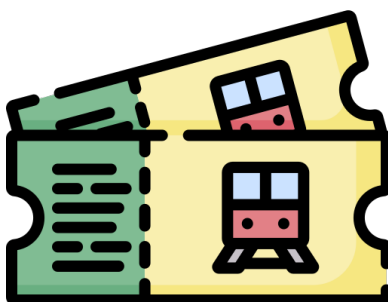# OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Iompar: Live Public Transport Tracking

College of Science & Engineering

Bachelor of Science (Computer Science & Information Technology)

Project Report

Author:

Andrew Hayes
Student ID: 21321503

Academic Supervisor:

Dr. Adrian Clear

2025–04–03

Contents

Chapter 1

Introduction

1.1 Project Overview

The purpose of this project is to create a useful & user-friendly application that can be used to track the current whereabouts & punctuality of various forms of Irish public transport, as well as access historical data on the punctuality of these forms of transport. In particular, this location-tracking takes the form of a live map upon which every currently active public transport service is plotted according to its current location, with relevant information about these services and filtering options available to the user.

The need for this project comes from the fact that there is no extant solution for a commuter in Ireland to track the current location and whereabouts of all the different public transport services available to them. There are some fragmented services that purport to display the live location of one particular mode of transport, such as the Irish Rail live map[14], but this can be slow to update, displays limited information about the services, and only provides information about one form of public transport.

The need for an application that tracks the live location & punctuality of buses in particular is felt here in Galway, especially amongst students, as the ongoing housing shortage drives students to live further and further away from the university and commute in, with Galway buses often being notoriously unreliable and in some cases not even showing up, many commuters could benefit from an application that tells them where their bus actually is, not where it's supposed to be.

The name of the application, "Iompar" (IPA: /ˈʊmˠpˠəɾˠ/), comes from the Irish word for "transport" but also can be used to mean carriage, conveyance, transmission, and communication[16]; it was therefore thought to be an apt name for an application which conveys live Irish public transport information to a user.
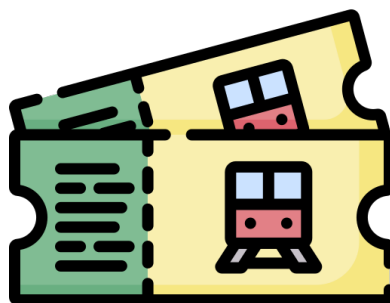

Figure 1.1: Iompar project icon[18]

1.2 Objectives

1.2.1 Core Objectives

The core objectives of the project are as follows:

- Create a live map of train, DART, bus, & Luas services in Ireland, which displays the real-time whereabouts of the service, relevant information about that particular service, and the punctuality of the service, to the extent that is possible with publicly-available data.

- Make the live map searchable to facilitate easy navigation & use, such as allowing the user to find the particular service in which they are interested.

- Provide an extensive array of filters that can be applied to the map to limit what services are displayed, including filtering by transport mode & punctuality.

- Collect & store historical data about services and make this available to the user as relevant, either via a dashboard or via relevant predictions about the punctuality of a service based off its track record.

- An easy-to-use & responsive user interface that is equally functional on both desktop & mobile devices.

1.2.2 Secondary Objectives

In addition to the core objectives, some secondary objectives include:

- Many of those who commute by bus don't have a specific service they get on as there are a number of bus routes that go from their starting point to their destination, and therefore it would be useful to have some kind of route-based information rather than just service-based information.

- A feature which allows the user to "favourite" or save specific services such as a certain bus route.

- Implement unit testing and obtain a high degree of test coverage for the application, using a unit testing framework such as PyUnit.

- The ability to predict the punctuality of services that will be running in the coming days or weeks for precise journey planning.

- User accounts that allow the user to save preferences and share them across devices.

- User review capability that allows users to share information not available via APIs, such as how busy a given service is or reports of anti-social behaviour on that service.

- Make the web application publicly accessible online with a dedicated domain name.

- Port the React application to React Native and make the application run natively on both Android & iOS devices.

- Publish the native applications to the relevant software distribution platforms (Apple App Store & Google Play Store).

1.2.3 Additional Objectives

Some additional objectives beyond the objectives that were specified before beginning development of this project were added as the project was developed, including:

- Remember a user's preferences the next time they visit the page.

- Optimise the performance of the frontend to process data as fast and as efficiently as possible.

- Make the UI design comply with Nielsen's 10 Usability Heuristics[41].

- Utilise the user's geographical location to allow them to filter results by proximity to them.

- Optimise the backend to stay within the limits of AWS Free Tier[27].

- Create a CI/CD pipeline to automatically test and deploy new code.

1.3 Use Cases

The use cases for the application are essentially any situation in which a person might want to know the location or the punctuality of a public transport service, or to gain some insight into the historical behaviour of public transport services. The key issue considered was the fact that the aim of the project is to give a user an insight into the true location and punctuality of public transport: where a service actually is, not where it's supposed to be. The application isn't a fancy replacement for schedule information: the dissemination of scheduling information for public transport is a well-solved issue. Schedules can be easily found online, and are printed at bus stops and train stations, and displayed on live displays at Luas stops. Public transport users know when their service is *supposed* to be there, what they often don't know is where it *actually* is. The application is to bridge this gap between schedules and reality.

Figure 1.2: Photograph of a TFI display erroring due to the clocks going forward [Taken: 2025–03–30]

Furthermore, any existing solution that attempts to give public transport users live updates can be unreliable, slow to update, difficult to use, and only supports one type of transport which forces users to download or bookmark numerous different website and apps, and learn the different interfaces & quirks for each. Figure 1.2 above is a recent example of this: the few bus stops in Galway that actually have a live information display all displayed error messages on Sunday the 30th of March because the clocks went forward by an hour and the system broke. There is a need for a robust and reliable solution for public transport users who want to know where their service is.

With this being said, the main use cases that were kept in mind during the design process were:

- A bus user waiting for their bus that hasn't shown up when it was supposed to and needs to know where it actually is so they can adjust their plans accordingly;

- A train user waiting for their train that hasn't shown up;

- A train user on a train wondering when they will arrive at their destination;

- A Luas user who wants to catch the next Luas from their nearest stop.

3

1.4   Constraints

The primary constraint on this project is the availability of data. Different public transport providers have different APIs which provide different types of data: some don't provide location data, others don't provide have punctuality data, and others don't have any API at all. Other constraints include:

- API rate limits & update frequencies;

- Cost of compute & storage resources;

- API security policies which limit what kind of requests can be made and from what origin.

Chapter 2

Research

2.1 Similar Services

2.2 Data Sources

2.3 Technologies

2.3.1 Frontend Technologies

2.3.2 Backend Technologies

2.3.3 Project Management Technologies

Chapter 3

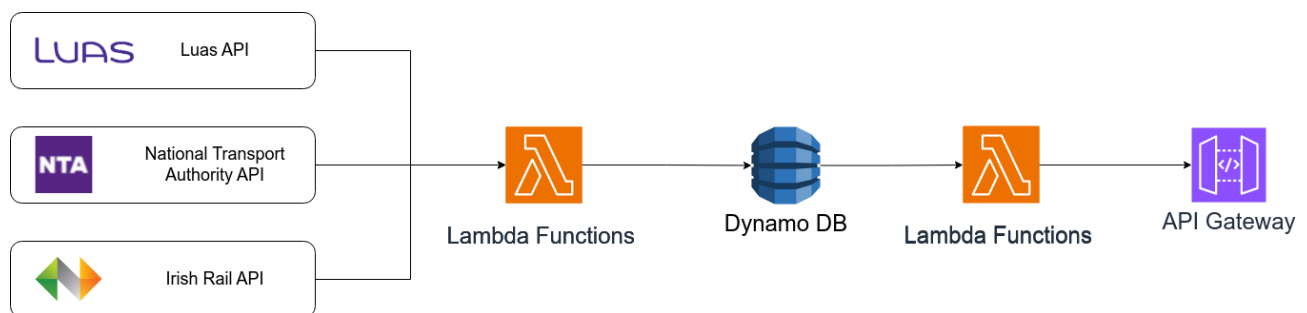# Backend Design & Implementation

Figure 3.1: High-level backend architecture

3.1 Database Design

Since the chosen database system was DynamoDB, a No-SQL database, the question of how best to separate the data is more open-ended: unlike a relational database, there is no provably correct, optimised structure of separated tables upon which to base the database design. The decision was made that data would be separated into tables according to the type of data, how its used, and how its updated, thus allowing separation of concerns for functions which update the data and allowing different primary keys and indices to be used for different querying patterns.

Permanent Data Table

The permanent data table holds the application data which is unchanging and needs to be updated only rarely, if ever. This includes information about bus stops, train stations, Luas stops, and bus routes. This data does not need to be updated regularly, just on an as-needed basis. Since this data is not temporal in nature, no timestamping of records is necessary.

```
1   [
2     {
3       "objectID": "IrishRailStation-GALWY",
4       "objectType": "IrishRailStation",
5       "trainStationCode": "GALWY",
6       "trainStationID": "170",
7       "trainStationAlias": null,
8       "trainStationDesc": "Galway",
9       "latitude": "53.2736"
10      "longitude": "-9.04696",
11    },
12    {
13      "objectID": "BusStop-8460B5226101",
```

```
14        "objectType": "BusStop",
15        "busStopID": "8460B5226101",
16        "busStopCode": "522611",
17        "busStopName": "Eyre Square",
18        "latitude": "53.2750947795551"
19        "longitude": "-9.04963289544644",
20      },
21      {
22        "objectID": "BusRoute-4520_67654",
23        "objectType": "BusRoute",
24        "busRouteID": "4520_67654"
25        "busRouteAgencyName": "City Direct",
26        "busRouteAgencyID": "7778028",
27        "busRouteShortName": "411",
28        "busRouteLongName": "Mount Prospect - Eyre Square",
29      },
30      {
31        "objectType": "LuasStop",
32        "objectID": "LuasStop-STS",
33        "luasStopCode": "STS"
34        "luasStopID": "24",
35        "luasStopName": "St. Stephen's Green",
36        "luasStopIrishName": "Faiche Stiabhna",
37        "luasStopIsParkAndRide": "0",
38        "luasStopIsCycleAndRide": "0",
39        "luasStopLineID": "2",
40        "luasStopZoneCountA": "1",
41        "luasStopZoneCountB": "1",
42        "luasStopSortOrder": "10",
43        "luasStopIsEnabled": "1",
44        "latitude": "53.3390722222222",
45        "longitude": "-6.26133333333333",
46      }
47    ]
```

Listing 1: Sample of the various types of items stored in the permanent data table

As can be seen in Listing 1, two additional fields are included for each item beyond what is returned for that item by its source API: the `objectType` to allow for querying based on this attribute and the `objectID`, an attribute constructed from an item's `objectType` and the unique identifier for that item in the system from which it was sourced, thus creating a globally unique identifier for the item. However (for reasons that will be discussed shortly), this attribute is *not* used as the primary key for the table; instead, it exists primarily so that each item has a unique identifier that does not need to be constructed on the fly on the frontend, thus allowing the frontend to treat specific items in specific ways. An example of a use for this is the "favourites" functionality: a unique identifier must be saved for each item that is added to a user's favourites. Defining this unique identifier in the backend rather than the frontend reduces frontend overhead (important when dealing with tens of thousands of items) and also makes the system more flexible. While the "favourites" functionality is implemented fully on the frontend at present, the existence of unique identifiers for items within the table means that this functionality could be transferred to the backend without major re-structuring of the database.

There are two ways in which a primary key can be created for a DynamoDB table[4]:

- A simple primary key, consisting solely of a **partition key**: the attribute which uniquely identifies an item, analogous to simple primary keys in relational database systems.

- A composite primary key, consisting of a partition key and a **sort key**, analogous to composite primary keys in relational database systems. Here, the partition key determines the partition in which an item's data is stored, and the sort key is used to organise the data within that partition.

While the objectID could be used as a partition key and thus a simple primary key, it was decided not to use the attribute for this purpose as it was not the most efficient option. The primary function of the permanent data table is to provide data for a user when they want to display a certain type of object, such as bus stops, train stations, Luas stops, or some combination of the three. Therefore, the most common type of query that the table will be dealing with is queries which seek to return all items of a certain objectType. Partitioning the table by objectID would make querying by objectID efficient, but all other queries inefficient, and querying by objectID is not useful for this application. Instead, the permanent data table uses a composite primary key, using the objectType as the partition key and the objectID as the sort key. Thus, it is very efficient to query by objectType and return, for example, all the bus stops and Luas stops in the country.

Technically speaking, there is some redundant data in each primary by using the objectID as the sort key when the partition key is the objectType: since the objectID already contains the objectType, it is repeated. However, the unique identifier for each item is different depending on the system from which it was sourced: for train stations, the unique identifier is named trainStationCode, while the unique identifier for bus stops is named busStopID. To use these fields as sort key, they would have to be renamed in each item to some identical title, thus adding overhead to the process of fetching data, and making the table less human-readable. Since the objectID was to be constructed regardless for use on the frontend, it is therefore more efficient to re-use it as the sort key, even if it does result in a few bytes of duplicated data in the primary key of each item.

