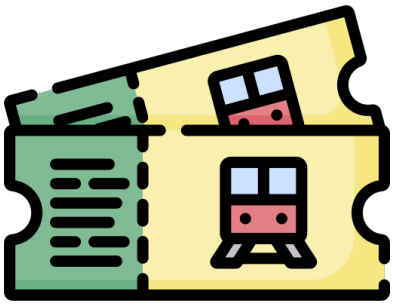
Ollscoil na Gaillimhe
University of Galway

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

2025–04–05

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[liveir], 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: /ˈʊmˠ pˠ əɾˠ/), comes from the Irish word for "transport" but also can be used to mean carriage, conveyance, transmission, and communication[iompar_teanglann]; it was therefore thought to be an apt name for an application which conveys live Irish public transport information to a user.

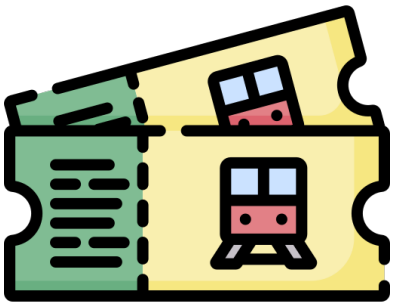


Figure 1.1: Iompar project icon[trainticket]

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[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[react] application to React Native[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[nielsenheuristics].

- 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[awsfree].
- 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.

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 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;
- Availability of live data for certain geographical locations.

Chapter 2

Research & Planning

2.1 Similar Services

As background research for this project, a list of transport-tracking applications available online was compiled, with each application then being analysed to determine their positive characteristics from which inspiration should be taken and their negative characteristics which should be avoided or fixed.

2.1.1 Irish Rail Live Map

The Irish Rail Live Map[liveir] displays the current location of Irish Rail intercity, commuter, & DART services, with clickable icons that display information about the selected service including the lateness of the train and its next stop.

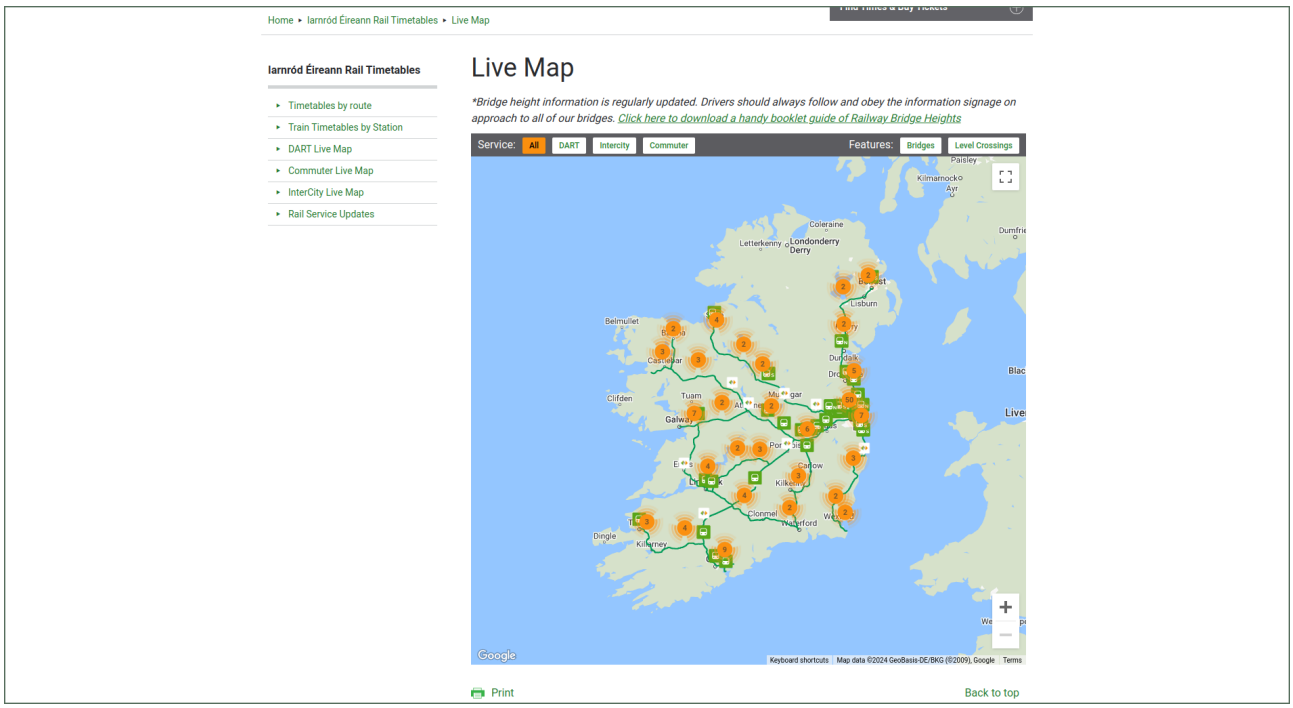


Figure 2.1: Irish Rail live map

Strengths of the Irish Rail live map that were identified include:

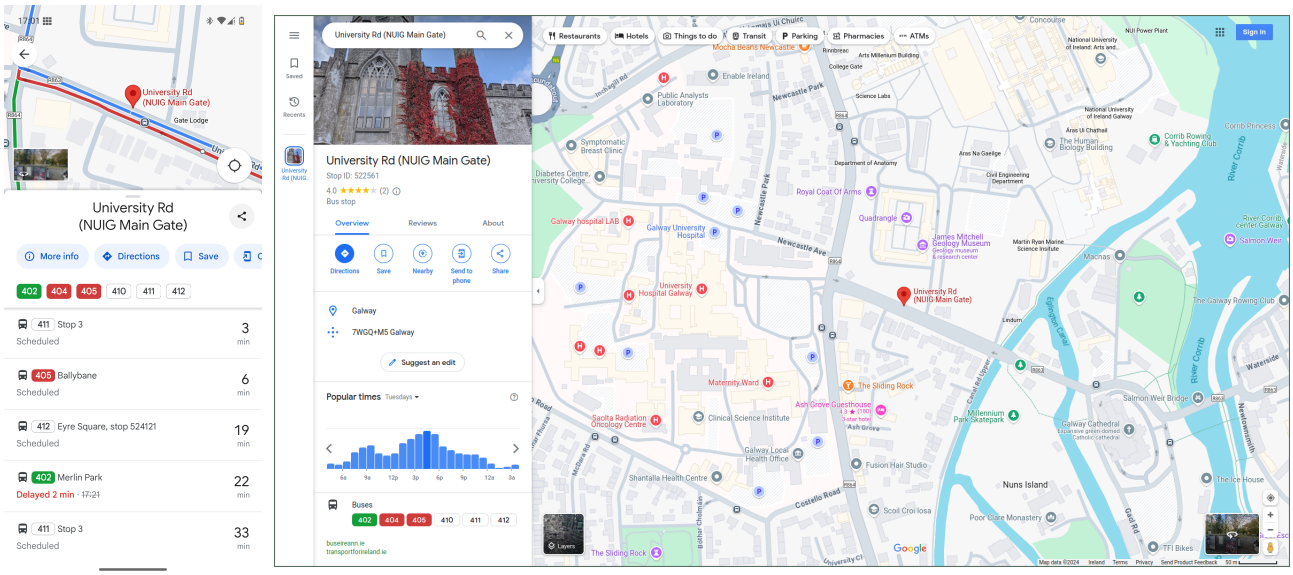
- Services can be clicked on to display a pop-up panel showing the punctuality of that service, the next stop of that service, and the stops on that service's route;
- There are basic filtering options to display a specific type of service such as DARTs;
- Bridges, level crossings, & stations are shown on the map. The stations can also be selected and information about them viewed.

Limitations of the Irish Rail live map that were identified include:

- The pop-up information panel covers the entire map and hides it;
- There is no search feature to find a specific service;
- The filtering options are greatly limited;
- The UI is slow and not particularly responsive;
- There is no visual distinction between the icons for different kinds of services.

2.1.2 Google Maps

Google Maps[gmaps] is a common choice for finding the next the upcoming nearby services and route planning. It covers a number of different types of service, include buses & trains. The mobile UI has been included here alongside the desktop UI as there is no live service information available on the desktop version of Google Maps.



(a) Mobile view

(b) Desktop view

Figure 2.2: Google Maps mobile & desktop views

Strengths of Google Maps include:

- It facilitates route determination, and can tell the user about what services they need to take to get any arbitrary location;
- The mobile version lists when services are due and how long they are expected to be delayed.

Limitations of Google Maps include:

- There is no live service information on the desktop version;
- Vehicle locations are not plotted on the map and thus the user cannot tell where the service actually is;
- Specific services cannot be searched for, and there is no filtering to display only a certain type of service.

2.1.3 TFI Live Departures

The TFI Live Departures map[tfilive] shows live departure information for both bus stops and train stations.

