Final Year Project Report

Full Unit – Interim Report

Live Underground TFL Train Tracker

Vara Uswaransigul

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Matthew Hague

Department of Computer Science
Royal Holloway, University of London

December 05, 2021

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 9548

Student Name: Vara Uswaransigul

Date of Submission: 04/12/2021

Signature: Vara Uswaransigul

Table of Contents

Chapter 1: **Introduction**

1.1 Context and Motivations

When waiting for a train at a station, the only information available for that given train is the destination and predicted time for arrival to the station. This presents a problem to the user as they lack information such as the next station the train is heading towards, current location and potential disruptions. This information can be helpful to a user as it puts the predicted arrival time of the train into context, which may help with frustration with long wait times or disruptions. Additionally, some stations serve trains that go to different destinations and therefore the train that a user might want to catch may only be shown for a brief period as other train times are shown in a scrolling manner. This can annoy the user as they must wait for their train to show up to see the predicted time or they may miss the cycle when it is shown and must wait again.

The motivation for this project is to show the user additional information about the train they wish to board. This will be done through a map on a website which has train icons shown with their live positions. Clicking on a train icon will show the additional information the user needs, such as next station and predicted time for arrival.

1.2 Aims and Objectives

The final version of the project aims to be a website that displays a map. This map will have all the London Underground tube lines drawn over it in their respective colours. All the stations serving the underground will be shown as icons on the map. Hovering over a station icon will display its name. All trains currently running on the underground will be shown as icons on the map and will be shown moving along the line they serve. Clicking on a train icon will display the next station it is heading towards and the predicted time to arrival to the that station.

To achieve the stated aims of the final project. A web framework will be used to support the development of the website. To display the map, a library will be used. This library will support features such as drawing over the map and displaying information when hovering over icons. Then to display the trains live location, the TFL unified API will be used to query for that data.

1.3 Purpose of this report

The purpose of this report is to show the current progress I am at with the project. I will be going over background theory I have learnt prior to starting the project. I will also explain the theory used to create proof of concept programs. Then I will detail my plan for the second term on finishing up the project.

Chapter 2: **Background Theory Reports**

2.1 Basic web page development in HTML5: html, CSS, JavaScript

2.1.1 How html, CSS, and JavaScript work together

Webpage development uses HTML5. This is made up of 3 kinds of code, HTML, CSS, and JavaScript [1]. HTML is a markup language used by browsers to structure the words, images, and videos a page will display. CSS is a style sheet language used to describe the presentation of a document written in a markup language, in this case being HTML. JavaScript is a scripting language that allows implementation of complex features on web pages. These 3 languages work together to allow web pages to be dynamic and interactive.

All 3 languages work together with HTML being the main file. Theoretically the HTML file doesn't need to work with any external files as the CSS and JS functionality can be put into the file. However, this is bad practice as it leads to non-cohesive code. Therefore, the HTML file will contain a link to the CSS file that will be used to style its elements. The CSS file will then reference the objects in the HTML file and the desired styles upon them. In the HTML file JavaScript code can be placed in <script> tags or be referenced from external .js files.

Using all 3 languages together allows for dynamic and interactive web pages. This is mainly done through JavaScript. JavaScript can interact with the webpage via the DOM (document object model) to query the page state and modify. An example would be waiting for a click event on a certain object and modifying the CSS. HTML and CSS are modifiable by JavaScript allowing for dynamic web pages.

2.1.2 Example - Hello world with button

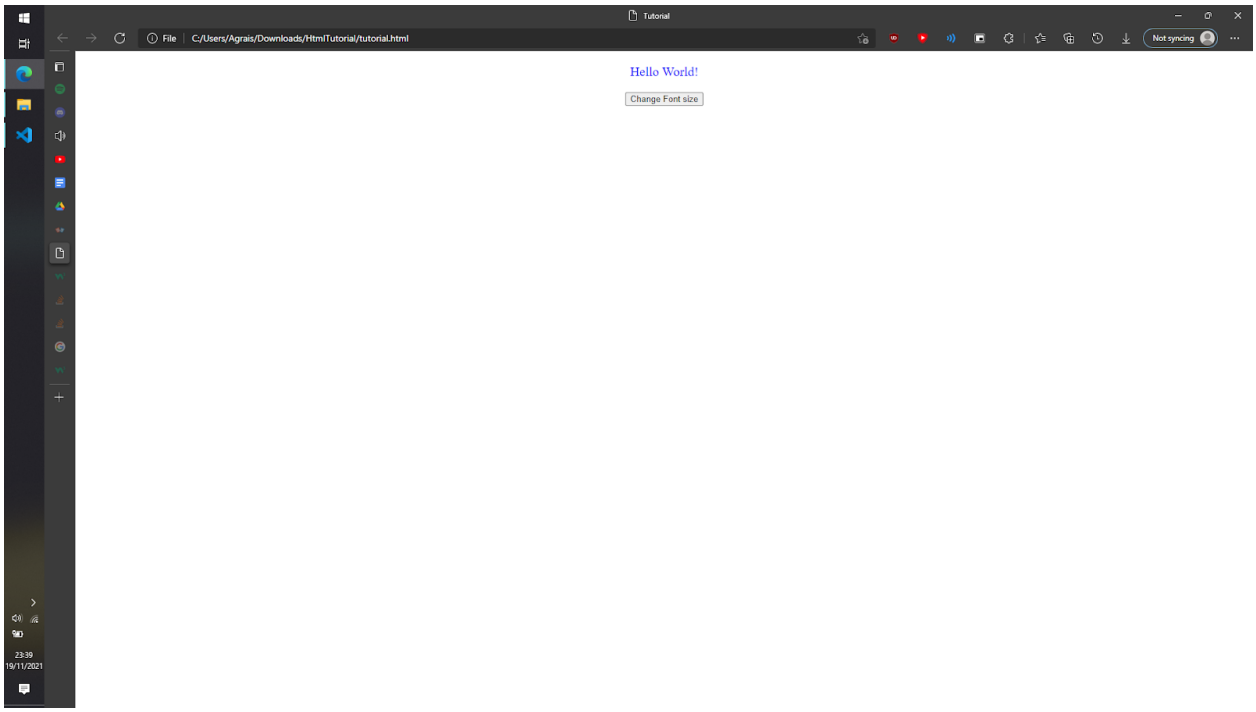


Figure 1.

This example website displays a ‘Hello World!’ text with a button that increases the font size of the text. The figure below shows the text increasing in size after clicking the button a few times. This example demonstrates how JavaScript code can be used to modify CSS styling to create an interactive website.

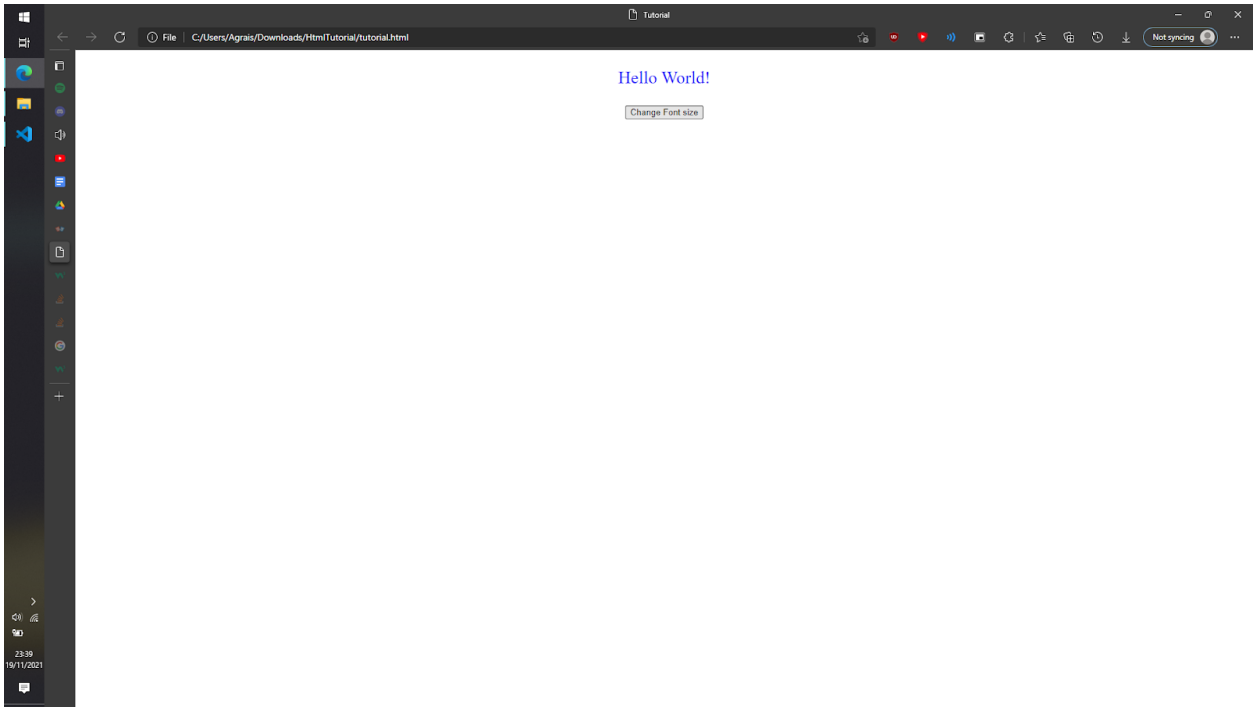


Figure 2.

Example.html

```
<!DOCTYPE html>
<html>
    <head>
        <link rel="stylesheet" href="styles.css">
        <title>Tutorial</title>
    </head>
    <body>
        <p id="text">Hello World!</p>
        <button id="button" type="button">Change Font size</button>
        <script src="tutorial.js"></script>
    </body>
</html>
```

The html file works based on tags. Each tag has a start and end, for example the paragraph tag Text . The first tag, <!DOCTYPE> tells the browser what version of html is being used. The <html> tag is the root of the file and is the container of all other html elements.