Transient Data Table

The transient data table holds the live tracking data for each currently running public transport vehicle in the country, including information about the vehicle and its location. Similar to the permanent data table, a unique objectID is constructed for each item. A sample of the data stored in the transient data table can be seen below in Listing 2:

```
1  [
2    {
3      "objectType": "IrishRailTrain",
4      "latenessMessage": "On time",
5      "timestamp": "1742897696",
6      "trainDirection": "Southbound",
7      "trainStatus": "R",
8      "trainDetails": "09:41 - Maynooth to Grand Canal Dock ",
9      "trainType": "S",
10     "objectID": "IrishRailTrain-P656",
11     "averagePunctuality": "0",
12     "trainUpdate": "Departed Pelletstown next stop Broombridge",
13     "trainStatusFull": "Running",
14     "longitude": "-6.31388",
15     "trainPublicMessage": "P656\\n09:41 - Maynooth to Grand Canal Dock (0 mins late)\\nDeparted
       ↪ Pelletstown next stop Broombridge",
16     "trainPunctuality": "0",
17     "trainPunctualityStatus": "on-time",
18     "trainTypeFull": "Suburban",
19     "trainDate": "25 Mar 2025",
20     "latitude": "53.3752",
21     "trainCode": "P656"
22   },
23   {
24     "objectType": "Bus",
```

```
25        "busScheduleRelationship": "SCHEDULED",
26        "timestamp": "1742908007",
27        "busID": "V598",
28        "busRoute": "4538_90219",
29        "busRouteAgencyName": "Bus Éireann",
30        "objectID": "Bus-V598",
31        "busRouteLongName": "Galway Bus Station - Derry (Magee Campus Strand Road)",
32        "longitude": "-8.50166607",
33        "busDirection": "1",
34        "busStartDate": "20250325",
35        "busRouteShortName": "64",
36        "latitude": "54.2190742",
37        "busTripID": "4538_114801",
38        "busStartTime": "10:30:00"
39    },
```

Listing 2: Sample of the various types of items stored in the transient data table

There are only two types of objects stored in the transient data table: Irish Rail Trains and Buses. There is no per-vehicle data provided in the Luas API, and thus no way to track the live location of Luas trams. For the two types of objects stored in the transient data table, additional fields are added beyond what is returned by their respective APIs (and beyond the `objectType` & `objectID` fields) to augment the data.

The following additional pieces of data are added to each `IrishRailTrain` object:

- The `trainStatus` & `trainType` fields are single-character codes returned by the API, representing longer strings; for example a `trainStatus` of "R" indicates that the train is *running*. To avoid having to construct these strings on the frontend, the fields `trainStatusFull` & `trainTypeFull` are automatically added to the record when the data is retrieved.

- The Irish Rail API compacts much of its interesting data into a single field: `trainPublicMessage`. This field contains the `trainCode` (which is also supplied individually in its own field by the API), a string containing details about the train's origin & terminus, a string describing how late the train is, a string containing an update about the train's current whereabouts, all separated by \\n characters. This string is parsed into several additional fields to prevent additional computation on the frontend, including:

    - `latenessMessage`: a human-readable string which describes whether a train is early, late, or on time.
    - `trainDetails`: a string describing the train service itself, its start time, origin, & terminus.
    - `trainUpdate`: a string containing an update about the current whereabouts of the train, such as what station it last departed and what station it will visit next.
    - `trainPunctuality`: an integer which represents how many minutes late the train is (where a negative number indicates that the train is that many minutes early).
    - `trainPunctualityStatus`: a whitespace-free field which gives the same information as `latenessMessage` but for use in filtering rather than information presentation to the user. While one of these fields could be derived from the other on the frontend, the extra computation necessary when repeated for multiple trains and multiple users dwarfs the few extra bytes in the database to store the data in the machine-readable and human-readable forms.

- The `averagePunctuality` field is a field which contains the average recorded value of the `trainPunctuality` for trains with that `trainCode` in the database, thus giving a predictor of how early or late that particular train usually is.

The following additional pieces of data are added to each `Bus` object:

- `busRouteAgencyName`.

- `busRouteShortName`.

- `busRouteLongName`.

These details are not included in the response from the GTFS API, but can be obtained by looking up the given `busRoute` attribute in the permanent data table to find out said information about the bus route. In a fully-normalised relational database, this would be considered data duplication, but storing the data in both places allows for faster querying as no "joins" need to be performed.

Since the primary role of the transient data table is to provide up-to-date location data about various public transport services, each item in the table is given a `timestamp` attribute. This `timestamp` attribute is a UNIX timestamp in seconds which uniquely identifies the batch in which this data item was obtained. Each train & bus obtained in the same batch have the same `timestamp`, making querying for the newest data in the table more efficient. Because the data is timestamped, old data does not have to be deleted, saving both the overhead of deleting old data every time new data is fetched, and allowing an archive of historical data to be built up over time.

Since the primary type of query to be run on this table will be queries which seek to return all the items of a certain `objectType` (or `objectTypes`) for the latest timestamp, it would be ideal if the primary key could be a combination of the two for maximum efficiency in querying; however, such a combination would fail to uniquely identify each record and thus would be inappropriate for a primary key. Instead, the primary key must be some combination of the `timestamp` attribute and the `objectID` attribute. It was decided that the partition key would be the `objectID` and the sort key to be the `timestamp` so that all the historical data for a given item could be retrieved efficiently. Equivalently, the partition key could be the `timestamp` and the sort key could be the `objectID` which would allow for queries of all items for a given timestamp, but this was rejected on the basis that such scenarios were covered by the introduction of a Global Secondary Index.

A **Global Secondary Index (GSI)** allows querying on non-primary key attributes by defining an additional partition and sort key from the main table[31]. Unlike a primary key, there is no requirement for a GSI to uniquely identify each record in the table; a GSI can be defined on any attributes upon which queries will be made. The addition of GSIs to a table to facilitate faster queries is analogous to **SELECT** queries on non-primary key columns in a relational database (and the specification of a sort key is analogous to a relational **ORDER BY** statement); the structured nature of a relational database means that such queries are possible by default, although an index must be created on the column in question for querying on that column to be *efficient* (such as with the SQL **CREATE INDEX** statement). In a No-SQL database like DynamoDB, this functionality does not come for free, and instead must be manually specified.

To facilitate efficient querying of items in the table by `objectType` and `timestamp`, a GSI was created with partition key `objectType` and sort key `timestamp`, thus making queries for the newest data on a public transport type as efficient as querying on primary key attributes. The downside of creating a GSI is the additional storage requirements, as DynamoDB implements GSIs by duplicating the data into a separate index: efficient for querying, but less so in terms of storage usage.

Average Punctuality by `objectID` Table

To give the user punctuality predictions based off the historical data stored for a given service, it's necessary that the average punctuality be calculated. The most obvious way to do this would be to calculate the average of the punctuality values for a given `objectID` in the transient data table every time data a new data item with that `objectID` is added to the transient data table. However, this would be greatly inefficient, as it would require scanning the entire table for each item uploaded to the table, greatly slowing down the fetching of new data and consuming vast amounts of DynamoDB read/write resources. It is also intractable, as the historical data archive in the transient table grows, it will become linearly more expensive to compute the average punctuality for an item.

Instead, it was decided that the average punctuality for an item would be stored in a table and updated as necessary. By storing the `objectID`, the `average_punctuality`, and the `count` of the number of records upon which this average is based, the mean punctuality for an item can be updated on an as-needed basis in an efficient manner. The

new mean value for an item can be calculated as:

$$\bar{x}_{\text{new}} = \frac{(\bar{x}_{\text{old}} \times c) + x}{c + 1}$$

where x is the punctuality value for a given item, $\bar{x}_{\text{old}}$ is the previous mean punctuality value for that item, c is the count of records upon which that mean was based, and $\bar{x}_{\text{new}}$ is the new mean punctuality value. By calculating the average punctuality in this way, the operation is $O(1)$ instead of $O(n)$, thus greatly improving efficiency.

```
[
  {
    "average_punctuality": "0.5",
    "count": "2",
    "objectType": "IrishRailTrain",
    "objectID": "IrishRailTrain-P746"
  },
  {
    "average_punctuality": "-4",
    "count": "1",
    "objectType": "IrishRailTrain",
    "objectID": "IrishRailTrain-A731"
  },
  {
    "average_punctuality": "9.333333333333333333333333333",
    "count": "3",
    "objectType": "IrishRailTrain",
    "objectID": "IrishRailTrain-E112"
  },
]
```

Listing 3: Sample of items from the average punctuality by `objectID` table

At the time of writing, Irish Rail is the only Irish public transport provider to offer any kind of punctuality data in their public APIs, and therefore, this table only stores items with `"objectType": "IrishRailTrain"`. It could be argued that including this value in the table is therefore redundant, as it can be inferred, but the decision was made to include this additional value to make the table expandable and updatable. If another transport provider were to begin to offer punctuality data via their API, this table would require no updates to start including, for example, bus punctuality data. If the `objectType` were not included, this table would have to be replaced with a re-structured table in the event that a new category of public transport items were to be added.

In the same vein as including the `objectType` in each record, the primary key for this table was created with partition key `objectType` and sort key `objectID`, like in the permanent data table. This means that if an additional type of public transport were to be added to the table, querying based on that `objectType` would be fast & efficient by default. Since the primary key of a table cannot be changed once the table has been created, not using the `objectType` in the primary key would mean that adding an additional public transport type to the table would require deleting the table and starting again, or at the very least the creation of an otherwise unnecessary GSI to facilitate efficient querying.

Punctuality by `timestamp` Table

To provide historical insights such as punctuality trends over time, it is necessary to keep a record of the average punctuality for each timestamp recorded in the database. Similarly to the punctuality by `objectID` table, it is more efficient to calculate this value and store it than to calculate the average for every item in the table as the data is needed. Unlike the punctuality by `objectID` table, however, the average punctuality value for a `timestamp` need never be updated, as the average is calculated for each data upload run.

```
1   [
2     {
3       "average_punctuality": "0.8823529411764706",
4       "timestamp": "1742908007"
5     },
6     {
7       "average_punctuality": "1.0625",
8       "timestamp": "1742905796"
9     }
10  ]
```

Listing 4: Sample of items from the average punctuality by `timestamp` table

The partition key for this table is the `timestamp` value, and there is no need for a sort key or secondary index.

3.2   API Design

To make the data available to the frontend application, a number of API endpoints are required so that the necessary data can be requested as needed by the client. AWS offers two main types of API functionality with Amazon API Gateway[24]:

- **RESTful APIs:** for a request/response model wherein the client sends a request and the server responds, stateless with no session information stored between calls, and supporting common HTTP methods & CRUD operations. AWS API Gateway supports two types of RESTful APIs[28]:

    - **HTTP APIs:** low latency, fast, & cost-effective APIs with support for various AWS microservices such as AWS Lambda, and native CORS support[a], but with limited support for usage plans and caching. Despite what the name may imply, these APIs default to HTTPS and are RESTful in nature.

    - **REST APIs:** older & more fully-featured, suitable for legacy or complex APIs requiring fine-grained control, such as throttling, caching, API keys, and detailed monitoring & logging, but with higher latency, cost, and more complex set-up & maintenance.

- **WebSocket APIs:** for real-time full-duplex communication between client & server, using a stateful session to maintain the connection & context.

It was decided that a HTTP API would be more suitable for this application for the low latency and cost-effectiveness. The API functions needed for this application consist only of requests for data and data responses, so the complex feature set of AWS REST APIs is not necessary. The primary drawback of not utilising the more complex REST APIs is that HTTP APIs do not natively support caching; this means that every request must be processed in the backend and a data response generated, meaning potentially slower throughput over time. However, the fact that this application relies on the newest data available to give accurate & up-to-date location information about public transport means that the utility of caching is somewhat diminished, as the cache will expire and become out of date within minutes or even seconds of its creation. This combined with the fact that HTTP APIs are 3.5× cheaper[25] than REST APIs resulted in the decision that a HTTP API would be more suitable.

It is important to consider the security of public-facing APIs, especially ones which accept query parameters: a malicious attacker could craft a payload to either divert the control flow of the program or simply sabotage functionality. For this reason, no query parameter is ever evaluated as code or blindly inserted into a database query; any interpolation of query parameters is done in such a way that they are not used in raw query strings but in **parameterised expressions** using the `boto3` library[29]. This means that query parameters are safely bound to named placeholder attributes in

---

[a]**Cross-Origin Resource Sharing (CORS)** is a web browser security feature that restricts web pages from making requests to a different domain than the one that served the page, unless the API specifically allows requests from the domain that served the page[55]. If HTTP APIs did not natively support CORS, the configuration to allow requests from a given domain would have to be done in boilerplate code in the Lambda function that handles the API requests for that endpoint, and duplicated for each Lambda function that handles API requests.

queries rather than inserted into raw query strings and so the parameters have no potential for being used to control the structure or logic of the query itself. The AWS documentation emphasises the use of parameterised queries for database operations, in particular for SQL databases which are more vulnerable, but such attacks can be applied to any database architecture[30]. This, combined with unit testing of invalid API query parameters means that the risk of malicious parameter injection is greatly mitigated (although never zero), as each API endpoint simply returns an error if the parameters are invalid.