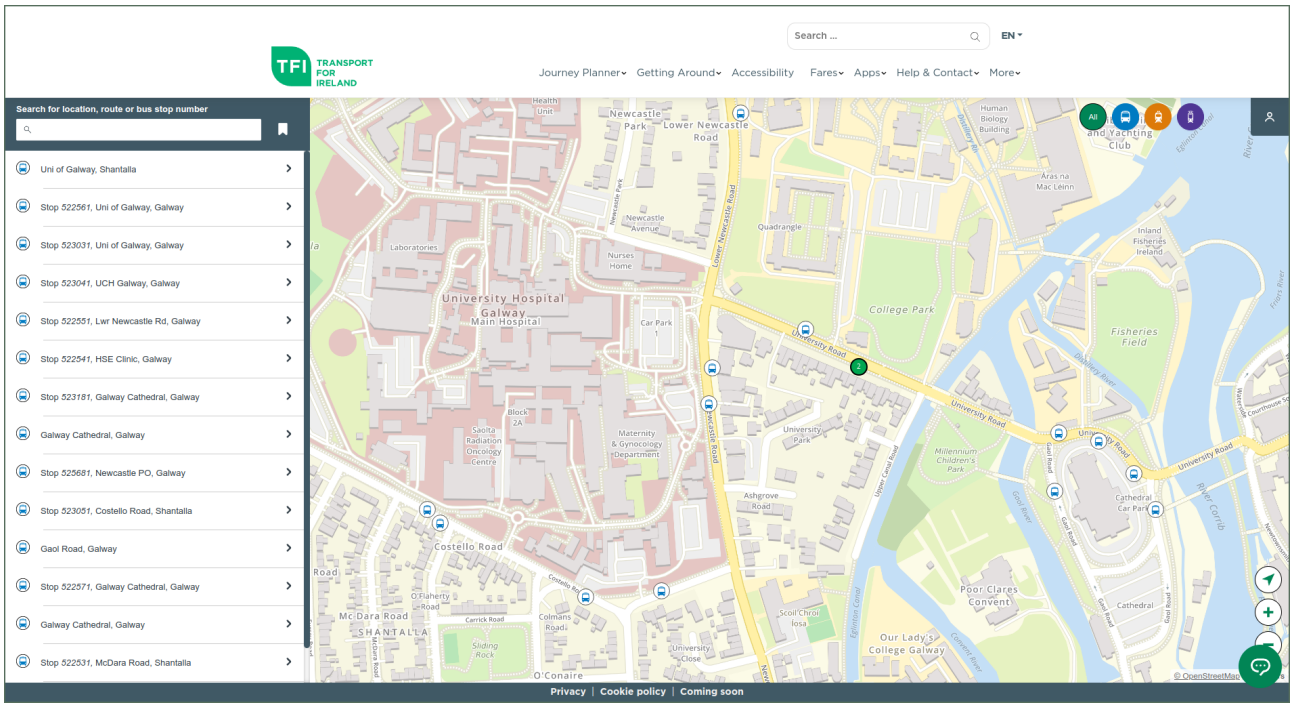


Figure 2.3: TFI Live Departures map

Strengths of the TFI Live Departures map include:

- A stop or a station can be clicked on to show a pop-up information panel that appears at the side of the screen and does not cover the map;
- There is a powerful search feature that allows the user to search by location, route, or stop number;
- If a specific route is selected, the route is highlighted on the map and its stops are plotted;
- The map is highly detailed, making it easier to find a particular location on the map and find nearby services.

Limitations of the TFI Live Departures map include:

- The map doesn't show the live locations of the services, it shows the stops or stations from which the services depart;
- The map has no filtering options beyond the search feature;
- The map has to be zoomed in very far to actually display the stations in that area, making the map more difficult to use & navigate;
- It doesn't say whether or not services will be delayed or where they currently are, only saying when they are scheduled for.

2.1.4 Flightradar24

Flightradar24[radar] is a live-tracking application for monitoring aeroplane & helicopter flights; while not public transport and therefore not within the scope of this project, this application has nonetheless chosen to be included in this analysis as it is particularly good example of transport-tracking UI, and is highly successful

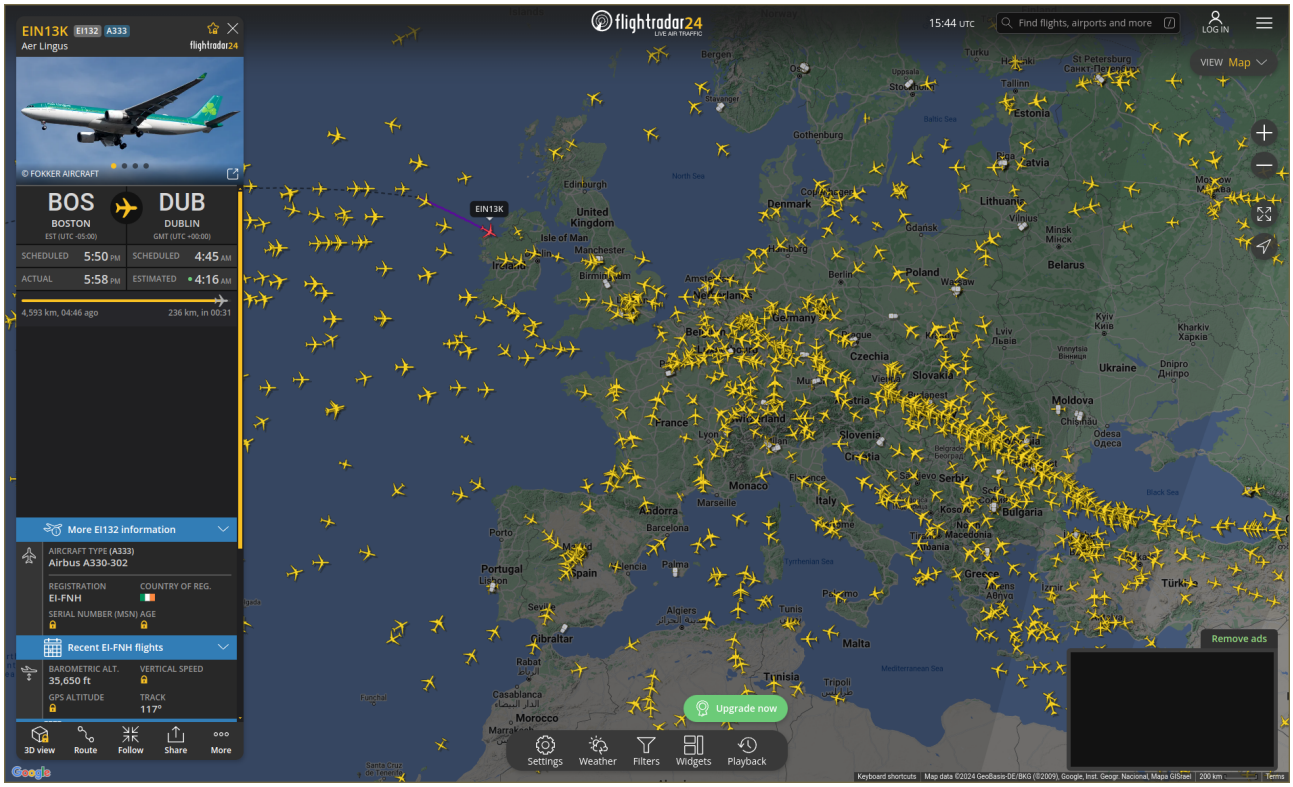


Figure 2.4: Flightradar24 UI

Strengths of Flightradar24 include:

- Ability to select a given service and open a pop-up panel that displays details about that service;
- Selected services are highlighted in red on the map;
- The information panel shows the scheduled departure time, the actual departure time, the scheduled arrival time, & the estimated arrival time;
- The information panel displays an image of the selected vehicle;
- Searching & filtering features;
- Larger planes have larger icons on the map and helicopters have distinct icons from planes.

Limitations of Flightradar24 include:

- Bookmarking a vehicle requires a paid account;
- The UI contains advertisements for free users.

2.2 Data Sources

2.2.1 Irish Rail API

Live train data, including DART information, can be sourced from the Irish Rail Realtime API[irishrailapi]. This is a mostly RESTful-like API which seems to be a legacy hybrid between REST[fielding2000rest] (Representational State Transfer) and SOAP[box2000soap] (Simple Object Access Protocol): unlike SOAP, it accepts HTTP `GET` requests instead of a HTTP `POST` of an XML envelope containing the request, but it doesn't use resource-based URLs and returns a namespaced XML response instead of plain XML. Nonetheless, it can be interacted with in more or less the same way as one would a REST API to obtain data. It provides a number of endpoints, with the relevant endpoints to this project being:

- `/getAllStationsXML` and `getAllStationsXML_WithStationType?StationType=<A,M,S,D>` which returns information about Irish Rail stations including latitude & longitude.
- `/getCurrentTrainsXML` and `/getCurrentTrainsXML_WithTrainType?=<A,M,S,D>` which returns information about all the currently running trains (with latitude & longitude), including trains that are due to depart within 10 minutes of the query time.
- `/getStationDataByCodeXML?StationCode=<station_code>` which returns the trains due to serve the station in question in the next 90 minutes.

The documentation page for the API warns that some areas are not fully supported for real-time information due to the central signalling system being subject to "ongoing work to support this real-time facility" and that, in the case that a train is in an area of the rail system where real-time data is unavailable, the scheduled data for the train will be returned instead. The extent to which coverage of real-time data is available today is unclear, as the API documentation page has not been updated since the 8th of December 2015 at the very latest[irishrail-api-archive] (although a more realistic estimate would be 2011 based off example dates used in the documentation) and so is now likely nearly a decade out-of-date, but this does not affect how the data will have to be processed very much; since the scheduling information is just returned in the absence of real-time data, this can be treated as a best approximation of the real-time data and does not need to be handled differently.

2.2.2 Luas API

The Luas API[luasapi] is an XML-over-HTTP API with a REST-like interface that provides real-time updates of expected Luas arrivals for a given Luas stop, based upon the location data of the trams from their Automatic Vehicle Location System (AVLS). No location data about individual trams is made publicly available; instead each stop must be queried individually using the `/get.ashx?action=forecast&stop=<stop_code>` endpoint, which returns an XML file with a list of all inbound & outbound trams due into that stop.

2.2.3 NTA GTFS API

The National Transport Authority (NTA) provides a General Transit Feed Specification (GTFS)[gtfs] REST API named GTFS-Realtime[gtfsapi] which provides real-time location data about buses operating in Ireland. GTFS is a standardised format for public transport schedules & associated geographic information that provides both static data (such as timetables, routes, & stop locations) and real-time data (such as live location data). The static GTFS feed is made up of comma-separated value files (as is standard) and the real-time data is returned in JSON format from the REST API endpoint `gtfsr/v2/Vehicles[?format=json]`. It is free to use, but requires an API key for access.

