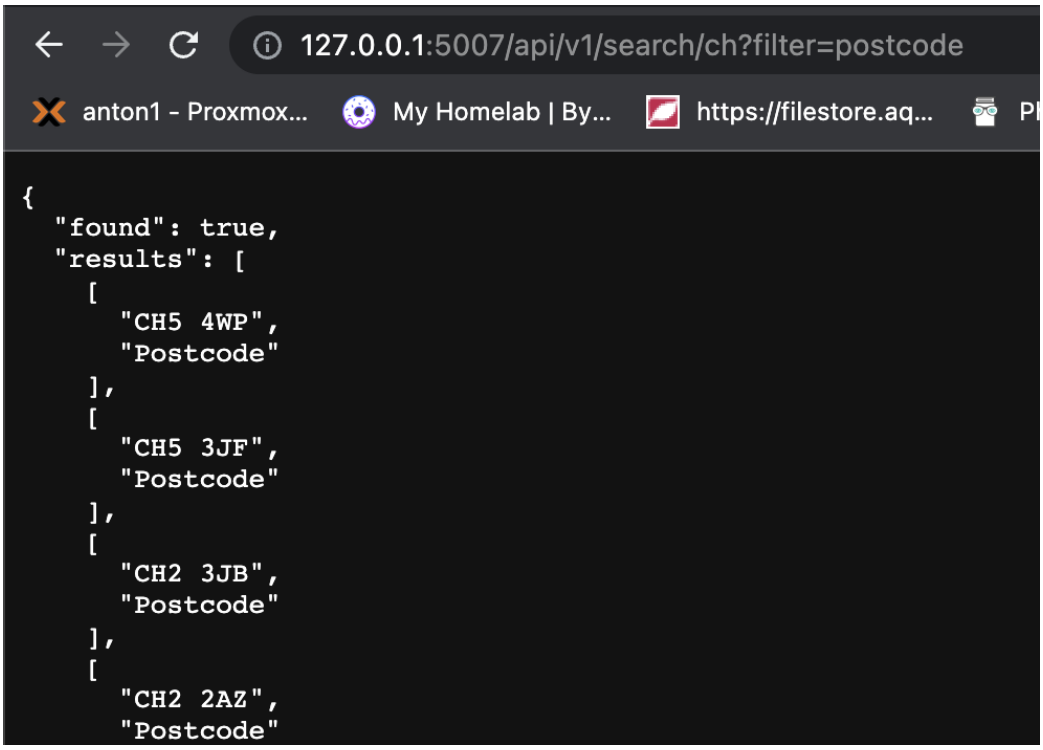
```
# Sorts list using key
```

You can see in the image above it put the towns before the street names. This will make it easier for the user when searching. After finishing this I realised I will also need to use the search function when looking up houses. For this, though I only want to search postcodes so I will need to add a way of filtering the search. To do this i will check to see if a filter argument has been passed to the URL. If it has i will modify the SQL query to only search for the area type specified in the filter argument.

```
query_filter = request.args.get("filter") # Check for
    filter argument

if query_filter is not None:
    if query_filter in ["postcode", "street", "town", "
        district", "county", "outcode", "area", "sector"
        ]:
        # Validate it is a correct area type

        sql_query = f"""SELECT area, area_type
                    FROM areas WHERE substr(area, 1,
                        50)
                    LIKE '{query}%' AND area_type = '{
                        query_filter}'
                    ORDER BY char_length(area)
                    LIMIT 10;"""
        # Edit SQL query to only search for that type
    else:
        return abort(404) # Return 404 if not valid
            type
```

```
else: # Act as normal if no filter passed
    sql_query = f"""SELECT area, area_type
                FROM areas WHERE substr(area, 1, 50)
                LIKE '{query}%'
                ORDER BY char_length(area)
                LIMIT 10;"""
```

I then tested it by only searching for postcodes and the result is below. As you can see it was successful in only searching the postcodes and no other area type.



3.3.5 House Lookup

My other feature as mentioned in the search section is the ability to look up a specific house and see all of its previous sales. The user will first search the postcode and then they will be given a list of all of the houses in this postcode which they can then click on to view a specific house. Firstly I will

need to write a route to get all of the houses in a given postcode. This will be quite simple as it uses one SQL query with a join to get all the houses from a postcode. The SQL query looks like this.

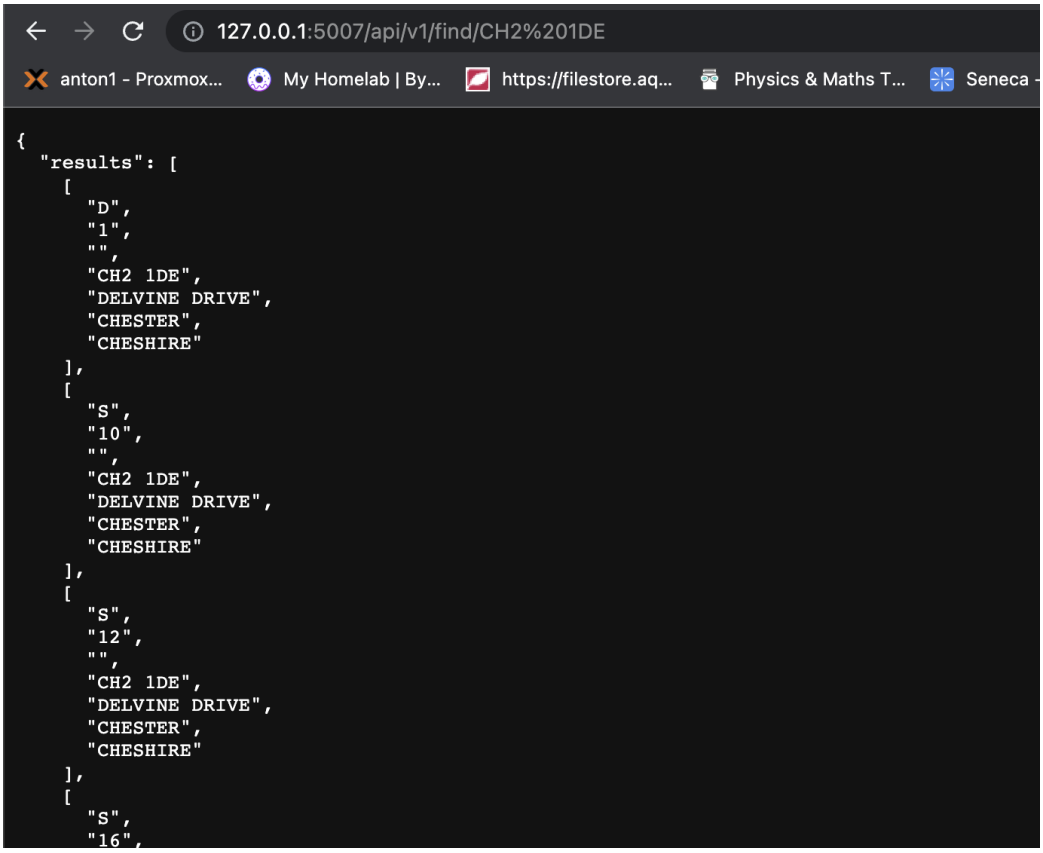
```
SELECT h.type, h.paon, h.saon, h.postcode, p.street, p.
    town, p.county
     FROM postcodes AS p
     INNER JOIN houses AS h ON p.postcode = h.postcode
        AND p.postcode = %s;
```

Onces all of the houses have been fetched they will then be sorted by house number and returned to the user. It will look like this.

```
@bp.route("/find/<string:postcode>")
def search_houses(postcode):
    sql_query = """SELECT h.type, h.paon, h.saon, h.
        postcode, p.street, p.town, p.county
                    FROM postcodes AS p
                    INNER JOIN houses AS h ON p.
                        postcode = h.postcode AND p.
                        postcode = %s;"""
    # Generate SQL query

    with current_app.app_context(): # Connect to app
        context
        cur = current_app.sql_db.cursor()
        cur.execute(sql_query, (postcode.upper(),)) #
            Execute SQL
        results: List[Tuple[str,str,str,str,str,str]] =
            cur.fetchall()
        # Store results

    if results != []:
        results = sorted(list(set(results)), key=lambda
            x: x[1])
        # Sort results by house number

        return jsonify(
            results=results,
```

```
        ) # Return results as JSON
    else:
        return abort(404, "Cannot Find Houses for
            Postcode")
        # Return error if no houses are found
```

After testing it with a postcode it was able to successfully return all the houses for a given postcode as seen below.



Now that they can search for houses in a postcode I need to create a route where they can get all of the sales for a house. This will work by getting the postcode, SAON, and PAON for a house. These will then be used to search for all of the sales in a house. The SQL queries are shown below in the Python Code.

```
@bp.route("/find/<string:postcode>/<path:house>")
def get_house_saon(postcode, house):
    try:
        paon, saon = house.split("/") # Extract paon
            and saon from path
    except ValueError:
        paon = house # Only extract paon if no saon
            found
        saon = ""
    sql_house_query = """SELECT h.houseid, h.type, h.
        paon, h.saon, h.postcode, p.street, p.town
                    FROM postcodes AS p
                    INNER JOIN houses AS h ON p.
                        postcode = h.postcode AND p.
                        postcode = %s
                    WHERE h.paon = %s AND h.saon = %s;
                        """
    # Generate query for getting all the info about the
        houses

    sql_sales_query = """SELECT *
                    FROM sales
                    WHERE houseid = %s
                    ORDER BY date DESC;"""
    # Generate query for getting all of the sales
```

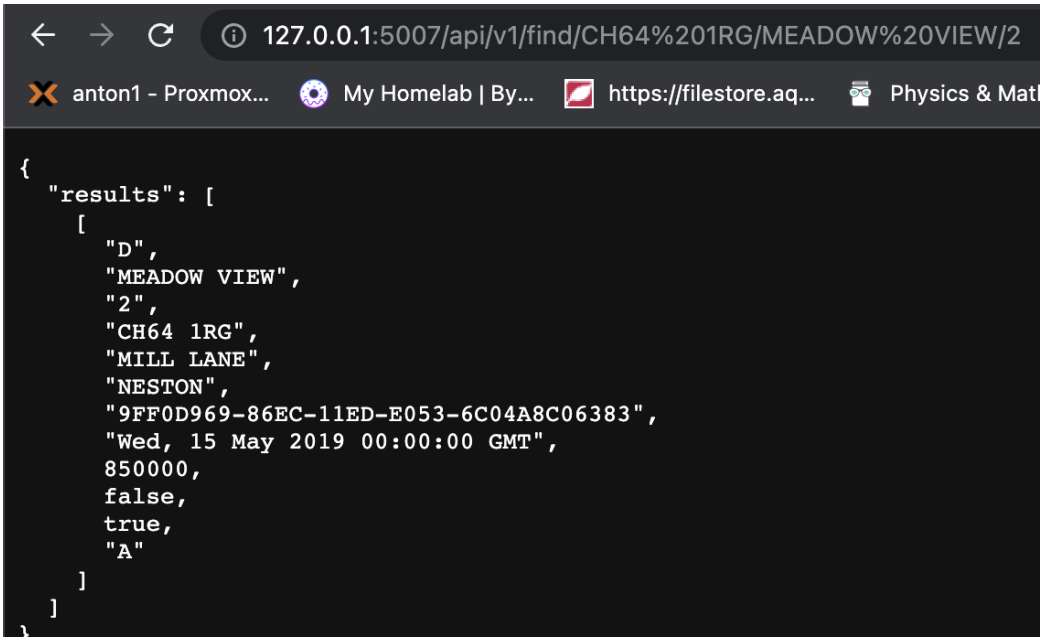
It will now search for a house with these values. Once it has found a house it can then search for sales.

```
    with current_app.app_context(): # Connect to app
        context
        cur = current_app.sql_db.cursor()
        cur.execute(sql_house_query, (postcode.upper(),
            paon.upper(),saon.upper(),))
        # Executes sql query for house
        house: List[Tuple] = cur.fetchone() # Stores
            result
```

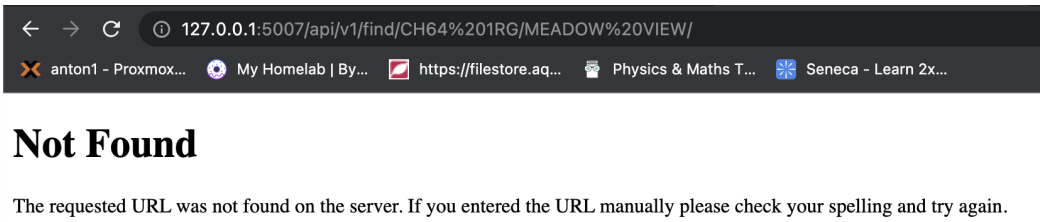
If it has found a house it will then search for all of the sales for that house. Once it has all of these the data will be returned in JSON format so it first must be formatted in a python dict

```
        if house != []:
            cur.execute(sql_sales_query, (house[0],)) #
                gets all sales for the house
            sales = cur.fetchall()
            # Load all sales
            house_info = {
                "paon": house[2],
                "saon": house[3],
                "postcode": house[4],
                "street": house[5],
                "town": house[6],
                "type": house[1],
                "sales": sales
            }
            # Format data in dict

            return jsonify(house_info)
            # Return as json
        else:
            return abort(404, "No House Found")
            # Return error if no house found
```

I tested this with a valid house and an invalid house. It was able to return all the sales for the house along with the correct address information as shown below.



I then tested it with the invalid address and it successfully returned a 404 page not found.



3.3.6 Review

All of the routes for the API are fully functional and have been tested. Now that this module has been completed I can tick off the success criteria for creating an API to interface with. Unfortunately, the API is not a human-

interactable module so I won't be able to get any feedback from my stakeholder until I create the web interface for it.

3.3.7 Code Layout

```python
import urllib.parse
from datetime import datetime
from pickle import dumps
from typing import List, Tuple

from app.api import bp
from flask import current_app, jsonify, url_for, abort, request


@bp.route("/analyse/<string:area_type>/<string:area>")
def index(area_type, area):…

@bp.route("/fetch/<string:query_id>")
def fetch_results(query_id):…

@bp.route("/search/<string:query>")
def search_area(query):…

@bp.route("/find/<string:postcode>")      You, 2 months ago • Added house Searching
def search_houses(postcode):…

@bp.route("/find/<string:postcode>/<string:paon>")
def get_house(postcode, paon):…

@bp.route("/find/<string:postcode>/<string:paon>/<string:saon>")
def get_house_saon(postcode, paon, saon):…

def get_last_updated():…
```

3.4 Web Interface

The web interface is one of the most important modules as it is what interfaces with the user and it determines how they view the data. For the web interface, I will be using svelte.js for the javascript framework and chart.js to make the graphs. Together these should provide a smooth user experience and allow me to create reactful content for the user.

3.4.1 Boilerplate Code

With javascript libraries, a lot of the code is auto-generated for you since it will be the same for most projects. The default files will be shown below.

```
web-ui/
    src/
        lib/
        routes/
            +page.svelte
        app.css
        app.d.task
        app.html
    static/
        favicon.png
        robots.txt
    .npmrc
    package-lock.json
    package.json
    postcss.config.js
    svelte.config.js
    tailwind.config.js
    tsconfig.json
    vite.config.js
```

All of these files are created automatically and in my case don't need to be edited so I won't show the contents of them here but it can be found in the appendix. For my website, I used tailwind CSS to style my elements. This allowed me to use classes instead of having to write actual css. Below is an example showing how they are different.

```
        # With tailwind
        <a class="text-xl text-red-600">Hello World</a>

        # Without tailwind
        <--CSS-->
        .text {
            color: #ff0000;
            font-size: 25rem;
        }
```

```
        <--CSS-->

        <a class="text">Hello World</a>
```

You can see how much easier the top example is compared to the bottom. It also makes the HTML easier to read as the styles are there next to the code and I don't have to go hunting in a style sheet to find it.

3.4.2 Search Bar

The key component of my website is the search bar as it is what all the analysis and lookups originate from so it has to work well otherwise they won't be able to access anything else on the website. The search bar will be accessible in two places the top right of the page to search for an area to analyse and on the house lookup page to find a postcode. It will work by detecting when the user types or removes a letter from the input box. When this happens it will call the autofill function which will then request the web API with the contents of the input bar as the query. Once the results are returned they will be shown under the input box as clickable links.

```
async function autoComplete(search_value: string){
    if (search_value) {
        const response = await fetch('https://api.
            housestats.co.uk/api/v1/search/' +
            search_value + '?filter=' + filter);
        // Sends http request to api with query string
            and filter

        const data = await response.json();
        // Saves json response
        if (data.found == true){
            suggestions = data.results;
            // If there are results saves them to
                variable
            results = true;
        } else {
            results = false;
        }
    } else {
```

```
        results = false;
    }
}
```

To make this function run when keys are typed I need to add an event listener to the input field like below.

```
<div class="flex flex-col">
    <input type="text"
    name="area"
    class="p-3 rounded"
    autocomplete="off"
    on:input={e => autoComplete(e.target.value)} //
        Event listener to run when typing
    >
</div>
```

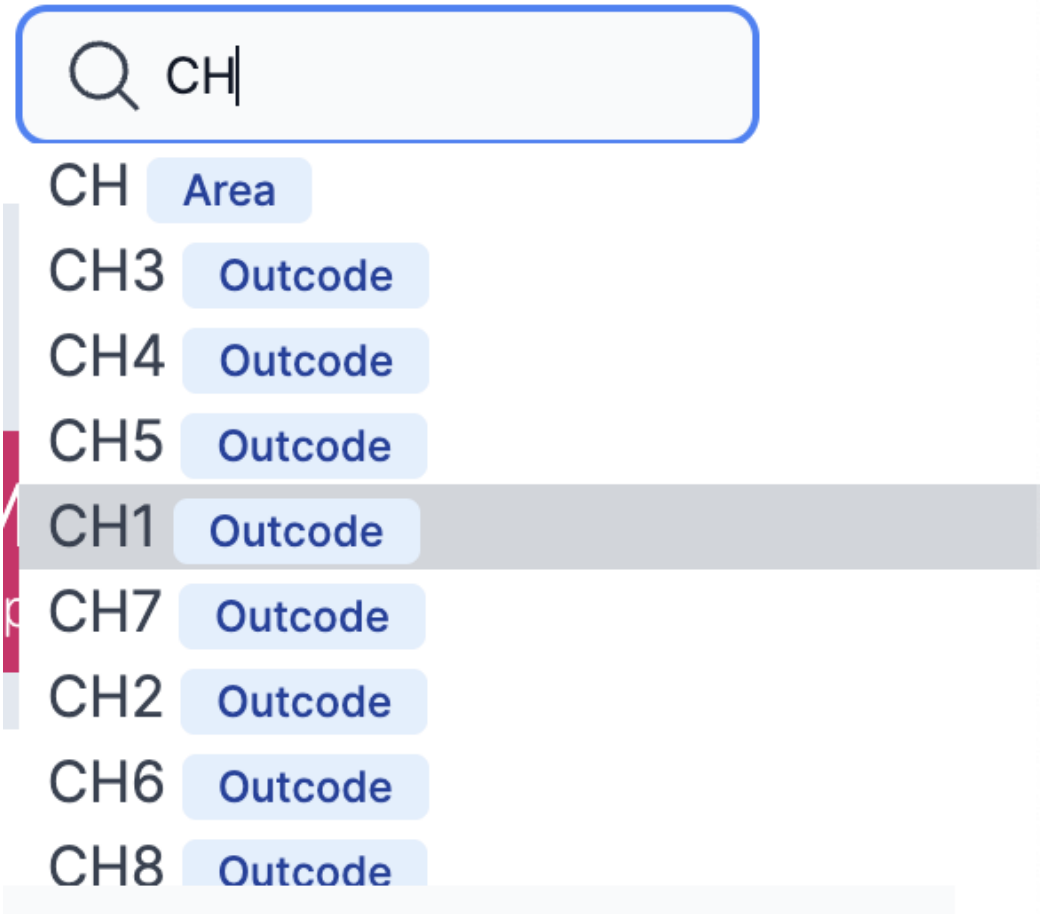
Now that it sends a request and saves the results every time you type in it I now need to output the results in a list below the search bar so that the user can click on them. I will do this using a condition in svelte to only show a block of HTML if the statement is true and then use a for loop to show all the results in the list. Svelte is a reactful library so whenever a variable changes value the interface is automatically updated e.g. when a new search request is done the list is automatically updated.

```
{#if isFocused == true} // If the user is clicked on
    the search bar
    <div class="">
    {#if results == true} // If there are results to
        show
        {#each suggestions as suggestion} // Loop
            through them
            <a href={'/analyse/' + suggestion[0]}> //
                Create a link to the page
            {suggestion[0]}
            <span class="">{suggestion[1]}</span>
            </a>
        {/each}
    {:else}
        <p class="ml-2">No Results</p>
```

```
    {/if}
    </div>
{/if}
```

I removed the styling so the code can be understood more easily. All of this code is then put together to create a component. Components in svelte are reusable pieces of code that are used to produce parts of an interface. It's the same concept as functions but for a GUI. This allows me to write the code for it once and use it all over the website by simply importing it into the page. Each component gets its file in the lib folder. Components can also take arguments so for example in the search bar it takes a filter as an argument so it can be used in the house lookup to search postcodes.