Figure 3.2: CORS configuration for the HTTP API

The Cross-Origin Resource Sharing (CORS) policy accepts only `GET` requests which originate from `http://localhost: 5173` (the URL of the locally hosted frontend application) to prevent malicious websites from making unauthorised requests on behalf of users to the API. While the API handles no sensitive data, it is nonetheless best practice to enforce a CORS policy and a "security-by-default" approach so that the application does not need to be secured retroactively as its functionality expands. If the frontend application were moved to a publicly available domain, the URL for this new domain would need to be added to the CORS policy, or else all requests would be blocked.

3.2.1 API Endpoints

`/return_permanent_data[?objectType=IrishRailStation,BusStop,LuasStop]`

The `/return_permanent_data` endpoint accepts a comma-separated list of `objectType` query parameters, and returns a JSON response consisting of all items in the permanent data table which match those parameters. If no query parameters are supplied, it defaults to returning *all* items in the permanent data table.

`/return_transient_data[?objectType=IrishRailTrain,Bus]`

The `/return_transient_data` endpoint accepts a comma-separated list of `objectType` query parameters, and returns a JSON response consisting of all the items in the transient data table which match those parameters *and* were uploaded to the transient data table most recently, i.e., the items which have the newest `timestamp` field in the table. Since the `timestamp` pertains to the batch of data uploaded to the table in a single run, each item in the response will have the same `timestamp` as all the others. If no `objectType` parameter is supplied, it defaults to returning all items from the newest upload batch.

`/return_historical_data[?objectType=IrishRailTrain,Bus]`

The `/return_historical_data` endpoint functions in the same manner as the `/return_transient_data` endpoint, with the exception that it returns matching items for *all* `timestamp` values in the table, i.e., it returns all items of the given `objectTypes` in the transient data table.

`/return_luas_data?luasStopCode=<luas_stop_code>`

The `/return_luas_data` returns incoming / outgoing tram data for a given Luas stop, and is just a proxy for the Luas real-time API. Since the Luas API returns data only for a queried station and does not give information about individual vehicles, the Luas data for a given station is only fetched on the frontend when a user requests it, as there is no

information to plot on the map beyond a station's location. However, this request cannot be made from the client to the Luas API, as the Luas API's CORS policy blocks requests from unauthorised domains for security purposes; this API endpoint acts as a proxy, accepting API requests from the `localhost` domain and forwarding them to the Luas API, and subsequently forwarding the Luas API's response back to the client.

This endpoint requires a single `luasStopCode` query parameter for each query to identify the Luas stop for which incoming / outgoing tram data is being requested.

/return_station_data?stationCode=<station_code>

The `return_station_data` returns information about the trains due into a given station in the next 90 minutes. This data is only shown to a user if requested for a specific station, so it is not stored in a DynamoDB table. Like the `/return_luas_data` endpoint, it too is just a proxy for an (Irish Rail) API, the CORS policy of which blocks requests from any unauthorised domain for security purposes. It requires a single `stationCode` query parameter for each query to identify the train station for which the incoming train data is being requested.

/return_punctuality_by_timestamp[?timestamp=<timestamp>]

The `/return_punctuality_by_timestamp` returns the contents of the `punctuality_by_timestamp` DynamoDB table. It accepts a comma-separated list of `timestamps`, and defaults to returning the average punctuality for *all* `timestamps` in the table if no `timestamp` is specified.

/return_all_coordinates

The `/return_all_coordinates` endpoint returns a JSON array of all current location co-ordinates in the transient data table for use in statistical analysis.

3.3 Serverless Functions

All the backend code & logic is implemented in a number of **serverless functions**[22], triggered as needed. Serverless functions are small, single-purpose units of code that run in the cloud and reduce the need to manage servers & runtime environments. In contrast to a server program which is always running, serverless functions are event-driven, meaning that they are triggered by events such as API calls or invocations from other serverless functions and do not run unless triggered. Each serverless function is *stateless*, which means that each function invocation is independent and that no state data is stored between calls; they are also *ephemeral*, starting only when triggered and stopping when finished, running only for a short period of time. This means that they automatically scale up or down depending on usage, and because they only run when they need to, they are much cheaper in terms of compute time than a traditional server application.

AWS Lambda[32] is a serverless compute service provided by Amazon Web Services that allows for the creation of serverless functions without provisioning or managing servers. A Python AWS Lambda function typically consists of a single source code file with specified entrypoint function, the name of which can vary, but is typically called `lambda_handler()`. They can be created and managed via the GUI AWS Management Console[33] or via the AWS CLI tool[26]. Each Lambda function can be configured to have a set memory allocation, a timeout duration (how long the function can run for before being killed), and environment variables.

Often, when a serverless application replaces a legacy mainframe application, the time & memory needed to perform routine tasks is drastically reduced because it becomes more efficient to process events individually as they come rather than batching events together to be processed all at once; the high computational cost of starting up a mainframe system means that it's most efficient to keep it running and to batch process events. For this reason, serverless functions often require very little memory and compute time. This application, however, is somewhat unusual as it requires the processing of quite a large amount of data at once: status updates for public transport don't come from the source APIs individually for each vehicle on an event-by-event basis, but in batches of data that are updated regularly. Therefore, the serverless functions for this application will require more memory and more compute time to complete. In this

context, memory and compute time have an inversely proportional relationship: more memory means more items can be processed quickly, thus reducing computational time, and less memory means that fewer items can be processed quickly, thus increasing the computational time.

One common approach to tuning the configuration of AWS Lambda functions is to use **AWS Lambda Power Tuning**[5], an open-source tool that is designed to help optimise the memory & power configurations for AWS Lambda functions. It works by invoking the function to be tuned multiple times across various memory allocations, recording metrics such as execution duration and cost for each configuration, and visualises the trade-off between cost and performance for each tested memory configuration, allowing the user to decide the most suitable memory allocation based on minimising cost, maximising speed, or balancing the two. While this is a very powerful & useful tool for Lambda function optimisation, it was not used in this project in order to (somewhat ironically) manage costs and remain with the AWS Free Tier: running the tuner involves several invocations of the target Lambda function at various memory levels, and a typical tuning run involves dozens of Lambda invocations. With the amount of data being written to the database per Lambda run (thousands of items), this would quickly exceed the free tier and begin incurring costs. While these costs would not be prohibitively high, doing so would change the nature of the project from researching & implementing the optimal approach for this application to paying for a faster & more performant application. The tuner *could* be run with database writes disabled, but this would not generate meaningful results for the functions as writing to the DynamoDB database is the critical choke point for functions in this application.

Instead, each function was manually tuned to consume the least amount of resources (reasonably) possible by gradually incrementing the memory allocation until the function could run to completion in what was deemed to be a reasonable amount of time (somewhat subjectively) for this application; for this reason, the functions could almost certainly achieve superior execution times if they were allocated more resources, but this would bring the application closer and closer to exceeding the free tier limit of 400,000 GB-seconds (allocated memory in GB $\times$ execution time in seconds). As this is a student project, it was decided that the backend should be optimised to avoid incurring costs: while it wouldn't be prohibitively expensive to pay for additional resources, it was deemed inappropriate for this project as it does not demonstrate ability in the field of computer science but ability in the field of entering one's bank card details. In a business setting, the costs of running AWS Lambda Power Tuning would be completely negligible (in the order of fractions of cents per function invocation), and would pay for itself in the money saved via function optimisation; if this project were not a student project, there is no doubt that AWS Lambda Power Tuning would be the correct way to go about optimising the function configurations.

fetch_permanent_data

The `fetch_permanent_data` Lambda function is used to populate the permanent data table. As the data in question changes rarely if ever, this function really need only ever be triggered manually, such as when a new train station is opened or a new bus route created. However, for the sake of completeness and to avoid the data being out of date, a schedule was created with **Amazon EventBridge** to run the function every 28 days to ensure that no changes to the data are missed. Like all other schedules created for this application, the schedule is actually disabled at present to avoid incurring unnecessary AWS bills, but can be enabled at any time with the click of a button.
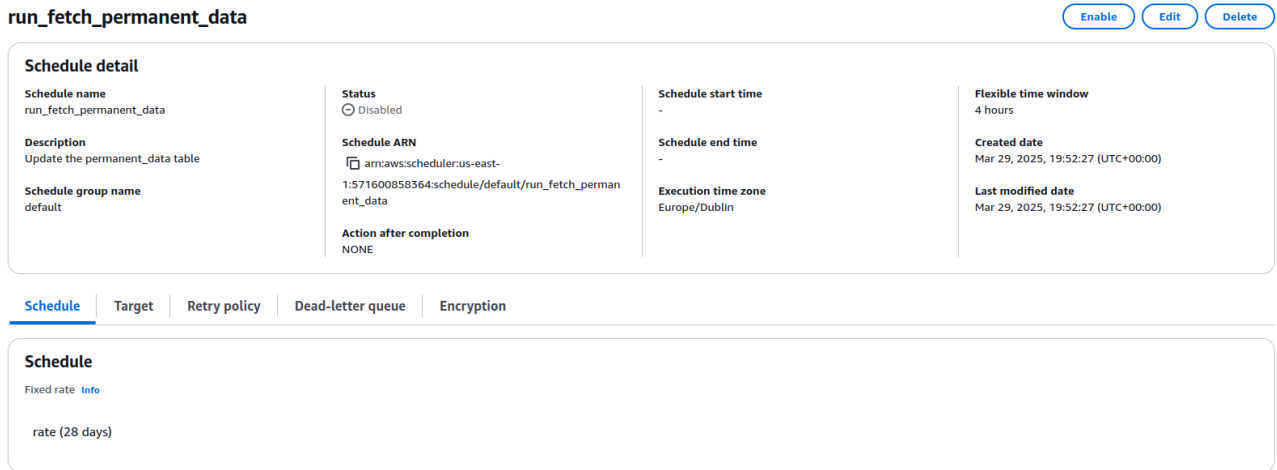
Figure 3.3: Screenshot of the Amazon EventBridge schedule to run the `fetch_permanent_data` Lambda function

The `fetch_permanent_data` function retrieves Irish Rail Station data directly from the Irish Rail API, but Luas stop data and bus data are not made available through an API; instead, the Luas stop data is made available online in a tab-separated `TXT` file, and the bus stop & bus route data are available online in comma-separated `TXT` files distributed as a single ZIP file. This makes little difference to the data processing however, as downloading a file from a server and parsing its contents is little different in practice from downloading an API response from a server and parsing its contents. The function runs asynchronously with a thread per type of data being fetched (train station data, Luas stop data, and bus stop & route data), and once each thread has completed, batch uploads the data to the permanent data table, overwriting its existing contents.

fetch_transient_data

The `fetch_transient_data` function operates much like the `fetch_permanent_data` function, but instead updates the contents of the transient data table. It runs asynchronously, with a thread per API being accessed to speed up execution; repeated requests to an API within a thread are made synchronously to avoid overloading the API. For example, retrieving the type (e.g., Mainline, Suburban, Commuter) of the trains returned by the Irish Rail API requires three API calls: the Irish Rail API allows the user to query for all trains or for trains of a specific type but it does not return the type of the train in the API response. Therefore, if a query is submitted for all trains, there is no way of knowing which train is of which type. Instead, the function queries each type of train individually, and adds the type into the parsed response data.

Additionally, the `return_punctuality_by_objectID` function is called when processing the train data so that each train's average punctuality can be added to its data for upload. Somewhat unintuitively, it transpired that the most efficient way to request this data was to request all data from the punctuality by `objectID` data table rather than individually request each necessary `objectID`; this means that much of the data returned is redundant, as many of the trains whose punctualities are returned are not running at the time and so will not be uploaded, but it means that the function is only run once, and so only one function invocation, start-up, database connection, and database query have to be created. It's likely that if bus punctuality data were to become available in the future, this approach would no longer be the most efficient way of doing things, and instead a `return_punctuality_by_objectType` function would be the optimal solution.

The bus data API doesn't return any information about the bus route beyond a bus route identifier, so the permanent data table is queried on each run to create a dictionary (essentially a Python hash table[17]) linking bus route identifiers to information about said bus route (such as the name of the route). As the bus data is being parsed, the relevant bus route data for each vehicle is inserted. Once all the threads have finished executing, the data is uploaded in a batch to the transient data table, with each item timestamped to indicate which function run it was retrieved on.