2.3 Technologies

2.3.1 Backend Technologies

Server-Based versus Serverless Architectures

The first choice to be made for the backend technologies for this project was whether the backend should be designed using a server-based or serverless architecture. A traditional server-based model was initially considered, either running on its own dedicated hardware or making use of a Virtual Private Server (VPS)[AWS:VPS] but was ultimately rejected. A server-based architecture requires extensive management of infrastructure, including the provisioning of hardware, system administration, and maintenance of the server application itself. Despite extensive personal experience with Linux-based and UNIX-like operating systems, the ongoing administration of a server would distract from the development of the application itself, and would be very difficult to scale to make the application available to a large number of users. While scaling issues could be partially mitigated with the utilisation of containerisation and a server-based microservices architecture[IBM:Microservices, AWS:Microservices] using technologies such as Docker[DockerEngineDocs], it would nonetheless take a large amount of administrative effort to scale the application as usage demands grew.

In contrast, serverless architectures[digitalocean_serverless_2023, google_serverless_2025, aws_serverless_2025] abstract away these infrastructure concerns, allowing the focus of the project to be solely on the application logic. Since serverless functions

are invoked on demand and billing for serverless models is based solely upon the usage, serverless architectures are generally much more cost-effective for the workloads with variable or intermittent traffic that would be expected for a user-facing application such as this one. The serverless model lends itself especially well to a microservices architecture, which allows the system to broken into small, independent services responsible for their own pieces of functionality, with each microservice being independently deployable and scalable. Instead of establishing or purchasing server infrastructure in advance, a serverless architecture can automatically scale up or down to meet the demand put upon it, preventing outages when large strain is put upon the service and preventing excessive financial or computational costs when there is low usage; the issue of over-provisioning or under-provisioning of compute resources is entirely circumvented, and the need for load balancers is eliminated entirely. Serverless functions support event-driven architecture, making them a natural fit for a system that must react to user demands as they occur. Furthermore, the fast time-to-production of serverless architectures results in faster iteration & experimentation, easier integration into CI/CD pipelines for continuous delivery, and simplifies rollbacks (should they be necessary) by deploying only small, isolated units, all without the need to spin up servers or configure deployment environment.

The ephemeral nature of serverless functions makes them highly fault-tolerant, as an error in a function execution will only affect that execution, whereas an improperly handled error in a server architecture could, in the very worst-case scenario, bring the entire system down and require a reboot. Moreover, although not especially relevant for this project as it does not handle sensitive data, the attack surface for a serverless architecture is much smaller and there is no need to manually configure server patches or firewall rules, as these are handled by the serverless provider. For these reasons, a serverless architecture was chosen for this project.

Serverless Platforms

A number of serverless platforms were considered for this project, including Amazon Web Services (AWS)[aws], Google Cloud[googlecloud], Microsoft Azure[azure], and Cloudflare[cloudflare]. While these platforms have innumerable differences, and are better and worse suited for different applications, a brief overview of their strengths & weaknesses as considered for this project is as follows:

- AWS has a mature & extensive ecosystem with broad service integration and a generous free tier, but its complexity can result in a steeper learning curve and a more complex initial set-up.
- Google Cloud has a simpler set-up, and tight integration with Firebase & Google APIs along with a similarly generous free tier, but fewer advanced features, less control over configuration, and a less extensive ecosystem.
- Microsoft Azure has a deep integration with the Microsoft ecosystem such as SQL server[azuresql], but less community support and a steep learning curve for non-Windows developers.
- Cloudflare has extremely low latency, excellent support for edge computing[ibm_edge_computing] (running on servers geographically close to users), and is great for lightweight apps, but has extremely limited memory and runtime, making it unsuitable for heavy backend logic. Its database & storage solutions are also not as mature or as scalable as comparable solutions.

AWS and Google cloud offer the most generous free tiers, while Cloudflare has very tight limits (CPU time is limited to 10ms maximum in the free tier[cloudflareworkers]) and thus is only suitable for very lightweight tasks, not the kind of extensive data processing that this application will be engaging in. Therefore, AWS was chosen as the serverless platform for this project as it is was deemed to be the most flexible & affordable option.

Relational versus Non-Relational Databases

AWS offers a wide array of relational and non-relational database services, so the next choice was between a relational & non-relational database. **Relational databases** store data in tables using fixed schemata consisting of rows & columns, and use SQL for querying; relationships between tables are strictly enforced with the use of foreign keys. These rigid, pre-defined schemata are excellent for data integrity & relational consistency, and are suitable for applications in which relationships between entities are complex and querying across those relationships is common.

Non-relational databases store data in more flexible formats, such as key-value pairs, JSON documents, or graphs.

They are more suitable to scaling horizontally and for handling unstructured or semi-structured data. The simplified structure means that they can be very high performance for simple read/write patterns, and are ideal for applications that handle large volumes of fast-changing or unstructured data.

For this application, data is semi-structured and varies greatly by API source. The data is updated very frequently, which would require many `JOIN` operations or schema migrations for a normalised relational database. Non-relational databases, on the other hand, offer flexibility for heterogeneous transport data with different attributes, are more suitable for real-time data ingestion & retrieval due to low-latency reads & writes, and can support on-demand scaling without the need for the manual intervention required by relational databases to be scaled up; a non-relational database structure was selected for this reason.

AWS offers several non-relational database services, each of which is optimised for a different use case[awsdatabases]:

- **Amazon DynamoDB**[dynamodb] is a fully-managed, serverless No-SQL database designed for low-latency, high-throughput workflows that supports storing data as key-value pair or as JSON-like documents. Key features of DynamoDB include the choice between on-demand or provisioned capacity, global tables for replication across multiple geographic regions, and fine-grained access control with Amazon Identity and Access Management (IAM).
- **Amazon DocumentDB**[documentdb] is a fully-managed, document-based database using JSON-like BSON with MongoDB compatibilities, designed for applications using MongoDB APIs. However, unlike DynamoDB, it is not fully serverless, and is instead a managed cluster-based service that must be provisioned & scaled manually.
- **Amazon ElastiCache**[elasticache] is an in-memory data store for extremely fast caching (sub-millisecond response times) and changing data, with data stored only per session. However, it cannot be used for persistent data storage, making it insufficient as a sole database service for this application.
- **Amazon Neptune**[neptune] is a managed database for storing & querying relationships using graph models, generally used for recommendation engines & knowledge graphs and therefore not suitable for this application.
- **Amazon Timestream**[timestream] is a time-series database that is purpose-built for the storage and analysis of time-series data, i.e., timestamped data. This would be a good choice for the historical analysis side of this application, but inappropriate for real-time data.
- **Amazon OpenSearch service**[opensearch] is a distributed full-text search & analytics engine based upon Elasticsearch that supports advanced filtering, sorting, & ranking of query results on an *ad-hoc* basis by default and supports JSON document ingestion. However, the higher querying flexibility comes at the cost of higher-latency queries, making it less appropriate for real-time data.

DynamoDB was chosen for the database service due to its suitability for the data being processed, its scalability, and its low-latencies.

Programming Language

AWS Lambda functions officially support many programming languages, including Node JS, Python, Java, C#, & Go; custom runtime environments can also be created, making practically any language usable within AWS Lambda functions with some effort. Due to previous experience with these programming languages, the main options considered for this project were Node JS, Python, & Java:

- Node JS[node] is a good choice for event-driven & real-time web applications because of its non-blocking IO, fast cold starts for latency-sensitive functions, and extensive library ecosystem. Asynchronous operations are supported out-of-the-box, making it a good choice for querying external APIs. However, it can become complex to manage for lengthy source files, and can be cumbersome to write compared to something like Python. An advantage of writing the backend in Node JS would be that both the frontend & the backend would utilise JavaScript code, meaning that the same tooling could be used for both.

- Python[python] is a good choice for polling of APIs & data ingestion due to its clean syntax, data parsing features, and extensive library ecosystem. Development & prototyping in Python is fast and easy to manage, but it does not have built-in support for asynchronous programming. Python is generally slightly slower than Node JS, and start-up time for Python Lambda functions can be longer.
- Java[java] is a good choice for complex, enterprise-grade applications due to its strong typing & compile-time safety, and has a mature library ecosystem. However, it can be slower to write due to its verbose syntax, and development environments can be more complex to set up & configure. Despite having fast execution times, it can be slow to start up for Lambda functions as the JVM must be initialised each time, leading to $\sim$1–2 seconds of delay.

Java was rejected on the basis that its slow start-up times made it unsuitable for this application, so the decision came down to whether the speed of execution for Node JS or the speed of development for Python should be favoured; ultimately, it was decided that Python would be a better choice for rapid prototyping & development, as the execution speed differences between the two languages are negligible (in this scenario), but the development speed differences are not.

2.3.2 Frontend Technologies

JavaScript Frameworks

While a web application like this could be developed using HTML and JavaScript alone, the complexity & dynamic nature of the application means that a JavaScript framework would more suitable for the benefits of re-usability & modularity, simplified application state management, optimised performance, and extensive library ecosystems. The main frameworks considered were React, Angular, & Vue due to their widespread popularity, good documentation, and extensive feature set:

- **React**[react] is the most popular choice of the three, with a large user community and a wide array of third-party libraries. It is component-based, making code easily re-usable and allowing the frontend design to be flexible for different design patterns & architectures. However, it can have a steep learning curve and can be needlessly complicated for simple applications. Technically speaking, it is not a full JavaScript framework, as it only provides built-in features for building UI components: it doesn't have built-in features for routing, data fetching, or tight integration with any backend services, although these can be added either with third-party libraries or by writing custom JavaScript code.

