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
21321503

Academic Supervisor:

Dr. Adrian Clear

2025–03–30

Contents

Chapter 1

Introduction

1.1 Project Overview

1.1.1 Problem Statement

1.1.2 Background

1.2 Document Structure

Chapter 2

Research

2.1 Introduction

2.2 Data Sources

2.3 Similar Services

2.4 Technologies

2.4.1 Frontend Technologies

2.4.2 Backend Technologies

2.4.3 Project Management Technologies

2.5 Conclusion

Chapter 3

Requirements

3.1 Functional Requirements

3.2 Non-Functional Requirements

3.3 Use Cases

3.4 Constraints

Chapter 4

Backend Design & Implementation

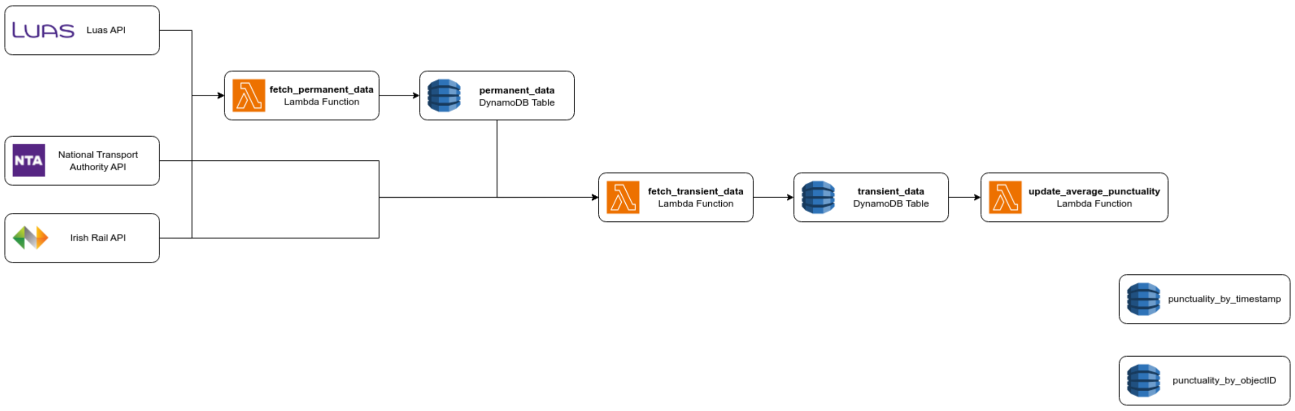

Luas API
National Transport Authority API
Irish Rail API
fetch_permanent_data Lambda Function
permanent_data DynamoDB Table
fetch_transient_data Lambda Function
transient_data DynamoDB Table
update_average_punctuality Lambda Function
punctuality_by_timestamp
punctuality_by_objectID

Figure 4.1: Backend architecture

4.1 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

Beyond what is returned for an item by its source API, two additional fields are included for each item: 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, 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[1]:

- 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.

```json
[
  {
    "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 ran 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[9]. 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.

```json
[
  {
    "average_punctuality": "0.5",
    "count": "2",
    "objectType": "IrishRailTrain",
    "objectID": "IrishRailTrain-P746"
  },
  {
    "average_punctuality": "-4",
    "count": "1",
    "objectType": "IrishRailTrain",
    "objectID": "IrishRailTrain-A731"
  },
  {
    "average_punctuality": "9.3333333333333333333333333333",
    "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 meant 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.