This function is run as part of an **AWS Step Function** with a corresponding Amazon EventBridge schedule (albeit disabled at present). A step function is an AWS service which facilitates the creation of state machines consisting of various AWS microservices to act as a single workflow. The state machine allows multiple states and transitions to

be defined, with each state representing a step in the workflow and the transitions representing how the workflow moves from one state to another and what data is transferred. Step functions have built-in error handling and retry functionality, making them extremely fault-tolerant for critical workflows.
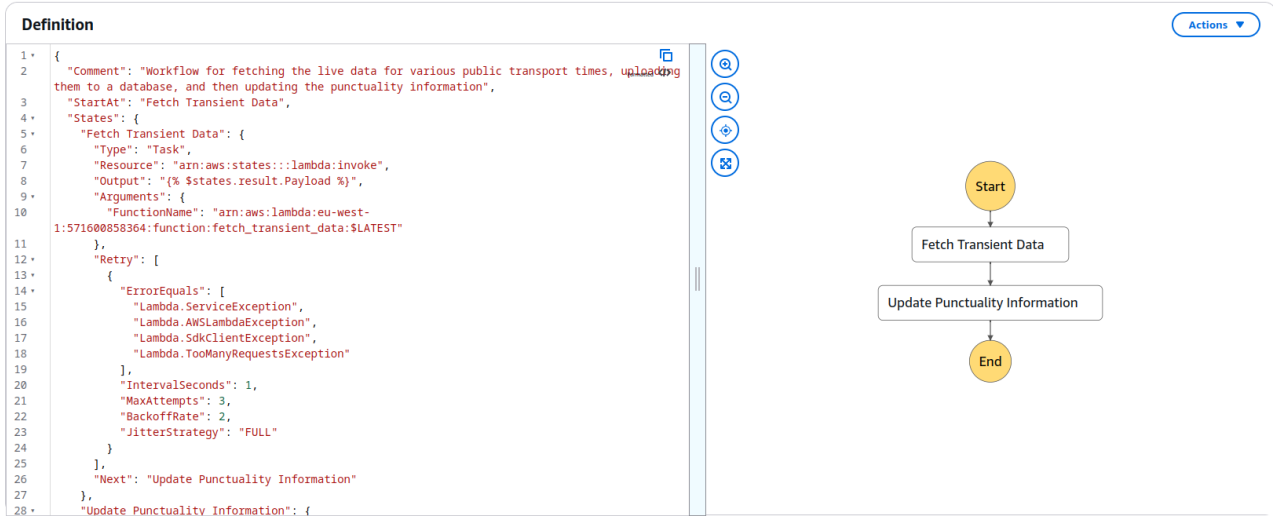

Figure 3.4: Screenshot of the `get_live_data` step function definition

The step function runs the `fetch_transient_data` function and then runs the `update_average_punctuality` function, if and only if the `fetch_transient_data` function has completed successfully. This allows the average punctuality data to be kept up to date and in sync with the transient data, and ensures that they do not become decoupled and therefore incorrect. This step function is triggered by a (currently disabled) Amazon EventBridge schedule which runs the function once a minute, which is the maximum frequency possible to specify within a cron schedule, and suitable for this application as the APIs from which the data is sourced don't update much more frequently than that. Furthermore, the data from which bus data is sourced will time out if requests are made too frequently, so this value was determined to be appropriate after testing to avoid overwhelming the API or getting timed-out. It is possible to run EventBridge schedules even more frequently using the *rate-based schedule* schedule type instead of the *cron-based schedule* schedule type but a more frequent schedule would be inappropriate for this application.

update_average_punctuality

The `update_average_punctuality` function runs after `fetch_transient_data` in a step function and populates the average punctuality by `objectID` and average punctuality by `timestamp` tables to reflect the new data collected by `fetch_transient_data`. For each item in the new data, it updates the average punctuality in the average punctuality by `objectID` table according to the aforementioned formula:

$$\bar{x}_{\text{new}} = \frac{(\bar{x}_{\text{old}} + c) + x}{c + 1}$$

As the function iterates over each item, it adds up the total punctuality and then divides this by the total number of items processed before adding it to the average punctuality by `timestamp` table, where the `timestamp` in question is the `timestamp` that the items were uploaded with (the `timestamp` of the `fetch_transient_data` run which created them).

There are a number of concerns that one might reasonably have about using the mean punctuality for the average displayed to users:

- Means are sensitive to outliers, meaning that if, for example, a train is very late just once but very punctual the rest of the time, its average punctuality could be misleading.

- The punctuality variable is an integer that can be positive or negative, which could have the result that the positive & negative values could cancel each other out for a train that is usually either very late or very early, giving the misleading impression of an average punctuality close to zero.

- Considering the entire history of a train for its average punctuality may not be reflective of recent trends: a train may historically have been consistently late, but become more punctual as of late and therefore the average punctuality may not be reflective of its recent average punctuality.

These questions were carefully considered when deciding how to calculate the average punctuality, but it was decided that the mean would nonetheless be the most appropriate for various reasons:

- The mean lends itself to efficient calculation with the $O(1)$ formula described above. No other average can be calculated in so efficient a manner: the median requires the full list of punctualities to be considered to determine the new median, and the mode requires at the very least a frequency table of all the punctualities over time to determine the new mode, which requires both additional computation and another DynamoDB table.

- Similarly, considering only recent data would destroy the possibility for efficient calculation: the mean could not be updated incrementally, and instead a subset of the historic punctualities would have to be stored and queried for each update.

- The outlier sensitivity is addressed by the sheer number of items that are considered for the mean: since this will be updated every minute of every day, an outlier will quickly be drowned out with time.

- Finally, the average is being calculated so that it can be shown to the user and so that they can make decisions based off it. The average person from a non-technical or non-mathematical background tends to assume that any average value is a mean value, and so it would only serve to confuse users if they were given some value that did not mean what they imagined it to mean. While calculating additional different measures of averages would be possible, displaying them to the user would likely be at best not useful and at worst confusing, while also greatly increasing the computation and storage costs. This aligns with the second of Nielsen's famous *10 Usability Heuristics for User Interface Design*, which were consulted throughout the design process: "**Match between the System and the Real World:** The design should speak the users' language. Use words, phrases, and concepts familiar to the user, rather than internal jargon"[41].

For these reasons, it was decided that the mean was the most suitable average to use.

return_permanent_data

The `return_permanent_data` function is the Lambda function which is called when a request is made from the client to the `/return_permanent_data` API endpoint. It checks for a comma-separated list of `objectType` parameters in the query parameters passed from the API event to the Lambda function, and scans the permanent data table for every item matching those `objectTypes`. If none are provided, it returns every item in the table, regardless of type. It returns this data as a JSON string.

When this function was first being developed, the permanent data table was partitioned by `objectID` alone with no sort key, meaning that querying was very inefficient. When the table was re-structured to have a composite primary key consisting of the `objectType` as the partition key and the `objectID` as the sort key, the `return_permanent_data` function was made 10× faster: the average execution time was reduced from ∼10 seconds to ∼1 second, demonstrating the critical importance of choosing the right primary key for the table.

return_transient_data

The `return_transient_data` function is the Lambda function which is called when a request is made from the client to the `/return_transient_data` API endpoint. Like `return_permanent_data`, it checks for a comma-separated list of `objectType` parameters in the query parameters passed from the API event to the Lambda function, and scans the permanent data table for every item matching those `objectTypes`. If none are provided, it returns every item in the table, regardless of type.

Similar to `return_permanent_data`, when this function was originally being developed, there was no GSI on the transient data table to facilitate efficient queries by `objectType` and `timestamp`; the addition of the GSI and updating the code to exploit the GSI resulted in an average improvement in run time of ∼8×, thus demonstrating the utility which GSIs can provide.

`return_punctuality_by_objectID`

The `return_punctuality_by_objectID` function is invoked by the `fetch_transient_data` function to return the contents of the punctuality by `objectID` table. It accepts a list of `objectID`s and defaults to returning all items in the table if no parameters are provided.

`return_punctuality_by_timestamp`

The `return_punctuality_by_timestamp` function is similar to `return_punctuality_by_objectID` but runs when invoked by an API request to the `/return_punctuality_by_timestamp` endpoint and simply returns a list of JSON objects consisting of a `timestamp` and an `average_punctuality`. It is used primarily to graph the average punctuality of services over time.

`return_all_coordinates`

The `return_all_coordinates` function is used to populate the co-ordinates heatmap in the frontend application which shows the geographical density of services at the present moment. It accepts no parameters, and simply scans the transient data table for the newest items and returns their co-ordinates.

`return_historical_data`

The `return_historical_data` function operates much like the `return_transient_data` function, accepting a list of `objectTypes` or defaulting to all `objectTypes` if none are specified, with the only difference being that this function does not consider the `timestamps` of the data and just returns all data in the transient data table. This function, along with its corresponding API endpoint exist primarily as a debugging & testing interface, although they also give a convenient access point for historical data analysis should that be necessary.

`return_luas_data`

The `return_luas_data` function is a simple proxy for the Luas API which accepts requests from the client and forwards them to the Luas API to circumvent the Luas API's restrictive CORS policy which blocks requests from unauthorised domains. It simply accepts a `luasStopCode` parameter, and makes a request to the Luas API with said parameter, parses the response from XML into JSON, and returns it.

`return_station_data`

Like `return_luas_data`, the `return_station_data` is a proxy for the Irish Rail API so that requests can be made as needed from the client's browser to get data about incoming trains due into a specific section without running afoul of Irish Rail's CORS policy. It also accepts a single parameter (`stationCode`) and makes a request to the relevant endpoint of the Irish Rail API, and returns the response (parsed from XML to JSON).

Chapter 4

Frontend Design & Implementation

The frontend design is built following the Single-Page-Application (SPA)[47] design pattern using the React Router[46] library, meaning that the web application loads a single HTML page and dynamically updates content as the user interacts with the application, without reloading the webpage. Since there is just one initial page load, the content is dynamically updated via the DOM using JavaScript rather than by requesting new pages from the server; navigation between pseudo-pages is managed entirely using client-side routing for a smoother & faster user experience since no full-page reloads are necessary.

The web application is split into three "pages":

- The home (or map) page, which is the main page that displays live location data & service information to the user. This page is where the user will spend the majority of their time, and where the majority of the functionality is delivered.

- The statistics page, which is used to deliver statistical insights about the data to the user. This page is for providing deeper insights into the stored data on a collective basis, rather than on a per-service basis.

- The help page, which gives some basic usage information on how to complete basic tasks within the application, in keeping with Nielsen's usability heuristic of "Help and Documentation"[41].

The web application follows the Container/Presentational Pattern[21], which enforces separation of concerns by separating the presentational logic from the application logic. This is done by separating the functionality into two classes of components:

- **Container Components:** those which fetch the data, process it, and pass it to the presentational components which are contained by their container components, i.e., the presentational components are children of the container components.

- **Presentational Components:** those which display the data it receives from the container components to the user as it is received. This makes the components highly re-usable, as there isn't specific data handling logic within them.

React components are reusable, self-contained pieces of the UI which act as building blocks for the application[50]; they can receive properties from their parent components, manage their own internal state, render other components within themselves, and respond to events.

The Container/Presentational pattern can be contrasted with other design patterns for UI design such as the Model-View-Controller (MVC) pattern[45]: in many ways, the containers of the Container/Presentational pattern as a collective can be thought of as analogous to the controller of the MVC pattern, and the presentational components as analogous to the view. The key difference between the two patterns is that the Container/Presentational pattern defines only the architecture of the frontend layer, and does not dictate how the backend ought to be laid out; the MVC pattern defines the architecture of the entire application and so the backend (the model) is necessarily *tightly coupled*[39] with the frontend layer (the view) by means of the controller. Therefore, updating the backend code will most likely necessitate updating

the frontend code (and vice-versa). For this reason, MVC is most commonly used for applications in which the backend & the frontend are controlled by the same codebase, and especially for application in which the frontend rendering is done server-side. The Container/Presentational pattern, however, is *loosely coupled*[39] with the backend and therefore the frontend & the backend can be updated almost independently of one another as long as the means & format of data transmission remain unchanged, thus making development both faster & easier, and mitigating the risk of introducing breaking changes. The Container/Presentational pattern lends itself particularly well to React development, as React code is rendered client-side and makes extensive use of components: the Container/Presentational pattern just ensures that this use of components is done in a way that is logical & maintainable.

