IRISH RAIL TRAIN TRACKER

CT216 GROUP PROJECT

Andrew Hayes (21321503), Conor McNamara (21378116), Jack Lennox (21337636), Owen Guillot (21378921)

Train Enthusiasts

CT216: Software Engineering I

University of Galway

March 2023

Contents

1 Introduction

1.1 Problem Statement and Project Description

The initial idea of this project was to design and build a real-time tracker that plotted and displayed Irish trains on a map. We were aware of a public API ran by Irish Rail which provided (amongst other data) the live co-ordinates of every train being ran by Irish Rail at any given time. Our intention was to make a replacement to the official live maps provided by Irish Rail, which had numerous issues.

These problems include:

- The live map provided by the official Irish Rail app has accuracy issues and invents data to fill in the gaps between API calls so that the map can have a continuous movement animation. We thought that this was inaccurate because at any given time, there was no way of knowing if a train was actually in the location that was being reported on the map or if the information had been simply made up to present a more seamless façade to the user. Frequently, we observed discrepancies that were equivalent to about 30 minutes travel, where we knew a train to be in one location, but the app displayed it elsewhere, a great distance away.
- The Irish Rail maps have unintuitive and indistinct icons for the various types of trains and stations. For example, the web version of the map makes no visual distinction between mainline trains and DARTs at all, while the app version uses icons that are slightly different shades of green to distinguish between them. Neither of these seemed acceptable to us.

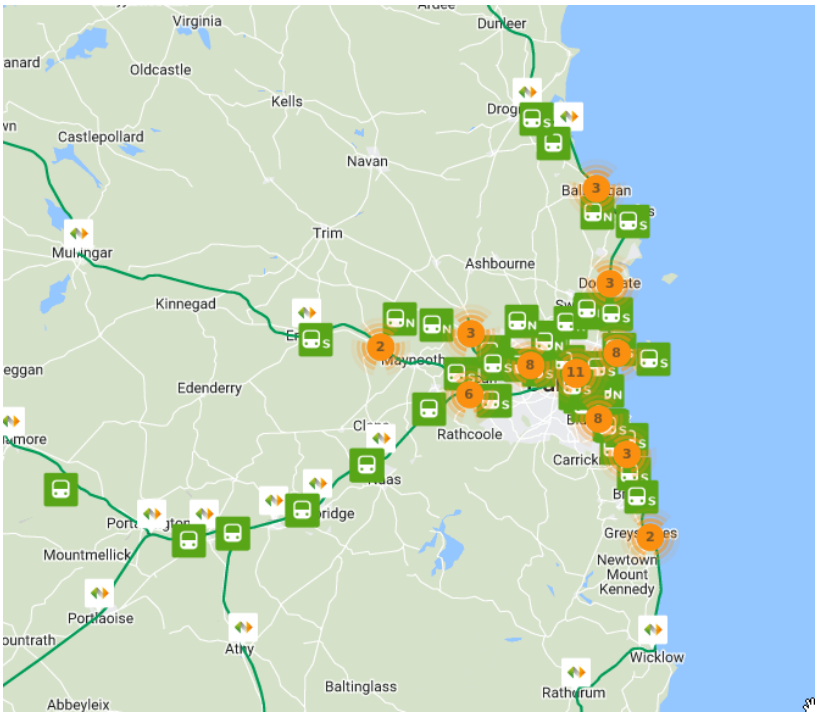

Figure 1: Indistinct Train and DART Icons [Source: Irish Rail Live Map (Web Version)]

- There is an overall lack of attention to detail in the design. The web version of the Irish Rail live map displays each train's public message (which is essentially the status of the train), rendering as plaintext the `\n` escape sequences that are supposed to represent a newline, resulting in strings such as "\n13:08 - Maynooth to Dublin Connolly (1 mins late)\nDeparted Maynooth next stop Leixlip (Louisa Bridge)" being displayed to the user verbatim. Clearly, the data received from the API is not groomed in any way to make it more intelligible or human-readable, especially for non-tech-savvy users who may be confused by the presence of the `\n`.

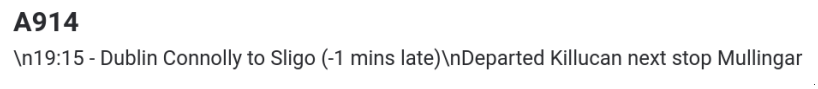

Figure 2: Unhandled Escape Sequences [Source: Irish Rail Live Map (Web Version)]

We intended to use newer technologies and software development methodologies that we had encountered to build a better, more user friendly, and intuitive live train tracker. We decided that we would do our web hosting using Firebase and our data storage with Firestore, the data in question being fetched from the Irish Rail public API. We decided that we would use the API data to plot each train on a map, and that we would use OpenLayers as the source of our map layers.

1.2 Project Goals

Our initial goals could be summarised as follows:

- Create a live train tracker that presents only data that is as accurate as possible, with none of the conjecturing that the Irish Rail's app employed.
- Create an intuitive, easy to use, and visually pleasing user interface that could be understood and made use of by almost anyone, regardless of their technical ability.
- Ensure that the map has clear visual distinctions between normal trains and DARTs, late trains and early trains, running and not running trains, etc.
- Implement a clear and precise data insights page on the website that gives data in a more objective format than the visual format of the map.
- Create a powerful filtering system for the trains shown on the map so that the user may choose exactly which categories they want to see.
- Implement a robust user system that allowed the user to save their map filter preferences, allowing them to re-use them whenever they visit the site, regardless of the device that they are on and without having to re-select them each time.
- Implement a search feature so that the user may search for individual trains or stations on the map, only displaying those which contain the searched-for keyword.