I then tested the search bar by typing in areas and seeing if it came up with the correct suggestions which it did. You can see an example below.



3.4.3 Analysis Page

Now that I had the search bar working I could move to the analysis page. For this page, I used a few components to display the data. I made a component for each of the charts bar, line, and pie chart. I also made what I call a badge which is a colourful box to house the 'quick stats' data.

Quick Stats Component

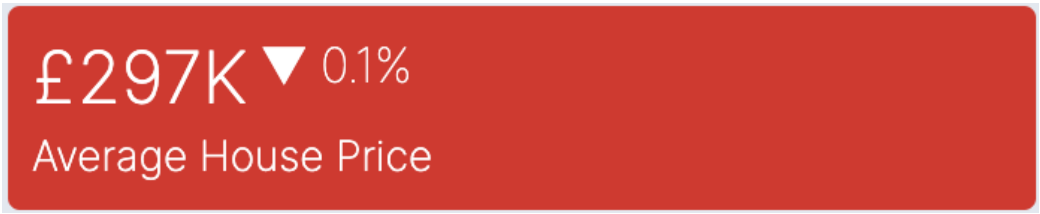
This was the simplest component as it was just a coloured box with the stat name, value, and percentage change. It took the name, value, and change as arguments which were then passed to the HTML to be rendered. It also

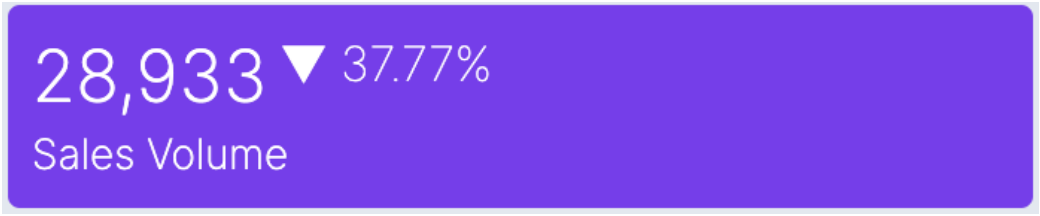
took an argument if the stat was a currency or not. This would round the number to two decimal places and insert a pound sign if it was. The code for this is below.

```
<script lang="ts">
    let formatter = Intl.NumberFormat('en',
        {
            notation: 'compact',
            unitDisplay: 'long',
            style: 'currency',
            currency: 'GBP'
        }); // Currency parser to round and add symbol

    // Arguments being defined for the component
    // with default values
    export let value: GLfloat;
    export let using_percentage: boolean = false;
    export let percentage: GLfloat = 0;
    export let title: string;
    export let currency: boolean = true;
    export let colour: string;
</script>

<div class="bg-{colour}-600">
    {#if currency} // Display as currency if chosen
        <p>{formatter.format(value)}</p>
    {:else} // Display as number if not
        <p >{value.toLocaleString()}</p>
    {/if}
    {#if using_percentage }
    // Showing the percentage and a sign for increase
       or decrease
    <p>{percentage < 0 ? 'down' : 'up'} {Math.abs(
       percentage)}%</p>
    {/if}
    <p>{title}</p>
</div>
```

Once written I then tested these with some data I just came up with and

the results are below. I did one with currency and one without to test the features along with different colours. They worked great so now i can move onto the next component.





Line Graph Component

The line chart is a slightly more complex component but a lot of the code is simply configuring and styling the graph for my preference. I will also need to transform the data to work with chart.js. The data is stored in the format shown below.

```
{
    "house_types": [<list of types>]
    "prices"[
        [<list of prices>],
        [<list of prices>],
        [<list of prices>],
        [<list of prices>]
    ],
    "dates": [<list of dates>]
}
```

Chart.js takes it in the format show below.

```
{
    "label": <house type>,
    "data": <list of prices>,
}
```

When it is creating the line chart it will need to match up the house type with its corresponding list of data points. This will be done by looping from 0 to however many property types and then getting the value from the house_types and the prices list corresponding to that index. I will then use a hashmap to get a human-readable version of the house type to display on the graph. Each property type will also be assigned a colour.

```
let house_types: { [key: string]: string } = {
    'D': "Detached",
    'S': "Semi-Detached",
    'T': "Terrace",
    'F': "Flat",
    'O': 'Other',
    "all": "All"
}; // Hashmap for human-readable version

let colours = [
    '#dc2626',
    '#9333ea',
    '#16a34a',
    '#db2777'
] # Different colours to use

export let labels: Array<string>;
export let title: string;
export let data: Array<Array<BigInt>>;
export let dates: Array<string>;

let data_length = data.length;
let datasets = [];

for (let i = 0; i < data_length; i++){
    datasets.push({ // Add to dataset list
```

```
        label: house_types[labels[i]], // Get human
            version of type
        data: data[i], // Get numbers from price list
        tension: 0.1,
        borderColor: colours[i], // Assign it a unique
            colour
        fill: false,
    });
}
```

Once the data has been transformed into the correct format it can be rendered into a graph. To do this it needs to have a target to render it on with a unique id. The graph will also need to be configured to have the x-axis display dates, be responsive and allow for panning and zooming in.

```
        const chart_data = {
            labels: dates.map((x) => {return new Date(x
                )}),
            datasets: datasets
        }; // Adds dates to a list and converts a
            string to DateTime object
        const config = {
            responsive: true,
            type: 'line',
            data: chart_data,
            options: {
                scales: {
                    x: {
                        type: 'time',
                        time: { // Sets smallest unit
                            for x-axis
                            round: 'month',
                            minUnit: 'month'
                        },
                        adapters: {
                            date: {
                                locale: enGB
                            } // Sets date format
                        }
```

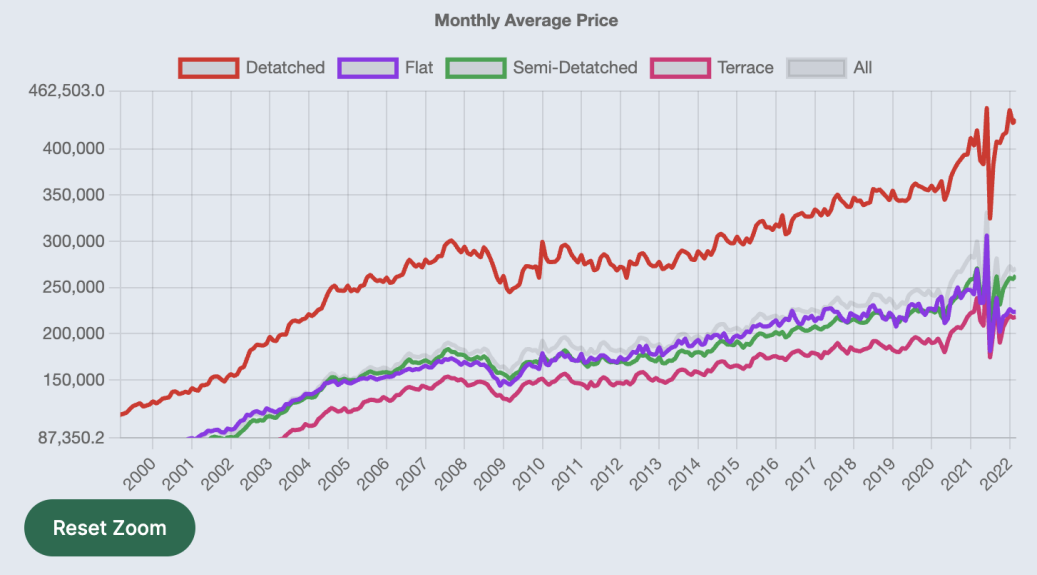
```
                    }
                },
                plugins: {
                    zoom: {
                        pan: {
                            enabled: true
                        }, // Enables panning
                        zoom: {
                        wheel: {
                            enabled: true,
                        }, # enables zooming
                        pinch: {
                            enabled: true
                        },
                        mode: 'xy',
                        }
                    },
                    title: {
                        display: true,
                        text: title
                    } # Sets the title for the graph
                }
            }
        };
        let line_chart: Chart;
        onMount(()=> { // Runs code when page is loaded
            let ctx = document.getElementById(graph_id)
               ; // Gets target canvas to render on
            if (ctx != null){
                line_chart = new Chart(ctx, config); #
                   Renders graph on target canvas
            }
        })
```

There also needs to be some HTML to go with this and there needs to be a reset button for zooming so the user can go back to the default view of they get stuck.

```
<canvas id={graph_id}> // Canvas to render graph on
```

```
</canvas>
<button on:click={line_chart.resetZoom('default')} type
    ="button">Reset Zoom</button>
```

Now that I have all the pieces working i can test it using some of the data. Below you can see the results.

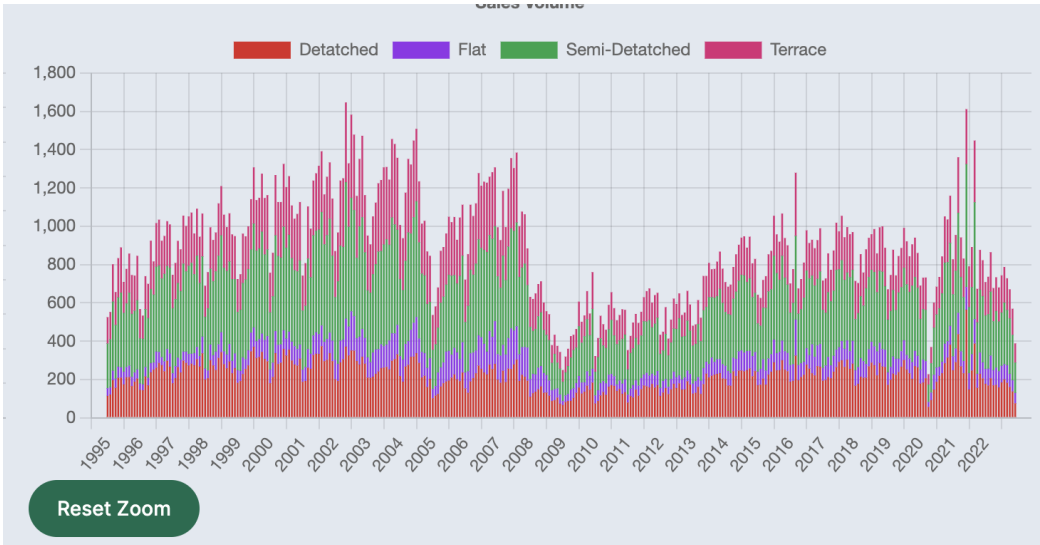


Bar Chart

This will be a very quick section as the bar chart is the exact same as the line graph except in the config section instead of line you write bar.

```
        const config = {
            responsive: true,
            type: 'bar',
            data: chart_data,
            ...}
```

As you can see below it works just as well as the line graph.



Pie Chart

The pie chart was quite easy as the data format that chart.js coincides with the format I'm storing in the database. I also used the bit of code to convert the property types to human-readable ones from the line and bar chart. The chart data code for the pie chart is shown below.

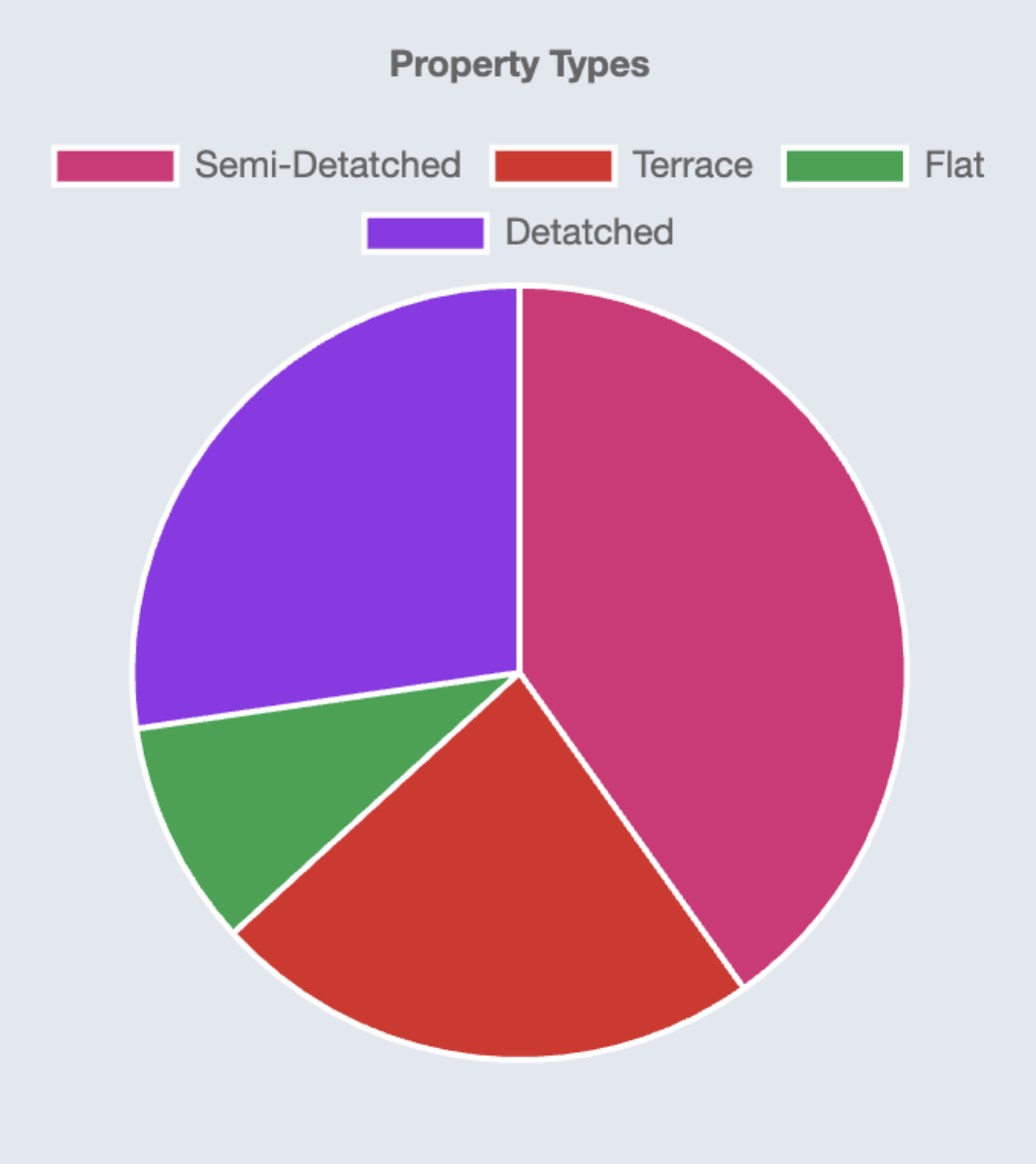
```
let house_types: { [key: string]: string } = {
    'D': "Detatched",
    'S': "Semi-Detatched",
    'T': "Terrace",
    'F': "Flat",
    'O': 'Other'
}; // Hashmap of human-readble types

// Defining arguments for component
export let labels: Array<string>;
export let title: string;
export let data: Array<BigInteger>;

const chart_data = {
    labels: labels.map((x) => {return house_types[x]}),
```

```
    // Convert types to human readable
    datasets: [{
        label: title,
        data: data, // Set data
        backgroundColor: [
        '#dc2626',
        '#9333ea',
        '#16a34a',
        '#db2777'
        ], // Set colours for each type
        hoverOffset: 4
    }],
};
```

Once the data has been set in the dataet it is then configured and rendered the same as the bar and line chart except there are less variables as the pie charts are a lot simpler and there is less to control.

```
const config = {
    type: 'pie',
    data: chart_data,
    options: {
        plugins: {
            title: {
                display: true,
                text: title // Sets title
            }
        }
    }
};
onMount(()=> { // Renders when page loads
    let ctx = document.getElementById('piechart');
    new Chart(ctx, config);
})
```

This is then rendered onto a canvas tag the same as the other graphs. Below is the pie chart.



Review

Now that I had all of the components made I sent the test images shown above of each one to my stakeholder. They each gave me feedback for them and it was mostly positive. The responses are summarised below as they had

mostly similar things to say.

Both Steven and Scott said they liked the colours as they contrasted well and made the graphs clear and easy to see. Neither of them had anything to say about the quick stats badges. Both of them did mention how on the line graph the line was a bit thick so they found it hard to see exactly where it was on the graph. Fortunately, this is just a setting I can tweak in the config section. Scott also said that the inconsistencies in the keys on the graphs annoyed him as flats would be green on the pie chart but purple on the line graph. He did say this was him being pedantic though fortunately this is also an easy fix.

The line width issues will be fixed as shown below.

```
const config = {
    responsive: true,
    type: 'line',
    data: chart_data,
    options: {
        borderWidth: 0.5, // Changes line
            thickness
        ...
    }...
}
```

The mismatching colours in the keys were simply fixed by changing the order of colours in the list for the pie chart.

3.4.4 Page Layout

Now that all the components have been created and I have gotten feedback from my stakeholders I can combine the components into the analysis page but before this happens I need to write the logic that fetches the data using the API. In svelte you can write logic that runs whilst the page is loading in a '+page.ts' file. This runs whilst loading so is useful for fetching any data that might be needed when the page gets rendered.

Loading Data

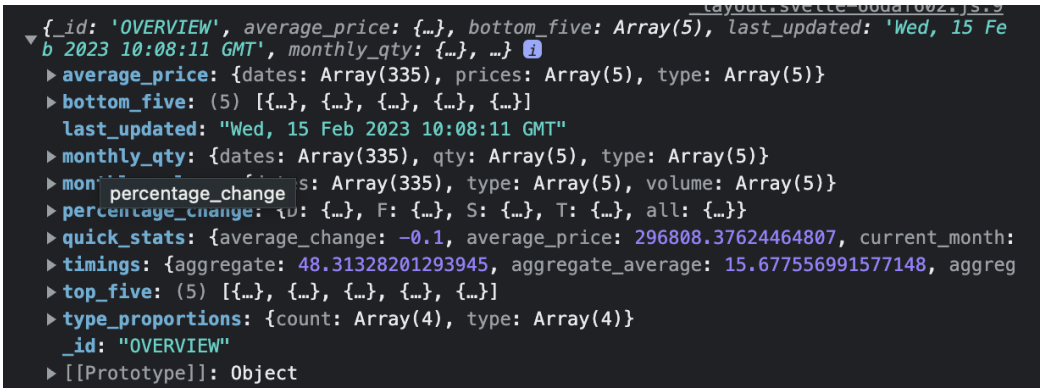
To get the data from the API i have to first make a request to the analyse endpoint to run a job on the data aggregators. Once that has been sent it

then needs to repeatedly request from the url returned until it says there are results. When these results are recieved they are then passed on to the page to be rendered.

```
import { error } from '@sveltejs/kit';

export async function load({ fetch, params }) {
        const sleep = (ms: number) => new Promise((r)
           => setTimeout(r, ms));
    // Sleep function to wait before requesting again

        let area: string = params.area; // area string
        let area_type: string = params.area_type; //
           area type string

        let stats;
        let counter = 0 // request counter if it times
           out

        const response = await fetch('https://api.
           housestats.co.uk/api/v1/analyse/' +
           area_type + '/' + area);
    // Send requests to analyse the endpoint to start
       job
        const data = await response.json();
    // Stores json response

        if (data.status == "ok") {
            while (true) {
                    const res_resp = await fetch(data.
                       result);
            // Checks returned endpoint to see if the
               job is done
            stats = await res_resp.json();

            if (stats.done == true) {
                                break // If it returns
                                   the data break out
```

```
                                    of the loop
                            } else if (counter > 60*4) {
                                    throw error(500, '
                                        Connection Timed Out
                                        ')
                        // If request take longer than 4
                            minutes raise an error
                                }
                                counter++;
                                await sleep(250) // sleeps for
                                    250ms between requests
                        }
                } else {
                        throw error(500, 'An Error Has Occured'
                            );
                }
                return stats
}
```

To test this I got it to 'console.log' the raw data so I know if it was loaded.



As you can see it was able to successfully load the data so now I can move on to laying out the components on the page.