The <head> tag is a container for metadata and is placed between the <html> and <body> tags. In this example I used the <link> tag to define a link to an external style sheet. Then I used a <title> tag for the title, which is shown in the pages title bar, this element is required for all html files.

The <body> tag contains all the contents of a html document. The element defines a paragraph where text can be added. Additionally, I added an id to this element so that it can be identified in CSS and JavaScript for modification. The id attribute is global and can be added to almost all tags. The <button> tag defines a clickable button, with the text of the button between tags. Finally, there is a <script> tag. A <script> tag can be used 2 ways, one way is to place JavaScript code between the tags, the other is to link towards an external script via the src attribute, which is what is done in

the example above. The <script> tag is placed at the end of the code as it manipulates DOM (document object model) and if placed before html elements, they will not be loaded in and return an error when attempted to use/modify.

Style.css

```
body {
    text-align:center;
}

#text {
    color: blue;
    font-size: 20px;
}
```

The CSS file works by setting style rules for specific DOM elements. The syntax for styling a DOM element starts with the element then the properties to be styled inside curly brackets. In the example file above, the body tag is styled such that all text contained in the tag is horizontally centred.

The text element to be styled is referenced via the id attribute. The # character signifies that the element to be styled is being referenced via id. Then I apply a blue colour and 20px font size to text inside the element.

Not shown here but a group of elements can be referenced by a class attribute, this is done via a "." character.

Example.js

```
document.getElementById('button').addEventListener('click', function() {
    let text = document.getElementById('text');
    let newFontSize = parseFloat(getComputedStyle(text).fontSize) + 1;
    text.style.fontSize = newFontSize + 'px';
});
```

The JavaScript file makes the website interactive by increasing the font size of the text by clicking the button. This is achieved through JavaScript code, first the DOM element for the button is retrieved by calling document.getElementById.

Then to make the button responsive to a click, an eventListener is added to the button which runs a function in response to a click. Here this function is defined as an anonymous function as it doesn't require a name and it is only used once in this eventListener. In the function the DOM element for the text is retrieved in a similar way as the button, then the font size is computed and incremented by 1 into a variable. Then the new font size is set to the text. The function getComputedStyle has to be used instead of simply accessing the property as the font size is defined in the CSS style file and therefore accessing the property directly will return " ". [2]

2.2 Advanced technologies: Django, Mapbox, Leaflet

2.2.1 What is a web framework

A web framework is a library that allows us to write web applications or services without having to handle low-level details such as protocols, sockets, or process/thread management. They offer a standard way to build and structure an application without too much work. This allows us to focus on implementing features rather than configuration details.

Common tasks that web frameworks can handle include:

- URL routing
- HTML, XML, JSON, and other output format templating
- Database manipulation
- Session storage and retrieval

2.2.2 Django

Django is a python-based web framework that follows the model-template-views architectural pattern.

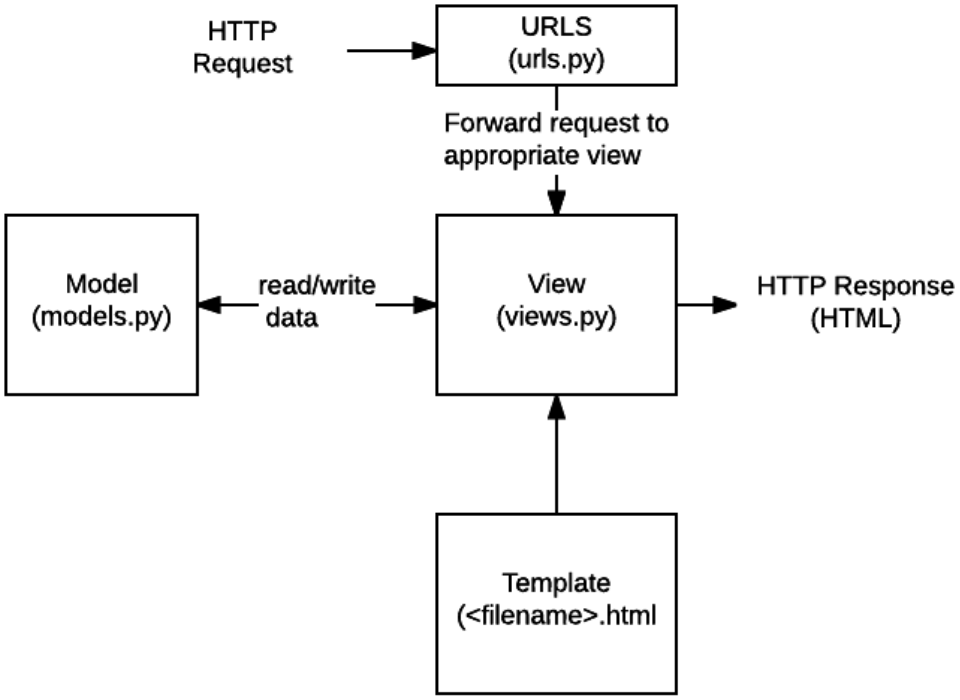


Figure 3. Diagram of Django application flow [3]

The image above shows how Django handles a HTTP request.

URLS: Contains a URL mapper that redirects a HTTP request to the corresponding view function. The URL mapper can parse for patterns in the URL and pass these as data to the view.

View: Contains a handler function for each valid HTTP request. Returns a HTTP response. Access data required for a request via the model. Gets the format for the HTTP response via the template. Server-side scripting can be done in the functions to create an interactive website.

Model: Models are python objects. They provide the structure of an application's data and provide mechanisms to manage and query records in a database.

Template: A template is a text file defining the structure or layout of a file (such as an HTML page), with placeholders used to represent actual content. A view can dynamically create an HTML page using an HTML template, populating it with data from a model.

As seen in the explanation above, the model-template-views pattern allows for separation of logic and code reuse. For example, a single HTML template can be used by 2 different view functions populating the template with different data.

Django follows the "Batteries included" philosophy and provides almost everything developers might want to do "out of the box". Because everything you need is part of the one "product", it all works seamlessly together, following consistent design principles. [3]

Another feature of Django is that it has over 4000 packages that can be used to build and run an application. This means it is likely that a specific configuration required already exists and can be used instead of building from scratch. [4]

2.2.3 Flask

Flask is a python micro web framework. It is defined as a microframework because it has no requirements for use of specific tools or libraries. It has no database abstraction layer compared to Django which has the model. Additionally, it doesn't support form validation or any other common tasks where other frameworks would have library support. [4]

This is what makes Flask different from Django as it is very flexible, with its main approach stated in its documentation “Flask aims to keep the core simple but extensible. Flask won’t make many decisions for you, such as what database to use. Those decisions that it does make, such as what templating engine to use, are easy to change. Everything else is up to you, so that Flask can be everything you need and nothing you don’t”. This flexibility comes at a cost to the developer as they must have prior knowledge on web development to know how to build up the framework themselves with their required extensions. However, the benefits come at no bloat to the framework as everything included directly supports the application. [5]

This means Flask is the opposite in terms of Django’s Batteries-included approach, where only a simple core is included. Tools and libraries needed are added onto the core as extensions.

2.2.4 Opinionated and Unopinionated frameworks

Opinionated frameworks are those with opinions about the "right way" to handle any task. They often support rapid development in a particular domain because the right way to do anything is usually well-understood and well-documented. However, they can be less flexible at solving problems outside their main domain and tend to offer fewer choices for what components and approaches they can use. [3]

Unopinionated frameworks have fewer restrictions on the best way to glue components together to achieve a goal, or even what components should be used. They make it easier for developers to use the most suitable tools to complete a particular task, albeit at the cost that you need to find those components yourself.

Django is a somewhat opinionated framework. This means it provides a set of components to handle most web development tasks and preferred ways to use them. However due to its decoupled architecture we can still pick and choose different options or add new ones ourselves.

On the other hand, Flask is an unopinionated framework, following its definition as a microframework, where it is very flexible in allowing a developer to build any type of website. However, it comes at the cost of having to find components and the knowledge of how to make multiple components work together.

2.2.5 Why I chose Django over Flask

Comparing the 2 frameworks, they both have different approaches to web development. Django is a powerful framework providing features such as ORM (object relational mapping) and uses data models out of the box. Additionally using a preferred architecture of MVT (model view template)

which makes structuring code easier to apply the DRY (don't repeat yourself) principle. This contrasts with Flask which only requires the use of a default template engine which can be easily changed. It doesn't have any of the mentioned Django features out of the box. Allowing for more flexibility and modularity but requiring previous knowledge in web development. [4]

One of the main reasons in choosing Django is due to this project being the first time I have done a web project in python. Due to having to learn everything about web development, choosing Django made the most sense as it is more beginner friendly than Flask. This is because Flask requires knowing what components are needed to build an application as out of the box the core is very simple and requires extensions to make a basic application. As a beginner having to find and know how to put together various components is too steep a learning curve. Whereas compared to Django it follows a Batteries-included philosophy, therefore following a basic tutorial is much easier and a basic application can be made faster.

Another reason for choosing Django is because it is much more popular with an active developer community compared to Flask. This means finding tutorials or questions is much easier. Compared to Flask, its community is not as big as Django and therefore finding answers to questions and basic tutorials will take more time. This made me choose Django as being my first web project in python, having a large community will mean more resources online to learn from with more basic tutorials that don't rely on assumed knowledge.