As our project went on, we also developed some further goals. It became clear that we wanted to:

- Manage and plan our project using Scrum and Agile methodologies, and to make extensive use of software such as Atlassian's Jira.
- Make extensive use of web development frameworks such as VueJS and Bootstrap, and to get as much as possible out of using Firebase hosting and cloud functions.
- Effectively use Git source management, making extensive use of branching and pull requests, and to never merge a pull request to the master branch without first getting a review from another teammate.
- Design and implement a good testing environment using the Firebase Emulator to ensure that our functions were working as intended.
- Implement an effective CI/CD pipeline that would automatically run tests and deploy to Firebase if the code passed the unit and integration tests.

2 User Requirements

2.1 User Stories

2.1.1 Real-Time Train Plotting

The principle functionality of the project was the real-time train plotting feature. We wanted to take real, live, and up-to-date data from the Irish Rail API and display it in a clear, intuitive, and easy-to-digest visual format on a map. This would allow a user to easily visually track the locations and punctuality of the various Irish Rail trains in operation at any given time.

We imagined a user story of a user coming to the website with the intention of simply observing the goings-ons of Irish Rail trains at that given time. Our target user for this basic functionality would be someone with an interest in the current locations of various trains, such as your average "train enthusiast" or someone who wants to know if there are any trains nearby before they move equipment over a railway line.

2.1.2 Real-Time Data Insights

Another functionality that would be of use to a user was the Insights page. Again, we wanted to take live data and display current statistics that could be inferred from this data. To achieve this, we made use of pie charts and bar graphs which informed the user of the proportion of trains that were late, the proportion of DARTs versus normal trains, etc. We also used a text-based leaderboard of the earliest and latest trains operating at any given time.

Again, the user story in this case was a user coming to the website with the intention of simply observing the goings-ons of Irish Rail trains at that given tiime. Our target user for this basic functionality would be someone with an interest in the punctuality of various trains, particularly "train enthusiasts", but also researchers and reporters, and perhaps even Irish Rail officials themselves.

2.1.3 Map Search and Filtering

The search and filtering functionality was the key functionality that enabled the broadest use cases. Up until this point, the user stories have been somewhat niche, and are not really something that your average person would be interested in. However, the ability to search for particular trains and stations makes this application extremely useful to everyone from your average commuter to hardcore train enthusiasts. There are a broad range of user stories in this category. A few, but not all, are summarised below:

- A commuter comes to the website with the intention of searching for their own train to keep track of it, e.g., see if it's late or early, track its location, etc. This is the principle usage of the application.
- Someone coming to the website with the intention of searching for a train that a friend or family member will be arriving on, to see if there are any delays and its location.
- Someone coming to the website with the intent to filter the map so that only running DARTs appear so that they may keep track of nearby DARTs and when they will be arriving to their local station. This might be more useful than the DART timetable, because while the timetable is a plan of intention, the map shows how things are actually going.
- Someone coming to the website with the intention of monitoring the punctuality of all the trains operating. The map can be filtered to only display late or early trains, or both, and details about each late train such as its unique train code and its current location can be obtained. This might be of interest to a researcher or an Irish Rail official, who might want to write a report on the punctuality of Irish Rail trains

2.1.4 User Authentication and Management

A core use case was to enable visitors of the site to create accounts. A successful user system would require people to do the following: signup, login, logout, delete their account, get a password reset email, and change an email or password. One of the services that Firebase offers is the provision of JavaScript-based functions to interact with their user management system. This means we wouldn't have to physically set up any actual REST requests ourselves for example. Their user authentication system also means we don't have to store passwords ourselves, which improves security.

2.1.5 User Map Preference Storage

The last of the key use cases for the map was the user map preferences storage. We imagined a user story of someone who was a frequent user of the website, who came to the website for the same thing every time, such as tracking a particular train to Galway, or monitoring the DARTs in their local area. It would therefore be important that this user could save these filter preferences to improve their experience.

2.2 Prototypes

The first prototype that we built was simply a map embedded on a pure-HTML webpage. For this prototype, OpenStreetMap (a crowdsourced world map built on top of OpenLayers) was used. We quickly decided that the use of OpenStreetMap was not needed and that we could simply use plain OpenLayers instead.

Figure 3: First Prototype

The first proof of concept was built shortly thereafter using pure HTML and JavaScript. The purpose of this was to demonstrate that what we wanted to do for the project was indeed going to be possible. Live data was not used for this POC; instead, a JSON string of data was hard-coded in, which is why if you compare early screenshots of this project to each other, none of the trains ever move. This proof of concept only displayed trains, and made no other data available to the user other than the position of the train.