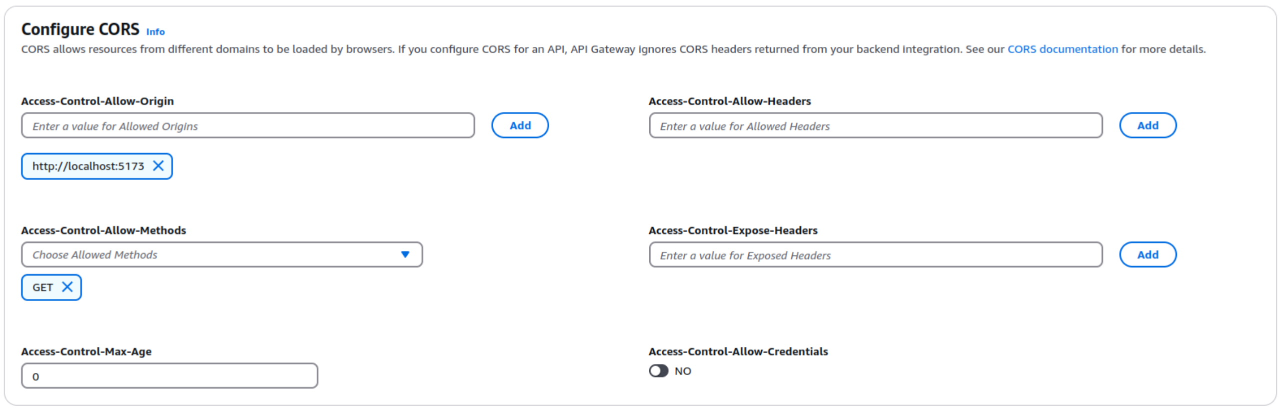
4.2 API Design

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[4]:

- **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[6]:
  - **HTTP APIs:** low latency, fast, & cost-effective APIs with support for various AWS microservices such as AWS Lambda, and native CORS support, 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, so 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 $3.5\times$ cheaper[5] 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[7]. 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[8]. 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.



Configure CORS Info

CORS allows resources from different domains to be loaded by browsers. If you configure CORS for an API, API Gateway ignores CORS headers returned from your backend integration. See our CORS documentation for more details.

Access-Control-Allow-Origin: Enter a value for Allowed Origins — Add — http://localhost:5173

Access-Control-Allow-Headers: Enter a value for Allowed Headers — Add

Access-Control-Allow-Methods: Choose Allowed Methods — GET

Access-Control-Expose-Headers: Enter a value for Exposed Headers — Add

Access-Control-Max-Age: 0

Access-Control-Allow-Credentials: NO

Figure 4.2: 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.

`/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`

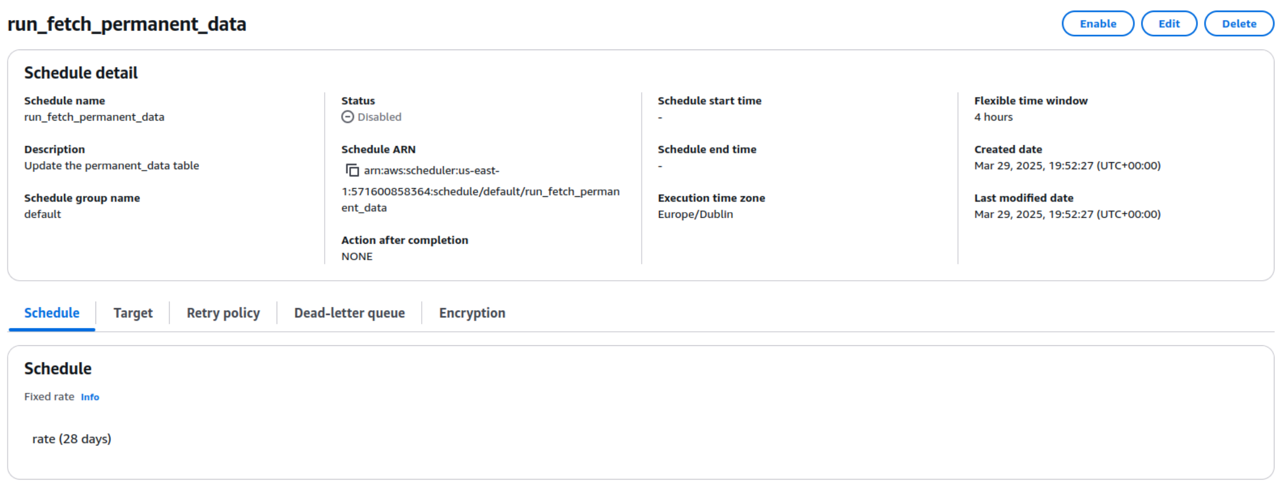
The `/return_all_coordinates` endpoint returns a JSON array of all current location co-ordinates in the transient data table for use in statistical analysis.

4.3 Serverless Functions

All the backend code & logic is implemented in a number of serverless functions, triggered as needed.