4.1 Main Page
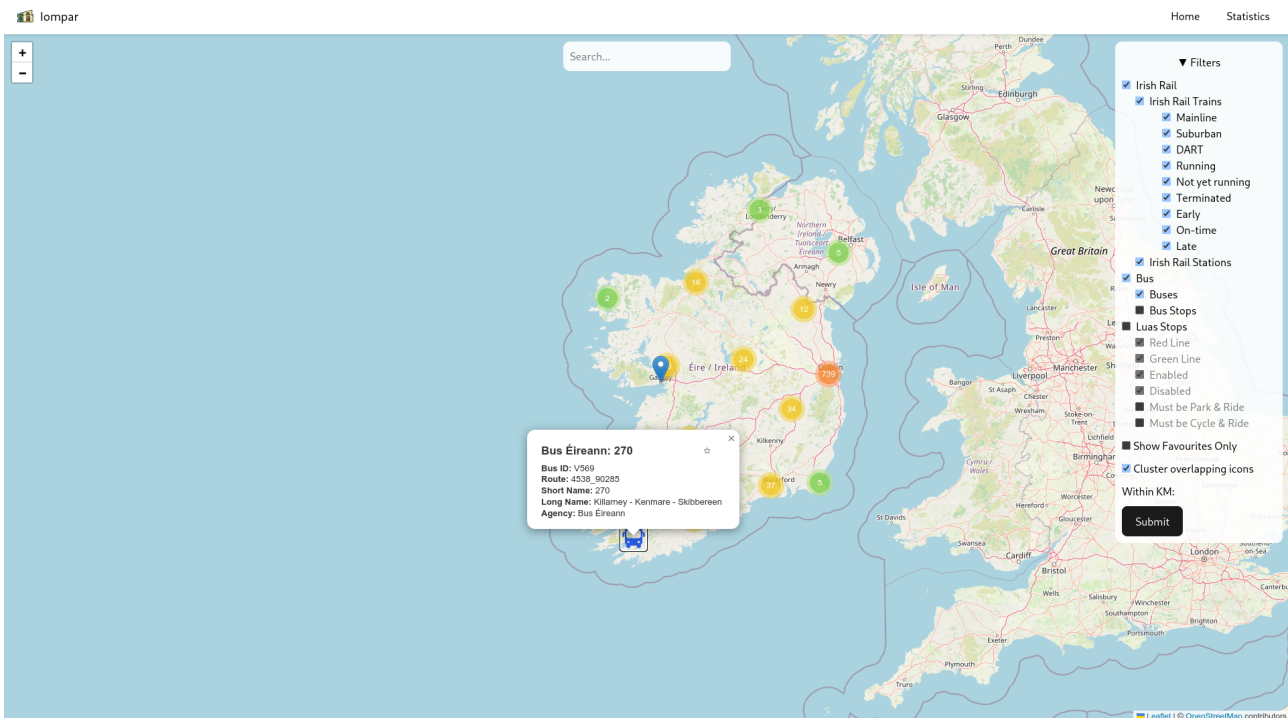
Figure 4.1: Screenshot of the front page of the web application

The main page of the application contains a map upon which the live location of each public transport object is marked, a navigation bar, a search bar to filter the vehicles shown on the map, and a filters side-panel which allows the user to select what kind of objects the user wants displayed on the map. The location markers for each item are clickable, and when clicked, display a pop-up with relevant information about the selected item. In line with Container/Presentational pattern, the main App.jsx file contains the data retrieval & processing logic, and passes the processed data to presentational components for display.

Data Retrieval & Processing

The data is retrieved and processed in a function named fetchData(). When the "Submit" button is clicked in the filters side-panel, the function is triggered and sends asynchronous API requests to the /return_permanent_data and /return_transient_data API endpoints as necessary. The App.jsx file contains a JavaScript object which identifies what checkboxes in the side-panel pertain to objectTypes in the backend, and which API endpoint they can be sourced from. When the function is fetching data, it constructs a comma-separated list of selected objectTypes to send to each API endpoint, thereby making only one request to each API endpoint to avoid unnecessary repeated queries.

Originally, the returned data was processed using JavaScript functional array methods[10] (very superficially similar to the Java Stream[13] interface), which allow for the creation of chained operations on arrays, not dissimilar to chained shell operations using the pipe (|) operator on UNIX-like systems[23]. Methods like .map(), .reduce(), etc. are some of

the methods covered in JavaScript functional array methods which allow data processing pipelines to be specified in a clean, elegant, & readable manner. However, the price of this elegance is slower and less efficient execution: these array methods add extra layers of abstraction, invoke a new function per element, create intermediate arrays for each result, and (unlike their Java counterparts) do not support short-circuiting and therefore cannot break early (with the exception of the `.some()` & `.many()` methods)[48]. The modern JavaScript engine also is less efficient at optimising these array methods than simple **for**-loops[43]. Indeed, re-writing the function to use a simple **for** loop resulted in an average loading speed increase of ~2 seconds when all data sources were selected.

While the data is loading, a loading overlay with a "spinner" icon is displayed on-screen which prevents the user from interacting with the UI, keeps the user informed of what's going on (in keeping with the first of Nielsen's 10 usability heuristics[41]), prevents the user from changing any settings as the data is loaded that may cause inconsistent behaviour, and gives the perception of a smoother loading experience. Studies have shown that displaying a loading screen to the user instead of just waiting for the UI to update gives the impression of a faster loading speed, so much so that even longer load times with a loading screen feel faster than shorter load times without one[11,44,54].

Once the JSON response is received, each item in the response is iterated over, as described above, and the marker for each item is constructed. The map plotting is done with Leaflet[2], and each marker on the map consists of an array of co-ordinates, some HTML pop-up content when the marker is clicked, an icon depending on the transport type, a Boolean display variable which dictates whether or not the marker is displayed, and some markup-free `markerText` which is not part of the Leaflet API[2] but added to each icon to give it some text to which the search text can be compared. This `markerText` is never displayed to the user, and is used solely for search & filtering purposes; it is created as an additional variable instead of just using the pop-up content for searching as the pop-up content contains HTML mark-up which would have to be removed at search time to make searching possible, leading to a large number of unnecessary & costly string-processing operations. Each item has a colour-coded icon depending on the type of data it represents:

Icon	Description
20	On-time & early, running Irish Rail trains
20	Late, running Irish Rail trains
20	Not-yet running & terminated Irish Rail trains
19	On-time & early, running DARTs
19	Late, running DARTs
19	Not-yet running & terminated DARTs
40	Irish Rail stations
1	Buses
15	Bus stops
3	Red line Luas stops
3	Green line Luas stops

Table 4.1: Marker icons & their descriptions

The value of the Boolean `display` variable for each function is calculated before any pop-up content or `markerText` is generated, so as not to waste computational time generating information that will not be shown to the user. Each `objectType` has its own Boolean formula that determines whether or not an item should be shown to the user, depending on what filters they have selected. Each formula is constructed in such a way that expensive computations are avoided if they are unnecessary, by taking advantage of JavaScript's ability to **short-circuit** logical expressions, that is, return immediately if the left-hand side of a Boolean `AND` is false[8] or if the left-hand side of a Boolean `OR` is true[9].

$$
\begin{aligned}
\text{display}_{\text{train}} = \quad & (\text{latitude} \neq 0) \wedge (\text{longitude} \neq 0) \\
& \wedge \big((\text{showMainline} \wedge (\text{trainType} = \text{Mainline})) \\
& \quad \vee (\text{showSuburban} \wedge (\text{trainType} = \text{Suburban})) \\
& \quad \vee (\text{showDart} \wedge (\text{trainType} = \text{DART}))\big) \\
& \wedge \big((\text{showRunning} \wedge (\text{trainStatus} = \text{Running})) \\
& \quad \vee (\text{showNotYetRunning} \wedge (\text{trainStatus} = \text{Not yet running})) \\
& \quad \vee (\text{showTerminated} \wedge (\text{trainStatus} = \text{Terminated}))\big) \\
& \wedge \big(((\text{trainStatus} = \text{Running}) \wedge \\
& \quad ((\text{showEarly} \wedge (\text{trainPunctualityStatus} = \text{early})) \\
& \quad\quad \vee (\text{showOnTime} \wedge (\text{trainPunctualityStatus} = \text{On time})) \\
& \quad\quad \vee (\text{showLate} \wedge (\text{trainPunctualityStatus} = \text{late})))) \\
& \quad \vee ((\text{trainStatus} = \text{Not yet running}) \wedge \text{showNotYetRunning}) \\
& \quad \vee ((\text{trainStatus} = \text{Terminated}) \wedge \text{showTerminated})\big) \\
& \wedge \big((\text{withinDistance} \wedge \text{userLocationAvailable}) \Rightarrow \\
& \quad \text{haversineDistance}(\text{userLocation}, [\text{latitude}, \text{longitude}]) < \text{withinDistance}\big) \\
& \wedge (\text{showFavouritesOnly} \Rightarrow \text{trainCode} \in \text{favourites})
\end{aligned}
$$

$$
\begin{aligned}
\text{display}_{\text{station}} = \quad & (\text{latitude} \neq 0) \wedge (\text{longitude} \neq 0) \\
& \wedge \big((\text{withinDistance} \wedge \text{userLocationAvailable}) \Rightarrow \\
& \quad \text{haversineDistance}(\text{userLocation}, [\text{latitude}, \text{longitude}]) < \text{withinDistance}\big) \\
& \wedge (\text{showFavouritesOnly} \Rightarrow \text{trainStationCode} \in \text{favourites})
\end{aligned}
$$

$$
\begin{aligned}
\text{display}_{\text{bus}} = \quad & (\text{latitude} \neq 0) \wedge (\text{longitude} \neq 0) \\
& \wedge \big((\text{withinDistance} \wedge \text{userLocationAvailable}) \Rightarrow \\
& \quad \text{haversineDistance}(\text{userLocation}, [\text{latitude}, \text{longitude}]) < \text{withinDistance}\big) \\
& \wedge (\text{showFavouritesOnly} \Rightarrow \text{busRoute} \in \text{favourites})
\end{aligned}
$$

$$
\begin{aligned}
\text{display}_{\text{bus stop}} = \quad & (\text{latitude} \neq 0) \wedge (\text{longitude} \neq 0) \\
& \wedge \big((\text{withinDistance} \wedge \text{userLocationAvailable}) \Rightarrow \\
& \quad \text{haversineDistance}(\text{userLocation}, [\text{latitude}, \text{longitude}]) < \text{withinDistance}\big) \\
& \wedge (\text{showFavouritesOnly} \Rightarrow \text{busStopID} \in \text{favourites})
\end{aligned}
$$

$$
\begin{aligned}
\text{display}_{\text{Luas stop}} = \quad & (\text{latitude} \neq 0) \land (\text{longitude} \neq 0) \\
& \land \big((\text{showGreenLine} \land (\text{luasLine} = \text{Green Line})) \\
& \quad \lor (\text{showRedLine} \land (\text{luasLine} = \text{Red Line}))\big) \\
& \land \big((\text{showEnabled} \land (\text{luasStopIsEnabled} = 1)) \\
& \quad \lor (\text{showDisabled} \land (\text{luasStopIsEnabled} = 0))\big) \\
& \land \big(\neg\text{showCycleAndRide} \lor (\text{showCycleAndRide} \land (\text{luasStopIsCycleAndRide} = 1))\big) \\
& \land \big(\neg\text{showParkAndRide} \lor (\text{showParkAndRide} \land (\text{luasStopIsParkAndRide} = 1))\big) \\
& \land \big((\text{withinDistance} \land \text{userLocationAvailable}) \Rightarrow \\
& \quad \text{haversineDistance}(\text{userLocation}, [\text{latitude, longitude}]) < \text{withinDistance}\big) \\
& \land (\text{showFavouritesOnly} \Rightarrow \text{luasStopID} \in \text{favourites})
\end{aligned}
$$

Although the formulae may appear excessively complex at first glance, they're relatively easy to understand when broken down: each line is essentially a conjunction of a variable that dictates whether or not a filter is applied and a check to see if that condition is fulfilled, e.g., (showMainline $\land$ (trainType = Mainline)) checks if the "Show Mainline" filter is applied and if the item in question fulfils that criterion. Each line of logic is either in conjunction or disjunction with the preceding line depending on whether or not that the filter is a *permissive filter* (that is, it increases the number of items shown) or if it is a *restrictive filter* (that is, it decreases the number of items shown).

As can be seen in the preceding formulae, if the user location is available and the user has specified that all items must be within a certain distance of their location, each marker is checked to see if it's within range by using **Haversine distance**, which calculates the shortest distance between two points on the surface of a sphere. The Haversine distance formula is defined as follows[6]:

$$
d = 2r \cdot \arcsin\left(\sqrt{\sin^2\left(\frac{\Delta\phi}{2}\right) + \cos\left(\phi_1\right)\cos\left(\phi_2\right)\sin^2\left(\frac{\Delta\lambda}{2}\right)}\right)
$$

where:

- d is the distance between the two points,

- r is the radius of the Earth ($\sim$6,371 kilometres[52]),

- ϕ_1, ϕ_2 are the latitudes of each point in radians,

- λ_1, λ_2 are the longitudes of each point in radians,

- $\Delta\phi = \phi_2 - \phi_1$, and

- $\Delta\lambda = \lambda_2 - \lambda_1$.