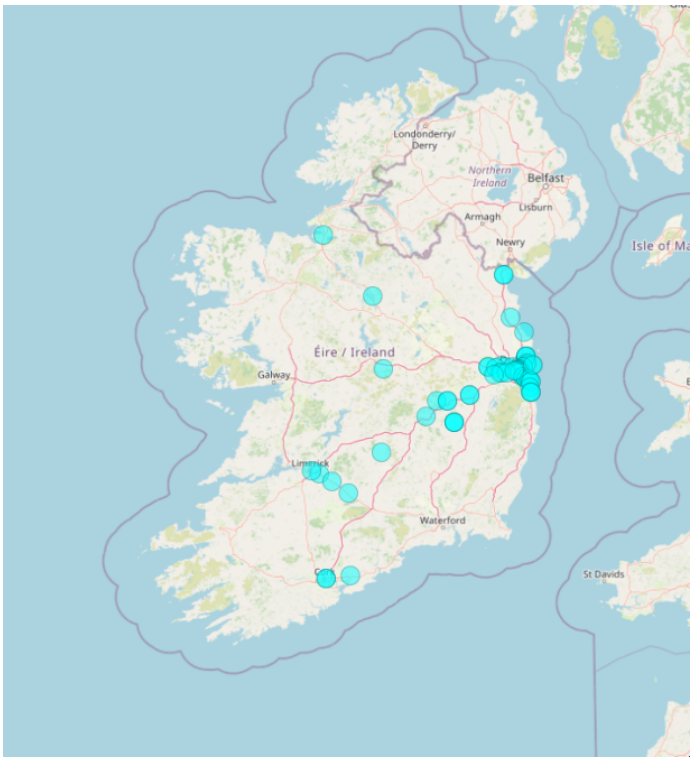

Figure 4: First Proof of Concept

We decided that we wanted a sidebar panel that would appear whenever a given train was clicked. The sidebar would provide information to the user about the train such as its origin, destination, etc.

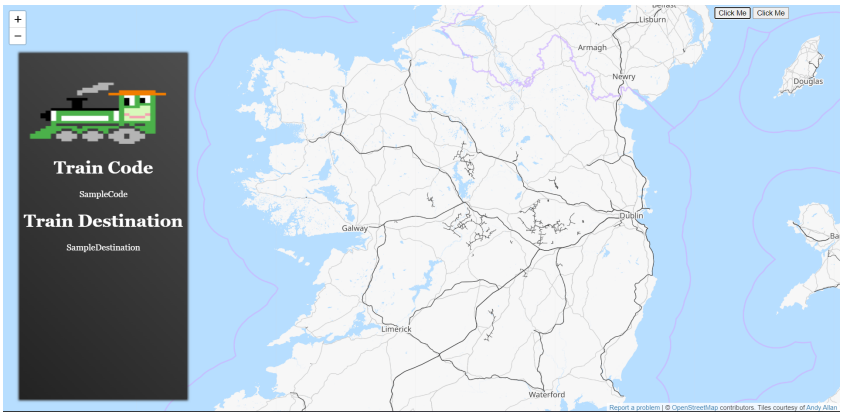

Figure 5: Sidebar Panel Proof of Concept

It became clear to us just how beneficial it would be to make the project using a JavaScript framework such as VueJS instead of just using plain HTML and JavaScript. We migrated all of our existing code to VueJS, which was a surprisingly challenging process, but well worth it, as it simplified the future development process.

Subsequently, we created basic on-click functionality for each train. When a given train was clicked, text information about that exact train would be inserted into the webpage.

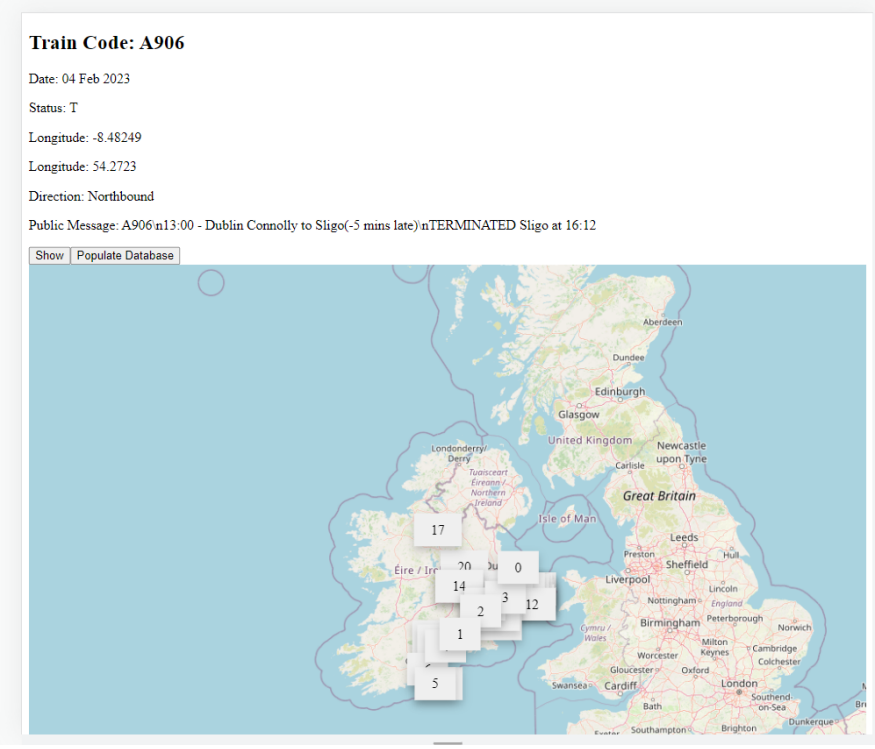

Figure 6: First Clickable Trains After Migration to VueJS

This on-click action was then changed to make a sidebar panel appear, which contained all the relevant information for a particular train, although it was not yet in a particularly nice or easy to read format. This panel would fade in from the side of the screen, rather than just instantly appearing on the map like in the previous prototypes.

Figure 7: Early Sidebar Panel Prototype

We also began expanding the application into more than just one page. We created the “Insights Page”, which displayed summary statistics on the current trains at any given point.

