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

2025–03–26

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

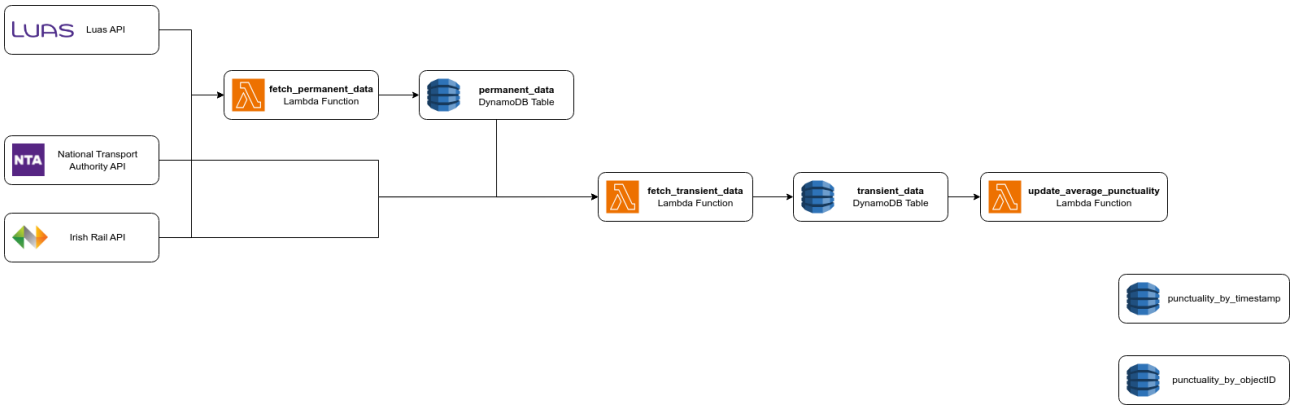

Figure 4.1: Backend architecture

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. The primary key of this table is the `objectID`, necessarily unique to each record in the table. This is constructed as a combination of the `objectType` (e.g., `BusStop`, `IrishRailStation`) and the unique identifier for that object returned by the API from which the data was retrieved. The prefix of the `objectType` is used here to guarantee uniqueness of the primary key in the case that two objects in the table of differing types have the same ID given to them by their respective source APIs.

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.

Instead of constructing a new attribute `objectID` for an item, it would also be possible to avoid creating a new attribute by instead using a composite primary key, with the partition key being the item's unique identifier in the system from which it came and the sort key being the `objectType`. This was rejected in favour of constructing a new attribute, `objectID` to serve as the simple primary key for this table for a number of reasons:

- The uniquely identifying attribute for each item given to it by the API from which said item was sourced has a different attribute name for every API; the unique identifier for bus stops is `busStopID`, for train stations is `trainStationCode`, et cetera. To use these values as the primary key in the table, each of these attributes would have to be re-named to some single, unifying title, creating additional parsing overhead when the data is being uploaded to the table, and making the item information more difficult to read for humans.
- Having a single uniquely identifying attribute for each item is useful on the frontend, allowing items to be easily uniquely identified without additional processing, useful for user functionality such as adding an item to the user's "favourites".
- The query efficiency improvements typically associated with a DynamoDB composite key would not apply to the type of queries this table is designed for with such a composite key structure. The data from this table will most often be queried by `objectType` in this application, such as in the event that a user wants to see bus stops or train stations or both on a map. The composite key would only speed up querying in the event that, for a number of different items with the same unique identifier, a query was ran on based on the type of those objects sharing an identifier, which is not a situation that is likely to arise for this application.

As mentioned in the final bullet-point above, this table is only intended for a single type of query: queries which seek to return all the items in the table of a certain `objectType` or `objectTypes`, such as when a frontend user requests to see bus stops, or train stations, or Luas stops, or some combination of the three. Therefore, it is imperative that such queries are efficient & fast. Since we cannot partition the data

4.1.2 API Design

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

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.