This formula was chosen as a good middle ground between accurate distance calculations and efficient computation. There are other distance formulae, such as Vincenty's formula[53] which is more accurate (because it considers the Earth to be an oblate spheroid rather than sphere) but slower to compute, or Euclidean distance[49] which is fast to compute, but inaccurate for points on the surface of the Earth (because it considers the points to be on flat 2D grid). Arguably, one could get away with using Euclidean distance, as the distances considered in this application are relatively short (covering only the island of Ireland) and in general, a user will be specifying only short distances to see services in their immediate vicinity. However, this approach cannot be scaled if the application scope was increased to include a broader geographical area as inaccuracies would rapidly build up, and even over the island of Ireland alone, there is potential for inconsistent behaviour due to measurement inaccuracies. Haversine distance is not too computationally expensive, and guarantees a reliable degree of accuracy for the user.
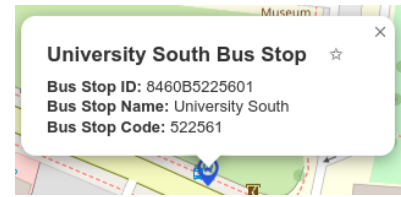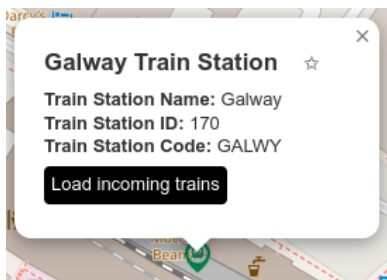
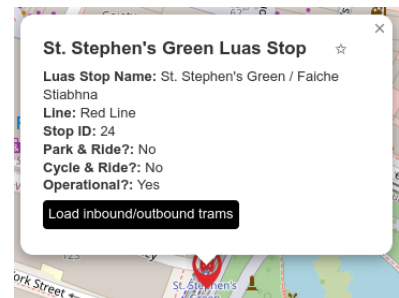**Marker Pop-Ups**
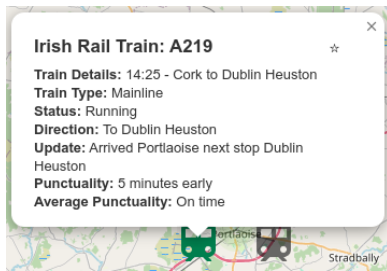
(a) `IrishRailTrain` pop-up
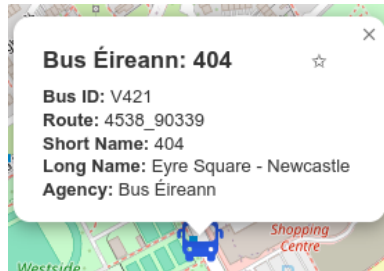
(b) `Bus` pop-up

(c) `BusStop` pop-up



(d) `IrishRailStation` pop-up

(e) `LuasStop` pop-up

Figure 4.2: The 5 pop-up types

When the data is being fetched, a pop-up is created for each new marker collected, to be displayed when the marker is clicked. For each `objectType`, there is a separate React component for its pop-up to which the item's data is passed so that it can be rendered differently depending on the service type. The `IrishRailTrain` pop-up, `Bus` pop-up, & `BusStop` are the most similar and the simplest, as they simply display information passed into their respective components, the only variance being that different `objectTypes` have different fields to display.

Each pop-up has a "star" button which, when clicked, adds the selected item to a list of the user's "favourite" services, which is stored as a cookie in the user's browser. This cookie is loaded when the main page is loaded and updated every time an item is toggled to be "starred" or not. This list of favourites is used to determine whether or not the item is displayed when the user selects "Show favourites only", as can be seen in the Boolean display formulae previously outlined. Favouriting behaves differently depending on the `objectType` of the item being favourited: notably, buses are favourited not on a per-vehicle basis using the item's `objectID`, but on a per-route basis. This means that if a user favourites, for example, a 401 Bus Éireann bus, every bus on this route will appear when the user applies the "Show favourites only" filter. This makes the favourites feature far more useful than it would be otherwise: users are generally interested not in a specific bus vehicle, but a specific bus route.
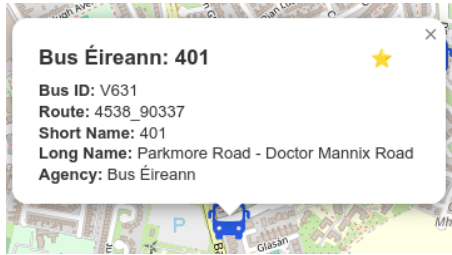
(a) Bus pop-up with bus "favourited"
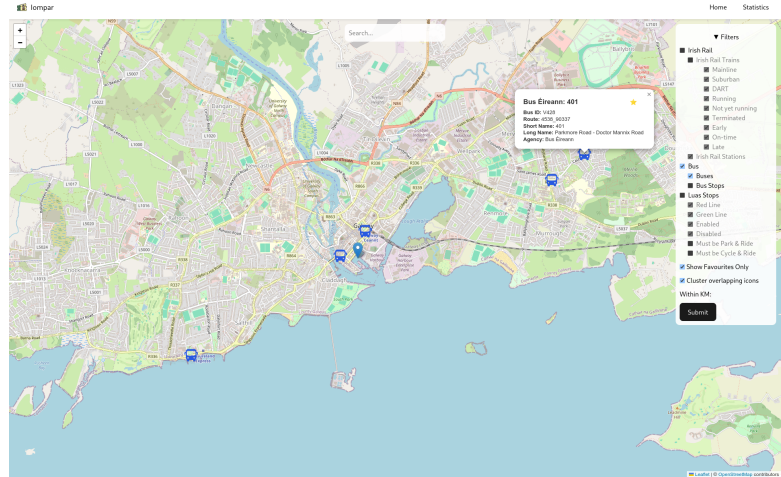
(b) Buses of the same route appearing when "favourites only" filter applied

Figure 4.3: Demonstration of the "favourite" functionality with buses

Train station pop-ups & Luas stop pop-ups also have an additional button which, when clicked, fetches data about the services relating to that station. This is necessary for Luas stops, as Luas data is only available on a per station basis, and just an extra feature for train stations, as train data is plotted on the map regardless. The "Load inbound/outbound trams" button on the Luas stop pop-up fetches data from the `/return_luas_data` API endpoint which contains the inbound and outbound trams due into that stop. The "Load incoming trains" button on the train station pop-up fetches data from the `/return_station_data` API endpoint which contains the trains due into that station in the next 90 minutes. This information is then displayed in the pop-up itself, with a scrollbar if the information is too long to display in the pop-up without it becoming unmanageably long.
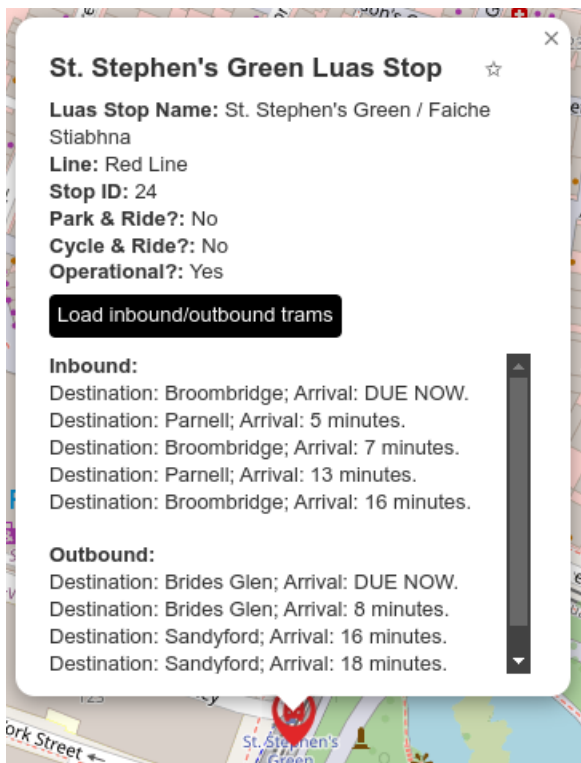
(a) Results of clicking "Load inbound/outbound trams"
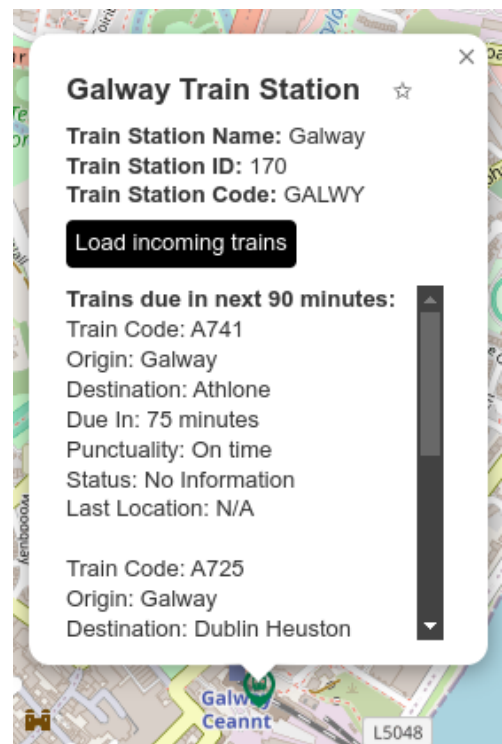
(b) Results of clicking "Load incoming trains"

Figure 4.4: Demonstration of the "favourite" functionality with buses
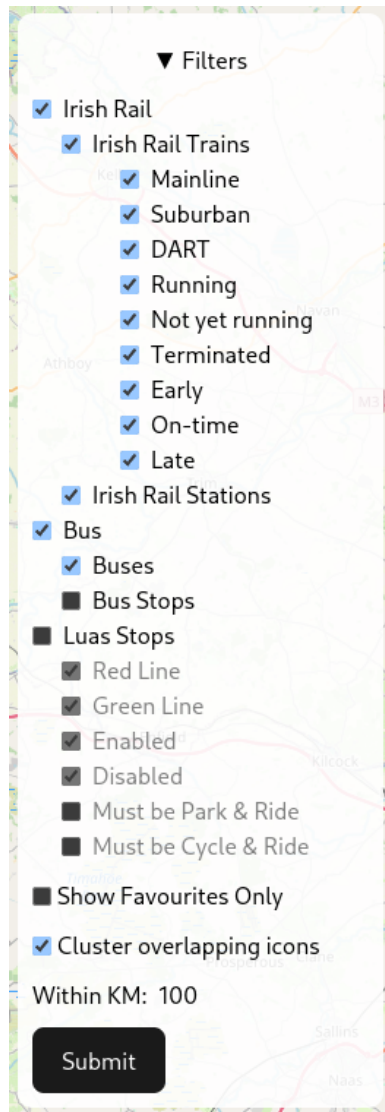
Filters Side-Panel

Figure 4.5: Screenshot of the filters side-panel

The Boolean filter variables mentioned in the display formulae above are selected by the user in the filters side-panel, implemented as a separate (presentational) component. The component appears in the top-right hand corner of the screen, and can be minimised while browsing the map. When a user clicks "Submit", the `fetchData()` function is invoked with the selected filters, and these selected filters are saved as a cookie to the user's browser; this cookie is loaded when the main page is first loaded, so that a user's last used filters are pre-selected for them when they open the application. Even the value of the "Within KM" number is stored as a cookie and loaded in with page load so that a user's experience is a seamless as possible between uses.

The first three top-level filters determine what data sources are used: Irish Rail, Bus, and/or Luas Stops. If a top-level or second-level filter is deselected, all its child filters are greyed out and cannot be interacted with until all its parents are selected. By default, if no cookie is stored for a user, the top-level filters are all deselected and all of their children are selected, making it as easy as possible for the user to select what they want to filter by. The only exception to this are the "Must be Park & Ride" and the "Must be Cycle & Ride" filters underneath "Luas Stops": these filters are more restrictive, and hide any item that does not match them, and so are not selected by default.

The "Cluster overlapping icons" option does not change what data is shown, but how it is shown; by default, any items whose icons overlap one another's at the current zoom level are *clustered* into a single icon that displays the number

of items which are contained within it using a Leaflet plug-in called `Leaflet.markercluster`[37]. This option toggles this behaviour, which may be desirable depending on the user's preferences, especially when only few items are being displayed in a small geographical area.
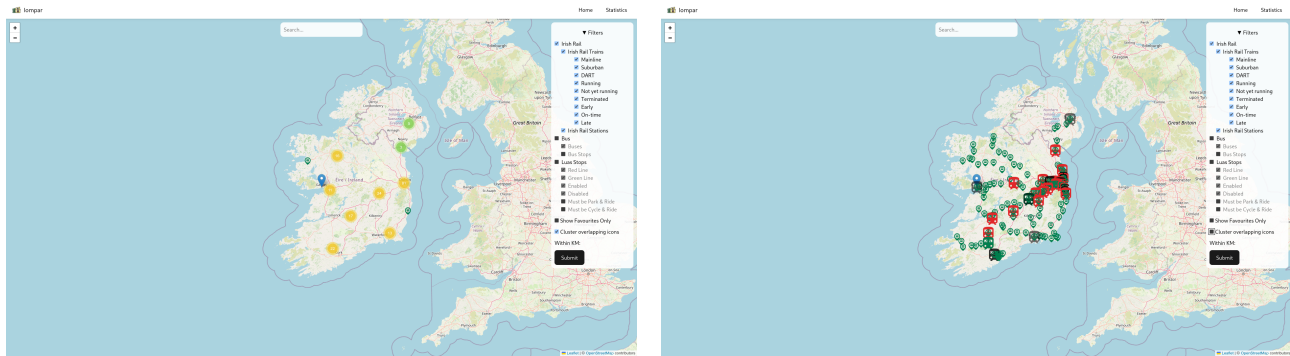
(a) Screenshot of the effects of enabling clustering

(b) Screenshot of the effects of disabling clustering

Figure 4.6: Effects of enabling/disabling clustering

The "Within KM" option accepts a number input value of the range within which an item must be for it to be displayed. This option only appears if the user's geographical location is available: if their browser does not support the option, or if they have rejected location access to the application, the option will not appear. The option can be reset to showing all items regardless of range by deleting the input to the textbox; if the user enters a value that doesn't make sense, such as a number ≤ 0, the application defaults to displaying all items regardless of distance, on the assumption that the user is either attempting to reset the filter by entering "0" or that they have accidentally entered a nonsensical value.