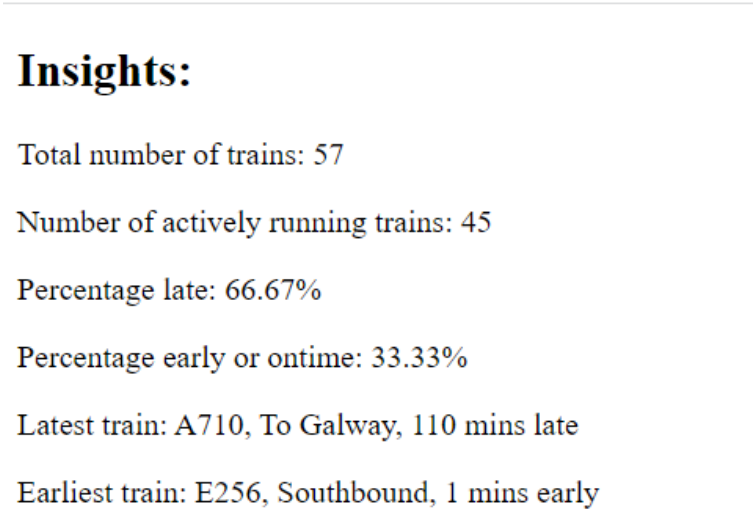

Figure 8: Early Prototype of the Insights Page

The square placeholders for the trains on the map were replaced by train icons, sourced under a permissive license. Originally, each train on the map was represented by a plain black train icon, but we quickly replaced this with either green or red icons, depending on whether the train was early/on-time or late. Any train that was late was red, and any other train (including those that were not running, terminated, early, or on-time) was green. However, we later made not-yet running or terminated train icons black to provide a clearer overview.

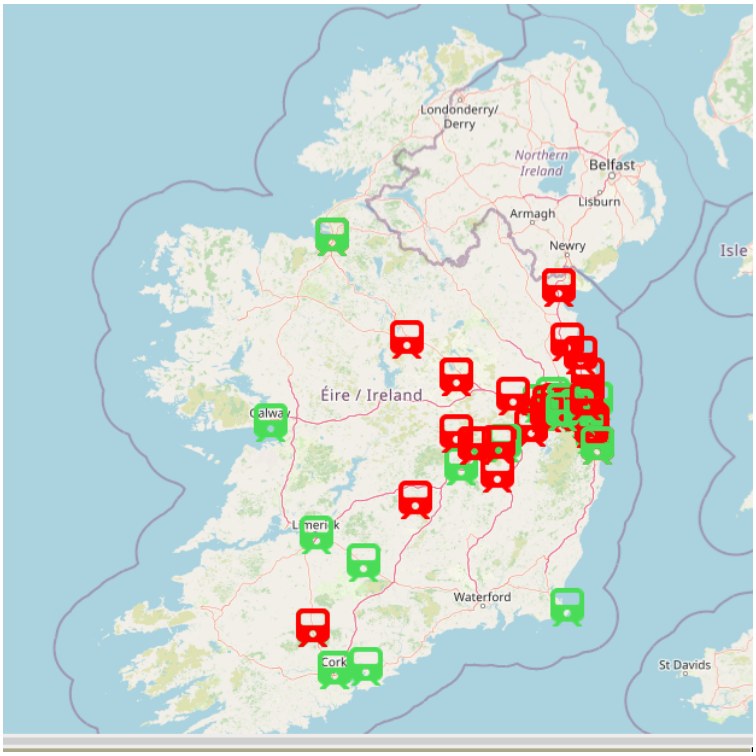

Figure 9: Colour-Coded Train Icons

Up until this point, the API was being called manually by the user via a button. This was clearly a sub-optimal solution, so the button was removed and the API was called on each page load instead, a temporary solution. Our ultimate plan was to have the API call be done server-side at regular intervals, regardless of how many people were using our application. This would prevent unnecessary strain on Irish Rail's API server. We also began to implement a user system, to which a user could sign up with their e-mail and a password. The ultimate plan for this user system was to allow the user to save their map filtering preferences, once they were implemented.

The next two features to be added were the public message ticker along the bottom of the screen and the "navbar" at the top of the page. The navbar gave the user access to the various other pages that we were adding to the application, including the insights page, a sign-up/login page, and an account settings page. The public message ticker idea was as follows: each train has a "public message" data field, which included up-to-date information about the train's whereabouts, its origin, its destination, and punctuality. Our idea was to display this information in a scrolling text bar along the bottom of the screen, as well as being available in the Sidebar panel.

