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

2025–03–28

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

Design

4.1 Backend Design

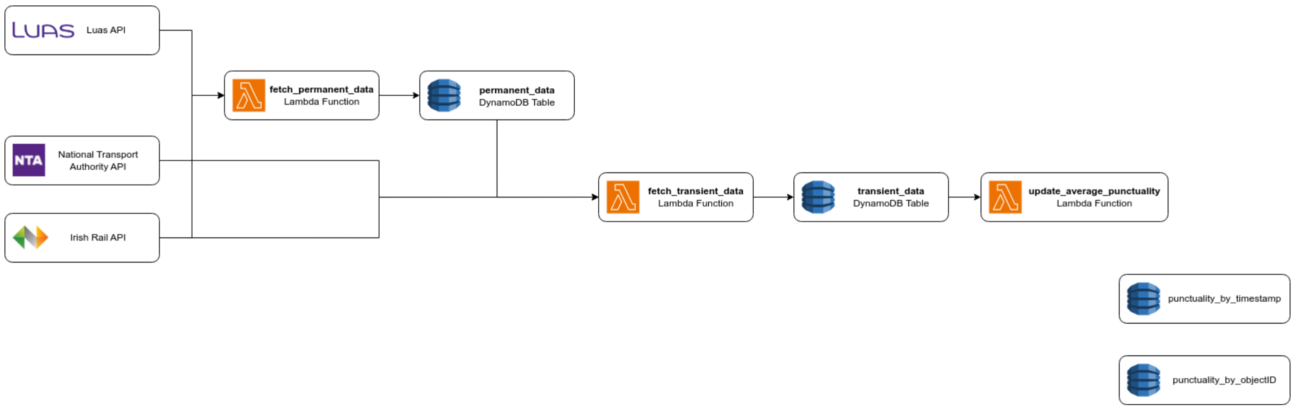


Figure 4.1: Backend architecture

4.1.1 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[5]. 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 relatively efficient by default as each column in the table functions as an index itself. In a No-SQL database, 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.

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

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

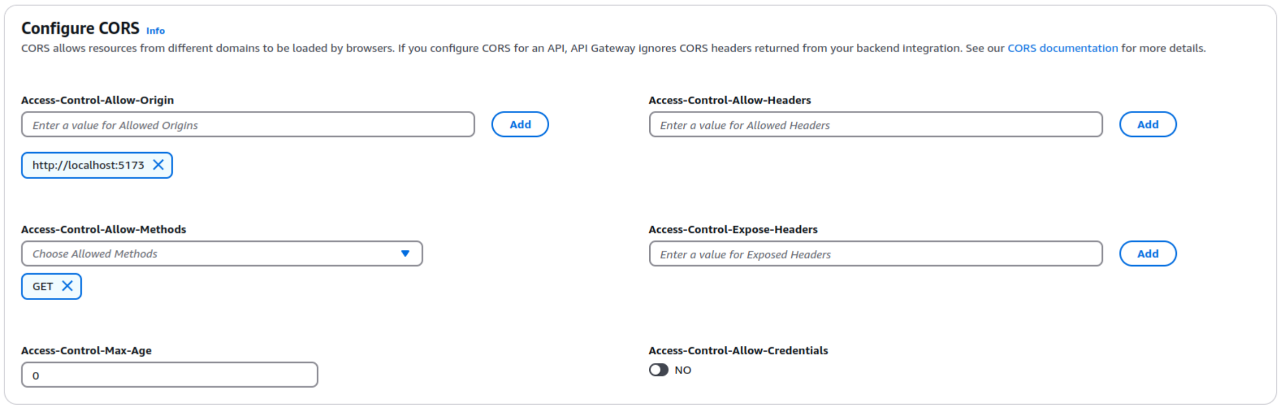
The partition key for this table is the `timestamp` value, and there is no need for a sort key or secondary index.

4.1.2 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[2]:

- **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[4]:
  - **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[3] than REST APIs resulted in the decision that a HTTP API would be more suitable.



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: http://localhost:5173

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

Access-Control-Allow-Methods: GET

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

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_objectID[?objectID=<object_id1>,<object_id2>]`

The `/return_punctuality_by_objectID` endpoint returns the contents of the `punctuality_by_objectID` DynamoDB table. It accepts a comma-separated list of `objectIDs` as query parameters, and defaults to returning the average punctuality for *all* items in the table if no `objectID` is specified.

`/return_punctuality_by_timestamp[?timestamp=<timestamp>]`

Like the `/return_punctuality_by_objectID` endpoint, 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 every historical co-ordinate stored in the transient data table, for use in statistical analysis.

4.1.3 Serverless Functions

4.2 Frontend Design

Chapter 5

Development

5.1 Introduction

5.2 Backend Development

5.3 Frontend Development

5.4 Development Considerations

Chapter 6

Code Quality

6.1 Introduction

6.2 Clean Coding Principles

6.3 Unit Testing

6.4 CI/CD

6.4.1 Continuous Integration

6.4.2 Continuous Deployment

Chapter 7

Conclusion

7.1 Evaluation

7.2 Reflection on Requirements

7.3 Reflection on Skill Development

7.4 Potential Future Work

Bibliography

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

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

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

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

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