2.2.6 Leaflet

Leaflet is a JavaScript library for creating interactive maps. Its main feature being lightweight (39KB of JS) and mobile friendly. By default, Leaflet only supports displaying a map via raster tiles. Raster tiles are images rendered on a server and then displayed on a website. A benefit of using raster tiles is that it works on all types of devices as it is just displaying an image. However, there are multiple downsides to using raster tiles. It is not possible to show/hide objects in the client, no way to customize or add new styles on the client side and zooming/moving the map displays temporary empty sections as the required tiles are loaded in. [6]

Since Leaflet is lightweight, its customization options on a map are limited. To get around this it allows for plugins for various features. However, these plugins are third party and therefore documentation isn't guaranteed. Additionally, different plugins compatibility with each other is unknown. [7]

2.2.7 Mapbox

Mapbox is a provider of online custom maps. It is the creator of the library Mapbox GL JS. Opposite to Leaflet, Mapbox GL JS displays maps via vector tiles. Vector tiles have many benefits. This includes being able to change the map's style dynamically client side, no rendering delays when the map is moved and a small size due to vector tiles being binary files containing information on how to generate the map instead of an image. However, there are some downsides to the flexibility vector tiles provide. Because rendering the map occurs client-side, it could cause performance issues on slower devices. Additionally, the library being used is specific for desktop users, therefore mobile users are not accounted for, and a separate library must be used. [6]

Because Mapbox is a commercial company, Mapbox GL JS is a powerful library with many customization options for the map. The default vector tiles provided by Mapbox can be modified on their website to suit a specific need. Additionally, custom made vector data can be created on the website and uploaded to an account. This vector data can then be accessed through API calls.

2.2.8 Creating a map

To create a map on Mapbox/Leaflet, a div element is created in HTML. Then in JavaScript a Mapbox/Leaflet map object is created and bound to that div. Then the map object is given a style layer from a source. In the tutorials they recommend using Mapbox vector tiles. Just from these simple steps an interactive map is created.

The process of creating a basic map is similar for both libraries because Leaflet was initially developed by someone who previously worked on Mapbox.js (older version of Mapbox GL JS).

2.2.9 Why I chose Mapbox over Leaflet

As explained above, Mapbox has much more customization available than Leaflet. This is one of the main reasons I chose Mapbox because I don't know the scope or what features will be required for the final project. Therefore, using Mapbox will mean I don't have to make a late switch between libraries when I find out it doesn't have a specific feature I need. I believe learning the more complicated system of Mapbox is worth it compared to the risk of switching late if Leaflet doesn't work. [6]

When comparing Leaflet and Mapbox GL JS in terms of official documentation, Leaflet's documentation is very basic and doesn't cover any advanced use of the library. This means having to search through the internet for tutorials and examples. Additionally, because it relies on plugins for more advanced features, the documentation will not be standardised. This is another reason why I chose Mapbox because the official documentation is extensive and covers most use cases.

2.2.10 Example - A Django application displaying a Mapbox map

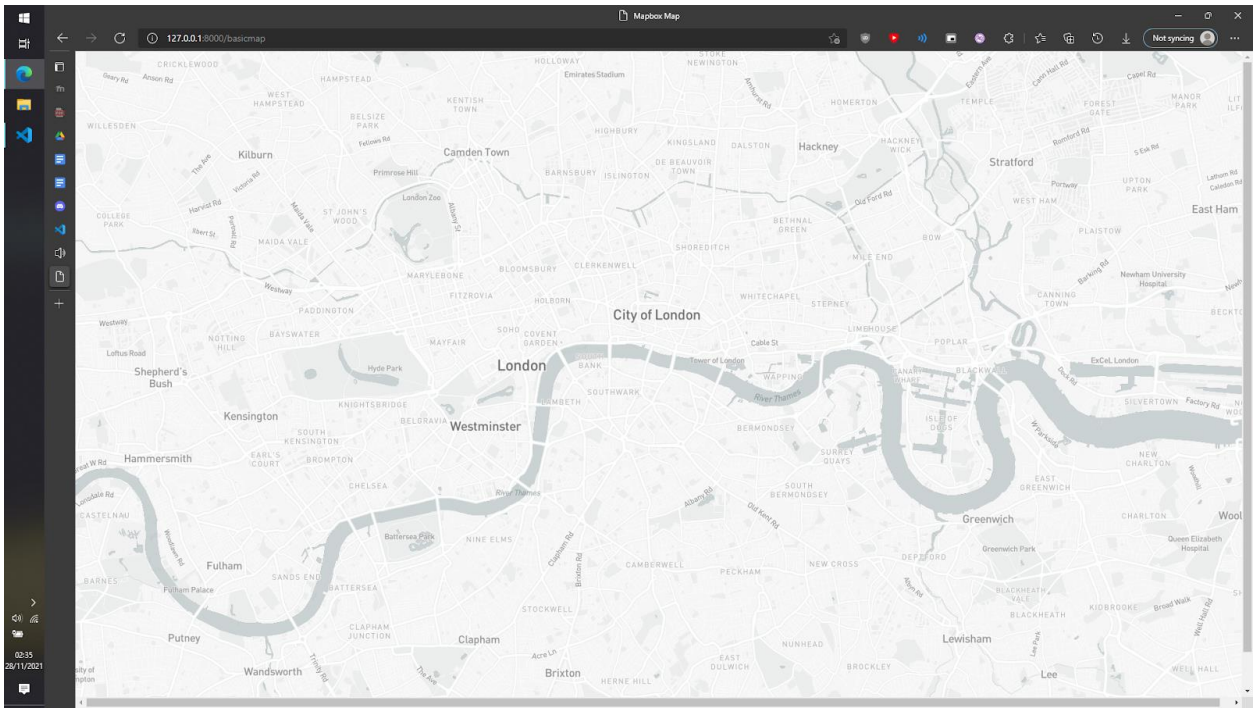


Figure 4.

This example website will return a map from the URL http://127.0.0.1:8000/basicmap. This map is interactive, allowing for moving the map via dragging the mouse and zooming in and out. The Django development server will be used to return the html, CSS, and JavaScript from the URL.

urls.py

```
from django.urls import path
from map import views

urlpatterns = [
    path("basicmap", views.map, name="map"),
]
```

This is the file used to map urls to their corresponding view function. The array urlpatterns contains all mappings. The path function returns an element for inclusion in urlpatterns. The path function's first argument is the route which is a string containing a url pattern, in this case it's the url ending with /basicmap. The next argument is the view function that will be run in response to the url. In this case it will be running the map() function in the views.py file. Finally, an optional argument for a name is used which allows for easier referencing to this mapping.

views.py

```
from django.http import HttpResponse
from django.shortcuts import render

def map(request):
    return render(
        request,
        'map/map.html'
    )
```

This file contains view functions that are run in response to a web request and return a web response. This response can be anything, e.g., the HTML contents of a Web page, or a redirect, or a 404 error, or an XML document. View functions can contain any logic that is necessary to return a response. View functions take a HttpRequest argument, which is called request here.

For the view function map, it runs in response to the url http://127.0.0.1:8000/basicmap. I have used the Django function render, which takes the request and a html file that may contain placeholders for values. Then the Django templating engine makes the substitutions if necessary and returns the html file. This provides separation of code as the view function only works with data values and the template only works with the markup.

map.html

```
<!DOCTYPE html>

<head>
    <title>Mapbox Map</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <script src='https://api.mapbox.com/mapbox-gl-js/v2.4.1/mapbox-
gl.js'></script>
    {% load static %}
    <link rel="stylesheet" type="text/css" href="{%static
'map/style.css' %}" />

</head>

<body>
    <div id='map'></div>
    <script src="{% static 'map/basicMap.js' %}"></script>
</body>

</html>
```

This file is the template that is used in the view function map. Here there are placeholders for values being used. In this case {% load static %} is used to signify that the template engine should fill out the static file pathway. "{% static 'map/style.css'%}" is the template and will be filled out when rendered in the view function. In this example the string "{% static 'map/style.css'%}" will be filled out to "map/static/map/style.css" by the templating engine. This same process is used to load the file basicMap.js.

In the head of the file, the library for Mapbox GL JS is loaded by using a hosted version of it found at the link https://api.mapbox.com/mapbox-gl-js/v2.4.1/mapbox-gl.js.

Finally, a div element with the id map is created. This will be the container for the map object created by Mapbox that displays an interactive map.

basicMap.js

```
mapboxgl.accessToken =  "Access token here";
const map = new mapboxgl.Map({
  container: "map", // container ID
  style: "mapbox://styles/mapbox/light-v10", // style URL
  center: [-0.09, 51.505], // starting position [lng, lat]
  zoom: 12, // starting zoom
  minZoom: 10,
});
```

This script is run at the place it is called in the html file. Mapbox GL JS provides a mapboxgl.Map class. This class is instantiated with arguments seen in the code above. The container used is the div element map shown in the html file above. Mapbox provides a set of styles to use. Here the light v10 style is used. By default, when the map class is created, it will use Mapbox's custom vector tiles. [8]

A mapboxgl.accessToken has its value set. This token is uniquely generated for my account and is used to allow the program to retrieve vector tiles from Mapbox servers.

style.css

```
html, body {
    margin:0px;
  }

#map {
    height: 100vh;
    width: 100vw;
}
```

The code above is the CSS used to make the map div fill the entire page. First the margin of all elements in the html document is set to 0px. This means there will be no space around any elements defined borders. Then the map div is referenced and its height and width are set to 100vh and 100vw. Vh and vw are short for viewport height and width with each unit being 1%. The viewport is the visible area on the screen. Therefore, setting the height and width of the map div to 100% of the viewport height and width.

2.3 Map Data: OpenStreetMap, Overpass turbo, GeoJSON

2.3.1 OpenStreetMap

OpenStreetMap is a free editable map of the whole world. The data that OpenStreetMap uses is populated by volunteers, meaning anyone can edit the map. The geographic database used for OpenStreetMap is freely available to query for any data needed.

OpenStreetMap uses 3 basic elements for its conceptual data model of the physical world. The types of elements are as follows: [9]

- Nodes (defining points in space)

- Ways (defining linear features and area boundaries)
- Relations (sometimes used to explain how other elements work together)

2.3.2 OpenStreetMap - Node

A node represents a specific point on the earth's surface defined by its latitude and longitude. Each node comprises at least an id number and a pair of coordinates.

Nodes can be used to define standalone point features. For example, a node could represent a park bench or a water well.

Nodes are also used to define the shape of a way.