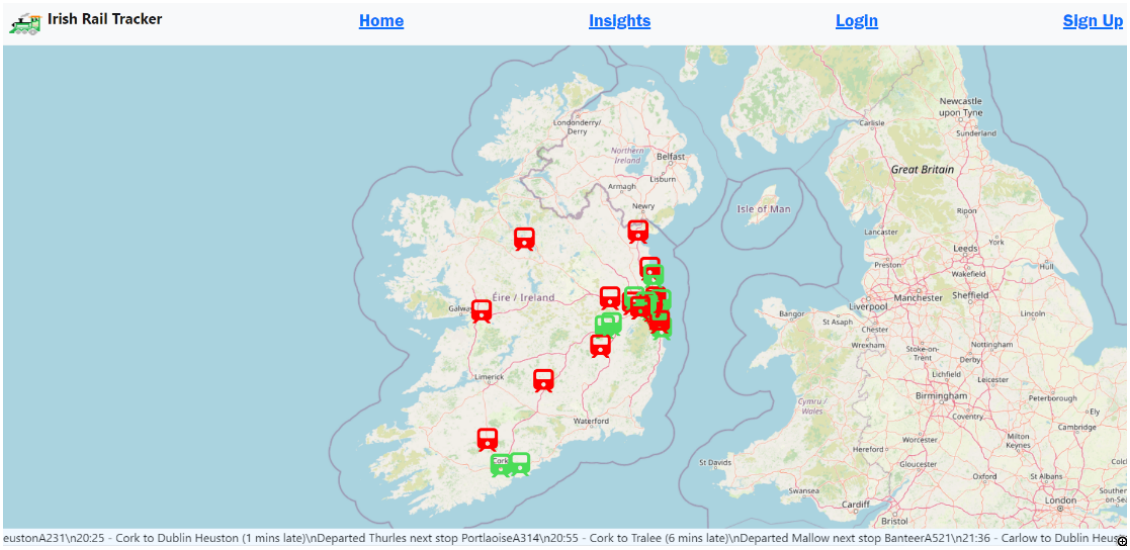

Figure 10: Screenshot of the Web App with Navbar and Public Messages Ticker

The ticker was later styled to be yellow (intended to be reminiscent of a news broadcast) and the delimiter between public messages was replaced with the Unicode bullet point symbol, for clarity and aesthetic purposes.

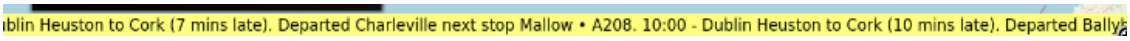

Figure 11: Styled Public Message Ticker

As time went on, we came to realise that rendering not-yet running and terminated trains as a green icon was confusing for the user, and was a lost opportunity to provide them with more information in an intuitive, visual manner. To rectify this, we brought back the plain black train icons that had previously been used for every train, and now just used them to distinguish trains that were not currently running or terminated. While we considered using some different kind of colour or pattern for the not-yet running and terminated trains, ultimately, we decided against it because we didn't feel that the user would be all too interested, or indeed, even care at all about the distinction between not-yet running and terminated trains. We felt that these trains broadly fell under the category of "Not Running".

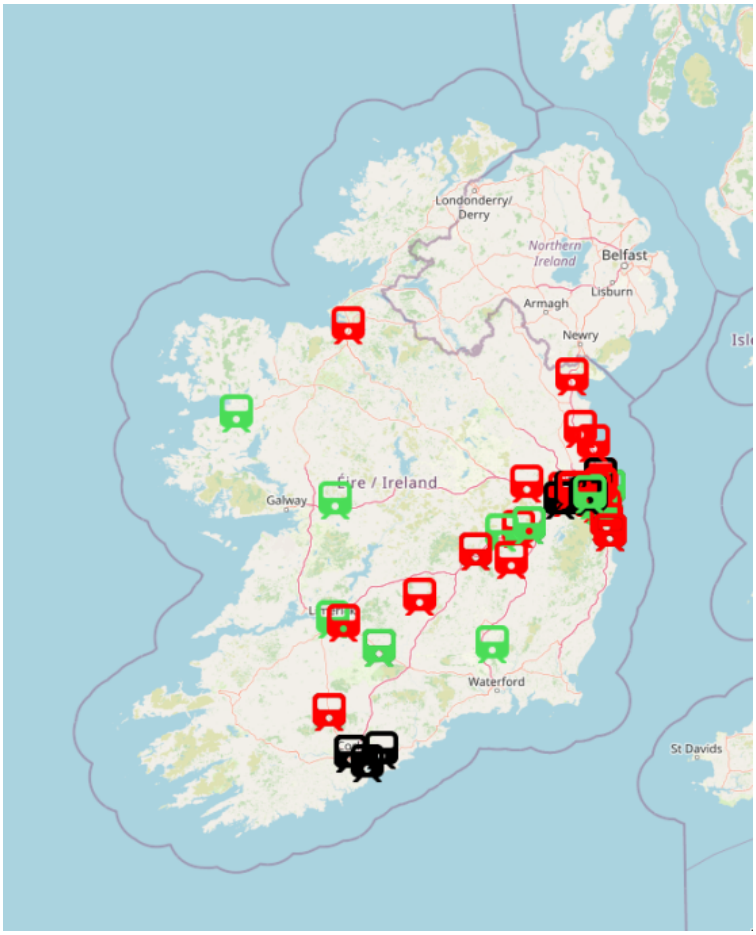

Figure 12: Added Black Icons to Distinguish Running and Not Running Trains

After doing this, we also added separate icons for DARTs to distinguish them easily from normal trains. We opted to use an icon that included the overhead power lines that the DARTs make use of. We followed the same colouring rules for these new DART icons, red if late, green if running and not late, and black if not running.

Figure 13: Added Distinct DART Icons

Finally, we implemented the search and filtering features on the map. The filtering allowed the user to filter in or out essentially any category of train imaginable, and it worked in conjunction with the search feature for extra-precise filtering. The search feature was case-insensitive, and allowed one to search the various fields in both the train data and the station data, including searching by keywords, codes, and public messages.