  There is also a mobile development framework built on top of React called **React Native**[native], which allows for native Android & iOS mobile application to be developed using JavaScript and React components. Instead of rendering to a DOM like vanilla React, React Native maps its UI components to native iOS/Android views (i.e., a button in React Native would use the operating system's native button component instead of being defined in HTML). This means that the same React Native codebase can be used to write an iOS and an Android application, and means that the performance will be superior to that of a webview application because of the utilisation of native components. However, React Native does not support any HTML or CSS, meaning that while a React web application is highly portable to React Native, it is not a simple drop-in solution, and a React web application would have to undergo extensive re-writing to use React Native.
- **Angular**[angular] is a complete JavaScript framework for large-scale enterprise applications that comes with complete routing, state management, and a HTTP client out-of-the-box, therefore eliminating much of the need for third-party libraries. However, it has a rather steep learning curve, and takes a lot more development time to get up and running than other comparable JavaScript frameworks. It requires a lot of boilerplate code and complex configuration, with relatively simple features sometimes taking a great deal of set-up. It also rigidly enforces certain best practices, which is both a positive and a negative, as this ties the development into the Angular way of doing things.
- **Vue**[vue] is a JavaScript framework known for its simplicity and ease-of-use, with very beginner-friendly syntax that is highly similar in appearance to standard HTML & JavaScript. Reactivity is baked into the framework, with the view updating automatically as data changes, reducing the need to manually update data values. All the HTML, CSS, & JavaScript for a Vue component is contained within a single component file, simplifying

development. However, it has a far less mature library ecosystem than the other two discussed frameworks, and fragmentation between the Vue 2 version and the Vue 3 version means that the few plug-ins that do exist may not be compatible with the version used. Vue's automatic handling of reactivity, while simplifying development, can be a black box that is difficult to understand for debugging purposes.

It was decided that React would be the most suitable JavaScript framework for this project, as it has extensive library support, is flexible & lightweight, and it lends itself to porting the codebase to a native application if so desired in the future. Since the consideration of native applications is only a secondary objective of this project, the decision was made to focus only on React development and to make use of React-only functionality wherever suitable (such as using HTML + CSS), but to write the components in a clean & modularised fashion so that a React Native port would not be excessively difficult.

CSS Frameworks

A **CSS framework** is a collection of pre-written styles, utility classes, or components that can be used to help build UIs quickly without reinventing basic functionality. CSS frameworks allow applications to be responsive by default on mobile devices, and maintain consistency throughout the application. The two CSS frameworks considered for this application were Tailwind CSS and Bootstrap CSS:

- **Tailwind CSS**[tailwind] is a utility-first CSS framework in which the user interface is built by composing utility classes directly in HTML / React code; in this context, "utility first" refers to the fact that Tailwind does not provide any pre-built components, but instead provides pre-defined utility classes. Layout and styling are all done in-line via class names. This utility-based approach makes the framework very flexible & customisable, but comes at the cost of not providing any out-of-the-box re-usable components and require for components to be designed from scratch instead.
- **Bootstrap CSS**[bootstrap] is a component-first CSS framework designed to help developers build responsive, styled UIs quickly. It comes with many pre-built components such as buttons and navigation bars, and JavaScript plug-ins for drop-down menus and image carousels. This makes it very useful for fast prototyping and building of UIs, but can look generic due to its pre-built components and overriding the default component styles can be time-consuming.

Tailwind CSS was chosen for this project due to its flexibility & customisability, and because its utility-first approach means it integrates very well with React's component-based approach as a component's style can be contained entirely within that component.

2.4 Project Management

Agile methodologies[agile] were employed throughout this project to facilitate iterative development with continuous feedback & adaptation: Agile is a development approach which divides work into phases, emphasising continuous delivery & improvement. Tasks are prioritised according to importance & dependencies, and are approached in a flexible manner so that approaches could be adjusted if something isn't working as intended. Progress was reflected upon regularly, with areas for improvement being identified and plans adjusted accordingly. Automated CI/CD pipelines were employed to streamline development, and to ensure that new changes never broke existing functionality. Progress was tracked in a Markdown diary file, and a physical Kanban[kanban] board with Post-It notes was used to track planned work, in progress work, & completed work.

The principles of **clean coding**[martin2008cleancode] (famously put forward by Robert "Uncle Bob" Martin) were also employed to ensure that all code written was easy to read, understand, & maintain. Key principles of clean coding involve ensuring that the code is readable with meaningful function & variable names, ensuring that the code is simple & concise with each function doing one thing only and doing that thing well, using consistent naming conventions, and employing the Don't Repeat Yourself (DRY) principle which involves the avoidance of duplicated code by creating re-usable functions & modules. Code should be structured in a way that is meaningful, and should have proper comments & documentation where necessary; comments should only explain *why* something is done, not *what* it does as properly written clean code should be self-explanatory. Code should also be written in a way that lends itself

easily to unit testing, and should be refactored to make testing as simple as possible rather than making the tests more complicated; if the code is decomposed into functions properly, it should be easier to test, upgrade, & maintain.

Proper functional decomposition also results in code that is less tightly coupled in to specific libraries, platforms, or services, making it more flexible & easier to port to different services if needs be. With proper separation of concerns, each function has a single responsibility and so dependencies can be isolated, and abstraction layers can prevent direct dependency on a specific implementation; if a function is abstracted properly switching libraries or services requires fewer changes and in fewer places. Isolated functions are also easier to test with mock functions, reducing reliance on real services during unit testing.

2.4.1 Version Control

Any software development project of substantial size should make use of version control software, and Git[git] was chosen for this project due to its ubiquitousness and extensive personal experience with the software. GitHub[github] was chosen as the hosting platform for the Git repository, again largely due to its ubiquitousness and extensive personal experience, but also because of its excellent support for CI/CD operations with GitHub Actions[githubactions]. The decision was made to keep all relevant files for this project, including frontend code, backend code, and document files in a single monolithic repository (monorepo) for ease of organisation, a simplified development workflow, and easier CI/CD set-up. While this is not always advisable for very large projects where multiple teams are collaborating, or projects in which the backend and frontend are truly independent of one another and are developed separately, this works well for projects like this one where development is done by a single individual or a small team[brito2018monorepos], and is even used on very large projects where unified versioning and simplified dependency management are of paramount importance, such as at Google[potvin2016google].

Commits to this repository were made in adherence with the philosophy of *atomic commits*, meaning that each commit represents a single, logical change: commits are kept as small as reasonably possible, and are made very frequently. This makes the Git history easier to read, understand, & revert if needed, makes debugging easier, keeps changes isolated and understandable, and reduces merge conflicts. This is the generally accepted best practice, as bundling unrelated changes into a single commit can make review, reversion, & integration of these commits much more difficult[dias2015untangling].

Meaningful commit messages are of great importance when using version control software, and even more so when making atomic commits as there will be a very large volume of commits & commit messages. For this reason, a custom scoped commit message convention was chosen, where each commit message follows the format `[scope]: Details of commit`, where the `scope` can be `[frontend]` for changes to frontend code, `[backend]` for backend code, `[report]` for changes to the LaTeX project report files, and so on. This ensure consistency & clarity in commit messages, and the strict formatting of the messages make it easy to parse commit messages in a script should that be necessary. Recent examples of commit messages to the GitHub repository can be seen in Figure 2.5 below.