Page Layout

Before I can write the HTML for the page I need to import the components and set some variable. This will make it easier to pass the data to the components as it is a heavily nested JSON object. The components will be imported first and then there will be some logic to check if the data has been loaded correctly and then it will be broken up into variables to make it easier.

```
// Import all of the components
import Badge from '$lib/components/Badge.svelte';
import QuickStat from '$lib/components/QuickStat.svelte
    ';
import PieChart from '$lib/components/PieChart.svelte';
import LineGraph from '$lib/components/LineGraph.svelte
    ';


let quick_stats, stats, results;
let last_updated: Date;
let current_month: Date;
// Init variables

export let data;
if (data.done == true) { // Check if data is loaded
    // Split data up into variable
    quick_stats = data.results.data.quick_stats;
    stats = data.results.data;
    results = data.results;

    // Get import dates
    last_updated = new Date(results.last_updated);
    current_month = new Date(quick_stats.current_month)
        ;
}

let title = "Analyse"; // Set page title

// Function to help with formatting some of the data
function toTitleCase(str: string) {
```

```
    return str.toLowerCase().split('-').map(function (
        word) {
        return (word.charAt(0).toUpperCase() + word.
            slice(1));
    }).join('-');
}
```

First, I want the quick stats to be displayed at the top front and centre so they are the first to be seen and easy to read. Alongside them will be the title of the area and what is the most recent month. The title will be accompanied by some statistics on the data processing like how long each step took and when it was last updated. The HTML for this is below.

```
<div class="m-2">
    <div class="items-center-align-middle-flex-flex-
        initial-flex-wrap">
        <p class="inline-block-text-2xl-m-2-align-
            middle">
            {toTitleCase(results.area)} ({toTitleCase(
                results.area_type)})
        </p>
        <Badge
            text="Last-Updated-{last_updated.
                toLocaleDateString()}"
            colour="green"
            classes="inline-block-align-middle"
        />
        <Badge
            text="Execution-Time-{Number((results.
                exec_time).toFixed(3))}s"
            colour="green"
            classes="inline-block-align-middle"
        />
        <Badge
            text="Data-Fetch-Time-{Number((results.
                load_time).toFixed(3))}s"
            colour="green"
            classes="inline-block-align-middle"
        />
```

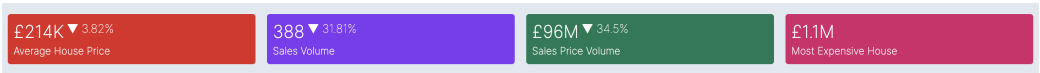
```
            <Badge
                text="Current Month {current_month.
                    toLocaleDateString()}"
                colour="green"
                classes="inline-block align-middle"
            />
        </div>
</div>
```

You can see all of the numerical values are rounded to 3 decimal places and all the dates are converted to the user's timezone and date format. The text is also all in title case. You can see how it looks below.

Now I can do the quick stats which will be laid out horizontally 4 across with individual colours and statistics.

```
            <QuickStat
                value={quick_stats.average_price}
                using_percentage={true}
                percentage={quick_stats.average_change}
                title="Average House Price"
                colour="red"
            />
            <QuickStat
                value={quick_stats.current_sales_volume}
                currency={false}
                using_percentage={true}
                percentage={quick_stats.sales_volume_change
                    }
                title="Sales Volume"
                colour="purple"
            />
            <QuickStat
                value={quick_stats.current_price_volume}
                using_percentage={true}
```

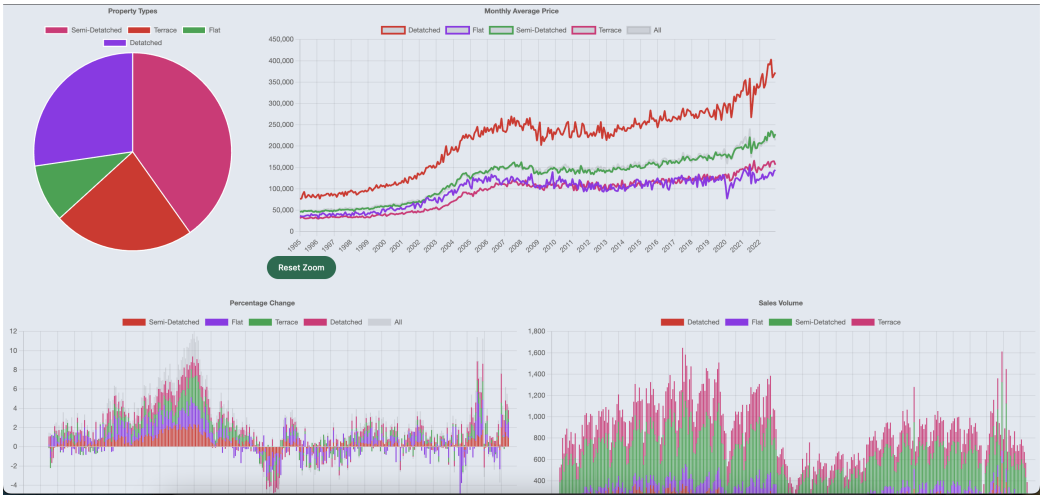
```
            percentage={quick_stats.price_volume_change
                }
            title="Sales Price Volume"
            colour="green"
        />
        <QuickStat
            value={quick_stats.expensive_sale}
            using_percentage={false}
            title="Most Expensive House"
            colour="pink"
        />
```



The last part for this page is the graphs. There will be four graphs average price, price volume, sale quantity, and property type proportions. Together these will provide the user with a comprehensive view of the property market for that area.

```
        <div class="xl:row-span-2">
            <PieChart
                title="Property Types"
                labels={stats.type_proportions.type}
                data={stats.type_proportions.count}/>
        </div>
        <div class="md:col-span-2 row-span-2">
            <LineGraph
                title="Monthly Average Price"
                labels={stats.average_price.type}
                data={stats.average_price.prices}
                dates={stats.average_price.dates}/>
        </div>
        <div class=" md:col-span-2 row-span-2">
            <BarGraph
                title="Sales Volume"
```

```
                labels={stats.monthly_sales_volume.type
                    }
                data={stats.monthly_sales_volume.volume
                    }
                dates={stats.monthly_sales_volume.dates
                    }/>
        </div>
        <div  class="md:col-span-2 row-span-2">
            <BarGraph
                title="Price Volume"
                labels={stats.monthly_price_volume.type
                    }
                data={stats.monthly_price_volume.volume
                    }
                dates={stats.monthly_price_volume.dates
                    }/>
        </div>
```



The analysis page is now finished along with all of the components for it. I sent copies of it over to both of my stakeholders and they both said they were very happy with it and were excited to see the finished product.

House Lookup

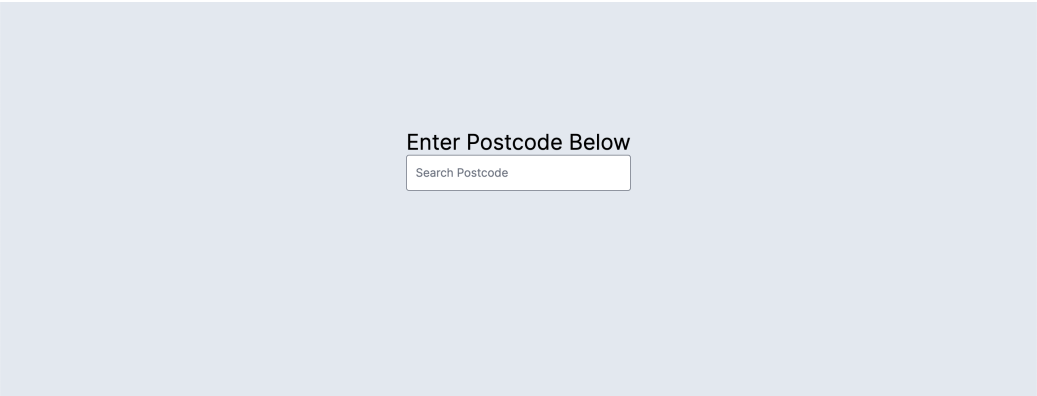
For this section, I need to make a page that uses the search bar to lookup postcodes and then takes the user on to another page where they can see all the houses for that postcode in a table. Each house in that table will then have a link where they can go see more information about that house and the sales for it.

Search Page

The search page will be a fairly simple page containing only a search bar in the centre and the suggestions list below. It will use the search bar component that was previously made and it will make use of the filter argument to only search postcodes. The code for this will be cery short so wont require much explanation.

```
<script lang='ts'>
        import SearchBar from 'lib/components/SearchBar
            .svelte';
</script>

<svelte:head>
        <title>House Stats | Valuation</title>
</svelte:head>

<div class="flex justify-center items-center max-w-
    screen">
        <div class="-mt-28">
                <p class="text-3xl">Enter Postcode
                    Below</p>
                <SearchBar filter="postcode"/>
        </div>
</div>
```

Below you can see how this page looks. There isn't much to this page so I didn't ask my stakeholder for any feedback.



House Table Page

Fortunately, this page has a bit more to it so there will be more to talk about. Like the analyse page it will need to load the data first. This is again done through the '+page.ts' file. Loading this data is a lot more simple as it only requires ones API call.

```
import { error } from '@sveltejs/kit';

export const load = (async ({ params }) => {
    let postcode: string = params.postcode;
        const response = await fetch('https://api.
           housestats.co.uk/api/v1/find/' + postcode.
           toUpperCase());
    // Fetch houses from postcode using API

        const data = await response.json();
    // Store results as JSON
        if (response.status == 200) { // Check for
           success
                return {
                        data: data.results,
                        postcode: postcode
        } // Return results to be rendered on page
        } else {
                throw error(404, 'No Postcode Found');
```

```
                    // Raise error if no houses found
        }
});
```

Now that the data has been loaded it can be rendered as HTML. It will be displayed in a table with 6 rows. These will be SAON & PAON, Street, Town, County, Postcode, and Action. The action column will house the link to see more information on the property. To show each house from the postcode I will use a for loop to iterate through the list and insert the value into the HTML as shown below.

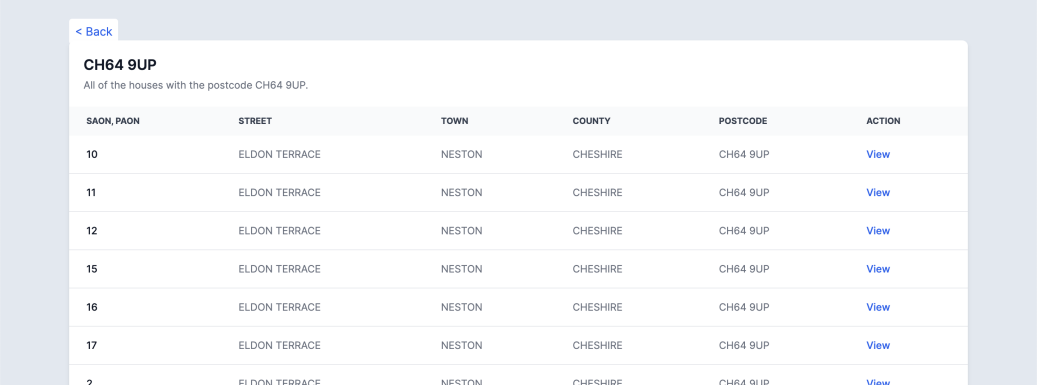
```
<tbody>
    {#each data.data as house} // Loop through houses
        <tr class="bg-white border-b dark:bg-gray-900
            dark:border-gray-700">
            <th scope="row" class="px-6 py-4 font-
                medium text-gray-900 whitespace-nowrap
                dark:text-white">
                {house[2]}{house[1] != '' && house[2]
                    != '' ? ',' : ''} {house[1]} //
                    Display house PAON and SAON
            </th>
            <td class="px-6 py-4">
                {house[4]}
            </td>
            <td class="px-6 py-4">
                {house[5]}
            </td>
            <td class="px-6 py-4">
                {house[6]}
            </td>
            <td class="px-6 py-4">
                {house[3]}
            </td>
            <td class="px-6 py-4">
                <a href="/valuation/{data.postcode.
                    toUpperCase()}/{house[1]}/{house[2]}
                    ">View</a>
                // Create link to see more info on the
```

```
                    house
                </td>
            </tr>
        {/each}
</tbody>
```

This code snippet will be embedded in the table

```
<div class="md:mx-24 my-8">
  <a href="/valuation">&lt; Back</a>
  <div >
    <table >
          <caption >
                {data.postcode.toUpperCase()}
                <p class="">All of the houses with the
                   postcode {data.postcode.toUpperCase
                   ()}.</p>
          </caption>
      <thead>
        <tr>
            <th scope="col" class="px-6 py-3"> SAON,
               PAON </th>
            <th scope="col" class="px-6 py-3"> Street <
               /th>
            <th scope="col" class="px-6 py-3"> Town </
               th>
            <th scope="col" class="px-6 py-3"> County <
               /th>
            <th scope="col" class="px-6 py-3"> Postcode
                </th>
            <th scope="col" class="px-6 py-3"> Action <
               /th>
        </tr>
      </thead>
          <<<Insert Snippet HERE>>>
        </table>
  </div>
</div>
```

The styling has been removed to make the HTML more readable.



The next part was displaying the sales data on a house. This would again require me to load the data from the API using the '+page.ts' file. This only required one API call so will look almost identical to the house list call except the URL will be slightly different.

```
import { error } from '@sveltejs/kit';

export const load = (async ({ params }) => {
    let postcode: string = params.postcode;
    let paon: string = params.paon;
    //parse information from path

        const response = await fetch('https://api.
            housestats.co.uk/api/v1/find/' + postcode.
            toUpperCase() + '/' + paon.toUpperCase());
        // Fetch house sales
    const data = await response.json();
    // Store house sales as json

        if (response.status == 200) { // Check for
            success
                return { // Return data upon success
                        sales: data,
                        postcode: postcode,
```

```
                        paon: paon
                    }
            } else {
                    throw error(404, 'No Postcode Found');
            // Raise error if no house or sales found
            }
});
```

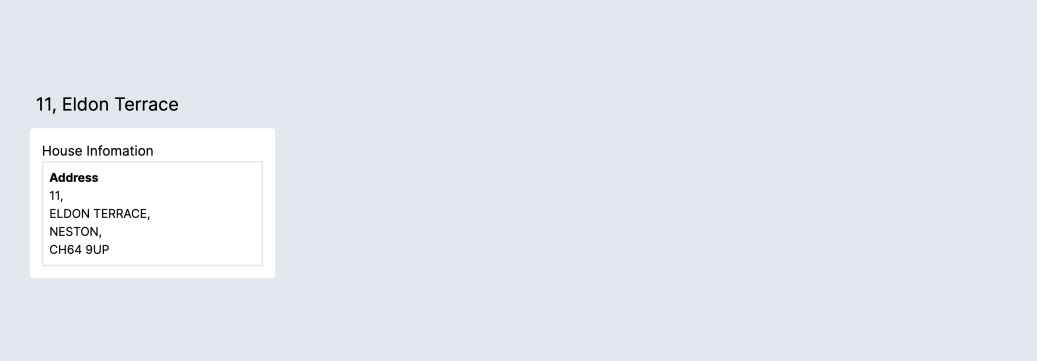
Once the information had loaded this was once again passed over to the HTML to be rendered. Since I will be displaying price information I will need to use the currency format as I did for the quick stats component. Again for simplicity, I will split the JSON data up into individual variables to be used making the code easier to read and work with. Below is the code for this.

```
function toTitleCase(str: string) {
    return str.toLowerCase().split('-').map(function (
        word) {
         return (word.charAt(0).toUpperCase() + word.
             slice(1));
    }).join('-');
} // Function to format text data as title

export let data: PageData;

// Divide json up into individual variables
let sales = data.sales;
let paon = data.paon;
let saon = data.saon;
let postcode = data.postcode;

// Initialize number formatter
let formatter = Intl.NumberFormat('en', {
    notation: 'compact',
    unitDisplay: 'long',
    style: 'currency',
    currency: 'GBP'
});
```

For the first part of the page, I want to have the title be the house number

and street and then below that show all the address information about the house. The address information will be housed within its white box.

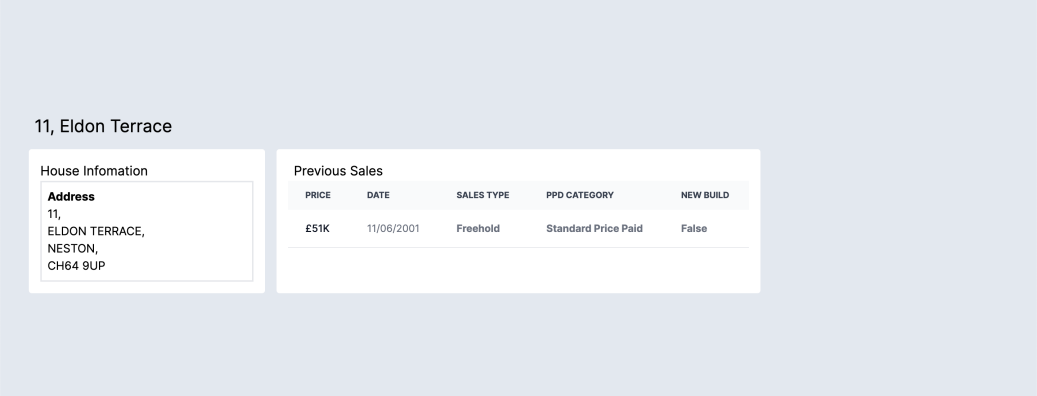
```
<p class="text-2xl my-4 mx-2">{toTitleCase(saon) + (
    saon != '' ? ',' : '')}{toTitleCase(paon)}, {
    toTitleCase(data.street)}</p>
// Convert house number and street to title case for
    the title


<div class="row-span-2 bg-white p-4 rounded">
    <p class="text-lg">House Infomation</p>
    <div class="border-2 p-2">
        <p class="font-bold">Address</p>
        <p>
            {saon} {paon}, <br>
            {data.street}, <br>
            {data.town}, <br>
            {postcode}
        </p> // Show the address for the house
    </div>
</div>
```



Now I'm going to create a table to show all of the sales a house has had since 1995. The table will have 5 columns price, date, sales, type, ppd_category, and new build. The code for the table will be the same as the house list minus the column names and row data.

```
{#each sales as sale}
    <tr class="bg-white border-b dark:bg-gray-800 dark:
        border-gray-700">
        <th scope="row" class="">
            {formatter.format(sale[1])}
        </th>
        <td class="px-6 py-4">
            {(new Date(sale[2])).toLocaleDateString()}
        </td>
        <th class="px-6 py-4">
            {sale[4] == true ? "Freehold" : "Leasehold"
                }
        </th>
        <th class="px-6 py-4">
            {sale[5] == "A" ? "Standard Price Paid" : "
                Additional Price Paid"}
        </th>
        <th class="px-6 py-4">
            {sale[3] ==  true ? "True" : "False"}
        </th>
    </tr>
{/each}
```

This is what the row part looks like. I won't show the rest of the table as it is identical to the other part except the column names. Now with all the parts done, you can get a final view of what the page will look like.



I sent all of the pictures of the pages to my stakeholders for some feedback. They both got back to me and were very impressed with it and said that they couldn't think of any meaningful changes that I could make to it.

Navigation Bar

The navigation bar is the glue for my website as it is where the user navigates between pages and where they search for new places to analyse. The navbar is a component that will be placed at the top of every single page so it is always accesible. The navbar will also be dynamic so whichever page the user is currently on will be blue text. This is done by using the page data store in svelte to see what the current path is.

```
// Import components and libraries
import { page } from '$app/stores';
import SearchBar from '$lib/components/SearchBar.svelte
    ';

$: current_page = $page.url.pathname; // Defines a
    variable to hold the current page
```

When a variable is defined using a dollar sign in svelte it means that when it changes it will automatically update the UI. This is required for the navbar as I want the text to change as soon as they change the page. The code to make the text blue looks like this.

```
<a
```

```
href="/valuation"
class="block py-2 pl-3 pr-4 rounded {current_page === "
    /valuation" ? 'text-blue' : 'text-gray-700'}"
>House Lookup</a>
```

So when the current page variable is the same as the page path the text is blue. This code is used on all the links in the navbar. Aswell as that the searchbar is embedded at the end of the navbar to allow users to search for an area no matter where they are on the website.

```
<nav >
    <div >
        <a href="/" class="flex items-center">
            <img src="/logo.svg" alt="House Stats Logo"
                />
            <span>House Stats</span>
        </a>
    </div>
    <div class="">
        <ul class="">
            <li>
                <a
                    href="/"
                    class="block py-2 pl-3 pr-4 rounded
                        {current_page === "/" ? 'text-
                        blue' : 'text-gray-700'}"
                >Dashboard</a>
            </li>
            <li>
                <a
                href="/counties"
                class="block py-2 pl-3 pr-4 rounded {
                    current_page === "/counties" ? '
                    text-blue' : 'text-gray-700'}"
                >Overview Counties</a>
            </li>
            <li>
                <a
                href="/valuation"
```

```
                    class="block py-2 pl-3 pr-4 rounded {
                        current_page === "/valuation" ? '
                        text-blue' : 'text-gray-700'}"
                    >House Lookup</a>
                </li>
                <li>
                    <a
                    href="/reports"
                    class="block py-2 pl-3 pr-4 rounded {
                        current_page === "/reports" ? 'text
                        -blue' : 'text-gray-700'}"
                    >Report Generator</a>
                </li>
            </ul>
        </div>
        <SearchBar></SearchBar>
</nav>
```

Below you can see the end result of this giving the user an easy way to navigate around the website.