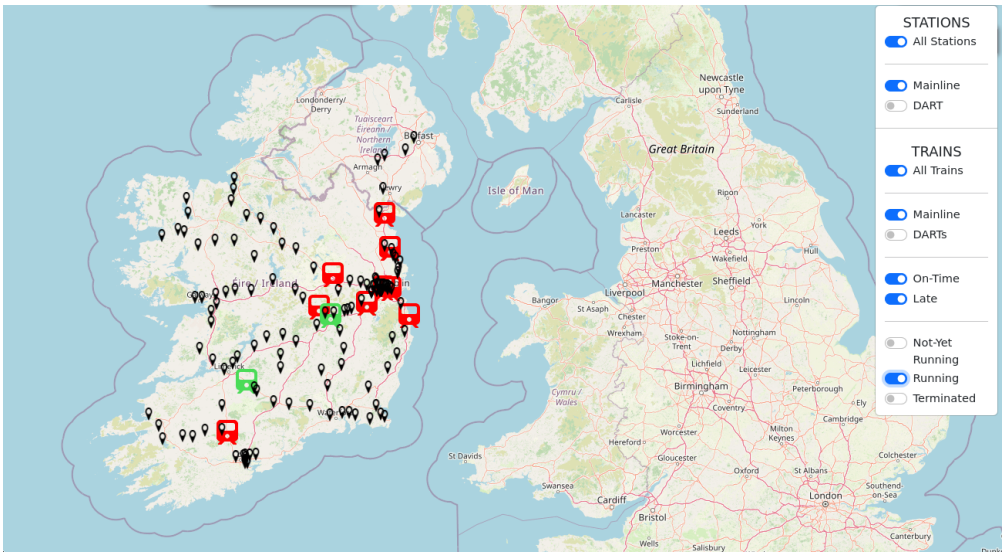

Figure 14: Displaying Only Running Mainline Trains and Mainline Stations

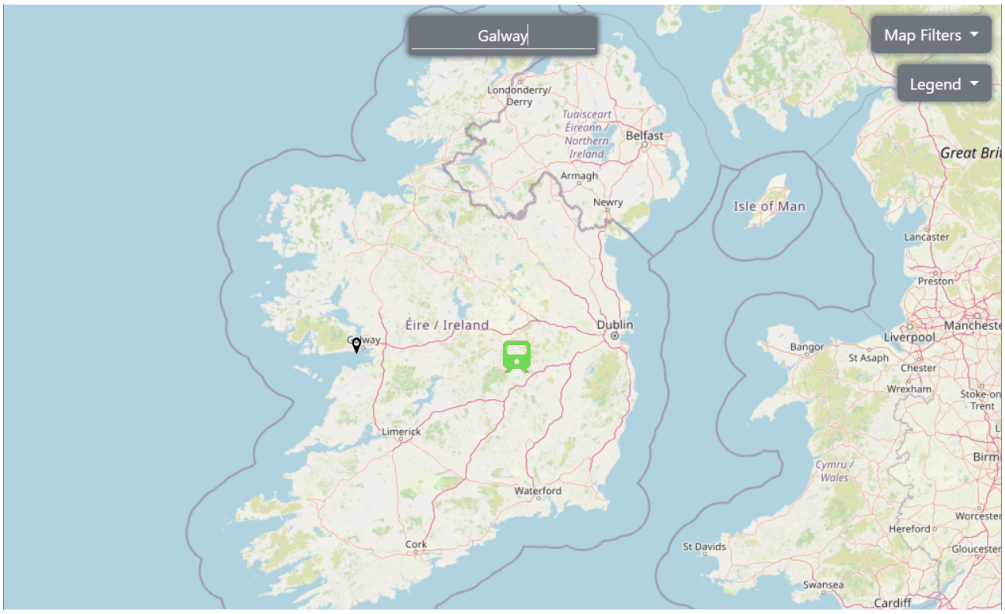

Figure 15: Effect of Searching for the Keyword “Galway”

3 System Design

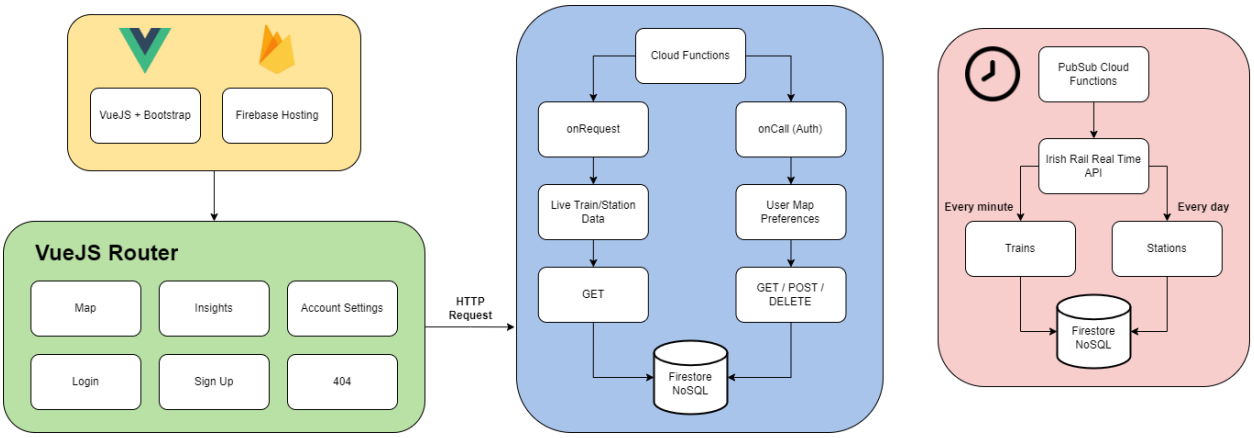

Figure 16: System Design Architecture

As shown above, the web app was created with VueJS and Bootstrap and hosted using Firebase. A client can view the following pages: map, insights, account settings, login, signup and a catch all 404 for any other route.