2.3.3 OpenStreetMap – Way

A way is an ordered list of between 2 and 2,000 nodes that define a polyline. Ways are used to represent linear features such as rivers, roads and railways.

Ways can also represent the boundaries of areas such as buildings or forests. In this case, the way's first and last node will be the same. This is called a "closed way". Closed ways occasionally represent loops, such as roundabouts on highways, rather than solid areas.

2.3.4 OpenStreetMap - Relation

A relation is a multi-purpose data structure that documents a relationship between two or more data elements (nodes, ways, and/or other relations). Relations can have different meanings. The relation's meaning is defined by its tags.

The relation is primarily an ordered list of nodes, ways, or other relations. These objects are known as the relation's members.

2.3.5 Overpass turbo

Overpass turbo is a web based data mining tool for OpenStreetMap. It works by running a given query and outputting the results on an interactive map. [10]

Overpass turbo will be used to query for the railway coordinates for the tube lines on the underground. This data is needed as I will be drawing the lube lines over the map. As an example, I will show the query and result for obtaining the railway coordinates for the Hammersmith line.

```
[out:json];
rel(102787);
way(r);
out geom;
```

The first line, out:json will return the result of the query in JSON format. The second line searches for all nodes and ways that are part of the relation “102787”. The relation 102787 corroesponds to the Hammersmith to Barking journey. The third line of the query takes the relation and filters for only the way elements. This is because I only need the railway coordinates and the relation includes the stations in the output. The result of running the query is shown in the figure below.

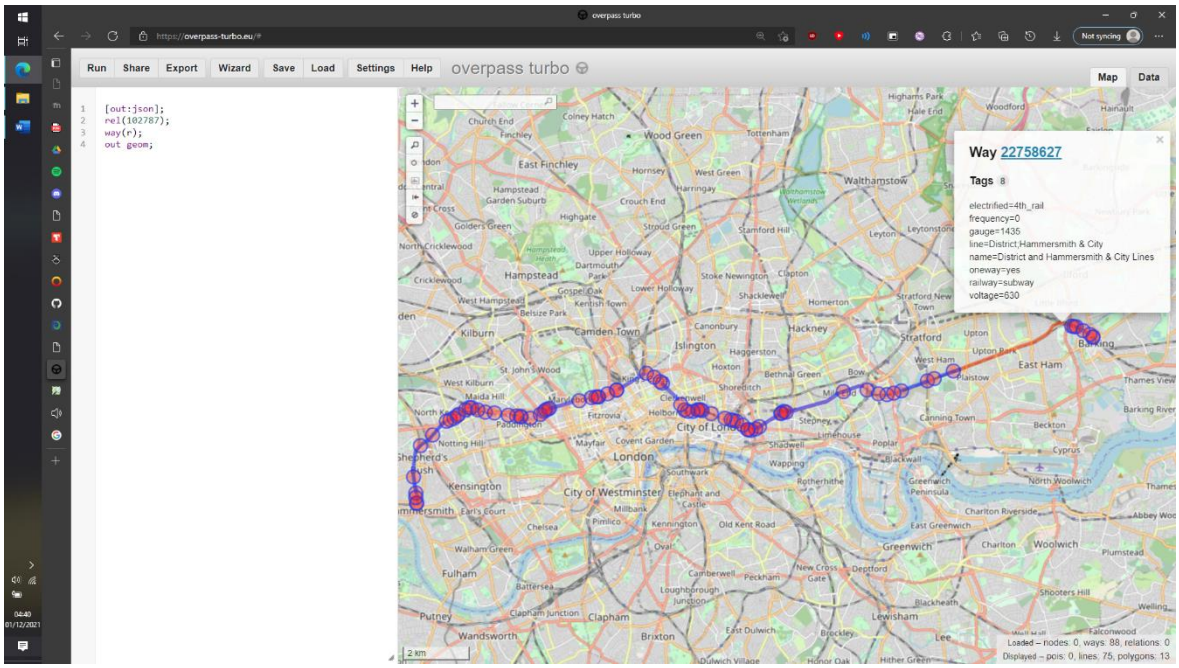


Figure 5.

After running this query, the data can be exported as a GeoJSON file.

2.3.6 GeoJSON

GeoJSON is a standard format designed for representing simple geographical features along with properties. Some of the features that are represented are:[11]

- Point (addresses and locations)
- LineStrings (streets, highways, and railways)
- Polygons (countries and provinces)

In the section above, I queried for the railway data for the Hammersmith line and exported it as a GeoJSON file. Below is an example of a LineString feature that represents a section of the railway.


```
{
      "type": "Feature",
      "properties": {
        "@id": "way/101298",
        "electrified": "4th_rail",
        "frequency": "0",
        "gauge": "1435",
        "line": "Circle;Hammersmith & City",
        "loading_gauge": "subsurface",
        "name": "Circle and Hammersmith & City Lines",
        "railway": "subway",
        "start_date": "1864-06-13",
        "voltage": "630"
      },
      "geometry": {
        "type": "LineString",
        "coordinates": [
          [
            -0.2249906,
            51.493682
```

```
          ],
          [
            -0.2251678,
            51.4945584
          ],
          [
            -0.2251941,
            51.4946959
          ]
        ]
      },
      "id": "way/101298"
    }
```

As shown above, features can contain properties that describe what the feature is representing. It also includes a geometry object which describes what type of feature it is, in this case LineString and the coordinates (latitude and longitude). For this LineString, the coordinates are parsed in sequential order to form the line.

Because of GeoJSON being a standard format, many applications natively support parsing such files. Mapbox, the map library I have chosen to use natively supports representing GeoJSON files on their map, this will be shown in the proof of concepts section.

Chapter 3: **Summary of completed work**

3.1 Proof on concept - Basic Map

The first proof of concept program I made was a website that loaded a map. This map allows for zooming in and out and moving the map around by dragging the mouse. The figure below shows the map that is returned when visiting the URL http://127.0.0.1:8000/proofofconcept/basicmap.

The following files are used in this proof of concept: basicMap.html, basicMap.css, basicMap.js.

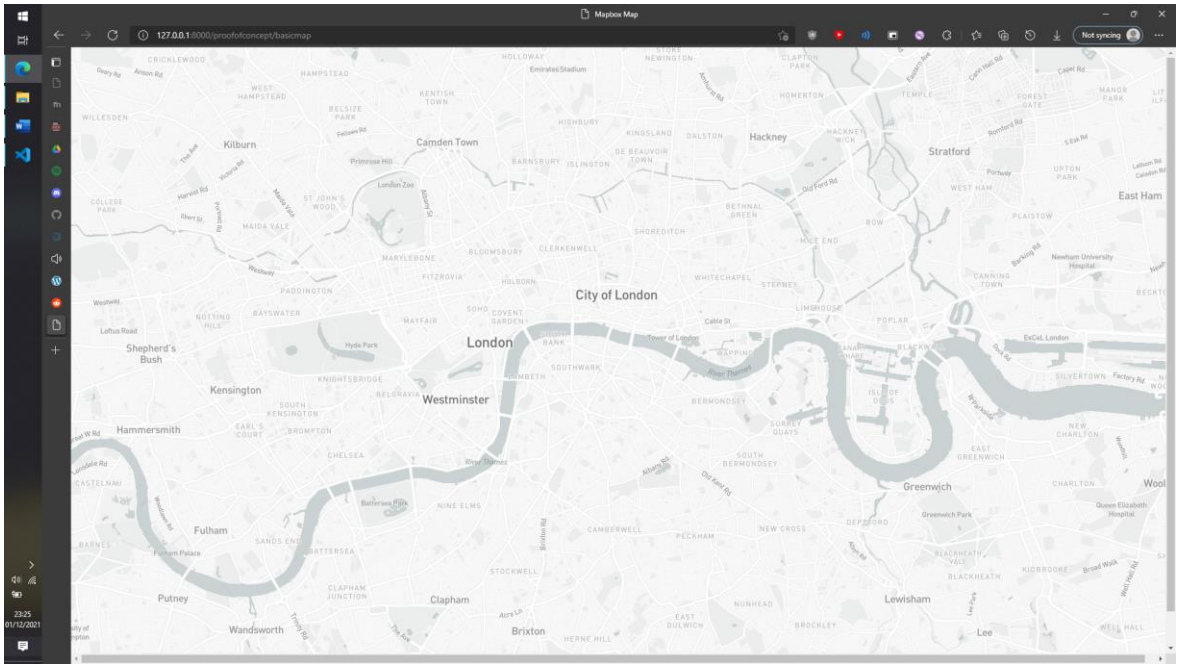


Figure 6.

This website was made with Django as its web framework. It is using Mapbox GL JS as the client-side JavaScript library for providing the map. The URL hostname is 127.0.0.1:8000 because that is what is used by the Django development server.

This proof of concept is almost the same as the example program explained in the report in section 2.2. The only difference is the URL used to access the website. Therefore, no code will be explained for this proof of concept as it has already been explained.

3.2 Proof of concept - Map with tube lines and stations

Building on from the first proof of concept, I used Mapbox GL JS functions to draw the Hammersmith line and the stations it serves over the map. Then I added the functionality to hover over a station icon, which will show a pop up that displays its name. This proof of concept can be viewed by visiting the URL http://127.0.0.1:8000/proofofconcept/mapdrawn.

The following files are used in this proof of concept: mapDrawn.html, mapDrawn.css, mapDrawn.js, GenerateGeoJson.py, OrderRailway.py, RawHammersmithCoordinates.geojson, HammersmithRail.geojson, HammersmithStations.geojson.