Review

Now that I have all of the pages completed I sent a link to my stakeholders where they could go and play around with pages and see if there were any issues or areas that needed improving.

Scott said that he liked the website and that it had a nice look and feel to it. He did say that it would be nice to have a short bit of text that explains all of the statistics and what they mean for people who may not be as well-versed. If I had more time I could've added some tooltips next to each stat explaining them.

Steven said he was very impressed with the site and how professional it looked. He had some slight issues when accessing it on his phone as the screen wasn't big enough to view the graphs as they were quire squash. Had I had more time I would've invested more time into making it more mobile-friendly

but my mine demographic was a professional user who would be accessing it on desktops or iPads with larger screens

Chapter 4

Evaluation & Stakeholder Testing

4.1 Stakeholder Testing

4.1.1 Questions

1. What device are you using and how does the website look on your device?
2. Did the page load promptly?
3. Do you understand what each statistic means on the home dashboard?
4. Were you easily able to search for statistics on a specific area?
5. How did you feel about the time taken to load the statistics?
6. How easy was it to view the history of a specific property?
7. Was the information laid out intuitively?

4.1.2 Responses

Scott

1. I opened the website on my laptop and the data was laid out fine all of the information was easy to read and the colour palette made it easy on the eyes.

2. The home page loaded as quickly as any other website so I wasn't sat there waiting

3. I was able to understand almost all of the statistics on the home page except what the 'Top Areas' meant but after googling it I realised it was referring to postcode area and I was initially confused by the suffixes used for the numbers on the quick stats so maybe they could be the full word like 'Million' instead of 'M'

4. The search was super easy and the autocomplete made it even easier as I couldn't go wrong with it and the suggestion loaded very quickly. One suggestion I would add tho is the ability to just press enter instead of having to click on the desired area

5. On my first search it took a fair few seconds but not so long to put me off. When I searched for another area though it loaded almost instantaneously

6. Searching for a property was super easy as I just needed to type in the postcode which was autocompleted for me and then click the house number.

7. The information was laid out fine and it was easy to understand each section of it. A nice addition would be to suggest similar properties in the area

Steven

1. I opened the website on my iPad. The website looked great on my iPad and I could read everything though it would be nice if the graphs were a bit bigger.

2. It took a few seconds to load but that might have been down to me being on-site and using cellular data.

3. I could understand all the statistics presented to me though my professional background probably played a part in that and the colours complemented each other nicely making it easy to read

4. The search function was very easy to use and the autocomplete made it even simpler to use. It might've been just me but I found I had to tap the area a few times before the website registered it

5. I was very impressed by how quickly the statistics loaded as I chose quite a large area to view

6. Initially I had a bit of trouble searching for a property as it wouldn't show up in the results after typing in the postcode. I tried another property and that worked fine. I would recommend looking into the first property as I'm 100% sure it exists

7. I liked the layout of the information and the Energy Proficiency Certificate was handy as it gave me an idea of the size of the property and the age

4.1.3 Review

Overall the stakeholders seemed very happy with the solution and had mostly positive things to say about it. They were happy with the loading time which was my biggest challenge and how it was laid out. One of the issues was not an issue with my programme but instead with the data provided. For example, after speaking with Steven the property he was searching for had not been sold after 1995 so it was not listed in the government dataset. Some of the other issues they put forward were to do with the usability of the programme for example Scott not understanding the suffixes and not understanding what some of the statistics mean. This could be solved by having a little question mark logo by each statistic which when hovered over could show a brief explanation of what it means and what it could indicate. The issues Steven raised about having to tap multiple times when trying to select an area, unfortunately, could not be replicated so I am unable to fix that issue though the suggestion to be able to press enter to select an area would be able to remediate this issue as they wouldn't have to tap the specific area. Another suggestion to show similar properties in an area when looking at another property would be tough to implement as I would have to find a dataset containing the attributes of each house so I can decide what is similar and what is not.

4.2 Sucess Criteria

Criteria	Met	Evidence
Insert the PPD data into an SQL database in 3NF	Yes	Screenshot below
Query data from the database using Python	Yes	Screenshot below
Performing statistical operations on data using Python	Yes	Screenshot below
Selecting data from specific areas and aggregating it	Yes	Screenshot below
Creating an API to interface with and get data	Yes	Screenshot below
Creating a user interface to show data from the API via graphs and figures	Yes	Link to website below
Having searched for historical data taking ¡500ms	No	Screenshot below with video
Generating statistics for an area and displaying taking ¡2000ms	No	Screenshot below with video
Upload new data to the website for analysing and searching	No	n/a
Setting time frames for analyses of the data	No	n/a

4.2.1 Evidence

Data Stored in 3NF

Sales Table

	tui	price	date	new	freehold	ppd_cat	houseid
1	2600B105-2E30-49D1-B7EE-4652EC1C1ED2	78000	1995-03-03	false	• true	A	21TF2 9NW
2	A42E2F04-2538-4A25-94C5-49E29C6C8FA8	18500	1995-01-31	false	false	A	VILLA PARADISOFLAT 10TQ1 1RY
3	10E5F080-7AF3-4982-AAEF-42C55DC955FC	30000	1995-11-28	false	• true	A	66WS10 9LD
4	E5B50DCB-BC7A-4E54-B167-49E2A6B4148B	59000	1995-03-31	false	• true	A	28BH12 2AE
5	F745AFC7-E616-49D0-8ABC-3F320DF886C7	40000	1995-06-09	false	• true	A	8BD6 3AR
6	1BA349E3-2579-40D6-999E-49E2A25D2284	73450	1995-10-09	• true	• true	A	6L26 7XJ
7	D30EEA6F-CFB6-489A-9D03-4652DB154C43	40000	1995-10-27	false	• true	A	17NR17 1EY
8	81E50116-D675-4B7F-9F8D-49E2B5D43271	31000	1995-12-04	• true	• true	A	NONSUCH COTTAGEIP13 0DR
9	5BBAE523-83F4-4829-BDE9-3BA0E3D921FD	39950	1995-10-27	false	• true	A	53NG15 7LY
10	B97455B9-75CB-40BB-A615-42C53683E143	95000	1995-09-22	false	• true	A	FOX COVER COTTAGEWS14 0BE
11	CD39CFC2-E6D1-47D9-A588-3F31DB3D3CA2	52000	1995-08-10	false	false	A	DUKES COURT, 17FLAT 4BH1 1QY

Houses Table

WHERE ORDER BY postcode DESC

	houseid	paon	saon	postcode 1	type
1	JOSEPH ROWNTREE THEATREYO91 1RT	JOSEPH ROWNTREE THEATRE		YO91 1RT	O
2	NORWICH UNION INSURANCE GROUPYO90 1WR	NORWICH UNION INSURANC…		YO90 1WR	O
3	2YO90 1UU	2		YO90 1UU	O
4	1YO8 9YE	1		YO8 9YE	D
5	2YO8 9YE	2		YO8 9YE	D
6	3YO8 9YE	3		YO8 9YE	D
7	1YO8 9YD	1		YO8 9YD	D
8	2YO8 9YD	2		YO8 9YD	D
9	4YO8 9YD	4		YO8 9YD	D
10	3YO8 9YD	3		YO8 9YD	D
11	12YO8 9YB	12		YO8 9YB	D

Postcodes Table

WHERE ORDER BY postcode DESC

	post... 1	street	town	district	county	outcode	area	sec
1	YO91 1RT		YORK	YORK	YORK	YO91	YO	YO91 1
2	YO90 1WR	WELLINGTON ROW	YORK	YORK	YORK	YO90	YO	YO90 1
3	YO90 1UU	ROUGIER STREET	YORK	YORK	YORK	YO90	YO	YO90 1
4	YO8 9YE	TOLL BAR CLOSE	SELBY	SELBY	NORTH YORKSHIRE	YO8	YO	YO8 9
5	YO8 9YD	ONE ACRE GARTH	SELBY	SELBY	NORTH YORKSHIRE	YO8	YO	YO8 9
6	YO8 9YB	YORKDALE DRIVE	SELBY	SELBY	NORTH YORKSHIRE	YO8	YO	YO8 9
7	YO8 9YA	YORKDALE COURT	SELBY	SELBY	NORTH YORKSHIRE	YO8	YO	YO8 9
8	YO8 9XP	HEATHER CLOSE	SELBY	SELBY	NORTH YORKSHIRE	YO8	YO	YO8 9
9	YO8 9XN	POPPY CLOSE	SELBY	SELBY	NORTH YORKSHIRE	YO8	YO	YO8 9
10	YO8 9XL	GORSE CLOSE	SELBY	SELBY	NORTH YORKSHIRE	YO8	YO	YO8 9
11	YO8 9XJ	THISTLE CLOSE	SELBY	SELBY	NORTH YORKSHIRE	YO8	YO	YO8 9

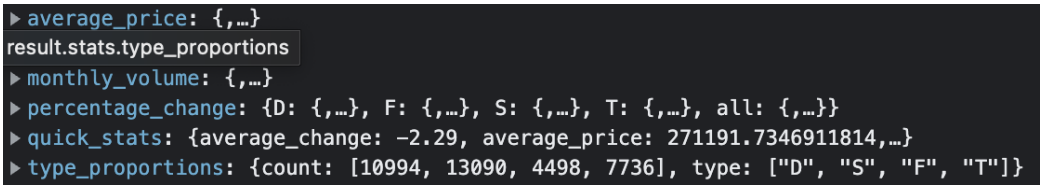
In the images above you can see the data that is stored in the database. Each image is a separate table and you can see that there is no duplicated data and referential integrity is maintained. The way it reduces duplicate data is by using foreign keys to link data to multiple records. For example, you can have multiple sales for a house and instead of storing the address data for the house with each sale you just link to it using a foreign key. This can dramatically reduce the amount of space the data take up compared to its raw form as a CSV file.

Querying Data Using Python

```
import psycopg2
from config import Config


config = Config()
sql_db = psycopg2.connect(f"postgresql://{config.SQL_USER}:{config.SQL_PASSWORD}@{config.SQL_HOST}:5432/house_data")
cur = sql_db.cursor()
cur.execute("""SELECT s.price, s.date, h.type, h.paon, h.saon, h.postcode, p.street, p.town, h.houseid
FROM postcodes AS p
INNER JOIN houses AS h ON p.postcode = h.postcode
INNER JOIN sales AS s ON h.houseid = s.houseid AND h.type != 'O'
WHERE s.ppd_cat = 'A' LIMIT 10;""")
results = cur.fetchall()
for res in results:
    print(res)
```

```
(.venv) (base)  morganthomas@Morgans-MacBook-Air ~/Documents/GitHub/housestats/web-api  main ±+  /Users/morganthomas/Documents/GitHub/housestats/web-api/.venv
/bin/python /Users/morganthomas/Documents/GitHub/housestats/web-api/web/test.py
(30000, datetime.date(1995, 11, 28), 'S', '66', '', 'WS10 9LD', 'HILL STREET', 'WEDNESBURY', '66WS10 9LD')
(42000, datetime.date(2000, 2, 4), 'S', '66', '', 'WS10 9LD', 'HILL STREET', 'WEDNESBURY', '66WS10 9LD')
(115000, datetime.date(2019, 6, 21), 'S', '66', '', 'WS10 9LD', 'HILL STREET', 'WEDNESBURY', '66WS10 9LD')
(57500, datetime.date(1995, 3, 20), 'D', '37', '', 'WS10 9LD', 'HILL STREET', 'WEDNESBURY', '37WS10 9LD')
(57000, datetime.date(1997, 9, 29), 'D', '37', '', 'WS10 9LD', 'HILL STREET', 'WEDNESBURY', '37WS10 9LD')
(25000, datetime.date(1998, 8, 3), 'T', '54', '', 'WS10 9LD', 'HILL STREET', 'WEDNESBURY', '54WS10 9LD')
(83000, datetime.date(2006, 3, 27), 'T', '54', '', 'WS10 9LD', 'HILL STREET', 'WEDNESBURY', '54WS10 9LD')
(49950, datetime.date(2000, 3, 10), 'T', '57', '', 'WS10 9LD', 'HILL STREET', 'WEDNESBURY', '57WS10 9LD')
(59950, datetime.date(2002, 8, 20), 'T', '61', '', 'WS10 9LD', 'HILL STREET', 'WEDNESBURY', '61WS10 9LD')
(50000, datetime.date(2013, 10, 4), 'T', '61', '', 'WS10 9LD', 'HILL STREET', 'WEDNESBURY', '61WS10 9LD')
(.venv) (base)  morganthomas@Morgans-MacBook-Air ~/Documents/GitHub/housestats/web-api  main ±+
```

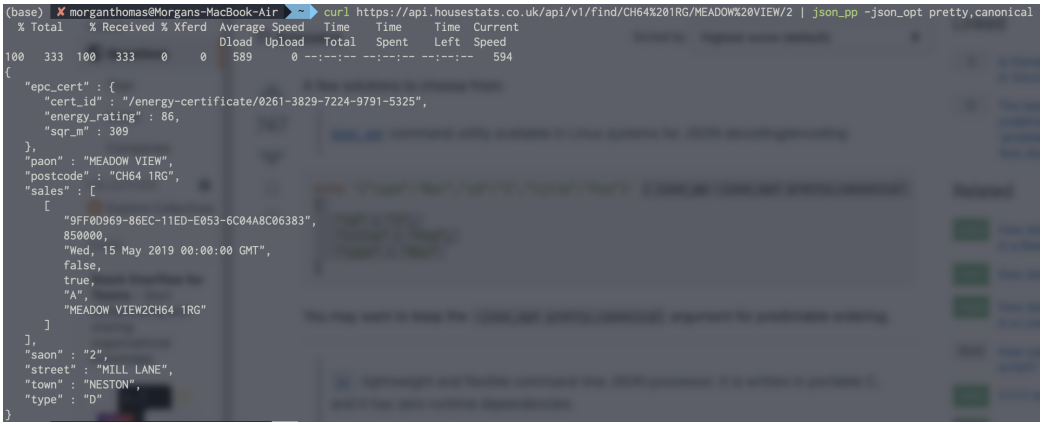
Above you can see the code to fetch the results from the database. It first connects to the database and then executes the query. The results are limited to 10 for testing purposes. These are then printed on each line individually. This is just a simple prototype of how the data is fetched as code similar to this is used all over the programme to fetch data to be processed and then displayed.



Performing Statistical Operation on Data

Below is a screenshot of the data produced after aggregating data for a specific area. This shows two of the criteria met 'Performing Statistical Operations' & 'Selecting data from specific areas and aggregating it'. The data is processed using the Polars library and then outputted in JSON format so it can be easily integrated with a web browser. In the image, the fields are collapsed so you can't see their contents. This is because there are lists containing up to 300 items in them which would take up many pages.

Web API Proof



Below is a screenshot showing the JSON response when sending an HTTPS request to my webserver running the Flask web API. This API can be called programmatically using Javascript. This allows the data to be shown in a web browser or displayed in graphs.

Web UI Proof

Here is the link to the website housestats.co.uk. Below are links to videos showing the website being used. The website can be accessed on any device with an up-to-date web browser and an internet connection.

- Home Dashboard
- Area Statistics
- Search for Historic Sales

Loading Historical Data

Below you can see a screenshot of the timings when making an HTTPS request to the web API route to fetch historical data. Unfortunately, I was not able to tick this one off of the success criteria. The cause of this was the server that the database was running on. This led to SQL transactions taking anywhere between 0.5 seconds and 5 minutes depending on how much data was being returned. This could be fixed by using an SSD on the server so it can read the data quicker resulting in a shorter query time. Although it may not have met the success criteria it is still well within acceptable limits as shown by my stakeholders.

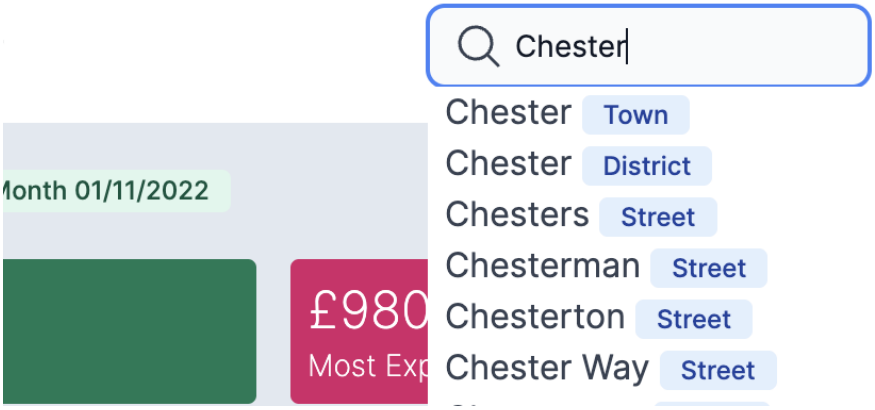
Loading Satistics for an Area

Below are the timings for loading statistics for a given area. You can see that this was able to fetch the data within 2000ms and then return it in the JSON format. There is also a video below showing the whole process in the web UI. Loading Area Statistics

Other Unmet Criteria

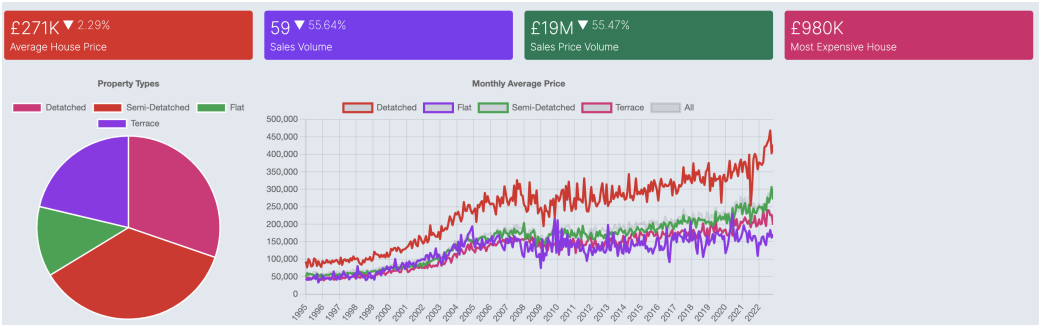
There were two other criteria I was unable to meet but these were due to design decisions I made. The loading of new data into the website was instead changed for a programme which would automatically detect new data on the government website and then insert it into the database and clear the cache. This is a lot more efficient as it removes the need for a human to be involved and is a lot quicker as it happens as soon as there is an update. The other unmet criteria is settings time frames for the data. This again was not met due to a design decision. Instead, the user can zoom in on the graph to see the data from a specific month.

4.3 Usability Features



I managed to incorporate a lot of usability features in my solution. For

example, when searching for a specific area suggestions come up making it easier to search. This is shown in the image above.



The web page is responsive so it works on many different types of devices and the content scales to fit the device making a better user experience. This means that it can be accessed on a portable device which is useful if a surveyor is on-site and they don't have access to a computer it also makes it more accessible as almost all web traffic is done by mobile phones so makes the site more appealing if it also works on a phone.

One of the complaints I got from my stakeholders though is that the data doesn't have any explanations so if you don't know much about housing you might not understand what any of them are. My solution for this would be to add a tooltip for each of the stats which when hovered over would show information about it.

4.4 Limitations

My biggest limitation was the hardware that I was running it on. This often causes my programme to crash as it ran out of memory since the data could often be more than 10GB when processing. It also meant that database queries would take a long time as the CPU had a relatively low clock speed and it was using HDD so it would take quite a while to read data since they aren't very good at random reads. All of this results in API requests are taking longer than they should. This could be quite easily fixed by just buying servers with CPUs that have higher clock speeds and more cores along with NVME SSDs but unfortunately, I did not have the capital for this.

One of the other limitations is the latency of the data provided by the government. House sales have 3 months to be registered. Most of the sales in the UK are registered within 2 months of selling. Therefore my programme will always be two months behind the actual figure and won't be able to provide the most accurate data to its users. I am unable to resolve this issue as the government is unfortunately out of my control.

The other limitation related to the government is how far back the sales go. The earliest sales in the dataset are from 1995 because before then the value of land sales was not required to be logged with HM Land Registry. This again cannot be fixed as there are no other datasets with house sales before 1995 that I can access as they are all behind paywalls and I do not have the capital to access them. If I had the capital a lot of large banks have datasets from their mortgages which can often date back hundreds of years.