These pages interacted with serverless Firebase cloud functions. Firestore had three collections: train data, station data, and user map filter preferences. PubSub (publish-subscribe) functions were used to continuously request data from the real-time Irish Rail API and to then parse and insert this information into our Firestore database. Train data was requested every minute, while station data was only requested once a day as it was unlikely it would change.

The frontend client interacted with onRequest and onCall functions via HTTP requests. The map home page makes continuous GET requests for live train and station data from Firestore. Map filter preferences can be saved by users and these onCall functions are protected, meaning if the function request is made by an unregistered user, it stops immediately. These functions are in charge of managing create, read, and delete operations on user preferences. So, a user can save their preferences, the frontend can read these, and the user can also delete these if they wish.

4 Core Technologies

4.1 Backend

4.1.1 Firebase Hosting and Firestore Database

We chose to make use of the Firebase ecosystem for our project as recommended by the lecturer, for a variety of reasons. The first reason was very simple: we had been provided with free credits for Firebase which saved us from having to pay out of our pockets for hosting. However, this was not the only reason. One of Firebase's services is Firestore, a NoSQL database, storing data in a document structure that was highly similar to JavaScript Object Notation (JSON). This was rather convenient for us, as we were treating each train & station as a JavaScript object. Being able to store each train's data in the database as what is essentially a JSON object saved us from constructing and de-constructing objects as they were read/written to/from the database, and prevented unnecessary logic and overhead.

We obtained a nice and memorable URL from the Firebase hosting: irishrailtracker.web.app.

Firebase also allowed easy manual deployment by simply entering `npm run build && firebase deploy` in a terminal, and it also integrated nicely with out CI/CD pipeline using GitHub Actions.

Firebase has a useful suite of user system technologies that made managing users convenient. It handles the hashing of user passwords, meaning that we did not have to implement that ourselves and ensured that user data was secure. Firebase also has some built-in password strength rules, and would throw an error if a user's password was too simple, which again, saved us from having to implement that ourselves and ensured that the user data was secure. It has a similar set of utilities for user e-mail addresses. When a user signs up, Firebase ensures that their e-mail is valid and that it is not already in use, which simplifies the process greatly. It also handles sending the password reset e-mails, which saves us from having to set up our own e-mail server, etc.

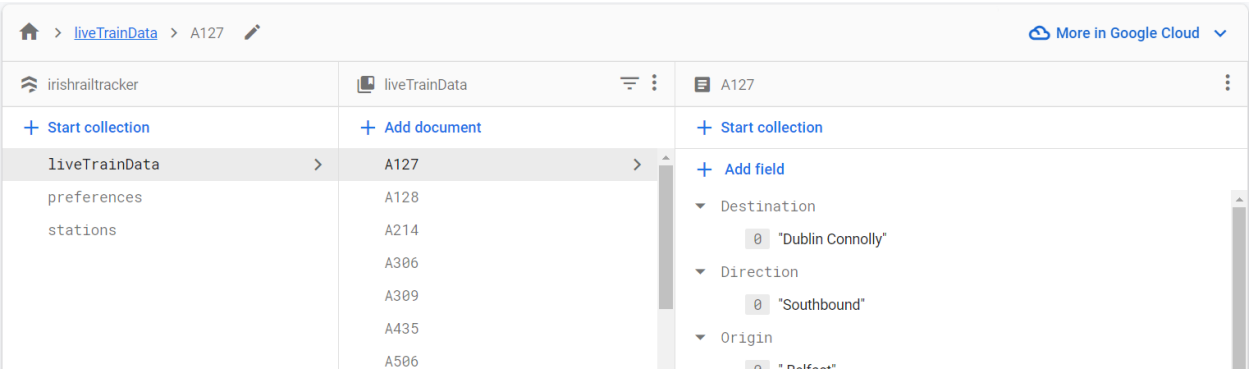

Figure 17: Firestore liveTrainData, Stations, and Preferences collections

Shown above are the NoSQL collections stored in Firestore. `liveTrainData` stores all the train objects, `stations` stores all the station objects, and `preferences` stores the map filter preferences for each user who saved them.

4.1.2 Serverless Functions

As mentioned in the System Design section, we used three types of Firebase serverless functions to essentially act as the backend. We needed to fetch new data from the Irish Rail API on a regular basis, which in our case was every minute for trains and every day for stations. Firebase offers PubSub (publish-subscribe) functions as a method to achieve this functionality. The Firebase emulator does not support PubSub functions, so regular onRequest functions that had the same logic we also implemented for local development. For example to specify that a function should run every minute, we can structure the function as follows:

```
exports.scheduledPostLiveTrainData = functions.pubsub.schedule('every 1 minutes')
    .onRun(async (context) => {}
```

Irish Rail's API returns data in XML format, which was more complicated than JSON. Given this, we converted the XML to JSON using the `xml2js` library. All HTTP requests were made using `axios`.

onRequest functions could be called anytime from anyone. onCall functions acted as secure functions that only ran when requested from a source where the client is a logged in user. These two types of functions were called from the client side.