A subject of much deliberation in the design process was the behaviour of the "Submit" button. It's important that the map only updates when the user asks it to so as to avoid situations where the user is looking at an item, the page refreshes automatically, and it disappears from view; therefore, fetching new data had to be a manually-specified process, rather than occurring automatically. The question of whether or not the filters should be applied as soon as they are selected or only upon a "Submit" also received much consideration: automatically updating them would minimise the amount of user interactions necessary, but to re-calculate the display value for every item when there are many being displayed could take a few seconds, and make the application unresponsive as the user makes many changes to the selected filters. Therefore, it was decided that the filters should only be applied when the user requests that they be applied, so that they can decide what filters to apply in a convenient & usable way. Finally, there was the question of optimisations: if the data sources selected did not change but a display filter did (for example, the user didn't change that they wanted to look at Irish Rail Trains, but added a selection to specify only Mainline trains), should another request be made to the API endpoints? Not requesting new data would mean a faster loading time and a more responsive program, at the cost of showing the user potentially out-of-date data. It was decided that each new click of "Submit" should always fetch new data, to align the program as much as possible with the user's mental model of the program, in keeping with the HCI principle of **conceptual model alignment**[42]: the behaviour of the application should match how the user imagines it to behave. Users are familiar with the "make selections, then confirm" paradigm, and will expect the submit button to do the same thing each time (which aligns with Nielsen's usability heuristic of "Consistency & Standards" and of "speaking the user's language"[41]).

Search Bar

The search bar allows the user to further refine the displayed items beyond what is possible with the filters side-panel by specifying text that should be present in the marker. The search is based upon a `markerText` variable that is constructed for each marker as the data is fetched, consisting of the pop-up content without any formatting, upper-case characters, or non-alphanumeric characters. Similarly, entered search terms are made lower-case and stripped of non-alphanumeric characters to facilitate searching, and an item is deemed to match the search term if the search term occurs in the `markerText`. This approach is relatively simplistic as search algorithms go, but is fast & efficient, and more than suitable for this application.
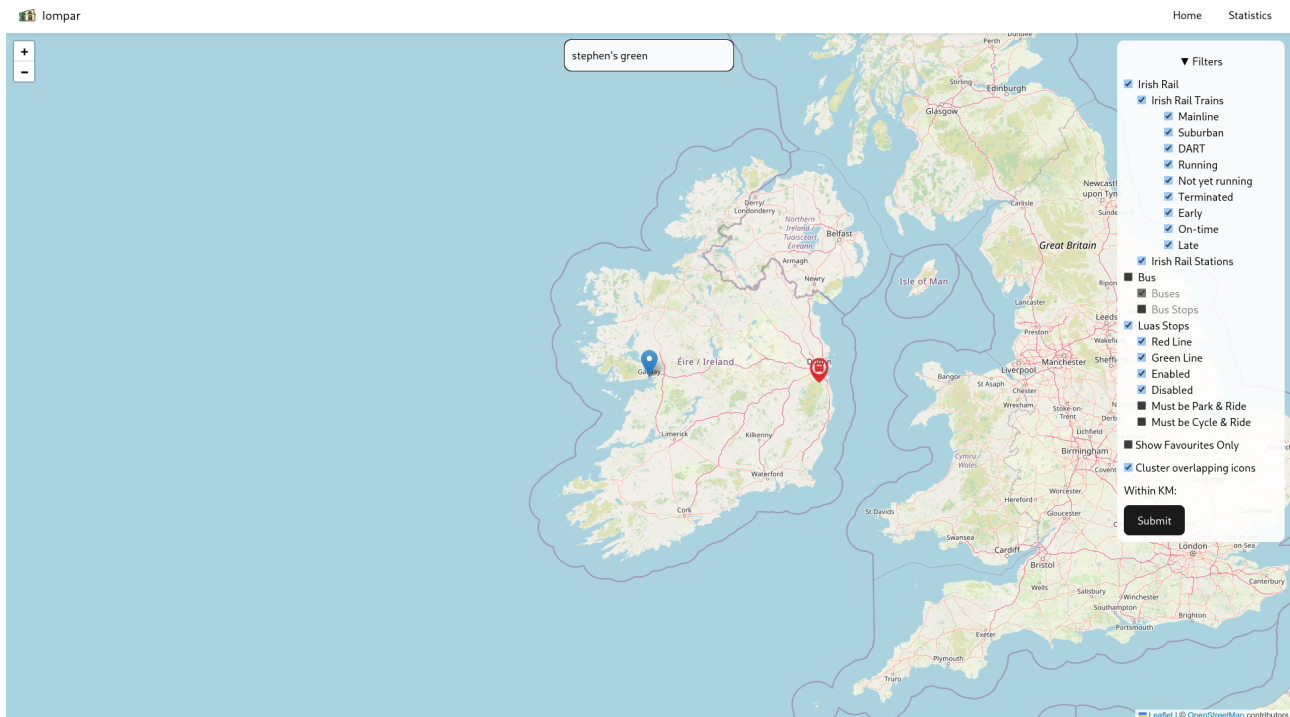
Figure 4.7: Screenshot of the search bar being used

Unlike the filters panel, the displayed items are automatically filtered as the user enters text, which is facilitated by the search-based filtering being separated from and less computationally complex than the side-panel based filtering. This allows the results to be quickly filtered in a responsive manner by the user, and makes the completion of tasks generally faster & more responsive. However, updating the filtered items for every keystroke that the user enters would be wasteful, as each key entry further refines their search and they would type it regardless, and constantly re-computing which items match the search term for each keypress would result in the application becoming less responsive. To prevent this, the search bar employs a **debounce function**[36] which waits for a very short interval after the user's last keypress to start searching for the entered text; the search will only commence once the user has finished typing. Experiments with different values on different users indicated that 300 milliseconds is the ideal value for this application: long enough that it won't be triggered until the user pauses typing, but short enough that the gap between the user stopping typing and the search being applied is nearly imperceptible.

However, if very large amounts of data are being displayed, such as in the event that the user has selected all data sources, the search can take a noticeable amount of time and make the UI sluggish & unresponsive as the search is executed. To address this, if the number of items being displayed is so high that it will induce noticeably slow loading times, the debounce time is increased to 400 milliseconds to be more careful in avoiding unnecessary computations, and the loading overlay is displayed as the filtering is performed to prevent the user from being exposed to a sub-optimally performant UI.

The search function makes use of the `useMemo`[51] React hook to cache the results of filtering the markers based off the search text, and making it so that it will only be re-calculated if the search term changes or if the markers change. Without `useMemo`, every re-render of the page would cause the filter function to run even if nothing had changed, which would be a huge waste of computational resources.

Map

The map component itself is the presentational component in which all of the mapping & plotting functionality is performed, implemented using the Leaflet[2] mapping libraries and map tiles from OpenStreetMap[12]. It receives the markers to be displayed, a Boolean determining whether clustering is enabled, and the geolocation of the user, if available. If the user's geolocation is available, a "You are here" marker is added to the map and the map is centered on those co-ordinates; otherwise, the map centers on the geographical centre of the island of Ireland[35] and displays no

marker.

Each marker variable component passed to the map component is added to the map, and if clustering is enabled, is wrapped in a `MarkerClusterGroup` to cluster the icons together if they overlap.
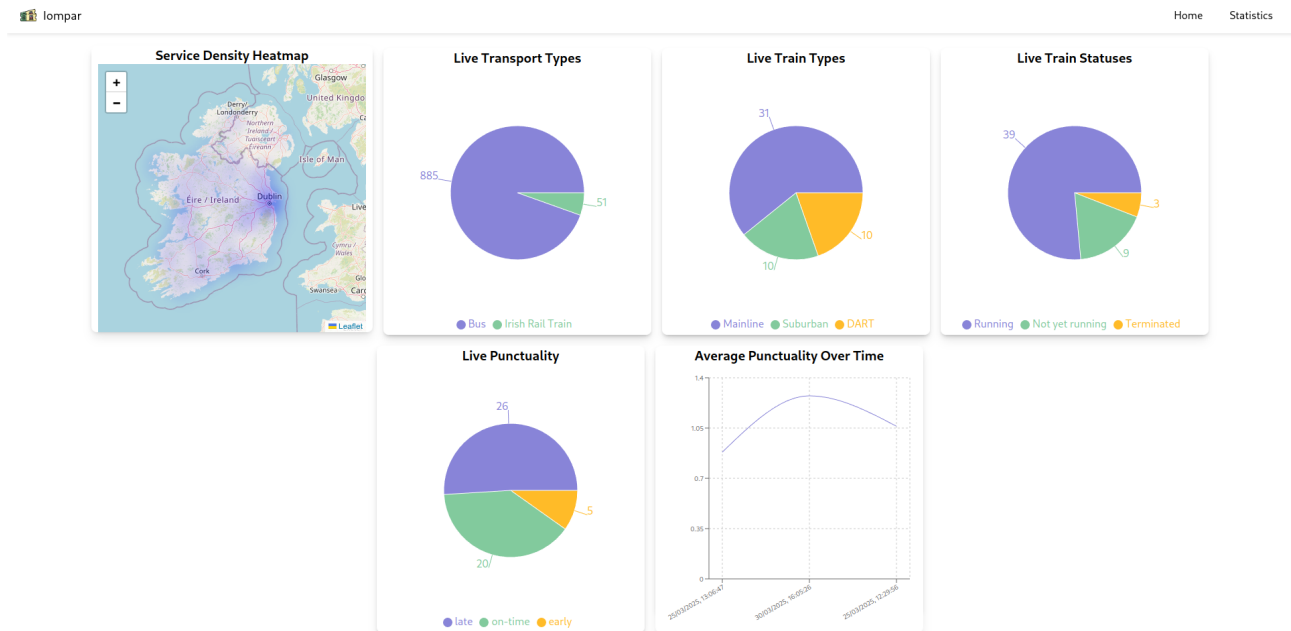
4.2 Statistics Page

Figure 4.8: Screenshot of the statistics page

The statistics page of the application exists to give the user a deeper insight into the data than simply live data. The grid layout is achieved with the use of Tailwind CSS flexbox[34]. It consists of 6 graphs, each displaying a different statistical insight about the application data:

- The Service Density Heatmap displays a heatmap depicting the geographical distribution of currently-running public transport services on the island of Ireland;

- The Live Transport Types pie chart displays the proportion of `objectTypes` for which there is currently live location data;

- The Live Train Types pie chart displays the proportion of mainline, suburban, & DART Irish Rail Trains currently in operation;

- The Live Train Statuses pie chart displays the proportion of running, not yet running, & terminated trains at the present moment;

- The Live Punctuality pie chart displays the proportion of late, on-time, & early services at the present moment.

- The Average Punctuality Over Time line graph displays how the average punctuality of services varies per collected `timestamp`.

Each pie chart uses the same re-usable presentation component which accepts some data and displays it. Pie charts are somewhat controversial in the realm of data visualisation and are often rejected in favour of bar charts, and are only recommended to be used under certain particular situations: when the data being illustrated is a part-to-whole representation, when there is no more than 5 categories in the pie chart, and when there are not small differences

between the categories[56,7,38]. Since the data for this application fulfils these criteria, and because testing with bar charts resulted with more difficult to understand results (as part of the proportion), pie charts were deemed suitable for this purpose.
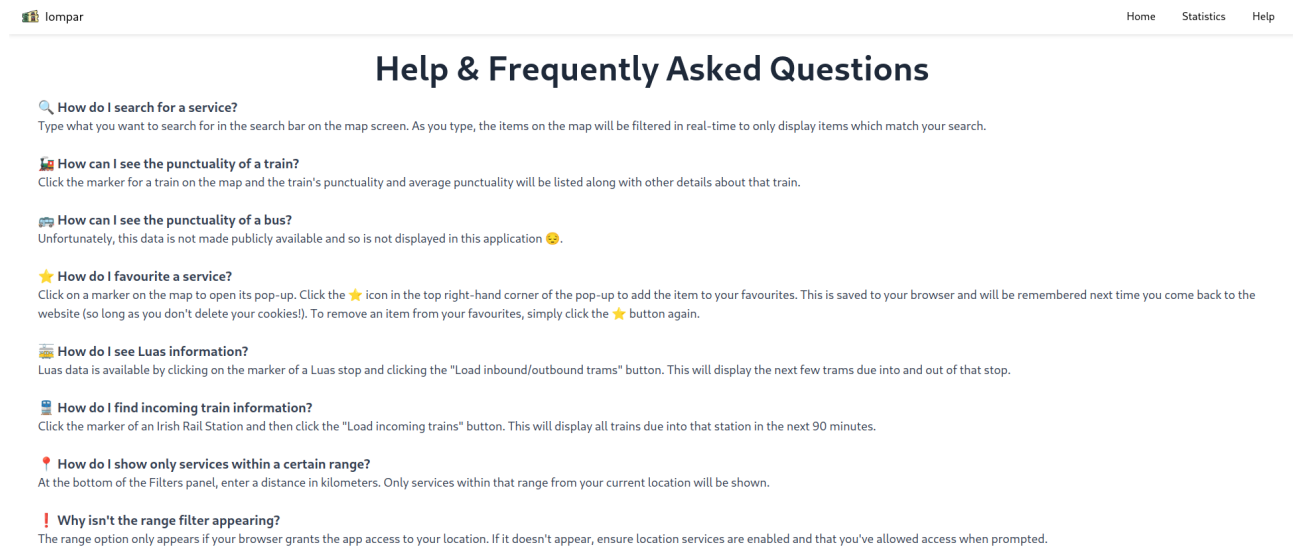
4.3 Help Page

Figure 4.9: Screenshot of the help page

The help page is a very simple web page that just contains questions and answers on how to perform basic tasks, such as favouriting an item or viewing real-time Luas information. It is intended to be user-friendly and easy to search through; this is in part facilitated through its minimalist design (achieved with TailwindCSS) which attempts to prevent the user from having to sift through irrelevant information to find out how to perform a basic task.