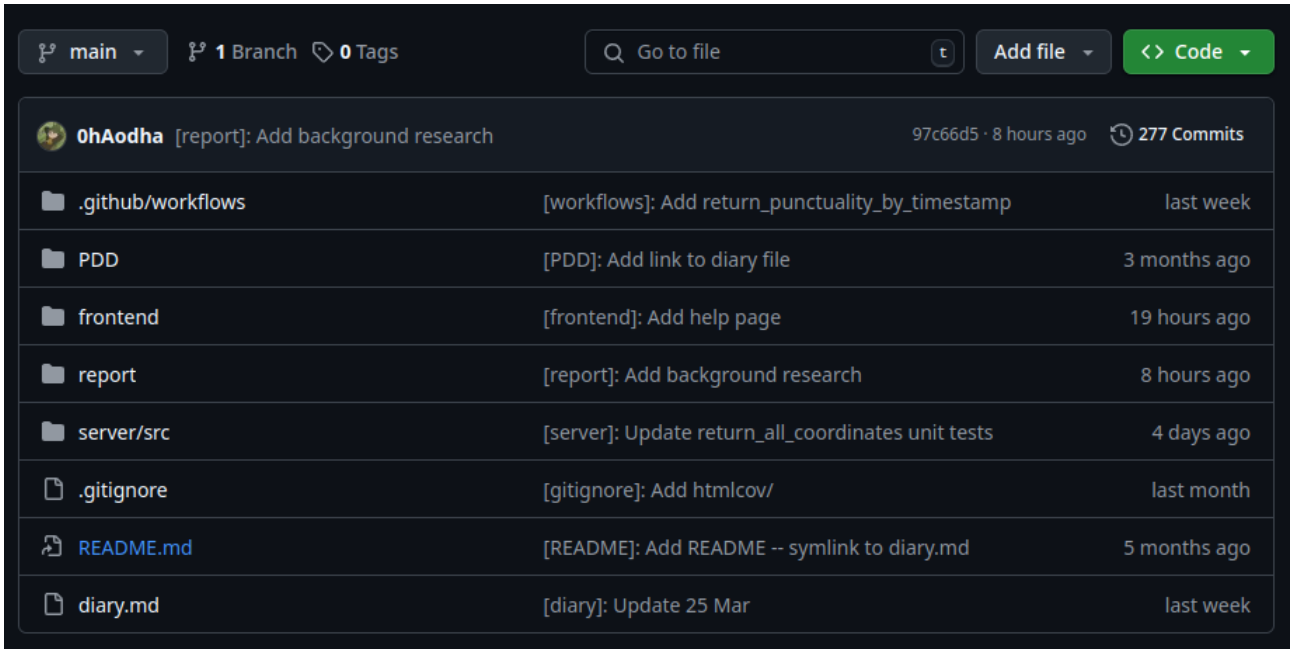


Figure 2.5: Screenshot of the repository files on GitHub

Chapter 3

Backend Design & Implementation

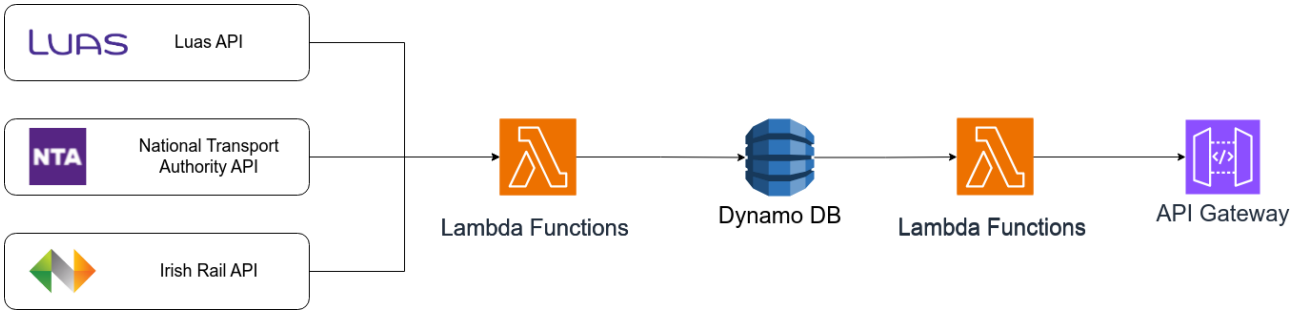


Figure 3.1: High-level backend architecture

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.

```
[
  {
    "objectID": "IrishRailStation-GALWY",
    "objectType": "IrishRailStation",
    "trainStationCode": "GALWY",
    "trainStationID": "170",
    "trainStationAlias": null,
    "trainStationDesc": "Galway",
    "latitude": "53.2736"
    "longitude": "-9.04696",
  },
  {
    "objectID": "BusStop-8460B5226101",
```

```
    "objectType": "BusStop",
    "busStopID": "8460B5226101",
    "busStopCode": "522611",
    "busStopName": "Eyre Square",
    "latitude": "53.2750947795551"
    "longitude": "-9.04963289544644",
  },
  {
    "objectID": "BusRoute-4520_67654",
    "objectType": "BusRoute",
    "busRouteID": "4520_67654"
    "busRouteAgencyName": "City Direct",
    "busRouteAgencyID": "7778028",
    "busRouteShortName": "411",
    "busRouteLongName": "Mount Prospect - Eyre Square",
  },
  {
    "objectType": "LuasStop",
    "objectID": "LuasStop-STS",
    "luasStopCode": "STS"
    "luasStopID": "24",
    "luasStopName": "St. Stephen's Green",
    "luasStopIrishName": "Faiche Stiabhna",
    "luasStopIsParkAndRide": "0",
    "luasStopIsCycleAndRide": "0",
    "luasStopLineID": "2",
    "luasStopZoneCountA": "1",
    "luasStopZoneCountB": "1",
    "luasStopSortOrder": "10",
    "luasStopIsEnabled": "1",
    "latitude": "53.3390722222222",
    "longitude": "-6.26133333333333",
  }
]
```

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[choosing-the-right-key]:

- 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:


```
[
  {
    "objectType": "IrishRailTrain",
    "latenessMessage": "On time",
    "timestamp": "1742897696",
    "trainDirection": "Southbound",
    "trainStatus": "R",
    "trainDetails": "09:41 - Maynooth to Grand Canal Dock ",
    "trainType": "S",
    "objectID": "IrishRailTrain-P656",
    "averagePunctuality": "0",
    "trainUpdate": "Departed Pelletstown next stop Broombridge",
    "trainStatusFull": "Running",
    "longitude": "-6.31388",
    "trainPublicMessage": "P656\\n09:41 - Maynooth to Grand Canal Dock (0 mins late)\\nDeparted
    ↪  Pelletstown next stop Broombridge",
    "trainPunctuality": "0",
    "trainPunctualityStatus": "on-time",
    "trainTypeFull": "Suburban",
    "trainDate": "25 Mar 2025",
    "latitude": "53.3752",
    "trainCode": "P656"
  },
  {
    "objectType": "Bus",
```

```
    "busScheduleRelationship": "SCHEDULED",
    "timestamp": "1742908007",
    "busID": "V598",
    "busRoute": "4538_90219",
    "busRouteAgencyName": "Bus Éireann",
    "objectID": "Bus-V598",
    "busRouteLongName": "Galway Bus Station - Derry (Magee Campus Strand Road)",
    "longitude": "-8.50166607",
    "busDirection": "1",
    "busStartDate": "20250325",
    "busRouteShortName": "64",
    "latitude": "54.2190742",
    "busTripID": "4538_114801",
    "busStartTime": "10:30:00"
  },
```

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[gsi]. 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": "9.33333333333333333333333333",
    "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.

```
[
  {
    "average_punctuality": "0.8823529411764706",
    "timestamp": "1742908007"
  },
  {
    "average_punctuality": "1.0625",
    "timestamp": "1742905796"
  }
]
```

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[awsapi]:

- **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[httpvsrest]:
  - **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[apipricing] 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[boto3query](the official AWS SDK for Python). This means that query parameters are safely bound

[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[w3c-cors]. 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.

to named placeholder attributes in 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[useparameterisedqueries]. 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.

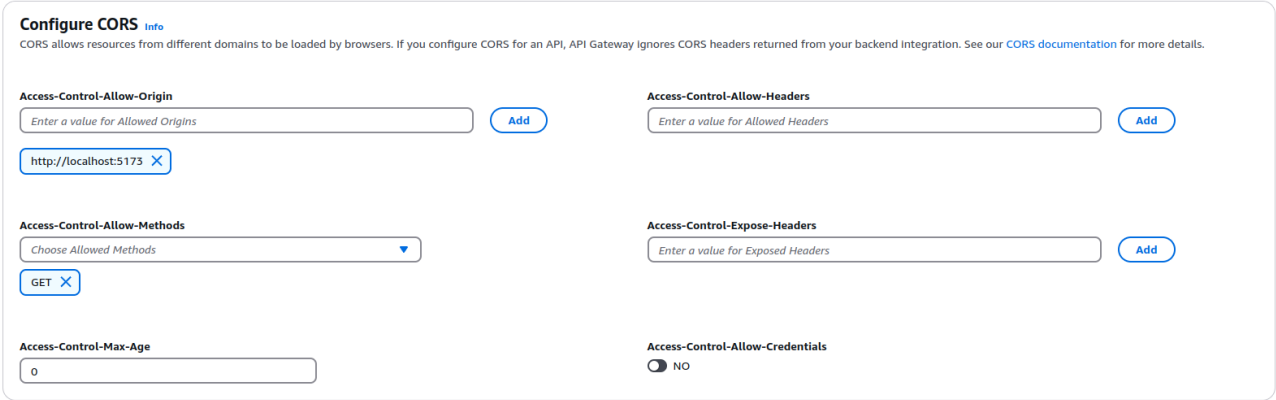


Figure 3.2: CORS configuration for the HTTP API

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**[hassan2021survey], 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[awslambda] 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[aws_management_console] or via the AWS CLI tool[aws_cli]. 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**[awsPowerTuning], 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).

`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.

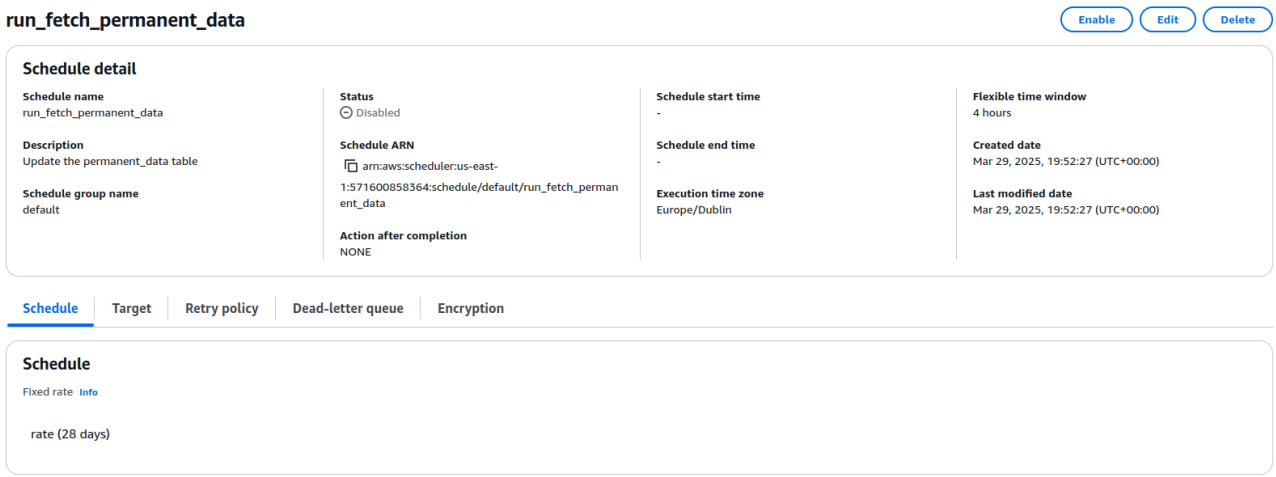


Figure 3.3: Screenshot of the Amazon EventBridge schedule to run the `fetch_permanent_data` Lambda function

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[pythondict]) 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.