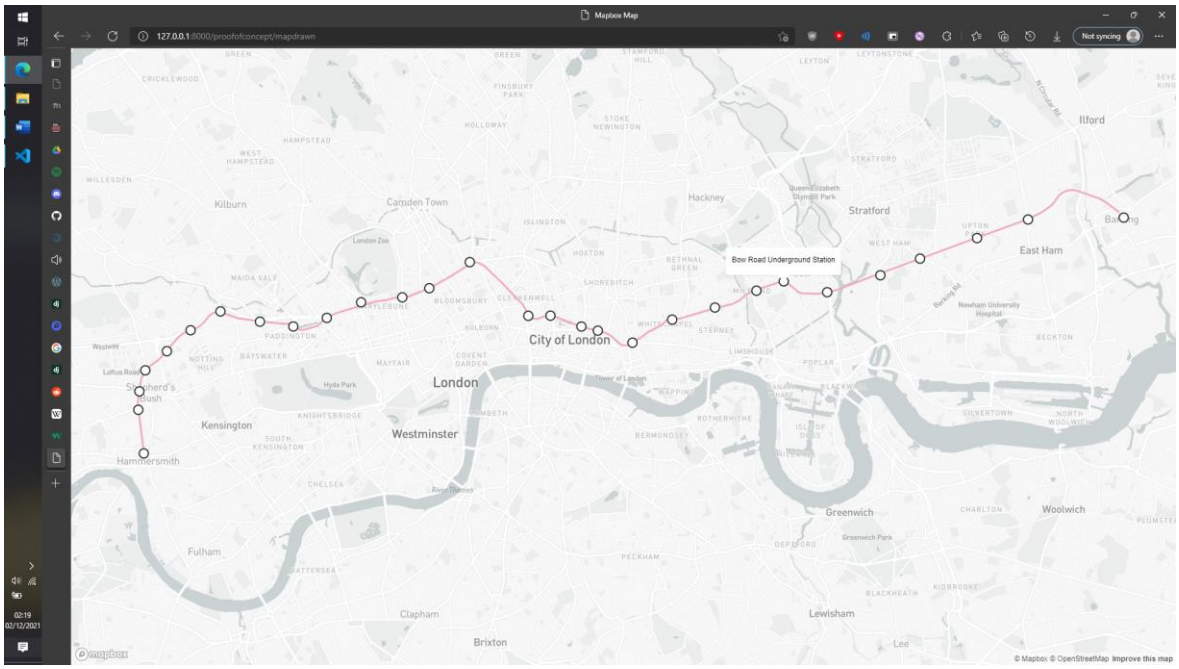


Figure 7.

3.2.1 Retrieving and parsing railway coordinates

The railway coordinates used to draw over the map were the same ones that were retrieved in section 2.3 of the report.

The exported data for the Hammersmith line has a problem, which is the order that the railway features are listed don't follow the route of Hammersmith to Barking. This is because the features are ordered by the id attribute, which doesn't follow the route. This is a problem because for the calculation to predict each trains live location; I need the coordinates to accurately place them on the map. Therefore, to extract all the coordinates from each railway feature and order them following the Hammersmith to Barking route, I created the script OrderRailway.py. This ordered the railway features to follow the route. Then it extracted the coordinates from each feature and placed them in a JSON file output.json.

Then I needed to group the railway coordinates between every 2 stations on the Hammersmith line. This was achieved by creating the script GenerateGeoJson.py. This script loaded the coordinates of every station on the Hammersmith line in order. Then it found the closest point on the railway coordinates to each station and grouped the coordinates until the next station is reached. Then it created a GeoJSON feature for these set of coordinates and output the result into the file HammersmithRail.geojson. Below is an example of a feature between the first 2 stations on the Hammersmith line.

```
"type": "Feature",
            "properties": {
                  "name": "Hammersmith (H&C Line) Underground Station to
Goldhawk Road Underground Station",
                "lineId": "hammersmith-city",
                "direction": "outbound",
                  "previousStation": "Hammersmith (H&C Line) Underground
Station",
                "previousStationId": "940GZZLUHSC",
                "nextStation": "Goldhawk Road Underground Station",
                "nextStationId": "940GZZLUGHK"
            },
            "geometry": {
```

```
                "type": "LineString",
                "coordinates": [
                    [
                        -0.2249906,
                        51.493682
                    ],
                    [
                        -0.2251678,
                        51.4945584
                    ],
                    [
                        -0.2251941,
                        51.4946959
                    ],
                    [
                        -0.2252555,
                        51.4949163
                    ],
                    [
                        -0.225488,
                        51.4961213
                    ],
                    [
                        -0.2255775,
                        51.496585
                    ],
                    [
                        -0.2256142,
                        51.4967693
                    ],
                    [
                        -0.2256854,
                        51.4972091
                    ],
                    [
                        -0.2259827,
                        51.4987234
                    ],
                    [
                        -0.2262316,
                        51.4999517
                    ],
                    [
                        -0.2266403,
                        51.5019688
                    ],
                    [
                        -0.22665393751804577,
                        51.502020963920785
                    ]
                ]
            }
```

3.2.2 Drawing over the map

```
map.addSource("Railways", {
        type: "geojson",
        data: railwayData,
      });

map.addLayer({
```

```
    id: "Railway Style",
    type: "line",
    source: "Railways",
    paint: {
      "line-color": "#F3A9BB",
      "line-width": 3,
    },
  });
```

The code snippets above come from the JavaScript file used to display and draw over the map. The addSource function takes the contents of the HammersmithRail.geojson file produced by the GenerateGeoJson script. The contents are formatted in GeoJSON which is specified as an argument in the function. Mapbox can natively parse GeoJSON data. Placing the GeoJSON data as a source in the map allows for styles to be applied on it. The addLayer function defines the styles that are used on the coordinate data. "line-color": "#F3A9BB" is what gives the line its pink colour in figure 6. [8]

3.3 Proof of concept - Map with animated trains

After successfully showing the Hammersmith line and its stations. The next step was to create train icons that would move along the line. After doing this I decided to make this proof of concept also query the TFL API for each trains predicted time of arrival. From the predicted time of arrival, I would estimate the trains live location and place it on the map. This is shown in the figure below. This proof of concept can be viewed by visiting the URL http://127.0.0.1:8000/proofofconcept/animatedtrains.

The purpose of using the TFL API to get the predicted arrival times and estimate their live locations was to familiarise myself with how the API works. The algorithm used to estimate the live locations was written quickly and contains bugs I have not fixed. This was done intentionally as I only wanted a rough idea of how it would work. All the algorithms will be refactored in the final build.

The following files are used in this proof of concept: animatedTrains.html, animatedTrains.css, animatedTrains.js, GenerateGeoJson.py, GenerateTimetable.py, OrderRailway.py, RawHammersmithCoordinates.geojson, HammersmithRail.geojson, HammersmithStations.geojson, HammersmithTimetable.geojson

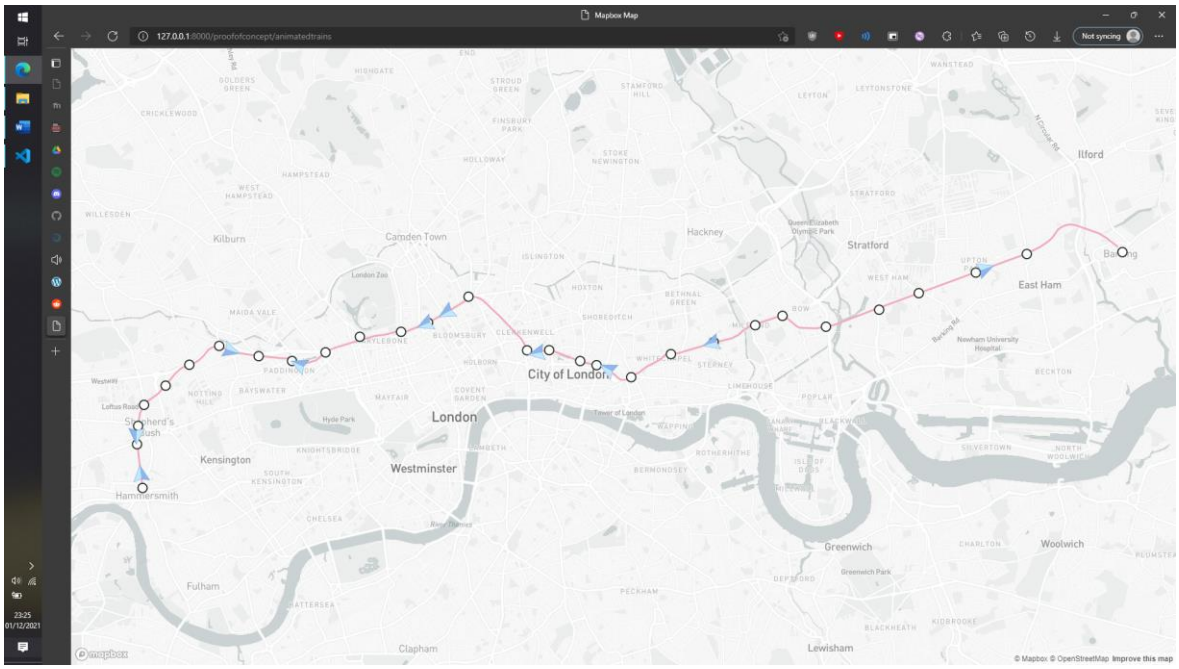


Figure 8.

3.3.1 Displaying / Animating the trains

```
.marker {
  display: block;
  border: none;
  background-image: url('TrainTrans.png');
  cursor: pointer;
  padding: 0;
}
```

The code snippet above is from the CSS file. This code modifies the marker class that is imported from the Mapbox GL JS library. It replaces the default image used for markers to a picture of a train icon. This is how the train icons shown in figure 7 are created, by modifying the default marker to represent a train.

```
train.marker.addTo(map);
```

This line of code is found in the JavaScript file in the createTrainMarkers function. It adds the marker representing a train to the map. This line of code is looped over for every train on the Hammersmith line, adding all the currently running trains to the map.

```
function animateTrain(timestamp) {
  *Logic to calculate and set new train location*
  window.requestAnimationFrame(animateTrain);
}
```

This function is found in the JavaScript file. It is the animation function used to make each train marker move along their railway line. It works by using the JavaScript inbuilt function window.requestAnimationFrame(animateTrain). This function tells the browser we wish to perform an animation and requests the browser calls the specified animation function before the next repaint. The number of calls to the animation function is usually 60 times per second but will generally match the display refresh rate in web browsers. [12]

The animation function works by looping over each train being shown on the map. During each loop it will calculate the trains' new location and update the marker representing the train on the map with the new coordinates. The trains movement is simulated by subtracting its predicted

arrival time by the time that has elapsed between each call of the animation function. The trains starting location comes from the TFL API.