Chapter 5

Evaluation

5.1   Objectives Fulfilled

5.2   Heuristic Evaluation: Nielsen's 10

5.3   User Evaluation

Chapter 6

Conclusion

Bibliography

[1]   Hilmy Abiyyu A. *Bus free icon*. Flaticon. URL: https://www.flaticon.com/authors/hilmy-abiyyu-a Accessed 2025-03-26.

[2]   Volodymyr Agafonkin. *Leaflet API Reference*. 2023. URL: https://leafletjs.com/index.html Accessed 2025-03-28.

[3]   Iconic Artisan. *Tram station free icon*. Flaticon. URL: https://www.flaticon.com/authors/iconic-artisan Accessed 2025-03-26.

[4]   Gowri Balasubramanian and Sean Shriver. *Choosing the Right DynamoDB Partition Key*. AWS Database Blog. 2017. URL: https://aws.amazon.com/blogs/database/choosing-the-right-dynamodb-partition-key/ Accessed 2025-03-26.

[5]   Alex Casalboni. *AWS Lambda Power Tuning*. 2023. URL: https://github.com/alexcasalboni/aws-lambda-power-tuning.

[6]   Nitin R. Chopde and Mangesh K. Nichat. "Landmark Based Shortest Path Detection by Using A* Algorithm and Haversine Formula". In: *International Journal of Innovative Research in Computer and Communication Engineering* 1.2 (2013-04), pp. 298–302. ISSN: 2320-9798. URL: https://www.researchgate.net/publication/282314348.

[7]   Inc. Cloud Software Group. *What is a pie chart?* Spotfire. URL: https://www.spotfire.com/glossary/what-is-a-pie-chart Accessed 2025-03-29.

[8]   MDN Contributors. *Expressions and operators > Logical AND (&&)*. MDN Web Docs. 2025. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_AND Accessed 2025-03-28.

[9]   MDN Contributors. *Expressions and operators > Logical OR (||)*. MDN Web Docs. 2025. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_OR Accessed 2025-03-28.

[10] MDN Contributors. *References > JavaScript > Reference > Standard built-in objects > Array: Instance Methods*. MDN Web Docs. 2025. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#instance_methods Accessed 2025-03-28.

[11] MDN Contributors. *Web performance > Perceived performance*. MDN Web Docs. 2025. URL: https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Performance/Perceived_performance Accessed 2025-03-28.

[12] OpenStreetMap Contributors. *OpenStreetMap: Getting Help*. OpenStreetMap. URL: https://www.openstreetmap.org/help Accessed 2025-03-29.

[13] Oracle Corporation. `Interface Stream<T>`. Java™ Platform, Standard Edition 8 API Specification. 2025. URL: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html Accessed 2025-03-28.

[14] Iarnród Éireann. *Irish Rail Live Map*. 2024. URL: https://www.irishrail.ie/en-ie/train-timetables/live-train-map.

[15] Boris Farias. *Location free icon*. Flaticon. URL: https://www.flaticon.com/authors/boris-farias Accessed 2025-03-26.

[16] Foras na Gaeilge. *iompar*. *Foclóir Gaeilge–Béarla*. Accessed: 2025-04-01. Teanglann. 2025. URL: https://www.teanglann.ie/en/fgb/iompar.

[17]   Python Software Foundation. *5. Data Structures ¶ Dictionaries*. Python 3.12.2 Documentation. 2025. URL: `https://docs.python.org/3/tutorial/datastructures.html#dictionaries` Accessed 2025-03-26.

[18]   Freepik. *Ticket free icon*. Flaticon. URL: `https://www.flaticon.com/authors/freepik` Accessed 2025-03-26.

[19]   Freepik. *Tram Front View free icon*. Flaticon. URL: `https://www.flaticon.com/authors/freepik` Accessed 2025-03-26.

[20]   Google. *Train free icon*. Flaticon. URL: `https://www.flaticon.com/authors/google` Accessed 2025-03-26.

[21]   Lydia Hallie and Addy Osmani. *Container/Presentational Pattern*. patterns.dev. 2025. URL: `https://www.patterns.dev/react/presentational-container-pattern/` Accessed 2025-03-29.

[22]   Hassan B. Hassan, Saman A. Barakat, and Qusay I. Sarhan. "Survey on serverless computing". In: *Journal of Cloud Computing: Advances, Systems and Applications* 10.1 (2021), pp. 1–29. DOI: `10.1186/s13677-021-00253-7`. URL: `https://journalofcloudcomputing.springeropen.com/articles/10.1186/s13677-021-00253-7`.

[23]   Ken Hess. *Working with pipes on the Linux command line*. Red Hat Blog. 2019. URL: `https://www.redhat.com/en/blog/pipes-command-line-linux` Accessed 2025-03-28.

[24]   Amazon Web Services Inc. *Amazon API Gateway*. 2025. URL: `https://aws.amazon.com/api-gateway/` Accessed 2025-03-26.

[25]   Amazon Web Services Inc. *Amazon API Gateway Pricing*. Amazon API Gateway Developer Guide. 2025. URL: `https://aws.amazon.com/api-gateway/pricing/` Accessed 2025-03-26.

[26]   Amazon Web Services Inc. *AWS Command Line Interface*. AWS Developer Center. 2025. URL: `https://aws.amazon.com/cli/` Accessed 2025-04-02.

[27]   Amazon Web Services Inc. *AWS Free Tier*. Amazon DynamoDB Developer Guide. 2025. URL: `https://aws.amazon.com/free/?all-free-tier.sort-by=item.additionalFields.SortRank&all-free-tier.sort-order=asc&awsf.Free%20Tier%20Types=*all&awsf.Free%20Tier%20Categories=*all` Accessed 2025-04-02.

[28]   Amazon Web Services Inc. *Choose between REST APIs and HTTP APIs*. Amazon API Gateway Developer Guide. 2025. URL: `https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-vs-rest.html` Accessed 2025-03-26.

[29]   Amazon Web Services Inc. *Table / Action / query*. Boto3 1.27.0 documentation. 2023. URL: `https://boto3.amazonaws.com/v1/documentation/api/1.27.0/reference/services/dynamodb/table/query.html` Accessed 2025-03-26.

[30]   Amazon Web Services Inc. *Use parameterized queries*. Amazon Athena User Guide. 2025. URL: `https://docs.aws.amazon.com/athena/latest/ug/querying-with-prepared-statements.html` Accessed 2025-03-26.

[31]   Amazon Web Services Inc. *Using Global Secondary Indexes in DynamoDB*. Amazon DynamoDB Developer Guide. 2025. URL: `https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html` Accessed 2025-03-26.

[32]   Amazon Web Services Inc. *What is AWS Lambda?* Amazon Lambda Developer Guide. 2025. URL: `https://docs.aws.amazon.com/lambda/latest/dg/welcome.html` Accessed 2025-04-02.

[33]   Amazon Web Services Inc. *What is the AWS Management Console?* AWS Management Console Documentation. 2025. URL: `https://docs.aws.amazon.com/awsconsolehelpdocs/latest/gsg/what-is.html` Accessed 2025-04-02.

[34]   Tailwind Labs Inc. *Flexbox & Grid*. Tailwind CSS Documentation. URL: `https://tailwindcss.com/docs/flex` Accessed 2025-03-29.

[35]   Ordnance Survey Ireland. *Where is the centre of Ireland?* Tailte Éireann. URL: `https://web.archive.org/web/20231004185017/https://osi.ie/blog/where-is-the-centre-of-ireland/` Accessed 2025-03-29.

[36]   Aneeqa Khan. *Throttling and Debouncing - Explained*. Accessed: 2025-03-29. 2023. URL: `https://dev.to/aneeqakhan/throttling-and-debouncing-explained-1ocb`.

[37]   Dave Leaver. *Leaflet.markercluster*. Accessed: 2025-03-29. 2025. URL: `http://leaflet.github.io/Leaflet.markercluster/`.

[38] Lumel. *Pie chart 101: How to use & when to avoid them*. Inforiver. URL: `https://inforiver.com/insights/pie-chart-101-how-to-use-when-to-avoid-them/` Accessed 2025-03-29.

[39] William B. McNatt and James M. Bieman. "Coupling of Design Patterns: Common Practices and Their Benefits". In: *Proceedings of the Computer Software & Applications Conference (COMPSAC 2001)*. To appear. Fort Collins, CO, USA: IEEE, 2001. URL: `https://www.cs.colostate.edu/~bieman/Pubs/McnattBieman01.pdf` Accessed 2025-04-02.

[40] nawicon. *Train free icon*. Flaticon. URL: `https://www.flaticon.com/authors/nawicon` Accessed 2025-03-26.

[41] Jakob Nielsen. *10 Usability Heuristics for User Interface Design*. 1994. URL: `https://www.nngroup.com/articles/ten-usability-heuristics/` Accessed 2025-03-26.

[42] Donald A. Norman. *The Design of Everyday Things*. Revised and Expanded. New York, NY: Basic Books, 2013. ISBN: 9780465050659.

[43] Andriy Obrizan. *Performance of JavaScript .forEach, .map and .reduce vs for and for..of*. 2021. URL: `https://leanylabs.com/blog/js-forEach-map-reduce-vs-for-for_of/` Accessed 2025-03-28.

[44] Samantha Persson. "Improving Perceived Performance of Loading Screens Through Animation". Bachelor's thesis. Växjö, Sweden: Linnaeus University, 2019. URL: `https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1333185&dswid=-9045`.

[45] Trygve Reenskaug. *The Model-View-Controller (MVC): Its Past and Present*. 2003. URL: `https://citeseerx.ist.psu.edu/document?doi=4ef90a7b9c1b1cd02acf273694e4059a70c7d198` Accessed 2025-04-02.

[46] React Router. *Routing*. React Router API Reference. 2025. URL: `https://reactrouter.com/start/declarative/routing` Accessed 2025-03-27.

[47] React Router. *Single Page App (SPA)*. React Router API Reference. 2025. URL: `https://reactrouter.com/how-to/spa` Accessed 2025-03-27.

[48] Marius Schulz. *The `some()` and `every()` Array Methods in JavaScript*. 2016. URL: `https://mariusschulz.com/blog/the-some-and-every-array-methods-in-javascript` Accessed 2025-03-28.

[49] Karl Smith. *Precalculus: A Functional Approach to Graphing and Problem Solving*. Jones & Bartlett Publishers, 2013, p. 8. ISBN: 978-0-7637-5177-7.

[50] Meta Open Source. *Component*. React API Reference. 2025. URL: `https://react.dev/reference/react/Component` Accessed 2025-03-28.

[51] Meta Open Source. *useMemo*. 2023. URL: `https://react.dev/reference/react/useMemo`.

[52] Donald L. Turcotte and Gerald Schubert. "The Earth's Radius". In: *Geodynamics*. 2nd ed. New York: Springer, 1992, pp. 61–74. DOI: `10.1007/978-1-4612-4434-9_4`. URL: `https://link.springer.com/chapter/10.1007/978-1-4612-4434-9_4`.

[53] Thaddeus Vincenty. "Direct and Inverse Solutions of Geodesics on the Ellipsoid with Application of Nested Equations". In: *Survey Review* 23.176 (1975-04), pp. 88–93. ISSN: 0039-6265.

[54] Gojko Vladić et al. "Analysis of the Loading Animation Performance and Viewer Perception". In: *GRID: Graphic Engineering and Design* 11.2 (2020). Professional paper, pp. 667–675. DOI: `10.24867/GRID-2020-p76`. URL: `https://www.researchgate.net/publication/346892743_Analysis_of_the_loading_animation_performance_and_viewer_perception`.

[55] World Wide Web Consortium. *Cross-Origin Resource Sharing*. W3C Proposed Edited Recommendation. 2020-06. URL: `https://www.w3.org/TR/2020/SPSD-cors-20200602/`.

[56] Mike Yi. *A Complete Guide to Pie Charts*. Atlassian. URL: `https://www.atlassian.com/data/charts/pie-chart-complete-guide` Accessed 2025-03-29.