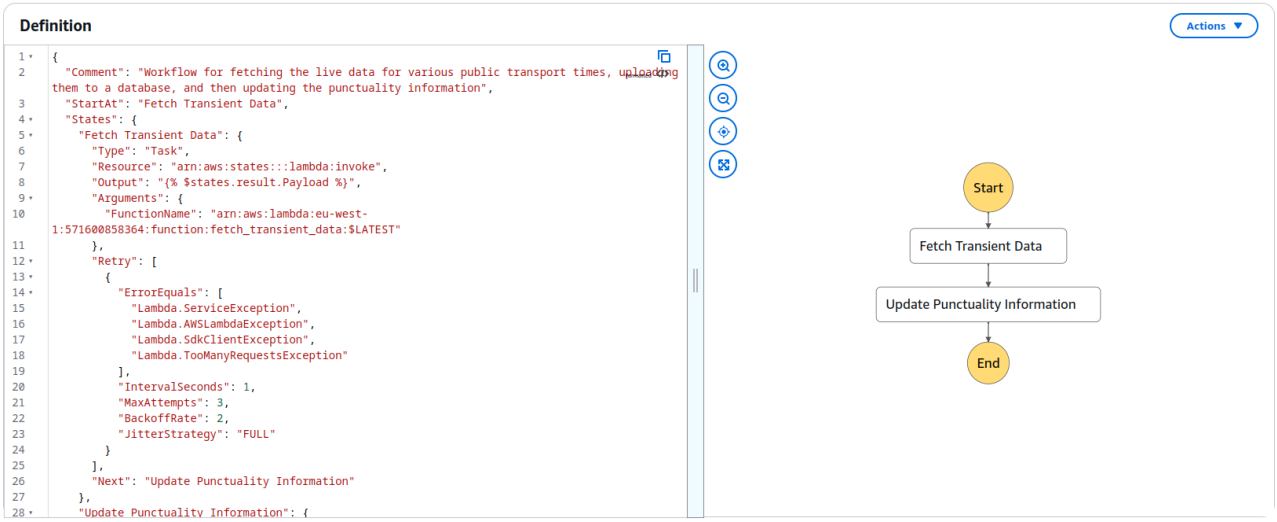


Figure 3.4: Screenshot of the `get_live_data` step function definition

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"[nielsenheuristics].

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 $10\times$ faster: the average execution time was reduced from $\sim$10 seconds to $\sim$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 $\sim 8\times$, 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).

3.4 CI/CD Pipeline

3.4.1 Unit Tests

As the code for each Lambda function was written, so too were corresponding PyUnit[pyunit] (based on the Java unit testing framework JUnit[junit]) unit tests for that function to ensure that the function was working as expected, that edge cases were covered, and to indicate if functionality was broken when updates were made. Indeed, this often proved invaluable, as the unit tests revealed on multiple occasions edge cases that were not handled properly, such as the handling of the API responses when only a single item was returned. Without the utilisation of unit testing, it's likely that these issues would have gone unnoticed in the codebase until they caused an error in deployment, which could have lead to Lambda function crashes, inconsistent database data, and wasted developer time spent on debugging the application.

These PyUnit tests are run using `pytest`[pytest] which automatically discovers Python test files in a given directory and can generate code coverage reports. While PyUnit tests can be run with `python -m unittest`, `pytest` makes it

easier to run the tests and can generate coverage reports automatically. `pytest` also allows for unit tests to be specified in its own non-standard format alongside to the standard PyUnit format, but this was not used in an effort to keep the tests as standard and as homogeneous as possible.

A statement coverage of 70% was selected to be aimed for when writing tests for this application; this is a high degree of test coverage, but allows the tests to focus on testing the core logic of the application rather than boilerplate code and allows for interactions with third-party services that cannot be fully covered by testing and instead have to be mocked[pyunitmock]. The actual total coverage achieved is 85%, which exceeds this minimum value and ensures that the application is well-covered with test cases.

server/src/test/fetch_transient_data/test_fetch_transient_data.py	35	1	0	97%
server/src/test/return_all_coordinates/test_return_all_coordinates.py	60	1	0	98%
server/src/test/return_historical_data/test_return_historical_data.py	60	1	0	98%
server/src/test/return_luas_data/test_return_luas_data.py	25	0	0	100%
server/src/test/return_permanent_data/test_return_permanent_data.py	60	1	0	98%
server/src/test/return_punctuality_by_objectID/test_return_punctuality_by_objectID.py	37	1	0	97%
server/src/test/return_punctuality_by_timestamp/test_return_punctuality_by_timestamp.py	56	1	0	98%
server/src/test/return_transient_data/test_return_transient_data.py	57	1	0	98%
server/src/test/update_average_punctuality/test_update_average_punctuality.py	99	1	0	99%
Total	**1071**	**164**	**0**	**85%**

Figure 3.5: Screenshot of (some of) the results from the HTML coverage report generated

3.4.2 CI/CD

To run the unit tests automatically, a GitHub Workflow[workflow] was created. This workflow is triggered when a pull request to merge code into the `main` Git branch is opened, or when code is pushed to the `main` branch. Since the Git repository used for this project is a monorepo containing all code files for the project, not just the backend Python files, many of the commits to this repository will not require re-running the unit tests, as the Python code will not have changed; therefore, the workflow only runs if the changes have been made to a `*.py` file.

```
name: Continuous Integration / Continuous Deployment

on:
  pull_request:
    branches:
      - main
    paths:
      - "**.py"
  push:
    branches:
      - main
    paths:
      - "**.py"
```

Listing 5: Snippet of the `ci.yml` workflow file

The workflow consists of two "jobs":

- The `test` job installs the necessary dependencies and runs the unit tests, generating coverage reports in both XML and HTML and uploading these as artifacts[artifacts] (a way to share data from a GitHub workflow run).
- The `deploy` job runs only on pushes to `main` (including direct pushes and merges of pull requests). It installs AWS CLI[aws_cli] and sets the AWS credentials using GitHub Secrets[githubsecrets] which allow for sensitive information such as environment variables to be stored privately for a repository, even if that repository is public, and avoids

having to hard-code these values. It then iterates over each Lambda function in the repository, installs the necessary Python libraries to a temporary directory, and bundles these along with the function source code into a ZIP file for deployment via the AWS CLI.

Chapter 4

Frontend Design & Implementation

The frontend design is built following the Single-Page-Application (SPA)[spa] design pattern using the React Router[reactrouter] 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"[nielsenheuristics].

The web application follows the Container/Presentational Pattern[containerpresentational], 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[reactcomponents]; 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[reenskaug2003mvc]: 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*[mcnatt2001coupling] 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*[mcnatt2001coupling] 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

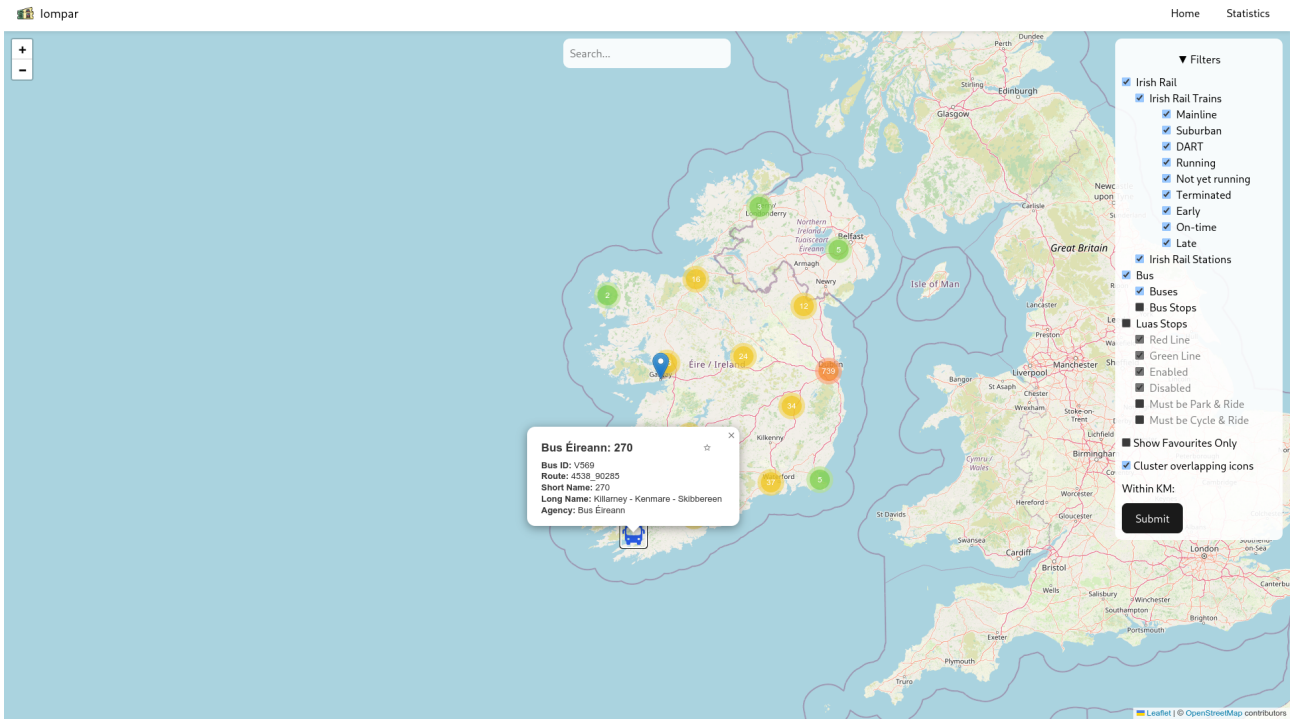


Figure 4.1: Screenshot of the front page of the web application

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[arraymethods] (very superficially similar to the Java `Stream`[javastream] 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[redhatpipe]. 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)[shortcircuit]. The modern JavaScript engine also is less efficient at optimising these array methods than simple **`for`**-loops[jsmethodopti]. 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[nielsenheuristics]), 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[mdnperceivedperformance, persson2019perceived, Vladic2020LoadingAnimations].