4.5 Maintenance

Fortunately, there are only two maintenance issues with my programme. One of the problems is the way the government publish their data. At the moment they publish their data in CSV format but they could decide to change this would result in the update module failing so no new data is added. As of right now, the Land Registry is not planning on changing the way they distribute their data but that could change.

Another issue is maintaining the front end to be compatible with modern browsers. Modern browsers are updated frequently resulting in the javascript API changing and old features being no longer supported and new features being added to replace them. Then you have to factor in that not all browsers get these new features at the same time so some browsers will get the new features before others. This can be combatted by using a javascript bundler called Babel. Babel converts the Javascript code into vanilla Javascript using only the most basic functions that are supported on all browsers.

Chapter 5

Appendix

5.1 Sales Ingest

5.1.1 __init__.py

```
from ingest import Ingest
import asyncio
import atexit


if __name__ == "__main__":
    ingester = Ingest()
    atexit.register(asyncio.run, ingester.remove_status
        ())
    asyncio.run(ingester.main_loop())
```

5.1.2 ingest.py

```
import os
import re
import socket
from datetime import datetime
from pickle import loads
from typing import List
```

```
from asyncpg import connect
from confluent_kafka import Consumer
from dotenv import load_dotenv


class Ingest():
    def __init__(self, test=False) -> None:
        self._load_env()
        if not test:
            self._consumer = Consumer({
                'bootstrap.servers': self._KAFKA,
                'group.id': 'INGESTER',
                'auto.offset.reset': 'earliest'
            })
        self._areas = ["postcode", "street", "town", "
            district", "county", "outcode", "area", "
            sector"]
        # Regex to split postcode into inward, outward
            & area
        self._postcode_re = re.compile("^(?:(?P<a1>[Gg
            ][Ii][Rr])(?P<d1>)(?P<s1>0)(?P<u1>[Aa]{2}))
            |(?:(?:(?:(?P<a2>[A-Za-z])(?P<d2>[0-9]{1,2})
            )|(?:(?:(?P<a3>[A-Za-z][A-Ha-hJ-Yj-y])(?P<d3
            >[0-9]{1,2}))|(?:(?:(?P<a4>[A-Za-z])(?P<d4
            >[0-9][A-Za-z]))|(?:(?P<a5>[A-Za-z][A-Ha-hJ-
            Yj-y])(?P<d5>[0-9]?[A-Za-z]))))))(?P<s2
            >[0-9])(?P<u2>[A-Za-z]{2}))$", flags=re.
            IGNORECASE)

    def _load_env(self):
        # Loads the enviroment variables
        load_dotenv()
        self._DB = os.environ.get("DBNAME", "house_data
            ")
        self._USERNAME = os.environ.get("POSTGRES_USER"
            )
```

```
        self._PASSWORD = os.environ.get("
            POSTGRES_PASSWORD")
        self._HOST = os.environ.get("POSTGRES_HOST")
        self._KAFKA = os.environ.get("KAFKA")

    async def _connect_db(self):
        self._conn = await connect(f"postgresql://{self
            ._USERNAME}:{self._PASSWORD}@{self._HOST}/{
            self._DB}")

    def extract_parts(self, postcode: str) -> List[str
        ]:
        try:
            if (parts := self._postcode_re.findall(
                postcode)[0]) != None:  # splits
                postcode & checks if it is valid
                parts = list(filter(lambda x : x != '',
                    parts))  # Removes empty parts from
                    postcode
                outcode = parts[0] + parts[1]
                area = parts[0]
                sector = parts[0] + parts[1] + "-" +
                    parts[2]
                return [outcode, area, sector]  #
                    Returns the parts of the postcode
            else:
                return ["","",""]
        except IndexError:
            return ["","",""]

    async def _set_status(self, status: str) -> None:
        consumer_id = socket.gethostname()
        try:
            await self._conn.execute("INSERT INTO
                settings (name, data) VALUES ($1, $2);",
                consumer_id, status)
        except:
```

```
            await self._conn.execute("DELETE FROM
                settings WHERE name = $1", consumer_id)
            await self._conn.execute("INSERT INTO
                settings (name, data) VALUES ($1, $2);",
                 consumer_id, status)

    async def remove_status(self):
        consumer_id = socket.gethostname()
        print("DELETING")
        await self._conn.execute("DELETE FROM settings
            WHERE name = $1", consumer_id)

    async def _insert_areas(self, sale: List,
        postcode_parts: List[str]):
        areas = [sale[3], sale[9], sale[11], sale[12],
            sale[13],
                 postcode_parts[0], postcode_parts[1],
                     postcode_parts[2]] # Extracts areas
                      values from sale
        values = []
        for idx, area_type in enumerate(self._areas):
            area_data = (area_type, areas[idx])
            values.append(area_data)
        await self._conn.executemany("""INSERT INTO
            areas (area_type, area)
                                    VALUES ($1,$2) ON
                                       CONFLICT (area_type,
                                        area) DO NOTHING;
                                    """,values)

    async def main_loop(self):
        print("Waiting for messages")
        await self._connect_db()
        self._consumer.subscribe(["new_sales"])
        while True:
            await self._set_status("WAITING")
            msg = self._consumer.poll(1.0)  # Fetches
                the latest message from kafka
```

```
            await self._set_status("PROCESSING")
            if msg is None:  #Checks the message isnt
               empty
                continue
            if msg.error():  # Checks there are no
               errors
                print("Consumer error: {}".format(msg.
                   error()))
                continue
            sale: List = loads(msg.value())  # Converts
                the bytes into a python list
            await self._process_sale(sale)

    async def _process_sale(self, sale):
        async with self._conn.transaction():
            if sale[-1] in ["C", "D"]:
                await self._conn.execute("DELETE FROM
                   sales WHERE tui=$1", sale[0]) #
                   Delete sale
            if sale[-1] in ["A", "C"]:
                postcode_parts = self.extract_parts(
                   sale[3])  # Fetches the postcode
                   parts
                await self._insert_areas(sale,
                   postcode_parts)

                houseID = str(sale[7]) + str(sale[8]) +
                    str(sale[3])
                await self._conn.execute("INSERT INTO
                   postcodes \
--------------------------------(postcode, street, town
   , district, county, outcode, area, sector) \
--------------------------------VALUES ($1, $2, $3, $4, $5,
   $6, $7, $8) ON CONFLICT (postcode) DO NOTHING;",
                                         sale[3], sale[9], sale
                                            [11], sale[12], sale
                                            [13], postcode_parts
                                            [0],
```

```
                                    postcode_parts[1],
                                        postcode_parts[2])
                                        # Insert into
                                        postcode table

                    await self._conn.execute("INSERT INTO
                        houses (houseID, PAON, SAON, type,
                        postcode) \
--------------------------------VALUES ($1,$2,$3,$4,$5)
    ON CONFLICT (houseID) DO NOTHING;",
                                    houseID, sale[7], sale
                                        [8], sale[4], sale
                                        [3])  # Insert into
                                        house table

                    new = True if sale[5] == "Y" else False
                        # Convets to boolean type
                    freehold = True if sale[6] == "F" else
                        False  # Converts to boolean type
                    date = datetime.strptime(sale[2], "%Y-%
                        m-%d %H:%M")  # Converts string to
                        datetime object
                    await self._conn.execute("INSERT INTO
                        sales (tui, price, date, new,
                        freehold, ppd_cat, houseID) \
----------------------------VALUES ($1,$2,$3,$4,$5,$6,
    $7) ON CONFLICT (tui) DO NOTHING;",
                                sale[0], int(sale[1]), date
                                    , new, freehold, sale
                                    [14], houseID) # Insert
                                    into sales table


if __name__ == "__main__":
    import asyncio
    x = Ingest()
    asyncio.get_event_loop().run_until_complete(x.
        main_loop())
```

5.1.3 Dockerfile

```
FROM python:3.9.7

WORKDIR /app

COPY ./ingester .
COPY ./requirements.txt .


RUN python3 -m pip install --upgrade pip setuptools
    wheel
RUN python3 -m pip install -r requirements.txt

CMD ["python3", "__init__.py"]
```

5.1.4 requirements.txt

```
asyncpg==0.27.0
certifi==2022.12.7
confluent-kafka==2.0.2
python-dotenv==0.21.1
sentry-sdk==1.14.0
urllib3==1.26.14
```

5.2 Update Checker

5.2.1 __init__.py

```
from check_update import checkForUpdate

if __name__ == "__main__":
```

```
    x = checkForUpdate()
    x.run()
```

5.2.2 check_update.py

```
import time
from csv import reader
from hashlib import sha256
from io import StringIO
from os import environ
from pickle import dumps

import requests
import schedule
from confluent_kafka import Producer
from dotenv import load_dotenv
from psycopg2 import connect
from requests import get


class checkForUpdate():
    def __init__(self) -> None:
        self._file_link = "http://prod.publicdata.
            landregistry.gov.uk.s3-website-eu-west-1.
            amazonaws.com/pp-monthly-update.txt"
        self._load_env()
        self._conn = connect(f"dbname={self._DB} \
            user={self._USERNAME} password={self.
            _PASSWORD} \
            host={self._HOST}")  # Connect to the
            database
        self._cur = self._conn.cursor()
        self._producer = Producer({"bootstrap.servers":
            self._KAFKA})  # Connect to Kafka cluster

    def _load_env(self):
        # Loads enviroment variables
```

```
        load_dotenv()
        self._DB = environ.get("DBNAME", "house_data")
        self._USERNAME = environ.get("POSTGRES_USER")
        self._PASSWORD = environ.get("POSTGRES_PASSWORD
            ")
        self._HOST = environ.get("POSTGRES_HOST")
        self._KAFKA = environ.get("KAFKA")

    def _fetch_file(self):
        print("fetching file")
        file = get(self._file_link).content  # Download
            monthly file from land registry
        file_hash = sha256(file).hexdigest()  #
            Calculate hash of file
        self._cur.execute("SELECT data FROM settings
            WHERE name='update_hash';")
        prev_hash = self._cur.fetchone()  # Check to
            see if file has been inserted already
        if prev_hash is not None:
            prev_hash = prev_hash[0]
        if file_hash != prev_hash:  # Compare hash to
            hash of old file
            print("New file being uploaded")
            self._update_database(file, file_hash)
        else:
            print("No new file yet")

    def _update_database(self, file, file_hash):
        self._send_file_db(file)
        self._cur.execute("""UPDATE settings SET data =
            %s
                          WHERE name='update_hash'
                              returning name;""",
                          (file_hash,))
        # Changes previous hash to most recent one

        if self._cur.fetchone() is None:
```

```
            self._cur.execute("""INSERT INTO settings (
                name, data)
                                 VALUES ('update_hash', %s
                                     )""",
                                 (file_hash,))
            # Inserts update_hash row if it doesn't
                exist
        self._conn.commit()

    def _send_file_db(self, file):
        csv_file = reader(StringIO(file.decode("UTF-8")
            ))
        csv_file = map(lambda x: dumps([x[0][1:-1]] + [
            i for i in x[1:]]), csv_file)  # Remove
            braces from tui
        while True:  # While is quicker than for loop
            try:
                list_bytes = next(csv_file) # Converts
                    list to byte array
                while True:
                    try:
                        self._producer.produce("
                            new_sales", list_bytes)  #
                            Send each sale as string to
                            kafka
                        self._producer.poll(0)
                        break
                    except BufferError:
                        print(time.time(), "Flushing")
                        self._producer.flush()
                        print(time.time(), "Finished-
                            flush")
            except StopIteration:
                self._producer.flush()
                break
        self._cur.execute("""UPDATE settings SET data =
            %s
```

```
                        WHERE name='last_updated';"""
                        ,
                        (time.time(),))
        self.aggregate_counties()

    def aggregate_counties(self):
        self._cur.execute("SELECT * FROM settings WHERE
            name = 'last_updated' OR name = '
            last_aggregated_counties' ORDER BY name DESC
            ;")
        times = self._cur.fetchall()
        print(times)
        if float(times[0][1]) > float(times[1][1]):
            self._cur.execute("SELECT * FROM settings
                WHERE data = 'WAITING';")
            res = self._cur.fetchall()
            if len(res) == 4:
                self._cur.execute("SELECT area FROM
                    areas WHERE area_type = 'area';")
                counties = self._cur.fetchall()
                self._cur.execute("""UPDATE settings
                    SET data = 'true'
                                WHERE name='
                                    agregating_counties
                                    ';""")
                self._conn.commit()
                for county in counties:
                    county = county[0] if county != ('
                        ',) else 'CH'
                    resp = requests.get(f"https://api.
                        housestats.co.uk/api/v1/analyse/
                        area/{county}")
                    print(county, resp.json()["status"
                        ])
                    if county == counties[-1][0]:
                        url = resp.json()["result"]
                        while True:
                            resp = requests.get(url)
```

```
                if resp.json()["status"] ==
                    "SUCCESS":
                    self._cur.execute("
                        UPDATE settings SET
                        data = %s WHERE name
                        = '
                        last_aggregated_counties
                        ';",
                                          (time.
                                              time
                                              (),)
                                          )
                    self._cur.execute("
                        UPDATE settings SET
                        data = 'false' WHERE
                        name='
                        agregating_counties
                        ';")
                    self._conn.commit()
                    break
                else:
                    time.sleep(5)

    def run(self):
        schedule.every(5).minutes.do(self._fetch_file)
        while True:
            schedule.run_pending()
            time.sleep(30)

if __name__ == "__main__":
    x = checkForUpdate()
    x.aggregate_counties()
    # x.run()
```

5.2.3 Dockerfile

```
FROM python:3.9.7
```

```
WORKDIR /app

COPY ./checker .
COPY ./requirements.txt .


RUN python3 -m pip install --upgrade pip setuptools
    wheel
RUN python3 -m pip install -r requirements.txt

CMD ["python3", "__init__.py"]
```

5.2.4 requirements.txt

```
certifi==2022.12.7
charset-normalizer==3.0.1
confluent-kafka==2.0.2
idna==3.4
psycopg2==2.9.5
python-dotenv==0.21.1
requests==2.28.2
schedule==1.1.0
sentry-sdk==1.15.0
urllib3==1.26.14
```

5.3 Data Processor

5.3.1 __init__.py

```
from main import Processor

if __name__ == "__main__":
    print("loading")
    processor = Processor()
```

```
    print("Running")
    processor.main_loop()
```

5.3.2 aggregations.py

```
from typing import Dict

import polars as pl
from load_data import Loader
from datetime import timedelta

class Aggregator():
    def __init__(self, data: Loader) -> None:
        self._data = data.data
        self._latest_date = data.latest_date

    def _calc_average_price(self) -> Dict:
        df = self._data.partition_by("type", as_dict=
            True)
        house_types_means = {}
        for house_type in df:
            temp_df = df[house_type]
            house_types_means[house_type] = temp_df \
                                                .sort("
                                                    date
                                                    ") \
                                                .
                                                    groupby_dynamic
                                                    ("
                                                    date
                                                    ",
                                                    every
                                                    ="1
                                                    mo")
                                                      \
                                                .agg(pl
                                                    .col
```

```
                                                    ("
                                                    price
                                                    ").
                                                    log
                                                    ().
                                                    mean
                                                    ().
                                                    exp
                                                    ())
                                                    \
                                                .
                                                    to_dict
                                                    (
                                                    as_series
                                                    =
                                                    False
                                                    )
    all_sales = self._data.lazy()
    std = all_sales.select(pl.col("price")).std().
        collect()[0, 0]
    mean = all_sales.select(pl.col("price")).mean()
        .collect()[0, 0]
    temp_df = all_sales.filter((pl.col("price") <
        mean+(2*std)))
    house_types_means["all"] = all_sales \
                                .sort("date") \
                                .groupby_dynamic("
                                    date", every="1
                                    mo") \
                                .agg(pl.col("price"
                                    ).log().mean().
                                    exp()) \
                                .collect() \
                                .to_dict(as_series=
                                    False)
    data = {
        "type": [key for key in sorted(
            house_types_means)],
```

```
            "prices": [house_types_means[key]["price"]
                for key in sorted(house_types_means)],
            "dates": house_types_means["all"]["date"]
        }
        return data

    def _remove_outliers(self, df: pl.DataFrame):
        std = df.select(pl.col("price")).std()[0, 0]
        mean = df.select(pl.col("price")).mean()[0, 0]
        df = df.filter(pl.col("price") < mean + (3*std)
            )
        return df


    def _calc_type_proportions(self) -> Dict:
        df = self._data
        df = df.unique(subset=["houseid"])
        df = df.groupby("type").count()
        data = df.to_dict(as_series=False)
        return data

    def _calc_monthly_volume(self) -> Dict:
        df = self._data.partition_by("type", as_dict=
            True)
        monthly_volumes = {}
        for house_type in df:
            temp_df = df[house_type].lazy()
            volume = temp_df \
                .sort("date") \
                .groupby_dynamic("date", every="1mo") \
                .agg(pl.col("price").count().alias("
                    volume")) \
                .collect() \
                .to_dict(as_series=False)
            monthly_volumes[house_type] = volume

        monthly_volumes["all"] = self._data.sort("date"
            ) \
```

```
                .groupby_dynamic("date", every="1mo") \
                .agg(pl.col("price").count().alias("
                    volume")) \
                .to_dict(as_series=False)

        data = {
            "type": [key for key in sorted(
                monthly_volumes)],
            "volume": [monthly_volumes[key]["volume"]
                for key in sorted(monthly_volumes)],
            "dates": monthly_volumes["all"]["date"]
        }
        return data

    def _calc_monthly_price_volume(self) -> Dict:
        df = self._data.partition_by("type", as_dict=
            True)
        monthly_price_volume = {}
        for house_type in df:
            temp_df = df[house_type].lazy()
            volume = temp_df \
                .sort("date") \
                .groupby_dynamic("date", every="1mo") \
                .agg(pl.col("price").sum().alias("
                    volume")) \
                .collect() \
                .to_dict(as_series=False)
            monthly_price_volume[house_type] = volume

        monthly_price_volume["all"] = self._data.sort("
            date") \
                .groupby_dynamic("date", every="1mo") \
                .agg(pl.col("price").sum().alias("
                    volume")) \
                .to_dict(as_series=False)
        data = {
            "type": [key for key in sorted(
                monthly_price_volume)],
```

```
            "volume": [monthly_price_volume[key]["
                volume"] for key in sorted(
                monthly_price_volume)],
            "dates": monthly_price_volume["all"]["date"
                ]
        }
        return data

    def _calc_all_perc(self) -> Dict:
        data = self._data.partition_by("type", as_dict=
            True)
        monthly_perc = {}
        for house_type in data:
            monthly_perc[house_type] = self.
                _calc_ind_percentage(data[house_type]).
                to_dict(as_series=False)
        monthly_perc["all"] = self._calc_ind_percentage
            (self._data).to_dict(as_series=False)
        return monthly_perc


    def _calc_ind_percentage(self, df: pl.DataFrame) ->
        pl.DataFrame:
        df = df.sort("date") \
                .groupby_dynamic("date", every="1mo") \
                .agg(pl.col("price").log().mean().exp()
                    .alias("avg_price"))
        df = df.with_columns([
            pl.col("date").dt.month().alias("month"),
            pl.col("date").dt.year().alias("year")
        ])
        df = df.groupby("month").apply(self.
            _calc_percentages_months)
        df = df.drop(["year", "month", "prev_year", "
            avg_price"])
        df = df.sort("date")
        return df
```

```
    def _calc_percentages_months(self, data: pl.
        DataFrame):
        df = data.sort("date").with_columns(
            pl.col("avg_price").shift().alias("
                prev_year")
        )
        df = df.filter(pl.col("prev_year").is_not_null
            ())
        df = df.with_columns(
            (((pl.col("avg_price")-pl.col("prev_year"))
                /pl.col("avg_price")*100)/12).alias("
                perc_change")
        )
        return df

    def _quick_stats(self, data) -> Dict[str, float]:
        try:
            current_month = data["average_price"]["
                dates"][-2]
            current_average = data["average_price"]["
                prices"][4][-2]
            prev_average = data["average_price"]["
                prices"][4][-3]
            current_average_change = round(100*(
                current_average-prev_average)/
                prev_average,2)
        except Exception:
            return {
                "current_month": 0,
                "average_price": 0,
                "average_change": 0,
                "current_sales_volume": 0,
                "sales_volume_change": 0,
                "current_price_volume": 0,
                "price_volume_change": 0,
                "expensive_sale": 0
            }
```

```
        try:
            current_sales_vol = data["
                monthly_sales_volume"]["volume"][4][-2]
            prev_sales_vol = data["monthly_sales_volume
                "]["volume"][4][-3]
            current_sales_vol_change = round(100*(
                current_sales_vol-prev_sales_vol)/
                prev_sales_vol,2)
        except IndexError:
            current_sales_vol = 0
            current_sales_vol_change = 0

        try:
            current_price_vol =  data["
                monthly_price_volume"]["volume"][4][-2]
            prev_price_vol = data["monthly_price_volume
                "]["volume"][4][-3]
            current_price_vol_change = round(100*(
                current_price_vol-prev_price_vol)/
                prev_price_vol,2)
        except IndexError:
            current_price_vol = 0
            current_price_vol_change = 0

        expensive_sale = (self._data
            .filter(pl.col("date").is_between(
                current_month, current_month + timedelta
                (days=31)))
            .filter(pl.col("price") == pl.col("price").
                max())
            )[0,0]

        quick_stats = {
            "current_month": current_month,
            "average_price": current_average,
            "average_change": current_average_change,
            "current_sales_volume": current_sales_vol,
```

```
            "sales_volume_change":
                current_sales_vol_change,
            "current_price_volume": current_price_vol,
            "price_volume_change":
                current_price_vol_change,
            "expensive_sale": expensive_sale
        }
        return quick_stats

    def get_all_data(self) -> Dict:
        data = {
            "average_price": self._calc_average_price()
                ,
            "type_proportions": self.
                _calc_type_proportions(),
            "monthly_sales_volume": self.
                _calc_monthly_volume(),
            "monthly_price_volume": self.
                _calc_monthly_price_volume(),
            "percentage_change": self._calc_all_perc()
        }
        data["quick_stats"] = self._quick_stats(data)
        return data



if __name__ == "__main__":
    import time
    import psycopg2

    start = time.time()
    conn = psycopg2.connect("postgresql://house_data:
        lriFahwbJwfv2388neiluOMI@192.168.4.30:5432/
        house_data")
    data_loader = Loader("CH3-5", "sector", conn.cursor
        ())
    print(f"loaded_data - {time.time() - start}")
    agg = Aggregator(data_loader)
```

```
    data = agg.get_all_data()

    inital_price = 249000
    final_price = 249000
    for month in data["percentage_change"]["S"]["
        perc_change"][281:]:
          final_price *= 1+(month/100)
    print(final_price)
```