3.3.2 Querying TFL API for prediction data

```
async function getJsonData(url) {
  const staticJsonData = await fetch(url);
  const jsonData = await staticJsonData.json();
  return jsonData;
}
```

This JavaScript function is used to query any given URL and return it in JSON format. It is an async function as it will allow for the await expression to be used on the function. This is used because the data being queried will be used later in the code. Therefore, waiting for the result must be done as errors will occur if the data hasn't been returned yet.

```
let arrivalPredicitons = await getJsonData(
    "https://api.tfl.gov.uk/Line/hammersmith-city/Arrivals"
```

This is a use of the function to query the TFL API for the live predicted arrival times for all the trains on the Hammersmith line. The returned JSON data from the API will be parsed and individual trains will be separated so that their live location can be calculated. [14]

3.3.3 Calculating the trains live location

```
function calculateTrainDistanceTravelled(train) {
  let milesPerSecond = train.railwayDistance / train.timetableToStation;
//How many miles a second travels for a train in terms of timetable
  let secondsTravelled = train.timetableToStation - train.timeToStation;
  if (train.direction == "outbound") {
    return milesPerSecond * secondsTravelled;
  } else {
    return train.railwayDistance - milesPerSecond * secondsTravelled;
  }
}
```

This is the main function used in the JavaScript to calculate the location of a train. It calculates the distance a train has travelled between 2 stations. It works based off the trains predicted time to arrival to the next station.

First it calculates how many miles a second the train can travel between the 2 stations. This is done by dividing the railway distance between the 2 stations by how long it takes to travel between the 2 stations in seconds according to the timetable. Then the amount of time the train has travelled between the 2 stations is calculated. Finally, it returns the result of milesPerSecond * secondsTravelled, which is the distance travelled by the train between the 2 stations. In this proof of concept, the direction the train travels affect how the train is displayed on the map, that is why there are 2 different return statements for either inbound or outbound directions.

```
newTrainLocationFeature = turf.along(
    railwayData["features"][train.currentRail],
    train.distanceTravelled,
    { units: "miles" }
  );
```

This JavaScript code calculates a trains new coordinate on the map after the calculateTrainDistanceTravelled() function has calculated a new distance travelled for the train. I use a library made for geographical calculations, turf.js. From this library I used the along function which takes a LineString and returns a point at a specified distance along the line. In this case the LineString is the collection of coordinates between the 2 stations the train is currently travelling along. The specified distance comes from the calculateTrainDistanceTravelled() function. The returned point (newTrainLocationFeature) is a GeoJson object like LineString but is just a single coordinate. [13]

```
train.marker.setLngLat([
    newTrainLocationFeature["geometry"]["coordinates"][0],
    newTrainLocationFeature["geometry"]["coordinates"][1],
  ]);
```

This is the JavaScript code used to set the marker representing a train with new coordinates. The code is called in the animation function for every train shown on the map. The new coordinate used comes from the point object that was explained above.

3.4 Final build - Current progress

After completing all my planned proof of concept programs, I decided to start making the final build of the project. The current progress is that I have been able to get the location of all the stations that serve the London underground and place them on the map. I have also worked on writing scripts to get all the routes that each line serves and timetables for all the routes. This data has been placed in JSON files. The current progress I have made can be viewed at the URL http://127.0.0.1:8000/finalmap. This is shown in the figure below.

The following files are used in this final build: finalMap.html, style.css, Main.js, GenerateRailway.py, GenerateStations.py, railways.json, stations.json. The other files in the static folder are classes that are currently work in progress and not being used on the website.

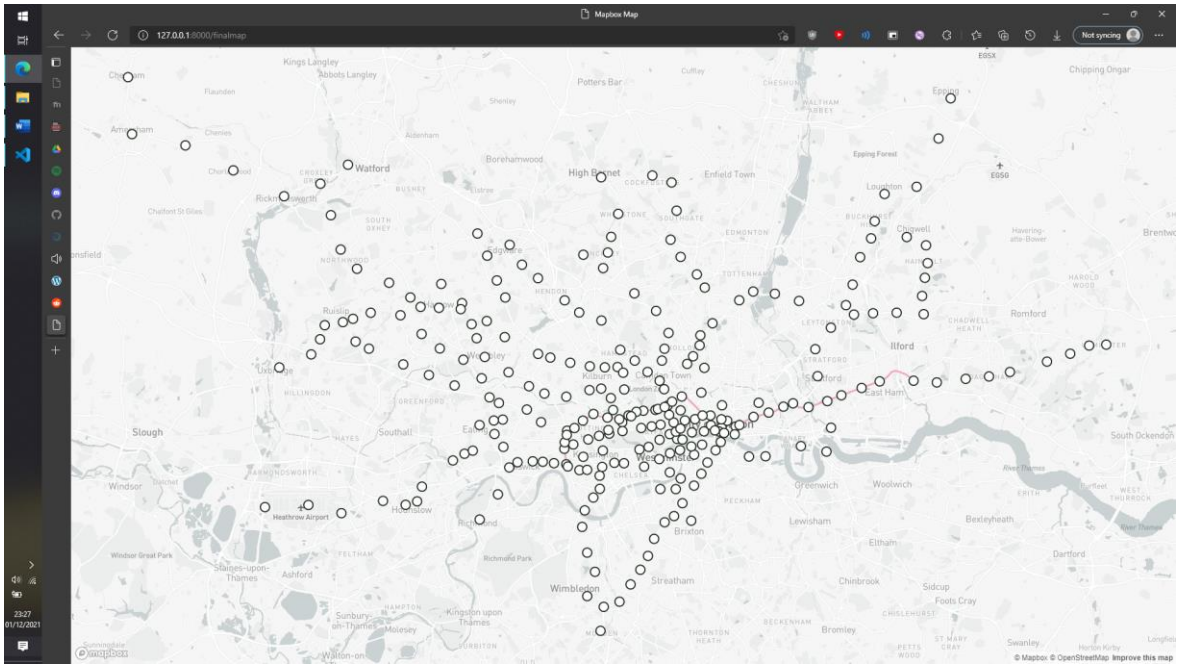


Figure 9.

With this final build I have been focusing on rewriting the scrtipts used to generate the stations, routes, railway, and timetable data. I have had to rewrite these scripts because the ones used in the proof of concept programs only generated data for the Hammersmith line. These reworked scripts generate data for all of the tube lines on the underground.

These scripts can be found in the bin folder of the root directory. This directory contains the JSON files produced by these scripts. I have chosen to use JSON files instead of GeoJSON files for storing data. This is a change from my proof of concept programs. This is because it is easier to write the scripts to output JSON files then convert them to GeoJSON format in the JavaScript code using turf.js functions that do so. Below is a sample of the the new format for the railway.json file.

```
"hammersmith-city": [
                {
                    "direction": "outbound",
                    "departureNaptanId": "940GZZLUHSC",
                    "destinationNaptanId": "940GZZLUBKG",
                    "railways": [
                        {
                            "departureNaptanId": "940GZZLUHSC",
                            "destinationNaptanId": "940GZZLUGHK",
                            "coordinates": [
                                [
                                    -0.2249906,
                                    51.493682
                                ],
                                [
                                    -0.2251678,
                                    51.4945584
                                ],
                                [
                                    -0.2251941,
                                    51.4946959
                                ],
                                [
                                    -0.2252555,
                                    51.4949163
                                ],
                                [
                                    -0.225488,
                                    51.4961213
                                ],
                                [
                                    -0.2255775,
                                    51.496585
                                ],
                                [
                                    -0.2256142,
                                    51.4967693
                                ],
                                [
                                    -0.2256854,
                                    51.4972091
                                ],
                                [
                                    -0.2259827,
                                    51.4987234
                                ],
                                [
                                    -0.2262316,
                                    51.4999517
                                ],
                                [
                                    -0.2266403,
                                    51.5019688
                                ],
                                [
                                    -0.22665393751804577,
                                    51.502020963920785
                                ]
                            ]
                        }
```

As you can see in the sample json data, I have included data about the stations the coordinates serve. This is a different format compared to the GeoJSON shown in the proof of concept in section 3.2.

As seen in the figure above, I have been able to display all the stations serving the underground and the Hammersmith line. I have not been able to make progress on displaying the location of trains as it is dependant on the JSON files I created which I just finished making.

Chapter 4: **Software Engineering used**

4.1 Design Patterns

Using Django, I have followed the Model-View-Template design pattern. This is a variation on the popular Model-View-Controller design pattern. The difference between MVC and MVT is that the controller logic, which is coded in manually, is taken care of by the framework itself. The 3 components of MVT are as follows:[15]

- This Model acts as an interface for data and is the logical structure behind the entire web application, which is represented by a database such as MySQL, PostgreSQL.

- The View executes the business logic and interacts with the Model and renders the template. It accepts HTTP request and then return HTTP responses.

- The Template is the component which makes MVT different from MVC. Templates act as the presentation layer and are the HTML code that renders the data. The content in these files can be static or dynamic.

Using the MVT design pattern has advantages for coding. One such advantage is that the components are loosely coupled, which makes modification easy to do. It also provides separation of concerns, which makes reusing and testing components easier.

4.2 Prototyping

The proof of concept programs I made were created as prototypes for the final build of the project. Each prototype built off the previous one which allowed me to test different ideas for each new feature added. For example, the first prototype, building a basic map I was able to test 2 different libraries to see which one would be best suited for the project.