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[leaflet], 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[leaflet] 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:

 trainicon	On-time & early, running Irish Rail trains
 trainicon	Late, running Irish Rail trains
 trainicon	Not-yet running & terminated Irish Rail trains
 darticon	On-time & early, running DARTs
 darticon	Late, running DARTs
 darticon	Not-yet running & terminated DARTs
 trainstaitionicon	Irish Rail stations
 busicon	Buses
 busstopicon	Bus stops
 tramicon	Red line Luas stops
 tramicon	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[logicaland] or if the left-hand side of a Boolean `OR` is true[logicalor].

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

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

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

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

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

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., $(\text{showMainline} \land (\text{trainType} = \text{Mainline}))$ 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[chopde2013landmark]:

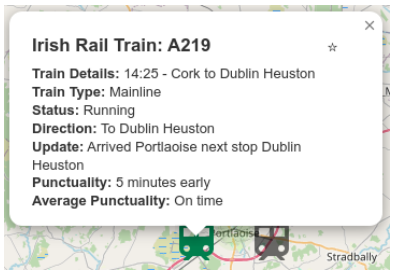
$$d = 2r \cdot \arcsin\left(\sqrt{\sin^2\left(\frac{\Delta\phi}{2}\right) + \cos\left(\phi_1\right)\cos\left(\phi_2\right)\sin^2\left(\frac{\Delta\lambda}{2}\right)}\right)$$

where:

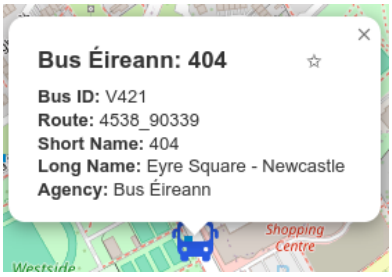
- d is the distance between the two points,
- r is the radius of the Earth ($\sim$6,371 kilometres[turcotte_earths_radius_1992]),
- ϕ_1, ϕ_2 are the latitudes of each point in radians,
- λ_1, λ_2 are the longitudes of each point in radians,
- $\Delta\phi = \phi_2 - \phi_1$, and
- $\Delta\lambda = \lambda_2 - \lambda_1$.

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[vincenty1975geodesics] which is more accurate (because it considers the Earth to be an oblate spheroid rather than sphere) but slower to compute, or Euclidean distance[smith2013precalculus] 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.

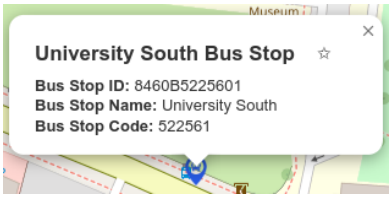
Marker Pop-Ups



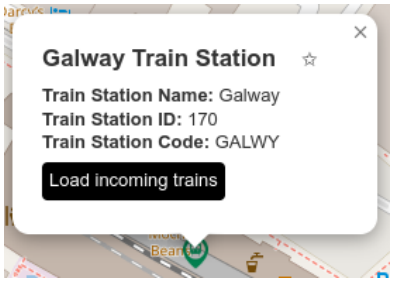
(a) `IrishRailTrain` pop-up



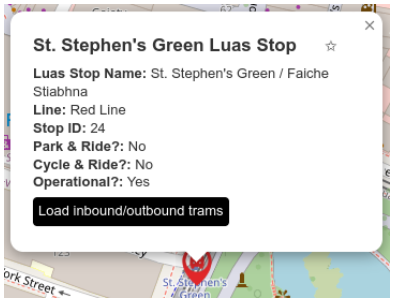
(b) `Bus` pop-up



(c) `BusStop` pop-up



(d) `IrishRailStation` pop-up

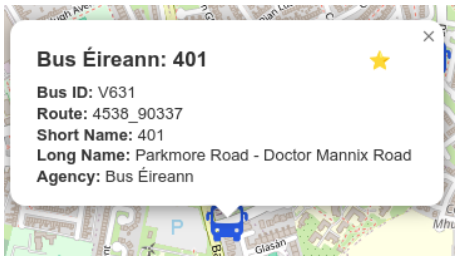


(e) `LuasStop` pop-up

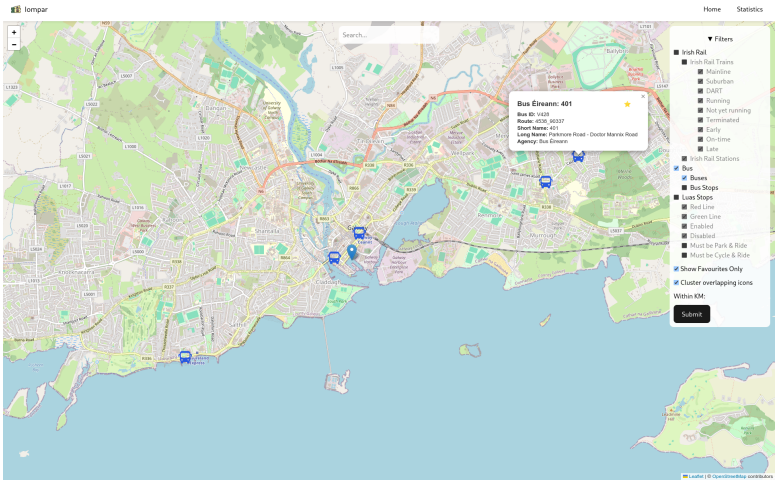
Figure 4.2: The 5 pop-up types

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.



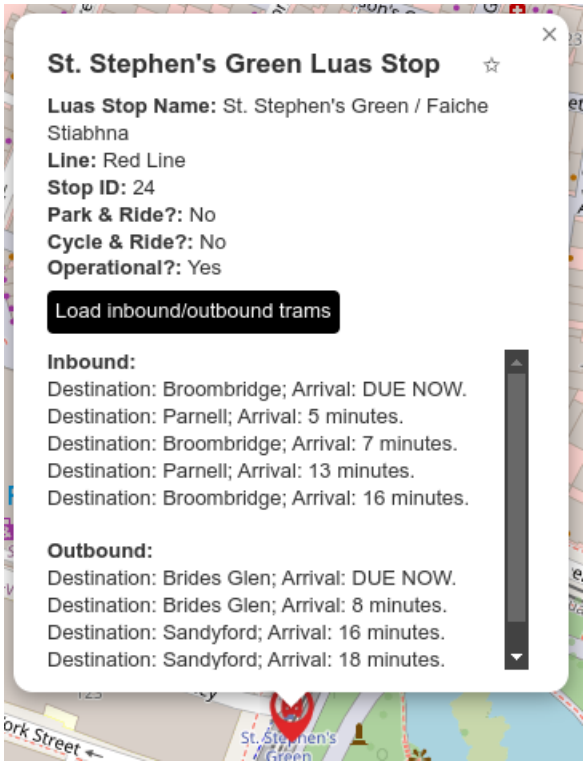
(a) Bus pop-up with bus "favourited"



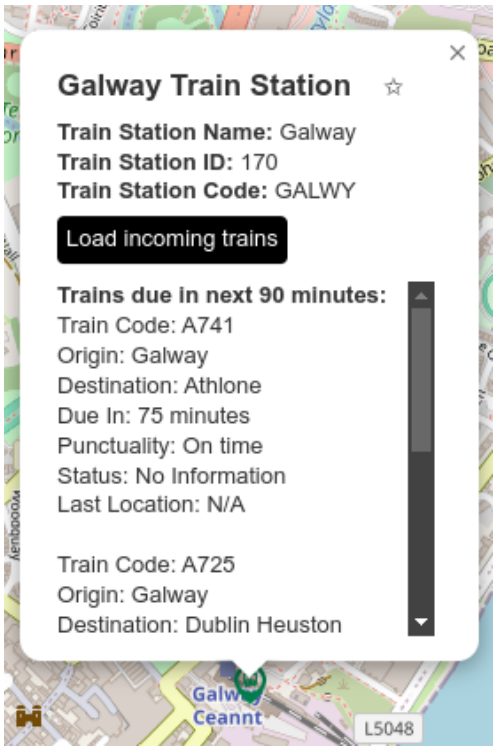
(b) Buses of the same route appearing when "favourites only" filter applied

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

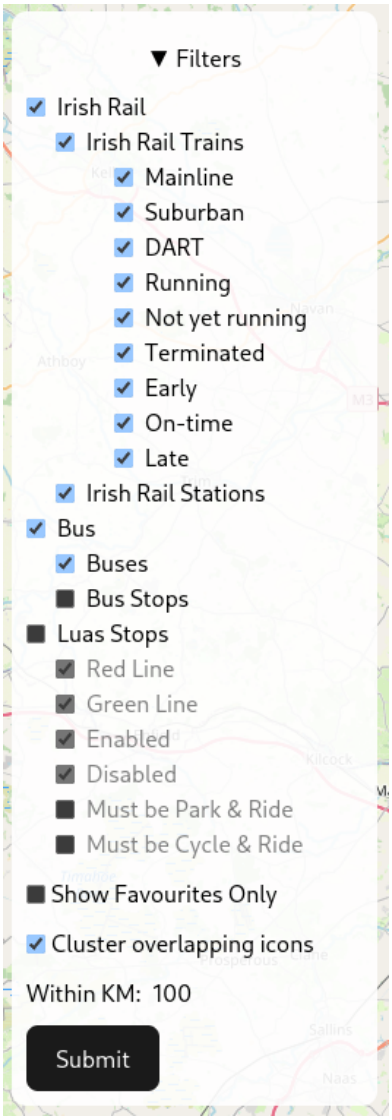


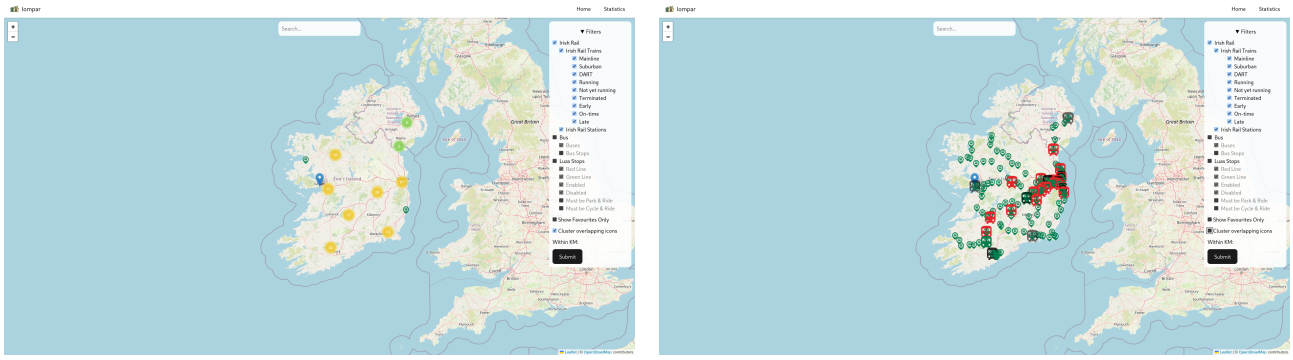
Figure 4.5: Screenshot of the filters side-panel

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`[leaflet_markercluster]. 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**[norman_design_2013]: 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"[nielsenheuristics]).