5.3.3 load_data.py

```
from datetime import datetime, timedelta

import polars as pl
from typing import List
from dateutil.relativedelta import relativedelta

class Loader():
    def __init__(self, area: str, area_type: str,
        db_cur) -> None:
          self._cur = db_cur
          self.area_type = area_type.lower()
          self.area = area.upper()
          self._areas = ["postcode", "street", "town", "
              district", "county", "outcode", "area", "
              sector"]
          if self.area == "" and self.area_type == "":
               self._cur.execute("""SELECT s.price, s.date
                   , h.type, h.paon, h.saon, h.postcode, p.
                   street, p.town, h.houseid
                        FROM postcodes AS p
                        INNER JOIN houses AS h ON p.
                            postcode = h.postcode
                        INNER JOIN sales AS s ON h.houseid
                            = s.houseid AND h.type != 'O'
                        WHERE s.ppd_cat = 'A';""")
               data = self._cur.fetchall()
```

```
            self._format_df(data)
        else:
            if self.area_type not in self._areas:
                raise ValueError("Invalid area type")
            else:
                if self.verify_area():
                    data = self.fetch_area_sales()
                    self._format_df(data)

    def verify_area(self):
        self._cur.execute(f"SELECT postcode FROM
            postcodes WHERE {self.area_type} = %s LIMIT
            1;", (self.area,))
        if self._cur.fetchall() is not []:
            return True
        else:
            raise ValueError(f"Invalid {self.area_type}
                entered")

    def fetch_area_sales(self) -> List:
        query = f"""SELECT s.price, s.date, h.type, h.
            paon, h.saon, h.postcode, p.street, p.town,
            h.houseid
                FROM postcodes AS p
                INNER JOIN houses AS h ON p.postcode =
                    h.postcode AND p.{self.area_type} =
                    %s
                INNER JOIN sales AS s ON h.houseid = s.
                    houseid AND h.type != 'O'
                WHERE s.ppd_cat = 'A' AND s.date < %s;
                """
        self._cur.execute(query, (self.area, self.
            latest_date))
        data = self._cur.fetchall()
        if data == []:
            raise ValueError(f"No sales for area {self.
                area}")
        else:
```

```
        return data

    def _format_df(self, data):
        self._data = pl.DataFrame(data,
                                  columns=["price","
                                      date","type","
                                      paon","saon",
                                             "postcode"
                                                 ,"
                                                 street"
                                                 ,"town"
                                                 ,"
                                                 houseid
                                                 "],
                                  orient="row")
        self._data = self._data.with_column(
            pl.col('date').apply(lambda x: datetime(*x.
                timetuple()[:-4])).alias("dt")
        )
        self._data = self._data.drop("date")
        self._data = self._data.with_column(
            pl.col("dt").alias("date")
        )
        self._data = self._data.drop("dt")

    @property
    def data(self) -> pl.DataFrame:
        return self._data

    @property
    def latest_date(self):
        self._cur.execute("SELECT date FROM sales ORDER
            BY date DESC LIMIT 1;")
        latest_date = self._cur.fetchone()
        if latest_date is not None:
            latest_date = datetime.combine(latest_date
                [0], datetime.min.time())
```

```
            if latest_date > (datetime.now() -
               timedelta(days=60)):
                start = datetime.now().replace(day=1)
                return start - relativedelta(months=2)
            else:
                return latest_date[0]

if __name__ == "__main__":
    import psycopg2
    conn = psycopg2.connect("postgresql://house_data:
       lriFahwbJwfv2388neiluOMI@192.168.4.30:5432/
       house_data")
    loader = Loader("CH", "area", conn.cursor())
    print(loader.latest_date)
    print(loader.data.head())
```

5.3.4 main.py

```
import os
import time
from datetime import datetime
from pickle import loads
from typing import Dict

import psycopg2
from aggregations import Aggregator
from confluent_kafka import Consumer
from load_data import Loader
from pymongo import MongoClient


class Processor():
    def __init__(self):
        self._load_env()
        self._sql_conn = psycopg2.connect(f"postgresql
           ://{self._SQL_USERNAME}:{self._SQL_PASSWORD}
           @{self._SQL_HOST}:5432/house_data")
```

```
        self._cur = self._sql_conn.cursor()
        self._mongo_conn = MongoClient(f"mongodb://{
            self._MONGO_USERNAME}:{self._MONGO_PASSWORD}
            @{self._MONGO_HOST}:27017/?authSource=
            house_data")
        self._mongo_db = self._mongo_conn["house_data"]
        self._consumer = Consumer({
                'bootstrap.servers': self._KAFKA,
                'group.id': 'PROCESSOR',
                'auto.offset.reset': 'earliest'
            })

    def _load_env(self):
        # Loads the enviroment variables
        self._DB = os.environ.get("DBNAME", "house_data
            ")
        self._SQL_USERNAME = os.environ.get("
            POSTGRES_USER")
        self._SQL_PASSWORD = os.environ.get("
            POSTGRES_PASSWORD")
        self._SQL_HOST = os.environ.get("POSTGRES_HOST"
            )
        self._KAFKA = os.environ.get("KAFKA")
        self._MONGO_HOST = os.environ.get("MONGO_HOST")
        self._MONGO_USERNAME = os.environ.get("
            MONGO_USERNAME")
        self._MONGO_PASSWORD = os.environ.get("
            MONGO_PASSWORD")

    def main_loop(self) -> None:
        self._consumer.subscribe(["query_queue"])
        print("Waiting for queries")
        while True:
            msg = self._consumer.poll(1.0)  # Fetches
                the latest message from kafka
            if msg is None:  #Checks the message isnt
                empty
                continue
```

```
            if msg.error():  # Checks there are no
                errors
                print("Consumer error: {}".format(msg.
                    error()))
                continue
            query: tuple = loads(msg.value()) # (area,
                area_type)
            query = tuple(map(lambda x: x.upper(),
                query)) # Makes all items upper case

            print(f"{time.time()} - {query[0]}({query
                [1]})")
            if not self._check_cache(*query):
                print(query, "- Aggregating data")
                self._get_stats(*query)
            else:
                print(query, "- Cache hit")
                continue

    def _check_cache(self, area, area_type) -> bool:
        query_id = self._calc_query_id(area, area_type)
        query = self._mongo_db.cache.find_one({"_id":
            query_id})
        if query is not None:
            last_updated = self._get_last_updated()
            if query["last_updated"] < last_updated:
                return False
            return True
        else:
            return False

    def _get_last_updated(self):
        self._cur.execute("SELECT * FROM settings WHERE
            name = 'last_updated'")
        last_updated = self._cur.fetchone()
        if last_updated == None:
            return datetime.fromtimestamp(0)
        else:
```

```
            if last_updated[1] is not None:
                return datetime.fromtimestamp(float(
                    last_updated[1]))
            else:
                return datetime.fromtimestamp(0)


    def _get_stats(self, area: str, area_type: str) ->
        bool:
        load_start = time.time()
        data = self._get_area_data(area, area_type)
        load_time = time.time()-load_start
        if data is not None:
            start = time.time()
            stats = self._get_aggregation(data)
            time_taken = time.time()-start
            query_id = self._calc_query_id(area,
                area_type)
            self._cache_query(stats, query_id, area,
                area_type, time_taken, load_time)
            return True
        else:
            return False

    def _cache_query(self, stats: Dict, query_id: str,
        area: str, area_type: str, exe_time: float,
        load_time: float):
        query = self._mongo_db.cache.find_one({"_id":
            query_id})
        if query is not None:
            self._mongo_db.cache.update_one(
                {"_id": query_id},
                {"$set": {
                    "data": stats,
                    "last_updated": datetime.now(),
                    "exec_time": exe_time,
                    "load_time": load_time
                    }
```

```
                }
                )
        else:
            document = {
                "_id": query_id,
                "area": area,
                "area_type": area_type,
                "data": stats,
                "last_updated": datetime.now(),
                "exec_time": exe_time,
                "load_time": load_time
            }
            self._mongo_db.cache.insert_one(document)


    def _calc_query_id(self, area: str, area_type: str)
        -> str:
        query_id = (area + area_type).replace("-", "")
        return query_id


    def _get_area_data(self, area: str, area_type: str)
        :
        try:
            if area == "ALL" and area_type == "COUNTRY"
                :
                lodr = Loader("", "", self._cur)
            else:
                lodr = Loader(area, area_type, self.
                    _cur)
            return lodr
        except Exception as e:
            pass # Store error in db with the query
                data

    def _get_aggregation(self, loader: Loader) -> Dict:
        agg = Aggregator(loader)
        data = agg.get_all_data()
```

```
        return data

if __name__ == "__main__":
    processor = Processor()
    processor.main_loop()
```

5.3.5 Dockerfile

```
FROM python:3.10.8

WORKDIR /app

COPY ./processor .
COPY ./requirements.txt .


RUN python3 -m pip install --upgrade pip setuptools
    wheel
RUN python3 -m pip install -r requirements.txt

CMD ["python3", "__init__.py"]
```

5.3.6 requirements.txt

```
certifi==2022.12.7
confluent-kafka==2.0.2
contourpy==1.0.7
cycler==0.11.0
dnspython==2.3.0
fonttools==4.38.0
kiwisolver==1.4.4
numpy==1.24.1
packaging==23.0
Pillow==9.4.0
polars==0.15.18
psycopg2==2.9.5
```

```
pymongo==4.3.3
pyparsing==3.0.9
python-dateutil==2.8.2
sentry-sdk==1.14.0
six==1.16.0
typing_extensions==4.4.0
urllib3==1.26.14
```

5.4 Web API

5.4.1 __init__.py

```
import os
import psycopg2
from config import Config
from flask import Flask, current_app
from flask_cors import CORS
from pymongo import MongoClient
from sentry_sdk.integrations.flask import
    FlaskIntegration


def create_app(config_class=Config) -> Flask:
    app = Flask(__name__)
    app.config.from_object(config_class)
    cors = CORS(app, resources={r"/api/*": {"origins":
        "*"}})


    mongo_db = MongoClient(f"mongodb://{app.config['
        MONGO_USER']}:{app.config['MONGO_PASSWORD']}@{
        app.config['MONGO_HOST']}:27017/?authSource=
        house_data")
    sql_db = psycopg2.connect(f"postgresql://{app.
        config['SQL_USER']}:{app.config['SQL_PASSWORD']}
        @{app.config['SQL_HOST']}:5432/house_data")
```

```
    with app.app_context():
        current_app.mongo_db = mongo_db.house_data
        current_app.sql_db = sql_db


    from app.api import bp as api_bp
    app.register_blueprint(api_bp, url_prefix="/api/v1"
        )

    @app.route("/")
    def checker():
        return "UP"

    return app
```

5.4.2 config.py

```
import os
from dotenv import load_dotenv

load_dotenv()
basedir = os.path.abspath(os.path.dirname(__file__))

class Config:
    SECRET_KEY = os.environ.get('SECRET_KEY')
    SQL_USER = os.environ.get("POSTGRES_USER")
    SQL_PASSWORD = os.environ.get("POSTGRES_PASSWORD")
    SQL_HOST = os.environ.get("POSTGRES_HOST")
    MONGO_HOST =os.environ.get("MONGO_HOST")
    MONGO_USER = os.environ.get("MONGO_USERNAME")
    MONGO_PASSWORD = os.environ.get("MONGO_PASSWORD")
```

5.4.3 api/__init__.py

```
from flask import Blueprint

```

```
bp = Blueprint("api", __name__)

from app.api import routes
```

5.4.4 routes.py

```
import urllib.parse
from datetime import datetime
from typing import List, Tuple

from app.api import bp, search_area_funcs
from app.celery import analyse_task, valuation_task
from flask import abort, current_app, jsonify, request,
    url_for
from app.api import epc_cert
from app.api import country

@bp.route("/analyse/<string:area_type>/<string:area>")
def index(area_type, area):
    with current_app.app_context():
        query_id = area.upper() + area_type.upper()
        result = current_app.mongo_db.cache.find_one({"
            _id": query_id})
    if result is None:
        task = analyse_task.delay(area, area_type)
        return jsonify(
            status="ok",
            task_id=task.id,
            result=f"https://api.housestats.co.uk{
                url_for('api.fetch_results',query_id=
                query_id)}?task_id={task.id}"
        )
    else:
        return jsonify(
            status="ok",
            result=f"https://api.housestats.co.uk{
                url_for('api.fetch_results',query_id=
```

```
                query_id)}"
        )

@bp.route("/get/<string:query_id>")
def fetch_results(query_id):
    task_id = request.args.get("task_id", None)
    if task_id is not None:
        task = analyse_task.AsyncResult(task_id)
        if task.state == "PENDING":
            return {
                "status": task.state
            }
        else:
            query_id = task.wait()
            with current_app.app_context():
                result = current_app.mongo_db.cache.
                    find_one({"_id": query_id})
            return {
                "status": task.state,
                "result": result
            }
    else:
        with current_app.app_context():
            result = current_app.mongo_db.cache.
                find_one({"_id": query_id})
        if result is not None:
            return {
                "status": "SUCCESS",
                "result": result
            }
        else:
            return {
                "status": "FAILED"
            }


@bp.route("/search/<string:query>")
def search_area(query):
```

```
    query = urllib.parse.unquote(query).upper()
    query_filter = request.args.get("filter", None)

    sql_query = search_area_funcs.generate_sql_query(
       query, query_filter=query_filter)
    if sql_query == "":
        return "Failed to generate query", 500

    with current_app.app_context():
        cur = current_app.sql_db.cursor()
        cur.execute(sql_query)
        results: List[Tuple[str,str]] = cur.fetchall()

    if len(results) > 0:
        sorted_res = search_area_funcs.sort_results(
           results)
        return jsonify(
            results=sorted_res,
            found=True
        )
    else:
        return jsonify(
            results=None,
            found=False
        )

@bp.route("/find/<string:postcode>")
def search_houses(postcode):
    sql_query = """SELECT h.type, h.paon, h.saon, h.
       postcode, p.street, p.town, p.county
                    FROM postcodes AS p
                    INNER JOIN houses AS h ON p.
                        postcode = h.postcode AND p.
                        postcode = %s;"""
    with current_app.app_context():
        cur = current_app.sql_db.cursor()
        cur.execute(sql_query, (postcode.upper(),))
```

```
        results: List[Tuple[str,str,str,str,str,str]] =
            cur.fetchall()
    results = sorted(list(set(results)), key=lambda x:
       x[1])
    if results != []:
        return jsonify(
            results=results,
        )
    else:
        return abort(404, "Cannot Find Houses for
           Postcode")

@bp.route("/find/<string:postcode>/<path:house>")
def get_house_saon(postcode, house):
    try:
        paon, saon = house.split("/")
    except ValueError:
        paon = house
        saon = ""
    sql_house_query = """SELECT h.houseid, h.type, h.
       paon, h.saon, h.postcode, p.street, p.town
                      FROM postcodes AS p
                      INNER JOIN houses AS h ON p.
                          postcode = h.postcode AND p.
                          postcode = %s
                      WHERE h.paon = %s AND h.saon = %s;
                          """
    sql_sales_query = """SELECT *
                      FROM sales
                      WHERE houseid = %s
                      ORDER BY date DESC;"""
    with current_app.app_context():
        cur = current_app.sql_db.cursor()
        cur.execute(sql_house_query, (postcode.upper(),
           paon.upper(),saon.upper(),)) # Gets house
        house: List[Tuple] = cur.fetchone()
        if house != []:
```

```
            cur.execute(sql_sales_query, (house[0],)) #
                gets all sales for the house
            sales = cur.fetchall()
            house_info = {
                "paon": house[2],
                "saon": house[3],
                "postcode": house[4],
                "street": house[5],
                "town": house[6],
                "type": house[1],
                "sales": sales
            }
            house_info["epc_cert"] = epc_cert.GetEPC().
               run(postcode, paon, saon)
            return jsonify(house_info)
        else:
            return abort(404, "No House Found")

@bp.route("/overview")
def overview():
    with current_app.app_context():
        data = current_app.mongo_db.cache.find_one({"
           _id": "OVERVIEW"})
        cur = current_app.sql_db.cursor()
        cur.execute("SELECT data FROM settings WHERE
           name = 'last_aggregated_counties'")
        last_update = cur.fetchone()
        if data is not None:
            if datetime.fromtimestamp(float(last_update
               [0])) < data["last_updated"]:
                return data
            else:
                data = country.get_overview(current_app
                    )
                data["_id"] = "OVERVIEW"
                data["last_updated"] = datetime.now()
                current_app.mongo_db.cache.delete_one({
                   "_id": "OVERVIEW"})
```

```
                current_app.mongo_db.cache.insert_one(
                    data)
        else:
            data = country.get_overview(current_app)
            data["_id"] = "OVERVIEW"
            data["last_updated"] = datetime.now()
            current_app.mongo_db.cache.insert_one(data)
        return data

@bp.route("/value/calc/<string:houseid>")
def value_house(houseid: str):
    task = valuation_task.delay(houseid)
    return jsonify(
            status="ok",
            task_id="/value/get/" + task.id,
        )

@bp.route("/value/get/<string:job_id>")
def get_value(job_id: str):
    if job_id is not None:
        task = analyse_task.AsyncResult(job_id)
        if task.state == "PENDING":
            return {
                "status": task.state
            }
        else:
            valuations = task.wait()
            return {
                "valuations": valuations,
                "status": "ok"
            }

def get_last_updated():
    cur = current_app.sql_db.cursor()
    cur.execute("SELECT * FROM settings WHERE name = '
        last_updated';")
    last_updated = cur.fetchone()
    if last_updated == None:
```

```
            return datetime.fromtimestamp(0)
        else:
            if last_updated[1] is not None:
                return datetime.fromtimestamp(float(
                    last_updated[1]))
            else:
                return datetime.fromtimestamp(0)
```