The final prototype, the animated trains was the most useful one. It was developed as a throwaway prototype to familiarise myself with how the TFL API works and create a basic algorithm to calculate each trains live location. This prototype won't be refined into the final build, but the idea behind the algorithms will be incorporated into the final build. This helped me in making a better plan for the final build as I now have a basic idea of how the algorithm works and can plan to code a better version into the final build.[16]

4.3 VCS, commits and branches

For this project I have used GitHub as the version control system. Using GitHub, I done small commits for uploading work. This makes it easier to understand the work that was done when reviewing commits. Additionally, it makes it easier to revert a commit if something has gone wrong as large commits may contain changes that you may want to keep. Finally small commits make the commit message clearer as it is easier to state what has been done.

I committed work at a regular rate throughout the first term. Working a regular rate allowed me to find bugs early and fix them. This also increased the quality of code written as having breaks between coding allowed me to plan before coding thus leading to less bugs.

While working on the project I have used branches to commit work. This allowed me to work on different parts of the project at the same time without interfering with each other. It also kept a working version of the project in the main branch available while working on other branches. Once I finished work on a branch, I would merge the commits back into the main branch. This keeps the main branch clean of bugs and a working version that can be referenced.

Chapter 5: **Planning**

5.1 Second term plan

My current progress on the final build is explained in section 3.4. The remaining work that needs to be done is as follows:

- Download railway coordinate data for all the remaining lines
- Fix the script used to generate timetable data as it is missing data for 2 lines
- Combine all the scripts used to generate route, railway, station, and timetable data under a façade object
- Create the algorithm to query TFL API for predicted arrival times and parse data into individual trains
- Create the algorithm to calculate each trains live location from its predicted arrival time
- Scale the algorithms to work with all the tube lines on the London underground
- Create the functionality for a pop up with information when clicking on a train icon

The first thing I will do is fix the script to generate all the timetable data. Then I will group all the scripts together under a façade. This will make it simpler to generate all the required data as only a single script needs to be run.

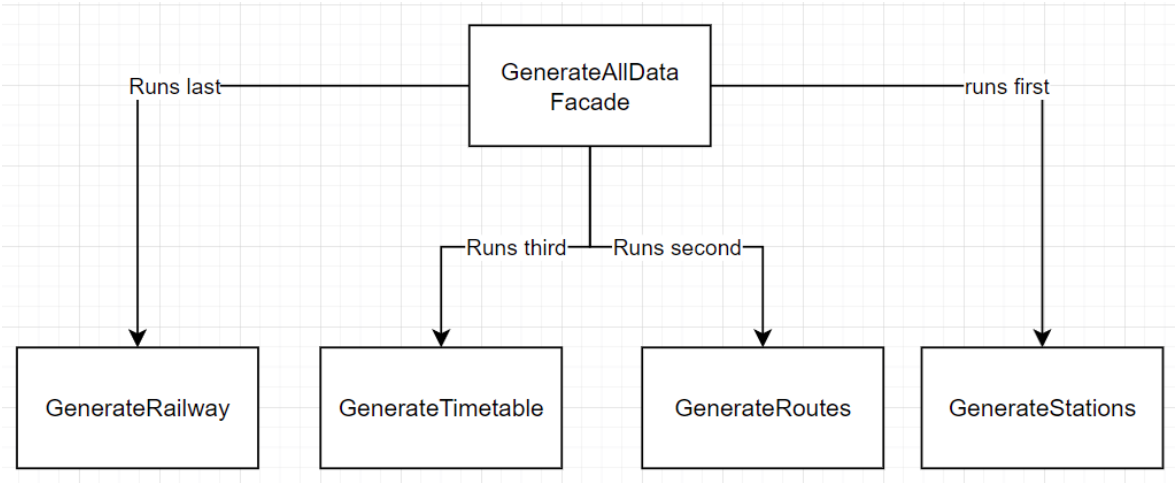


Figure 10.

This is a diagram showing how the façade will run each script to generate all required JSON data files.

Then I will create the new algorithms that will query and parse the TFL API predicted arrival times data. Using this data, I will then create the algorithm to calculate each trains live location. These algorithms will be designed so that they are scalable to work with all the tube lines in the underground.

Finally, after the core functionality of the application has been implemented, I will start working on less important features such as displaying information when a train icon is clicked.

5.2 Test Strategy

There are 3 main parts of my project that need to be tested. First the JSON data that is created by the scripts needs to be tested. Some of this JSON data is displayed on the map, such as the railway coordinates and stations. Therefore, testing is required to make sure that the data is displayed properly. Each JSON file will be tested differently. For example, the timetable file will be tested to make sure that it contains timetable data for all the routes that exist. This would be done by comparing all the timetable data with the routes file to see if all the routes are accounted for in the timetable file.

The second part that needs to be tested is the algorithm to parse the TFL API data into individual trains, then calculate their live location. This cannot be done with the live TFL API data as it is always changing and therefore a correct static output for the algorithms cannot be created. To work around this problem, I will take a sample of the TFL API data for predicted arrival data and manually create a correct output that I would expect from the algorithms. This will allow me to test the algorithms by running them against the sample data and I will know if the algorithms are parsing the data correctly as I have a correct output that I have manually checked.

Finally, the positions of the trains displayed on the map need to be tested to make sure they are correctly placed. This will use the sample data explained in the point above. I will run the algorithm to place the trains on the map with the sample data and a correct output would be the trains placed in the correct location. Since it is sample data, I will already know what the expected train positions should be.

Chapter 6: **Project Diary**

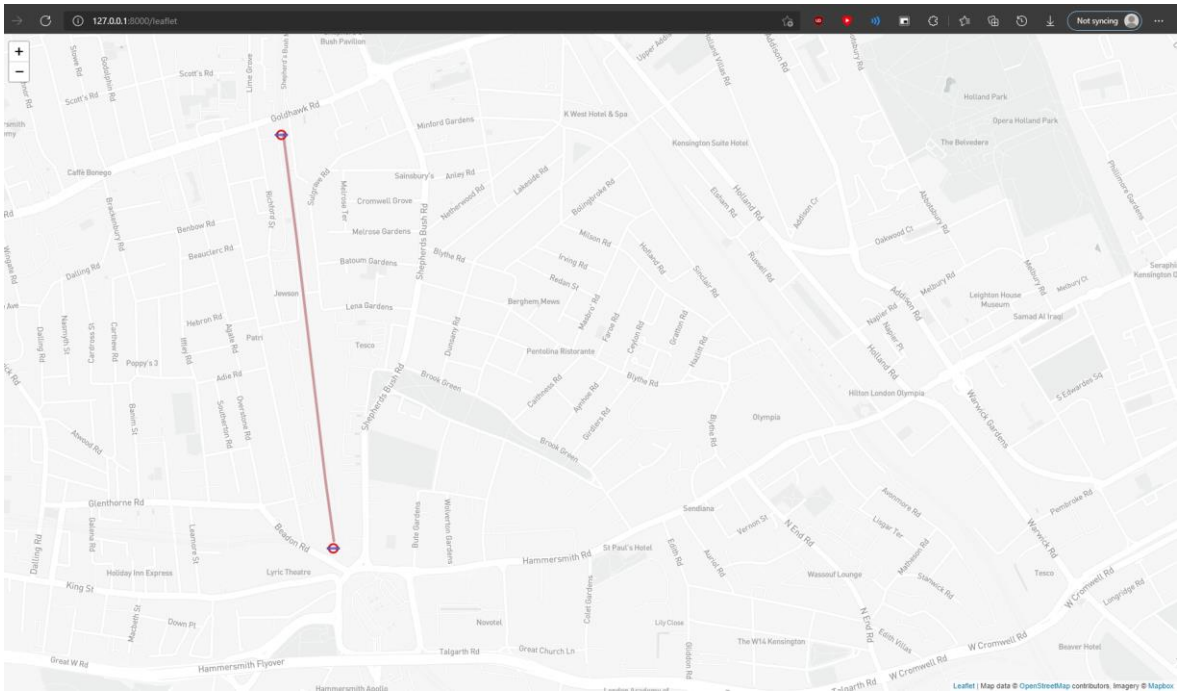
6.1.1 4:15 pm on September 27, 2021

I have finished a basic Django tutorial to setup a basic backend for the website. Using this I have created 2 webpages. One uses leaflet to draw a map, the other uses Mapbox. On the leaflet map I have drawn the hammersmith & city line over the map including stations it serves. I got this data using overpass turbo. However, there is a problem with OSM data itself as the relation I used to acquire this data doesn't include 3 stations. I have tried using the TFL API to get all the underground stations and retrieve their longitude and latitude, however this doesn't work as the coordinates for each station don't match with the coordinates for each station in the OSM data. If I were to use the TFL API data, then I would have to represent the railways as straight lines connecting each station instead of what they actually look like.