`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.



run_fetch_permanent_data

Schedule detail

Schedule name
run_fetch_permanent_data

Description
Update the permanent_data table

Schedule group name
default

Status
Disabled

Schedule ARN
arn:aws:scheduler:us-east-1:571600858364:schedule/default/run_fetch_permanent_data

Action after completion
NONE

Schedule start time
-

Schedule end time
-

Execution time zone
Europe/Dublin

Flexible time window
4 hours

Created date
Mar 29, 2025, 19:52:27 (UTC+00:00)

Last modified date
Mar 29, 2025, 19:52:27 (UTC+00:00)

Schedule

Fixed rate

rate (28 days)

Figure 4.3: 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.

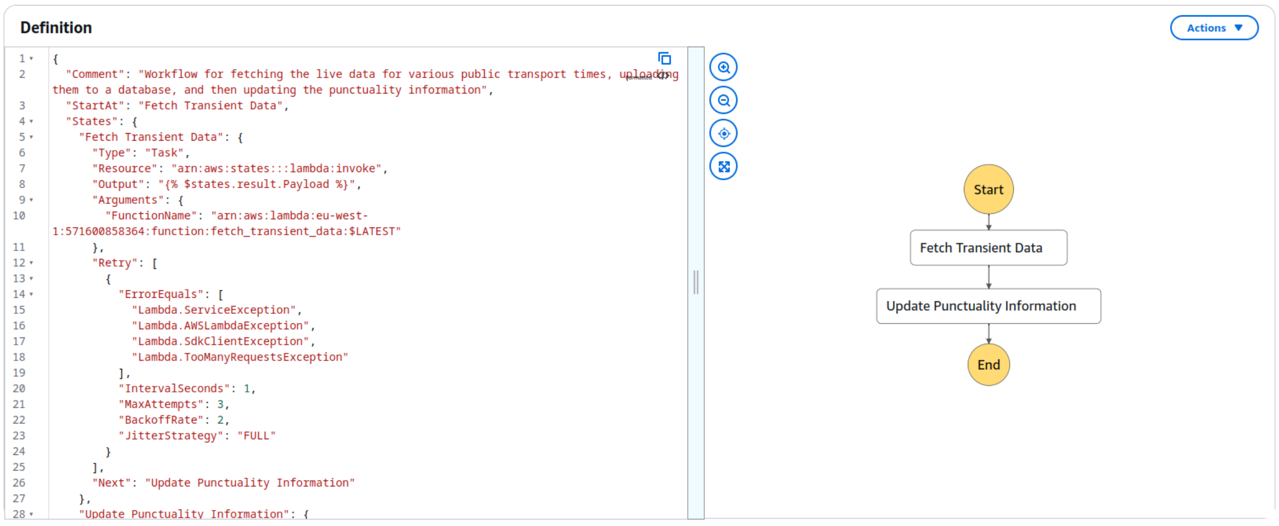
`fetch_transient_data`

The `fetch_transient_data` function operates much like the `fetch_transient_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 ran 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[2]) 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 ran 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.



```
{
  "Comment": "Workflow for fetching the live data for various public transport times, uploading them to a database, and then updating the punctuality information",
  "StartAt": "Fetch Transient Data",
  "States": {
    "Fetch Transient Data": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "Output": "{% $states.result.Payload %}",
      "Arguments": {
        "FunctionName": "arn:aws:lambda:eu-west-1:571600858364:function:fetch_transient_data:$LATEST"
      },
      "Retry": [
        {
          "ErrorEquals": [
            "Lambda.ServiceException",
            "Lambda.AWSLambdaException",
            "Lambda.SdkClientException",
            "Lambda.TooManyRequestsException"
          ],
          "IntervalSeconds": 1,
          "MaxAttempts": 3,
          "BackoffRate": 2,
          "JitterStrategy": "FULL"
        }
      ],
      "Next": "Update Punctuality Information"
    },
    "Update Punctuality Information": {
```

Figure 4.4: 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"[10].

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 `objectIDs` and defaults to returning all items in the table if no parameters are provided.

`return_punctuality_by_timestamp`

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

`return_all_coordinates`

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

`return_historical_data`

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

`return_luas_data`

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

`return_station_data`

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

Chapter 5

Frontend Design & Implementation

The frontend design is built following the Single-Page-Application (SPA)[12] design pattern using the React Router[11] 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 JavaSript 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 two "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 web application follows the Container/Presentational Pattern[3], 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 processing handling logic within them.

React components are reusable, self-contained pieces of the UI which act as building blocks for the application[13]; they can receive properties from their parent components, manage their own internal state, render other components within themselves, and respond to events.