5.4.5 epc_cert.py

```
from typing import Tuple

import requests
from bs4 import BeautifulSoup
from config import Config
from pymongo import MongoClient


class GetEPC():
    def __init__(self) -> None:
        config = Config()
        self._mongo_db = MongoClient(f"mongodb://{
            config.MONGO_USER}:{config.MONGO_PASSWORD}@{
            config.MONGO_HOST}:27017/?authSource=
            house_data")
        self._mongo = self._mongo_db.house_data

    def _get_houses(self, postcode: str) -> str:
        url_postcode = "+".join(postcode.split("-"))
        resp = requests.get(f"https://find-energy-
            certificate.service.gov.uk/find-a-
            certificate/search-by-postcode?postcode={
            url_postcode}")
        house_soup = BeautifulSoup(resp.content.decode(
            "UTF-8"), 'html.parser')
        house_tags = house_soup.select("#main-content >
            div > div > table > tbody > tr")
```

```
        houses = []
        for house in house_tags:
            properties = house.find(name="th").find("a"
                )
            address = properties.contents[0] \
                .replace("\n", "") \
                .strip() \
                .split(",")[0] \
                .upper()
            cert = properties["href"]
            houses.append((address, cert))
        return houses

    def get_cert(self, path: str):
        resp = requests.get(f"https://find-energy-
            certificate.service.gov.uk{path}")
        cert_soup = BeautifulSoup(resp.content.decode("
            UTF-8"), 'html.parser')
        sqr_m = cert_soup.select_one("#main-content ->
            div -> div.govuk-grid-column-two-thirds.epc-
            domestic-sections -> div.govuk-body.epc-blue-
            bottom.printable-area.epc-box-container -> dl
            -> div:nth-child(2) -> dd") \
            .contents[0] \
            .replace("\n", "") \
            .replace("square metres", "") \
            .strip()
        sqr_m = int(sqr_m)

        energy_rating = cert_soup.select_one("#main-
            content -> div -> div.govuk-grid-column-two-
            thirds.epc-domestic-sections -> div.govuk-
            body.epc-blue-bottom.printable-area.epc-
            rating-graph-section -> svg -> svg.rating-
            current -> text.current-potential-number") \
            .contents[0] \
            .replace("|", "") \
            .strip()
```

```
        energy_rating = int(energy_rating)

        return (sqr_m, energy_rating)

    def run(self, postcode: str, paon: str, saon: str):
        houses = self._get_houses(postcode)
        if saon != "":
            house_id = f"{saon}-{paon}".upper()
        else:
            house_id = paon.upper()
        try:
            house = list(filter(lambda x: x[0] ==
               house_id,houses))[0]
        except IndexError:
            return {
                "sqr_m": None,
                "energy_rating": None,
                "cert_id":  None
            }
        cert_stats = self.get_cert(house[1])
        self._insert_data(cert_stats, house[1],
           postcode, paon, saon)
        return {
                "sqr_m": cert_stats[0],
                "energy_rating": cert_stats[1],
                "cert_id":  house[1]
            }

    def _insert_data(self, cert_stats: Tuple[int,int],
       cert_id: str, postcode: str, paon: str, saon:
       str ):
        epc_doc = self._mongo.epc_certs.find_one({"_id"
           : f"{paon}{saon}{postcode}"})
        doc = {
                "_id": f"{paon}{saon}{postcode}",
                "sqr_m": cert_stats[0],
                "energy_rating": cert_stats[1],
                "cert_id": cert_id
```

```
            }
        if epc_doc is None:
            self._mongo.epc_certs.insert_one(doc)
        elif epc_doc != doc:
            self._mongo.epc_certs.update_one(
                {"_id": doc["_id"]},
                {"$set": {
                    "sqr_m": cert_stats[0],
                    "energy_rating": cert_stats[1],
                    "cert_id": cert_id
                }}
            )



if __name__ == "__main__":
    cert = GetEPC()
    cert.run("CH2 1DE", "16", "")
```

5.4.6 search_area_funcs.py

```
import urllib.parse

def generate_sql_query(query: str, query_filter: str =
    None):
    if query_filter is not None:
        if query_filter in ["postcode", "street", "town
            ", "district", "county", "outcode", "area",
            "sector"]:
            sql_query = f"""SELECT area, area_type
                        FROM areas WHERE substr(area,
                            1, 50)
                        LIKE '{query}%' AND area_type =
                            '{query_filter}'
                        ORDER BY char_length(area)
                        LIMIT 10;"""
        else:
```

```
            return ""
    else:
        sql_query = f"""SELECT area, area_type
                    FROM areas WHERE substr(area, 1, 50)
                    LIKE '{query}%'
                    ORDER BY char_length(area)
                    LIMIT 10;"""
    return sql_query

def sort_results(results):
    SORT_ORDER = {"area": 0, "outcode": 1, "sector": 2,
        "postcode": 3, "town": 4, "county": 5, "
       district": 6, "street": 7}
    return_list = []
    for area in results:
        if area[1] not in ["postcode","outcode","sector
           ","area"]:
            return_list.append((area[0].title(), area
               [1].title()))
        else:
            return_list.append((area[0], area[1].title
               ()))
    return_list.sort(key=lambda val: SORT_ORDER[val[1].
       lower()])
    return return_list
```

5.4.7 country.py

```
from flask import current_app

def get_overview(current_app: current_app):
    query = [
        {
            '$match': {
                'area_type': 'AREA'
            }
        }, {
```

```
            '$project': {
                '3_month_perc': {
                    '$avg': {
                        '$slice': [
                            '$stats.percentage_change.
                               all.perc_change', -3, 3
                        ]
                    }
                }
            }
        }, {
            '$sort': {
                '3_month_perc': -1
            }
        }, {
            '$limit': 5
        }
    ]
    top_5_towns = current_app.mongo_db.cache.aggregate(
       query)
    query[2]["$sort"]["3_month_perc"] = 1
    bottom_5_towns = current_app.mongo_db.cache.
       aggregate(query)
    country_data = current_app.mongo_db.cache.find_one
       ({"_id": "ALLCOUNTRY"})
    return_data = country_data["stats"]
    return_data["timings"] = country_data["timings"]
    return_data["top_five"] = list(top_5_towns)
    return_data["bottom_five"] = list(bottom_5_towns)
    return return_data
```

5.4.8 Dockerfile

```
FROM python:3.10.8s

WORKDIR /app
```

```
COPY ./requirements.txt ./

RUN python3 -m pip install --upgrade pip setuptools
    wheel
RUN python3 -m pip install -r requirements.txt

COPY ./web .

CMD ["gunicorn", "-w 4", "-b 0.0.0.0:8000", "run:app"]
```

5.4.9 requirements.txt

```
amqp==5.1.1
async-timeout==4.0.2
beautifulsoup4==4.11.2
billiard==3.6.4.0
blinker==1.5
celery==5.2.7
certifi==2022.12.7
charset-normalizer==3.0.1
click==8.1.3
click-didyoumean==0.3.0
click-plugins==1.1.1
click-repl==0.2.0
dnspython==2.3.0
Flask==2.2.2
Flask-Cors==3.0.10
gunicorn==20.1.0
idna==3.4
importlib-metadata==6.0.0
itsdangerous==2.1.2
Jinja2==3.1.2
kombu==5.2.4
lxml==4.9.2
MarkupSafe==2.1.2
polars==0.16.1
prompt-toolkit==3.0.36
```

```
psycopg2==2.9.5
pymongo==4.3.3
python-dateutil==2.8.2
python-dotenv==0.21.1
pytz==2022.7.1
redis==4.4.2
requests==2.28.2
sentry-sdk==1.14.0
six==1.16.0
soupsieve==2.3.2.post1
typing_extensions==4.4.0
urllib3==1.26.14
vine==5.0.0
wcwidth==0.2.6
Werkzeug==2.2.2
zipp==3.12.0
```

5.5 Website

5.5.1 app.html

```
<!DOCTYPE html>
<html lang="en">
        <head>
                <meta charset="utf-8" />
                <link rel="icon" href="%sveltekit.
                    assets%/favicon.png" />
                <meta name="viewport" content="width=
                    device-width, initial-scale=1,
                    maximum-scale=1">
                <meta name="description" content="View,
                    analyse and compare property data
                    from all over England and Wales">

                %sveltekit.head%
        </head>
```

```
        <body data-sveltekit-preload-data="hover">
                <div style="display: contents">%
                    sveltekit.body%</div>
        </body>
</html>
```

5.5.2 app.css

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

5.5.3 +routes.svelte

```
<script lang='ts'>
        import QuickStat from "$lib/components/
            QuickStat.svelte";
        import Badge from "$lib/components/Badge.svelte
            ";
        import PieChart from "$lib/components/PieChart.
            svelte";
        import LineGraph from "$lib/components/
            LineGraph.svelte";
        import BarChart from "$lib/components/BarChart.
            svelte";

        export let data;
    console.log(data);
        let quick_stats = data.quick_stats;
        let current_month = new Date(data.average_price
            .dates.slice(-1)[0] );
        let last_updated = new Date(data.last_updated);
        let timings = data.timings;

        function toTitleCase(str: string) {
```

```
        return str.toLowerCase().split(' ').map(
            function (word) {
             return (word.charAt(0).toUpperCase() + word
                .slice(1));
        }).join(' ');
    }

    let perc_change = {
        type: ["D","F","S","T","all"],
        perc: [data.percentage_change.S.perc_change,
            data.percentage_change.F.perc_change,data.
            percentage_change.T.perc_change,data.
            percentage_change.D.perc_change,data.
            percentage_change.all.perc_change],
        date: data.percentage_change.all.date
    };
</script>
<svelte:head>
        <title>House Stats | Home</title>
</svelte:head>

<div class="h-5/6">
    <div class="m-2">
        <div class="items-center align-middle flex flex
            -initial flex-wrap">
            <p class="inline-block text-2xl m-2 align-
                middle">England & Wales, {current_month.
                toLocaleString('default', { month: 'long
                ' })} {current_month.getFullYear()}</p>
            <Badge
                text="Last Updated {last_updated.
                    toLocaleDateString()}"
                colour="green"
                classes="inline-block align-middle"
            />
            <Badge
                text="Execution Time {Number((timings.
                    aggregate).toFixed(3))}s"
```

```
                colour="green"
                classes="inline-block align-middle"
            />
            <Badge
                text="Current Month {current_month.
                    toLocaleDateString()}"
                colour="green"
                classes="inline-block align-middle"
            />
        </div>
    </div>
    <div class="grid lg:grid-cols-4 md:grid-cols-2 grid
        -cols-1 gap-4 h-full m-2">
        <QuickStat
            value={quick_stats.average_price}
            using_percentage={true}
            percentage={quick_stats.average_change}
            title="Average House Price"
            colour="red"
        />
        <QuickStat
            value={quick_stats.sales_qty}
            currency={false}
            using_percentage={true}
            percentage={quick_stats.sales_qty_change}
            title="Sales Volume"
            colour="purple"
        />
        <QuickStat
            value={quick_stats.sales_volume}
            using_percentage={true}
            percentage={quick_stats.sales_volume_change
                }
            title="Price Volume"
            colour="green"
        />
        <QuickStat
            value={quick_stats.expensive_sale}
```

```
    using_percentage={false}
    title="Most Expensive House"
    colour="pink"
/>
        <div class="row-span-2 md:col-span-2 bg
            -white p-4 rounded">
    <p class="text-lg ml-2">Top 5 Areas</p>
    <div class="relative overflow-x-auto">
        <table class="w-full text-sm text-left
            text-gray-500 dark:text-gray-400">
            <thead class="text-xs text-gray-700
                uppercase bg-gray-50 dark:bg-
                gray-700 dark:text-gray-400">
                <tr>
                    <th scope="col" class="px-6
                        py-3">
                        Postcode Areas
                    </th>
                    <th scope="col" class="px-6
                        py-3">
                        3m Moving Average
                            Percentage
                    </th>
                </tr>
            </thead>
            <tbody>
                {#each data.top_five as town}
                    <tr class="bg-white border-
                        b dark:bg-gray-800 dark:
                        border-gray-700">
                        <th scope="row" class="
                            px-6 py-4 font-
                            medium text-gray-900
                            whitespace-nowrap
                            dark:text-white">
                            <a href={"/analyse/
                                area/" + town.
                                _id.split("AREA"
```

```
                                        )[0]}>{town._id.
                                        split("AREA")
                                        [0].toUpperCase
                                        ()}</a>
                                </th>
                                <td class="px-6 py-4">
                                    {Number((town["3
                                        _month_perc"]).
                                        toFixed(3))}%
                                </td>
                            </tr>
                        {/each}
                    </tbody>
                </table>
            </div>
            </div>
            <div class="row-span-2 md:col-span-2 bg
                -white p-4 rounded">
            <p class="text-lg ml-2">Bottom 5 Areas</p>
            <div class="relative overflow-x-auto">
                <table class="w-full text-sm text-left
                    text-gray-500 dark:text-gray-400">
                    <thead class="text-xs text-gray-700
                        uppercase bg-gray-50 dark:bg-
                        gray-700 dark:text-gray-400">
                        <tr>
                            <th scope="col" class="px-6
                                py-3">
                                Postcode Areas
                            </th>
                            <th scope="col" class="px-6
                                py-3">
                                3m Moving Average
                                    Percentage
                            </th>
                        </tr>
                    </thead>
                    <tbody>
```

```
                    {#each data.bottom_five as town
                        }
                        <tr class="bg-white border-
                            b dark:bg-gray-800 dark:
                            border-gray-700">
                            <th scope="row" class="
                                px-6 py-4 font-
                                medium text-gray-900
                                whitespace-nowrap
                                dark:text-white">
                                <a href={"/analyse/
                                    area/" + town.
                                    _id.split("AREA"
                                    )[0]}>{town._id.
                                    split("AREA")
                                    [0].toUpperCase
                                    ()}</a>
                            </th>
                            <td class="px-6 py-4">
                                {Number((town["3
                                    _month_perc"]).
                                    toFixed(3))}%
                            </td>
                        </tr>
                    {/each}
                </tbody>
            </table>
        </div>
            </div>
    <div class="xl:row-span-2">
        <PieChart title="Property Types" labels={
            data.type_proportions.type} data={data.
            type_proportions.count}/>
    </div>
    <div class="md:col-span-2 row-span-2">
        <LineGraph title="Monthly Average Price"
            labels={data.average_price.type} data={
            data.average_price.prices} dates={data.
```

```
                average_price.dates}/>
        </div>
        <div class="md:col-span-2 row-span-2">
            <BarChart title="Percentage Change" labels
                ={perc_change.type} data={perc_change.
                perc} dates={perc_change.date}/>
        </div>
        <div class="md:col-span-2 row-span-2">
            <BarChart title="Sales Volume" labels={data
                .monthly_qty.type} data={data.
                monthly_qty.qty} dates={data.monthly_qty
                .dates}/>
        </div>
        <div class="md:col-span-2 row-span-2">
            <BarChart title="Price Volume" labels={data
                .monthly_volume.type} data={data.
                monthly_volume.volume} dates={data.
                monthly_volume.dates}/>
        </div>
    </div>
</div>
```

5.5.4 +layout.svelte

```
<script>
        import Menu from "$lib/components/menu.svelte";
        import Loader from "$lib/components/Loader.
            svelte";
        import { navigating } from '$app/stores';
        import "../app.css";
        import "flowbite/dist/flowbite.css";
        import * as Sentry from "@sentry/svelte";
        import { BrowserTracing } from "@sentry/tracing
            ";
        import { dev } from '$app/environment';

        let current_year = new Date().getFullYear();
```

```
</script>



<style lang="postcss">
        :global(html) {
                background-color: theme(colors.slate
                    .200);
        }
</style>

<div class="flex flex-col min-h-screen justify-between"
    >
        <div class="inline">
                <Menu></Menu>
        </div>
        {#if $navigating}
                <div class="flex justify-center items-
                    center my-52">
                        <Loader></Loader>
                </div>
        {:else}
                <slot />
        {/if}

        <footer class="bg-green-800 text-sm text-white
            text-center inset-x-0 bottom-0 p-2">
          &copy; {current_year} <a href="https://github
              .com/emtee14">Morgan Thomas</a> | <a href=
              "mailto:contact@housestats.co.uk">
              contact@housestats.co.uk</a> | <a href="/
              tos">Terms of Use</a> | <a href="/pp">
              Privacy Policy</a> <br>
          Contains HM Land Registry data Crown
              copyright and database right 2021. This
              data is licensed under the Open Government
               Licence v3.0.
        </footer>
```

```
</div>
```

5.5.5 +page.ts

```
import { error } from '@sveltejs/kit';
import type { PageLoad } from './$types';

export const load = (async ({ params, fetch }) => {
        const response = await fetch('https://api.
           housestats.co.uk/api/v1/overview');
        const data = await response.json();
        if (response.status == 200) {
                return data
        } else {
                throw error(400, 'Unable to load main
                   dashboard new data is being added');
        }
  }) satisfies PageLoad;
```

5.5.6 +error.svelte

```
<script>
    import { page } from '$app/stores';
</script>

<div class="flex-h-screen-justify-center-items-center">
    <div class="text-center">
        <p class="text-2xl">{$page.error.message}</p>
        <p class="text-lg">This site is still under
           development so some features may be missing<
           /p>
    </div>

</div>
```

5.5.7 analyse/+page.svelte

```
<script lang="ts">
    /** @type {import('./$types').PageData} */
    import Badge from '$lib/components/Badge.svelte';
    import QuickStat from '$lib/components/QuickStat.
       svelte';
    import PieChart from '$lib/components/PieChart.
       svelte';
    import LineGraph from '$lib/components/LineGraph.
       svelte';
    import BarChart from '$lib/components/BarChart.
       svelte';

    let quick_stats, stats, results, timings,
       perc_change;
    let last_updated: Date;
    let current_month: Date;
    let area: string;


    export let data;
    if (data.status == "SUCCESS") {
        quick_stats = data.result.stats.quick_stats;
        stats = data.result.stats;
        results = data.result;
        timings = results.timings;
        last_updated = new Date(results.last_updated);
        current_month = new Date(quick_stats.
           current_month);

        perc_change = {
            type: ["S","F","T","D","all"],
            perc: [stats.percentage_change.S.
               perc_change, stats.percentage_change.F.
               perc_change, stats.percentage_change.T.
               perc_change, stats.percentage_change.D.
               perc_change, stats.percentage_change.all.
```

```
                perc_change],
            date: stats.percentage_change.all.date
        };

        let postcodes = ["POSTCODE","AREA","SECTOR","
            OUTCODE"]
        if (!postcodes.includes(results.area_type)){
            area = toTitleCase(results.area);
        } else {
            area = results.area;
        }
    }

    let title = "Analyse";

    function toTitleCase(str: string) {
        return str.toLowerCase().split(' ').map(
            function (word) {
            return (word.charAt(0).toUpperCase() + word
                .slice(1));
        }).join(' ');
    }
</script>


<svelte:head>
    <title>House Stats | {title}</title>
</svelte:head>


<div class="h-5/6">
    <div class="m-2">
        <div class="items-center align-middle flex flex
            -initial flex-wrap">
            <p class="inline-block text-2xl m-2 align-
                middle">{area} ({toTitleCase(results.
                area_type)}) {current_month.
                toLocaleString('default', { month: 'long
```

```
                ' })} {current_month.getFullYear()}</p>
            <Badge
                text="Last Updated {last_updated.
                    toLocaleDateString()}"
                colour="green"
                classes="inline-block align-middle"
            />
            <Badge
                text="Execution Time {Number((timings.
                    aggregate).toFixed(3))}s"
                colour="green"
                classes="inline-block align-middle"
            />
            <Badge
                text="Data Fetch Time {Number((timings.
                    loader).toFixed(3))}s"
                colour="green"
                classes="inline-block align-middle"
            />
            <Badge
                text="Current Month {current_month.
                    toLocaleDateString()}"
                colour="green"
                classes="inline-block align-middle"
            />
        </div>
    </div>
    <div class="grid lg:grid-cols-4 md:grid-cols-2 grid
        -cols-1 gap-4 h-full m-2">
        <QuickStat
            value={quick_stats.average_price}
            using_percentage={true}
            percentage={quick_stats.average_change}
            title="Average House Price"
            colour="red"
        />
        <QuickStat
            value={quick_stats.sales_qty}
```

```
        currency={false}
        using_percentage={true}
        percentage={quick_stats.sales_qty_change}
        title="Sales Volume"
        colour="purple"
    />
    <QuickStat
        value={quick_stats.sales_volume}
        using_percentage={true}
        percentage={quick_stats.sales_volume_change
            }
        title="Sales Price Volume"
        colour="green"
    />
    <QuickStat
        value={quick_stats.expensive_sale}
        using_percentage={false}
        title="Most Expensive House"
        colour="pink"
    />
    <div class="xl:row-span-2">
        <PieChart title="Property Types" labels={
            stats.type_proportions.type} data={stats
            .type_proportions.count}/>
    </div>
    <div class="md:col-span-2 row-span-2">
        <LineGraph title="Monthly Average Price"
            labels={stats.average_price.type} data={
            stats.average_price.prices} dates={stats
            .average_price.dates}/>
    </div>
    <div class=" md:col-span-2 row-span-2">
        <BarChart title="Percentage Change" labels
            ={perc_change.type} data={perc_change.
            perc} dates={perc_change.date}/>
    </div>
    <div class=" md:col-span-2 row-span-2">
```

```
            <BarChart title="Sales Volume" labels={
                stats.monthly_qty.type} data={stats.
                monthly_qty.qty} dates={stats.
                monthly_qty.dates}/>
        </div>
        <div class="md:col-span-2 row-span-2">
            <BarChart title="Price Volume" labels={
                stats.monthly_volume.type} data={stats.
                monthly_volume.volume} dates={stats.
                monthly_volume.dates}/>
        </div>
    </div>
</div>
```