6.1.2 1:10 am on September 28, 2021

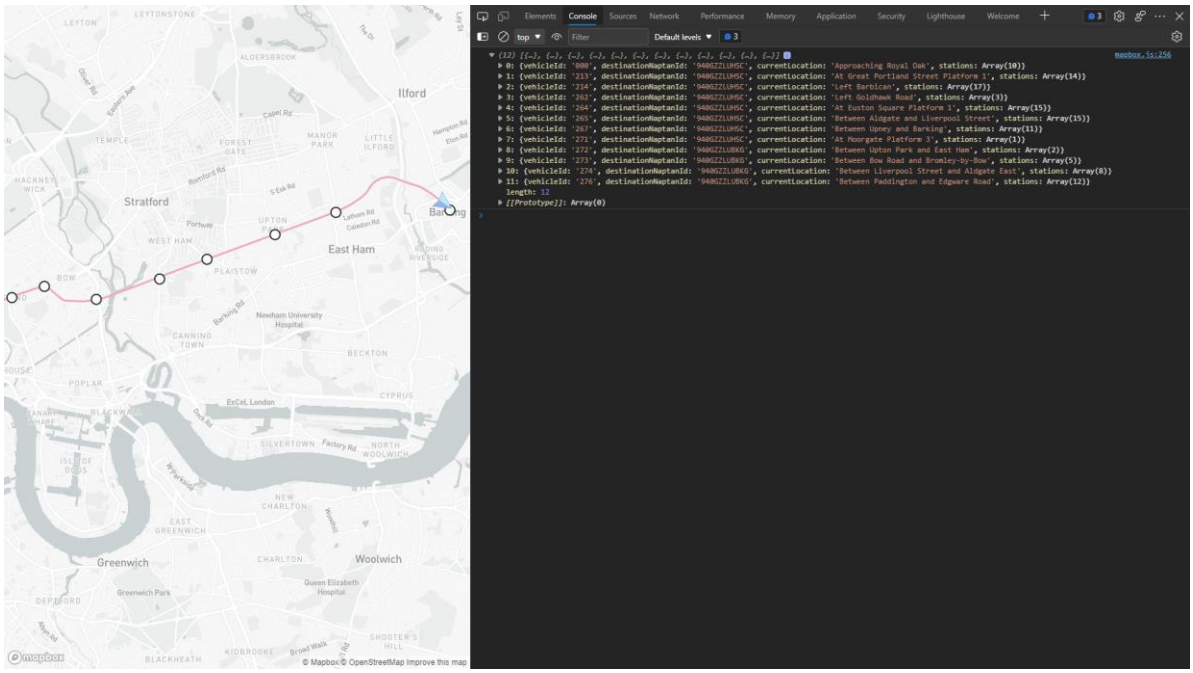
Using an algorithm that finds the closest point on a given line to another point. I can use the TFL station coordinate data and use that as a given point, then I go through each segment in the railway of a tube line and attempt to the find the closest point. This returns the closest point on the railway to the station, this will allow me to segment each part of the railway while still using the official TFL data for stations but using OSM data for railway line.

6.1.3 2:57 am on September 28, 2021



This is the first success I've had using the algorithm I mentioned below. Here the 2 stations shown are from the TFL API and therefore not connected to the railway. I used the algorithm to find the closest points on the railway and generate GeoJSON data that is shown in the image.

6.1.4 9:49 pm on October 7, 2021



Website uses TFL API to get arrival prediction for hammersmith line. Takes that data and gets all vehicleIds. These vehicleIds aren't unique therefore contact API for predictions for all vehicleIds on hammersmith line. Filter arrival prediction data for each vehicleId and print on website as seen in screenshot. It shows all unique trains running on the actual hammersmith railways. If it was on all hammersmith lines, then stations not on the railway but part of the hammersmith line would be output, however that isn't needed as I'm only trying to get trains on the hammersmith railway as that's the only data I must work with currently.

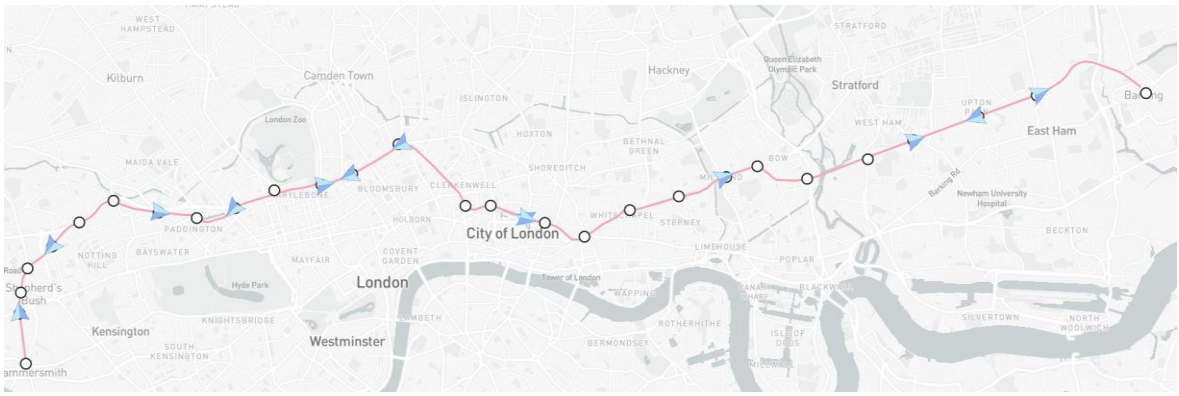
6.1.5 9:51 pm on October 7, 2021

Forgot to mention, changed from using leaflet to Mapbox. Mapbox is more powerful allowing for more options. Also seen in the screenshot is an icon on the railway. That is an example train that moves along the rail line and rotates to the direction it is facing. Just as an example going from hammersmith to barking. Will now use the data i have filtered to show real trains.

6.1.6 4:12 pm on October 28, 2021

Created a broken prototype. This prototype wasn't meant to be fully functional. It cut a lot of corners and isn't scalable. It was only meant for me to get a general idea on how train images will be placed on the map, how they will be animated and logic the trains will use in moving across a tube line. The prototype misses a lot of trains, showing only 13 trains on the hammersmith line at peak hours which isn't right. Now I am currently working on creating a scalable version of the website starting from scratch. The logic used in the prototype was made to only accommodate a single line. Therefore, it was too much work to refactor it to work with multiple lines as a lot of the underlying files that served json data to the website had to be changed. Right now, I am working on creating the routes each line serves which is a bit of a headache.

6.1.7 5:12 pm on October 28, 2021



As seen in the image above, this is the initial prototype, the positions of trains are wrong due a to bug in the code, however the main idea of trains facing the direction they are going and being placed along the railway line is shown here.

6.1.8 1:53 pm on November 1, 2021

I have written reports on basic web development and advanced technologies. I now plan to finish working on generating the json files needed for the website. I have checked the mark scheme and realised TDD is required for good marks. Therefore, I will start writing code for the backend through TDD. I am unsure if TDD is usable when writing JavaScript as I am working with live API data.

6.1.9 10:37 pm on November 8, 2021

Haven't been able to work on the project this past week because I have had 2 pieces of large coursework due. I will be spending this whole week working on the project coding wise.

6.1.10 11:16 pm on November 22, 2021

I have redone my basic web development report on html with an example website plus code explained. For my project, I have nearly completed working on the scripts to automatically generate backend data. The script used to generate the routes and all stations in the TfL are done. The script to generate the timetable data for all routes is nearly done, the only problem is that it is encountering a bug on the Piccadilly and circle lines, however all other lines generate correct timetable data.

6.1.11 1:42 am on December 1, 2021

I have extracted the proof of concept programs out of my main project and placed them in a separate folder. I have also added new url mappings for the proof of concept programs so it can be easily run instead of having to manually replace files.

Chapter 7: **Appendix**

7.1 Submission directory structure

The directory of the project contains 3 main folders. This is the bin, web_project, and map folders. The bin contains the scripts used to generate the routes, timetable, railways, and stations data to be displayed on the map. It will also contain the output JSON files of these scripts. Inside the bin folder will contain a proof of concepts folder, this will contain the old scripts and GeoJSON files used for the proof of concept programs.

The web_project folder contains files created by Django. The folder contains configuration files of the project. This includes urls.py which is the url mapper. It also contains the settings.py file which is used to add applications and middleware to the project. The other files in this folder aren't used for this project.

Finally, the map folder contains the files for the application used for the project. This will contain files such as view.py and urls.py which is explained in the reports section 2.2. It will also contain the templates folder which contains the html file for the website. Inside the templates folder is a prof of concepts folder which contains the html files for the proof of concept websites. Finally, the static folder contains other data needed by the website besides html files such as JavaScript and CSS files. Similar to the templates folder, the static folder contains a proof of concept folder which serves the same purpose.

I have placed proof of concept folders inside the project instead of as a separate folder. This is because it is simpler to visit different urls to see proof of concept programs working, instead of having to copy and paste these files into the main project.

7.2 Video showing project running

https://www.youtube.com/watch?v=gNMObciMiXE

This video shows all 3 of my proof of concept programs running, then finally showing the current progress on the final build.

Bibliography

[1] C. Marshall, "HTML5: what is it?," [Online]. Available: https://www.techradar.com/uk/news/internet/web/html5-what-is-it-1047393.

[2] "Window.getComputedStyle()," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Window/getComputedStyle.

[3] "Django introduction," [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction.

[4] "Flask vs Django: How to Understand Whether You Need a Hammer or a Toolbox," [Online]. Available: https://steelkiwi.medium.com/flask-vs-django-how-to-understand-whether-you-need-a-hammer-or-a-toolbox-39b8b3a2e4a5.

[5] "Building a Flask Web Application," [Online]. Available: https://alysivji.github.io/flask-part2-building-a-flask-web-application.html.

[6] "Mapbox Vs Leaflet: What Makes These Two So Different," [Online]. Available: https://mapsvg.com/blog/mapbox-vs-leaflet.

[7] "Leaflet Documentation," [Online]. Available: https://leafletjs.com/reference.html.

[8] "Mapbox GL JS Documentation," [Online]. Available: https://docs.mapbox.com/mapbox-gl-js/.

[9] "OpenStreetMap Wikipedia Elements," [Online]. Available: https://wiki.openstreetmap.org/wiki/Elements.

[10] "OpenStreetMap Wikipdeia Overpass turbo," [Online]. Available: https://wiki.openstreetmap.org/wiki/Overpass_turbo.

[11] "GeoJSON Documentation," [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7946#section-3.1.

[12] "Window.requestAnimationFrame()," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame.

[13] "Tuf,js Documentation," [Online]. Available: https://turfjs.org/docs/.

[14] "TFL Unified API Documentation - Line endpoint," [Online]. Available: https://api-portal.tfl.gov.uk/api-details#api=Line&operation=Line_ArrivalsByPathIds.

[15] “Difference between MVC and MVT design patterns,” [Online]. Available: https://www.geeksforgeeks.org/difference-between-mvc-and-mvt-design-patterns/.

[16] “Prototypes: Evolutionary vs Throwaway,” [Online]. Available: https://simplicable.com/new/evolutionary-prototype-vs-throwaway-prototype.