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.

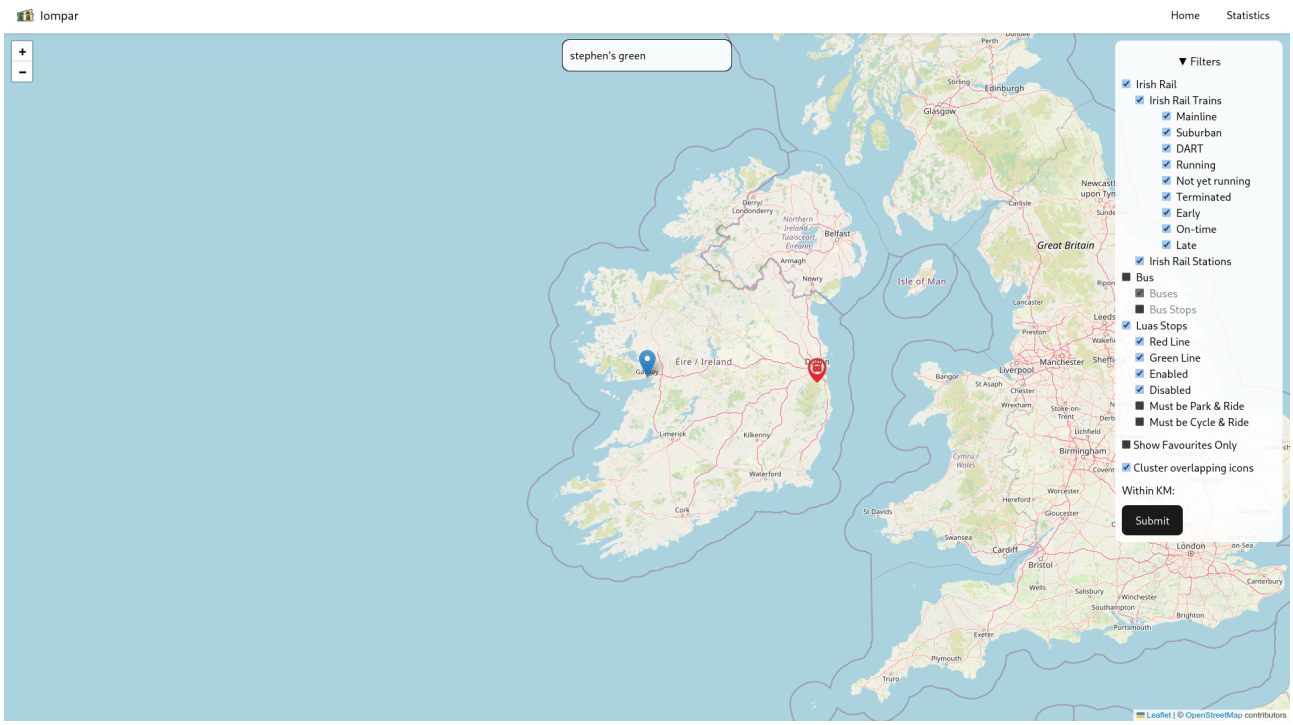


Figure 4.7: Screenshot of the search bar being used

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**[debounce] 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`[usememo] 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[leaflet] mapping library and map tiles from OpenStreetMap[osm]; this option was chosen as it is lightweight & performant, free to use, and is flexible & customisable. It is also by far the most popular mapping library, and has comprehensive documentation and a wide range of plug-ins. Other mapping options exist, such as Mapbox[mapbox] and the Google Maps API[gmaps], but these have limited free tiers compared to the completely free

& open-source option of Leaflet with OpenStreetMap. 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[osi] 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

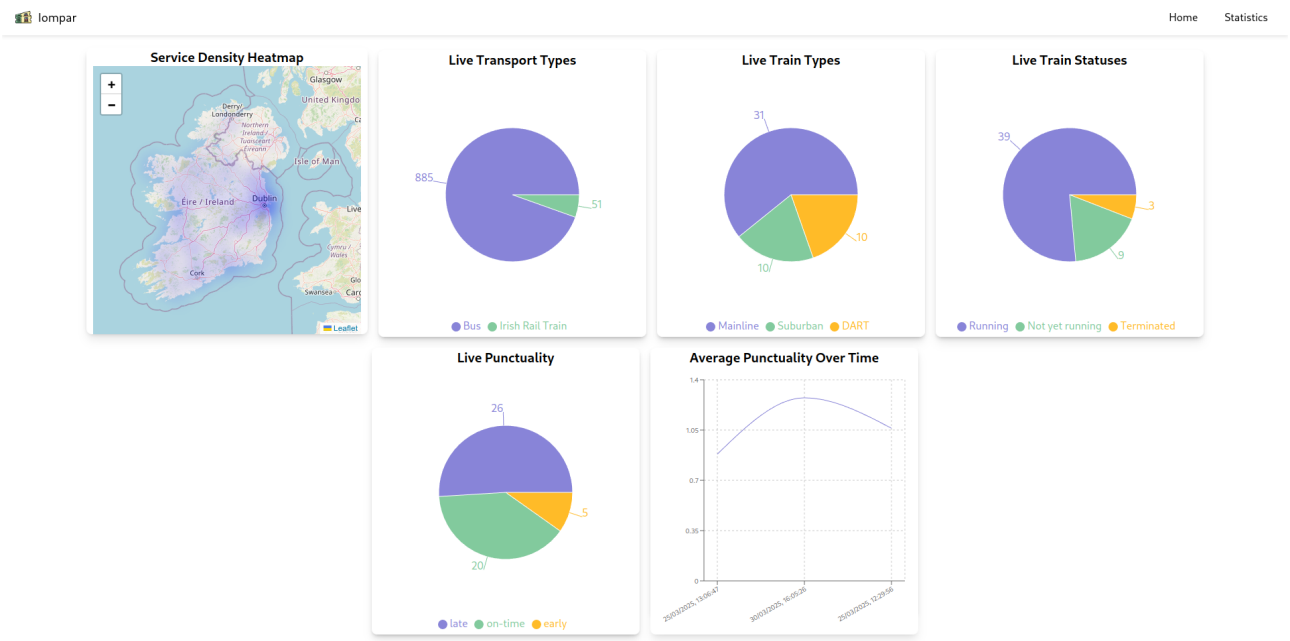


Figure 4.8: Screenshot of the statistics page

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[flexbox]. 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[atlassianpiechart, spotfirepie, inforirver]. 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

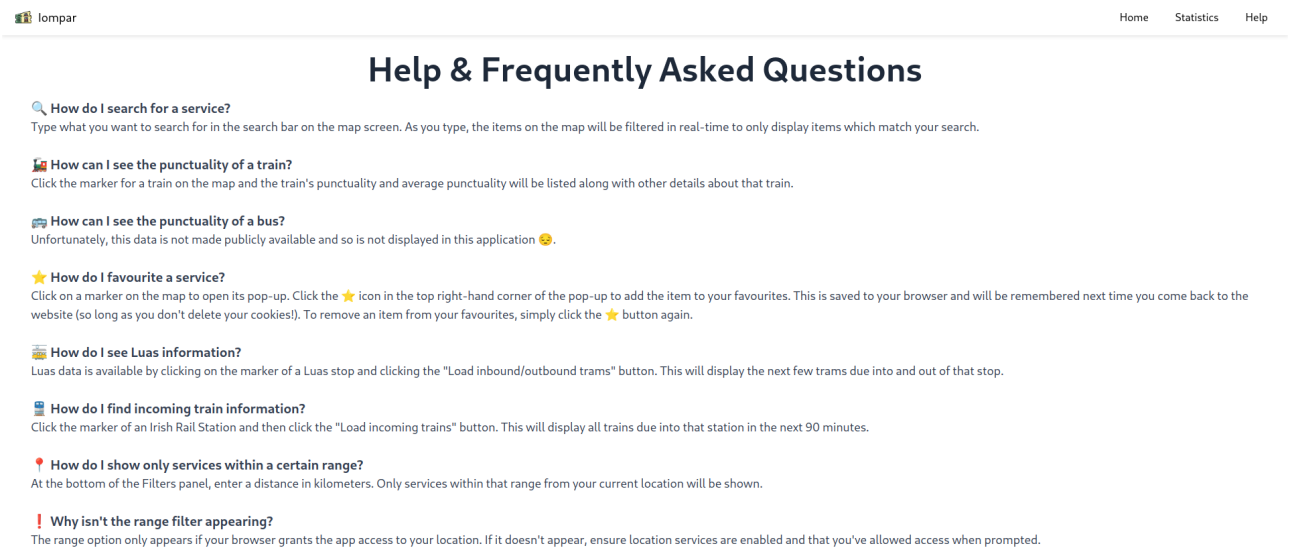


Figure 4.9: Screenshot of the help page

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