5.5.8 +page.svelte

```
import { error } from '@sveltejs/kit';

/** @type {import('./$types').PageLoad} */
export async function load({ fetch, params }) {
        const sleep = (ms: number) => new Promise((r)
           => setTimeout(r, ms));
        let area: string = params.area;
        let area_type: string = params.area_type;

        let stats;
        let counter = 0
        const response = await fetch('https://api.
           housestats.co.uk/api/v1/analyse/' +
           area_type + '/' + area);
        const data = await response.json();
        if (data.status == "ok") {
                while (true) {
                        const res_resp = await fetch(
                           data.result);
            stats = await res_resp.json();
```

```
            if (stats.status == "SUCCESS" || stats.
               status == "COMPLETED") {
                                break
                        } else if (counter > 60*4) {
                                throw error(500, '
                                   Connection Timed Out
                                   ')
                        }
                        counter++;
                        await sleep(250)
                 }
        } else {
                 throw error(500, 'An Error Has Occured
                    ');
        }
        return stats
}
```

5.5.9 compoents/Badge.svelte

```
<script lang="ts">
    export let text: string;
    export let colour: string;
    export let size: string = "xs";
    export let classes: string;
</script>

<span class="bg-{colour}-100 text-{colour}-800 text-{
   size} font-medium m-1 px-2.5 py-0.5 rounded {classes
   }">{text}</span>
```

5.5.10 BarChart.svelte

```
<script lang="ts">
    import Chart from 'chart.js/auto';
    import zoomPlugin from 'chartjs-plugin-zoom';
```

```
import { onMount } from 'svelte';
import 'chartjs-adapter-date-fns';
import {enGB} from 'date-fns/locale';

Chart.register(zoomPlugin);
let graph_id = Math.random().toString(36).substr(2,
    5)

let house_types: { [key: string]: string } = {
    'D': "Detatched",
    'S': "Semi-Detatched",
    'T': "Terrace",
    'F': "Flat",
    'O': 'Other',
    "all": "All"
};
let colours = [
    '#dc2626',
    '#9333ea',
    '#16a34a',
    '#db2777'
]

export let labels: Array<string>;
export let title: string;
export let data: Array<Array<BigInt>>;
export let dates: Array<string>;

let data_length = data.length;
let datasets = [];
for (let i = 0; i < data_length; i++){
    datasets.push({
        label: house_types[labels[i]],
        data: data[i],
        tension: 0.1,
        backgroundColor: colours[i],
        fill: false,
    });
```

```
    }
    const chart_data = {
        labels: dates.map((x) => {return new Date(x)}),
        datasets: datasets
    };
    const config = {
        responsive: true,
        type: 'bar',
        data: chart_data,
        options: {
            scales: {
                x: {
                    stacked: true,
                    type: 'time',
                    time: {
                        round: 'month',
                        minUnit: 'month'
                    },
                    adapters: {
                        date: {
                            locale: enGB
                        }
                    }
                },
                y: {
                    stacked: true
                }
            },
            plugins: {
                zoom: {
                    pan: {
                        enabled: true
                    },
                    zoom: {
                    wheel: {
                        enabled: true,
                    },
                    pinch: {
```

```
                        enabled: true
                    },
                    mode: 'xy',
                    }
                },
                title: {
                    display: true,
                    text: title
                }
            }
        }
    };
    let line_chart: Chart;
    onMount(()=> {
        let ctx = document.getElementById(graph_id);
        if (ctx != null){
            line_chart = new Chart(ctx, config);
        }
    })
</script>

<canvas id={graph_id}>
</canvas>
<button on:click={line_chart.resetZoom('default')} type
   ="button" class="text-white bg-green-700 hover:bg-
   green-800 focus:outline-none focus:ring-4 focus:ring
   -green-300 font-medium rounded-full text-sm px-5 py
   -2.5 text-center mr-2 mb-2 dark:bg-green-600 dark:
   hover:bg-green-700 dark:focus:ring-green-800">Reset
   Zoom</button>
```

5.5.11 LineGraph.svelte

```
<script lang="ts">
    import Chart from 'chart.js/auto';
    import zoomPlugin from 'chartjs-plugin-zoom';
    import { onMount } from 'svelte';
```

```
    import 'chartjs-adapter-date-fns';
    import {enGB} from 'date-fns/locale';

    Chart.register(zoomPlugin);
    let graph_id = Math.random().toString(36).substr(2,
        5)

    let house_types: { [key: string]: string } = {
        'D': "Detatched",
        'S': "Semi-Detatched",
        'T': "Terrace",
        'F': "Flat",
        'O': 'Other',
        "all": "All"
    };
    let colours = [
        '#dc2626',
        '#9333ea',
        '#16a34a',
        '#db2777'
    ]

    export let labels: Array<string>;
    export let title: string;
    export let data: Array<Array<BigInt>>;
    export let dates: Array<string>;

    let data_length = data.length;
    let datasets = [];
    for (let i = 0; i < data_length; i++){
        let label: string;
        if (labels[i].length < 4){
            label = house_types[labels[i]];
        } else {
            label = (new Date(labels[i])).
               toLocaleDateString();
        }
        datasets.push({
```

```
            label: label,
            data: data[i],
            tension: 0.1,
            borderColor: colours[i],
            fill: false,
        });
    }
    const chart_data = {
        labels: dates.map((x) => {return new Date(x)}),
        datasets: datasets
    };
    const config = {
        responsive: true,
        type: 'line',
        data: chart_data,
        options: {
            scales: {
                x: {
                    type: 'time',
                    time: {
                        round: 'month',
                        minUnit: 'month'
                    },
                    adapters: {
                        date: {
                            locale: enGB
                        }
                    }
                }
            },
            plugins: {
                zoom: {
                    pan: {
                        enabled: true
                    },
                    zoom: {
                    wheel: {
                        enabled: true,
```

```
                    },
                    pinch: {
                        enabled: true
                    },
                    mode: 'xy',
                    }
                },
                title: {
                    display: true,
                    text: title
                }
            },
            elements: {
                point:{
                    radius: 0
                }
            }
        }
    };
    let line_chart: Chart;
    onMount(()=> {
        let ctx = document.getElementById(graph_id);
        if (ctx != null){
            line_chart = new Chart(ctx, config);
        }
    })
</script>

<canvas id={graph_id}>
</canvas>
<button on:click={line_chart.resetZoom('default')} type
    ="button" class="text-white bg-green-700 hover:bg-
    green-800 focus:outline-none focus:ring-4 focus:ring
    -green-300 font-medium rounded-full text-sm px-5 py
    -2.5 text-center mr-2 mb-2 dark:bg-green-600 dark:
    hover:bg-green-700 dark:focus:ring-green-800">Reset
    Zoom</button>
```

5.5.12 Loader.svelte

```
<style>
        .loader {
                border: 16px solid #f3f3f3;
                border-radius: 50%;
                border-top: 16px solid #046C4E;
                width: 120px;
                height: 120px;
                -webkit-animation: spin 2s linear
                   infinite; /* Safari */
                animation: spin 2s linear infinite;
        }
        @-webkit-keyframes spin {
                0% { -webkit-transform: rotate(0deg); }
                100% { -webkit-transform: rotate(360deg
                   ); }
        }
        @keyframes spin {
                0% { transform: rotate(0deg); }
                100% { transform: rotate(360deg); }
        }
</style>

<div class="loader"></div>
```

5.5.13 menu.svelte

```
<script lang="ts">
    import { page } from '$app/stores';
    import { Navbar, NavBrand, NavLi, NavUl,
       NavHamburger, Button, Input } from 'flowbite-
       svelte'

    $: current_page = $page.url.pathname;
    let suggestions: Array<Array<[string,string]>> =
       [];
```

```
    let isFocused = false;
    const onSearchFocus =()=> isFocused=true;
    const onSearchBlur = () => setTimeout(() => {
       isFocused=false; }, 250);

    let results: boolean = false;

    async function autoComplete(search_value: string){
        if (search_value) {
            const response = await fetch('https://api.
               housestats.co.uk/api/v1/search/' +
               search_value);
            const data = await response.json();
            if (data.found == true){
                suggestions = data.results;
                results = true;
            } else {
                results = false;
            }
        } else {
            results = false;
        }
    }
</script>

<Navbar let:toggle let:hidden>
    <NavBrand href="/">
        <img src="/logo.svg" class="h-15 mr-3 sm:h-9"
           alt="House Stats Logo" />
        <span class="self-center whitespace-nowrap text
           -xl font-semibold dark:text-white">
            House Stats
        </span>
    </NavBrand>
    <div class="flex md:order-2">
        <Button color="none" data-collapse-toggle="
           mobile-menu-3" aria-controls="mobile-menu-3"
```

```
        aria-expanded="false" class="md:hidden text
      -gray-500 dark:text-gray-400 hover:bg-gray
      -100 dark:hover:bg-gray-700 focus:outline-
      none focus:ring-4 focus:ring-gray-200 dark:
      focus:ring-gray-700 rounded-lg text-sm p-2.5
      ml-1">
        <svg xmlns="http://www.w3.org/2000/svg"
            fill="none" viewBox="0 0 24 24" stroke-
            width="1.5" stroke="currentColor" class=
            "w-6 h-6 dark:text-white"><path stroke-
            linecap="round" stroke-linejoin="round"
            d="M21 21l-5.197-5.197m0 0A7.5 7.5 0 0
            105.196 5.196a7.5 7.5 0 0010.607 10.607z
            " /></svg>
    </Button>
    <div class="hidden relative md:block">
        <div class="flex absolute inset-y-0 left-0
            items-center pl-3 pointer-events-none">
            <svg xmlns="http://www.w3.org/2000/svg"
                fill="none" viewBox="0 0 24 24"
                stroke-width="1.5" stroke="
                currentColor" class="w-6 h-6 dark:
                text-white"><path stroke-linecap="
                round" stroke-linejoin="round" d="
                M21 21l-5.197-5.197m0 0A7.5 7.5 0 0
                105.196 5.196a7.5 7.5 0 0010.607
                10.607z" /></svg>
        </div>
        <input
            type="text"
            id="search-navbar"
            autocomplete="off"
            class="block w-full p-2 pl-10 text-sm
                text-gray-900 border border-gray-300
                rounded-lg bg-gray-50 focus:ring-
                blue-500 focus:border-blue-500 dark:
                bg-gray-700 dark:border-gray-600
                dark:placeholder-gray-400 dark:text-
```

```
                white dark:focus:ring-blue-500 dark:
                focus:border-blue-500"
            placeholder="Search Areas..."
            on:input={e => autoComplete(e.target.
                value)}
            on:focus={onSearchFocus}
            on:blur={onSearchBlur}>
        <div class="absolute">
            {#if isFocused == true}
                <div class="relative bg-white w-96"
                    >
                    {#if results == true}
                        {#each suggestions as
                            suggestion}
                            <a href="/analyse/{
                                suggestion[1]}/{
                                suggestion[0]}"
                                class="pl-2 hover:bg
                                -gray-300 block">
                                {suggestion[0]}
                                <span class="bg-
                                    blue-100 text-
                                    blue-800 text-xs
                                    font-medium mr
                                    -2 px-2.5 py-0.5
                                    rounded">{
                                    suggestion[1]}</
                                    span>
                            </a>
                        {/each}
                    {:else}
                        <p class="ml-2">No Results<
                            /p>
                    {/if}
                </div>
            {/if}
        </div>
    </div>
```

```
        <NavHamburger on:click={toggle} />
    </div>
    <NavUl {hidden}>
        <NavLi href="/" active={current_page === "/" ?
            true : false}>Dashboard</NavLi>
        <NavLi href="/counties" active={current_page
           === "/counties" ? true : false}>Overview
           Counties</NavLi>
        <NavLi href="/valuation" active={current_page
           === "/valuation" ? true : false}>House
           Lookup</NavLi>
        <NavLi href="/reports" active={current_page ===
            "/reports" ? true : false}>Report
           Generator</NavLi>
    </NavUl>
</Navbar>
```

5.5.14 PieChart.svelte

```
        <script lang="ts">
    import Chart from 'chart.js/auto';
    import { onMount } from 'svelte';

    let house_types: { [key: string]: string } = {
        'D': "Detatched",
        'F': "Flat",
        'S': "Semi-Detatched",
        'T': "Terrace",
        'O': 'Other'
    };

    export let labels: Array<string>;
    export let title: string;
    export let data: Array<BigInteger>;

    const chart_data = {
```

```
        labels: labels.map((x) => {return house_types[x
            ]}),
        datasets: [{
            label: title,
            data: data,
            backgroundColor: [
            '#db2777',
            '#dc2626',
            '#16a34a',
            '#9333ea',

            ],
            hoverOffset: 4
        }],
    };
    const config = {
        type: 'pie',
        data: chart_data,
        options: {
            plugins: {
                title: {
                    display: true,
                    text: title
                }
            }
        }
    };
    onMount(()=> {
        let ctx = document.getElementById('piechart');
        new Chart(ctx, config);
    })
</script>

<canvas id="piechart">

</canvas>
```

5.5.15 QuickStat.svelte

```
<script lang="ts">
    let formatter = Intl.NumberFormat('en',
        {
            notation: 'compact',
            unitDisplay: 'long',
            style: 'currency',
            currency: 'GBP'
        });
    export let value: GLfloat;
    export let using_percentage: boolean = false;
    export let percentage: GLfloat = 0;
    export let title: string;
    export let currency: boolean = true;
    export let colour: string;
</script>

<div class="bg-{colour}-600 text-white rounded p-2">
    {#if currency}
        <p class="text-2xl font-extralight inline">{
            formatter.format(value)}</p>
    {:else}
        <p class="text-2xl font-extralight inline">{
            value.toLocaleString()}</p>
    {/if}
    {#if using_percentage }
    <p class="inline font-thin align-top">{percentage <
        0 ? '>' : '<'} {Math.abs(percentage)}%</p>
    {/if}
    <p class="text-sm font-extralight">{title}</p>
</div>
```

5.5.16 SearchBar.svelte

```
<script lang='ts'>
    export let filter: string;
```

```
    let suggestions: Array<Array<[string,string]>> =
       [];

    let isFocused = false;
    const onFocus =()=>isFocused=true;
    const onBlur = () => setTimeout(() => { isFocused=
       false; }, 250);

    let results: boolean = false;

    async function autoComplete(search_value: string){
        if (search_value) {
            const response = await fetch('https://api.
               housestats.co.uk/api/v1/search/' +
               search_value + '?filter=' + filter);
            const data = await response.json();
            if (data.found == true){
                suggestions = data.results;
                results = true;
            } else {
                results = false;
            }
        } else {
            results = false;
        }
    }
    function titleCase(str: string) {
        return str.toLowerCase().split(' ').map(
           function(word: string) {
            return (word.charAt(0).toUpperCase() + word
               .slice(1));
        }).join(' ');
    }
</script>
```

```
<div class="flex flex-col">
    <input type="text"
    name="area"
    class="p-3 rounded"
    autocomplete="off"
    placeholder={"Search " + titleCase(filter)}
    on:input={e => autoComplete(e.target.value)}
    on:focus={onFocus}
    on:blur={onBlur}>
    {#if isFocused == true}
        <div class="bg-white absolute mt-12 w-52 block"
            >
            {#if results == true}
                {#each suggestions as suggestion}
                    <a href={'/valuation/' + suggestion
                        [0]} class="pl-2 hover:bg-gray
                        -300 block">
                        {suggestion[0]}
                        <span class="bg-blue-100 text-
                            blue-800 text-xs font-medium
                            mr-2 px-2.5 py-0.5 rounded"
                            >{suggestion[1]}</span>
                    </a>
                {/each}
            {:else}
                <p class="ml-2">No Results</p>
            {/if}
        </div>
    {/if}
</div>
```

5.5.17 valuation/+page.svelte

```
<script lang='ts'>
        import SearchBar from '$lib/components/
            SearchBar.svelte';
</script>
```

```
<svelte:head>
        <title>House Stats | House Lookup</title>
</svelte:head>
<div class="flex justify-center items-center max-w-
    screen">
        <div class="-mt-28">
                <p class="text-3xl">Enter Postcode
                    Below</p>
                <SearchBar filter="postcode"/>
        </div>
</div>
```

5.5.18 valuation/[postcode]/+page.svelte

```
<script lang='ts'>
        import type { PageData } from './$types';

        export let data: PageData;
</script>

<div class="md:mx-24 my-8">
        <a href="/valuation" class=" bg-white p-2
            rounded text-blue-600">&lt; Back</a>
        <div class="relative overflow-x-auto shadow-md
            sm:rounded-lg">
                <table class="w-full text-sm text-left
                    text-gray-500 dark:text-gray-400">
                        <caption class="p-5 text-xl
                            font-semibold text-left text
                            -gray-900 bg-white dark:text
                            -white dark:bg-gray-800">
                                {data.postcode.
                                    toUpperCase()}
                                <p class="mt-1 text-sm
                                    font-normal text-
                                    gray-500 dark:text-
                                    gray-400">All of the
```

```
                houses with the
                postcode {data.
                postcode.toUpperCase
                ()}.</p>
        </caption>
        <thead class="text-xs text-gray
            -700 uppercase bg-gray-50
            dark:bg-gray-700 dark:text-
            gray-400">
                <tr>
                        <th scope="col"
                            class="px-6
                            py-3"> SAON
                            , PAON </th>
                        <th scope="col"
                            class="px-6
                            py-3">
                            Street </th>
                        <th scope="col"
                            class="px-6
                            py-3"> Town
                            </th>
                        <th scope="col"
                            class="px-6
                            py-3">
                            County </th>
                        <th scope="col"
                            class="px-6
                            py-3">
                            Postcode </
                            th>
                        <th scope="col"
                            class="px-6
                            py-3">
                            Action </th>
                </tr>
        </thead>
        <tbody>
```

```
                {#each data.data as
                    house}
                        <tr class="bg-
                            white border
                            -b dark:bg-
                            gray-900
                            dark:border-
                            gray-700">
                                <th
                                    scope
                                    ="
                                    row"

                                    class
                                    ="px
                                    -6
                                    py-4
                                    font
                                    -
                                    medium
                                    text
                                    -
                                    gray
                                    -900
                                    whitespace
                                    -
                                    nowrap
                                    dark
                                    :
                                    text
                                    -
                                    white
                                    ">
```

```
                                                                {
                                                                    house
                                                                    [2]}{
                                                                    house
                                                                    [1]

                                                                    !=

                                                                    ' '

                                                                    &&

                                                                    house
                                                                    [2]

                                                                    !=

                                                                    ' '

                                                                    ?

                                                                    ' , '

                                                                    :

                                                                    ' '}

                                                                    {
                                                                    house
                                                                    [1]}
                                                  </th>
                                                  <td
                                                      class
                                                      =" px
                                                      -6
                                                      py-4
                                                      ">
```

```
                                {
                                    house
                                    [4]}
                            </td>
                            <td
                                class
                                =”px
                                -6-
                                py-4
                                ”>
                                {
                                    house
                                    [5]}
                            </td>
                            <td
                                class
                                =”px
                                -6-
                                py-4
                                ”>
                                {
                                    house
                                    [6]}
                            </td>
                            <td
                                class
                                =”px
                                -6-
                                py-4
                                ”>
                                {
                                    house
                                    [3]}
                            </td>
```

```
                                    <td
                                        class
                                        =" px
                                        -6
                                        py-4
                                        ">
                                                <
                                                    a

                                                    href
                                                    =
                                                    "
                                                    /
                                                    valuation
                                                    /{
                                                    data
                                                    .
                                                    postcode
                                                    .
                                                    toUpperCase
                                                    ()
                                                    }/{
                                                    house
                                                    [1]}/{
                                                    house
                                                    [2]}
                                                    "

                                                    class
                                                    =
                                                    "
                                                    font
                                                    -
                                                    medium
                                                    -
                                                    text
                                                    -
                                                    blue
```

```
                                                                    -600
                                                                    -
                                                                    dark
                                                                    :
                                                                    text
                                                                    -
                                                                    blue
                                                                    -500
                                                                    -
                                                                    hover
                                                                    :
                                                                    underline
                                                                    "
                                                                    >
                                                                    View
                                                                    <
                                                                    /
                                                                    a
                                                                    >
                                                            </td>
                                                        </tr>
                                                    {/each}
                                                </tbody>
                                            </table>
                                        </div>
                                    </div>
```

5.5.19 valuation/[postcode]/+page.ts

```
import { error } from '@sveltejs/kit';
import type { PageLoad } from './$types';

export const load = (async ({ params }) => {
  let postcode: string = params.postcode;
        const response = await fetch('https://api.
           housestats.co.uk/api/v1/find/' + postcode.
```

```
            toUpperCase());
        const data = await response.json();
        if (response.status == 200) {
                return {
                        data: data.results,
                        postcode: postcode
                }
        } else {
                throw error(404, 'No Postcode Found');
        }
}) satisfies PageLoad;
```