A comprehensive list of the functions that were created are as follows:

- getStationData: onRequest. To fetch station data from Firestore
- postStationData: onRequest. To populate Firestore with station data from the Irish Rail API (for local development)
- scheduledPostStationData: PubSub. To populate Firestore with station data from the Irish Rail API (for production)
- getLiveTrainData: onRequest. To fetch train data from Firestore

- postLiveTrainData: onRequest. To populate Firestore with train data from the Irish Rail API (for local development)
- scheduledPostLiveTrainData: PubSub. To populate Firestore with train data from the Irish Rail API (for production)
- getPreferences: onCall. A secure function to fetch a user's preferences from Firestore
- postPreferences: onCall. A secure function to set a user's preferences into Firestore
- deletePreferences: onCall. A secure function to delete a user's preferences from Firestore

Each of these functions log messages back to Firebase depending on the progress it makes. These were specified by calling the `functions.logger.log("")` methods. Logging the functions sped up our debuggging process when an unknown error occured.

The onRequest functions had to handle cross-origin resource sharing (CORS). CORS is a set of browser rules where certain headers must be added to HTTP requests to access resources from a different domain. Using the object return by calling "require('cors')(origin: true)" and calling "response.set('Access-Control-Allow-Origin', '*')" and "response.set('Access-Control-Allow-Credentials', 'true')" we can make CORS-safe requests to external domains from the browser.

4.2 Frontend

4.2.1 VueJS Framework and Libraries

The frontend was written entirely in VueJS, although earlier prototypes were written in pure HTML. We made use of the `vue3-openlayers` library for sourcing the map tiles used in the map, and for plotting on the map. The website functions as a single page application (SPA), giving the illusion of multiple pages using the VueJS Router.

Through using the VueJS framework, we were able to split up our code into components. For example, the sidebar started out as part of the main page in earlier versions of the project, but it eventually became its own component. This made it far more manageable to make small adjustments as the codebase increased towards the end of the project. The use of components was also exceptionally beneficial for reducing repetition in the code. The navbar which features at the top of every page was made into its own component, instead of hardcoding it into every page. This also meant any changes we made to the navbar would be instantly applied to all pages, greatly saving time.

The VueJS framework opened many doors into different libraries which we utilised to better our project. There was the aforementioned `vue3-openlayers` which was integral to the idea of the project. There was also `vue-chartjs`, which we see in action on the "Insights" page. This library allowed us to easily visualise the train data, and make the experience more engaging and aesthetically pleasing for the user. Along with `vue3-chartjs`, we were also able to use `vue-bootstrap` to further improve the UI of the site. We used Bootstrap in conjunction with CSS to deliver the cleanest user experience we could. All the dropdown menus, slider buttons, and normal buttons were taken directly from the `vue-bootstrap` library. Small additions such as these really helped to tie the site together and give it a more modern look in comparison to the official Irish Rail counterpart.

4.2.2 Router

The links in the navbar worked through the use of the Vue "router". The router would simply load the desired page when activated. The navbar would not display the option to go to the accounts page if the user wasn't signed in, however a user could still try to access the page by using the URL for the page. To combat this, we made sure to check if the user was logged in before loading the accounts page. If the user wasn't logged in then they would be returned to the home page.

This check was implemented by marking the /account route with a beforeEnter, as shown:

```
1 { path: "/account", component:loadPage('AccountPage'), beforeEnter: isAuth }
```

beforeEnter was a function that decided if the user would be allowed to access this route or whether they would be sent back to the home page. This is implemented below:

```
function isAuth(to, from, next) {
    const auth = getAuth(app)
    onAuthStateChanged(auth, (user) => {
        // user is logged in, continue to page
        if (user) {
            return next()
        }
        // user is logged out, send back to home page
        else {
            return next({path: "/"})
        }
    })
}
```

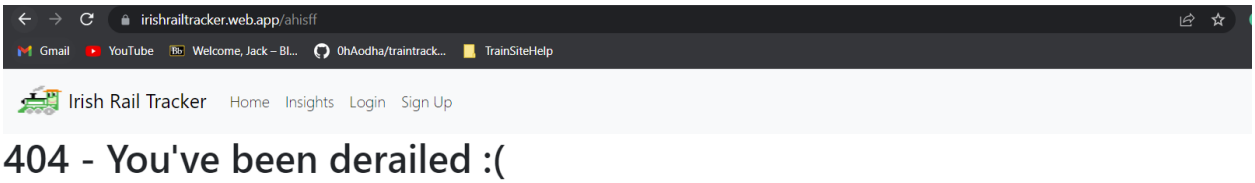

Figure 18: 404 Error Message

As shown above, a 404 page was implemented as a catch all route, which was activated if a client tried to access any route that was not already specified. This page would display “404 - You’ve been derailed”, and have the navbar at the top to allow the user to return to whatever page they wished. This was implemented by simply specifying:

```
{ path: "/:catchAll(.*)", component:loadPage('404Page') }
```

4.2.3 Store State Management

A VueJS store can be created by instantiating a `reactive` object. We use to store to have global access to variables across different components. This was essential to implement, as various components use the same data, and a mechanism was needed to have a central location where any component could access vital data. Any variables stored here, such as the loggedIn status, could get got or set from any component.

4.2.4 Responsiveness

The responsiveness in this project was achieved through CSS and Bootstrap. We used the in-built mobile view on Chrome to test the look of the site on mobile, as well as using our own phones. Conveniently, Bootstrap components often are responsive by default. For example, the navbar would offer a hamburger dropdown menu when the screensize became too small. For the rest of the CSS, we decided on a uniform cutoff point of 850px. Once the screen size was smaller than this we changed the website layout. This was implemented using CSS media queries.

The first issue we had was with the sidebar. We initially tried to keep the sidebar to the side on smaller screens, but as a result the information was harder to read. Eventually we came to the decision that it should take up a majority of the screen to be more readable. On mobile, the search bar moves from the center of the page to the left to make space for the sidebar.

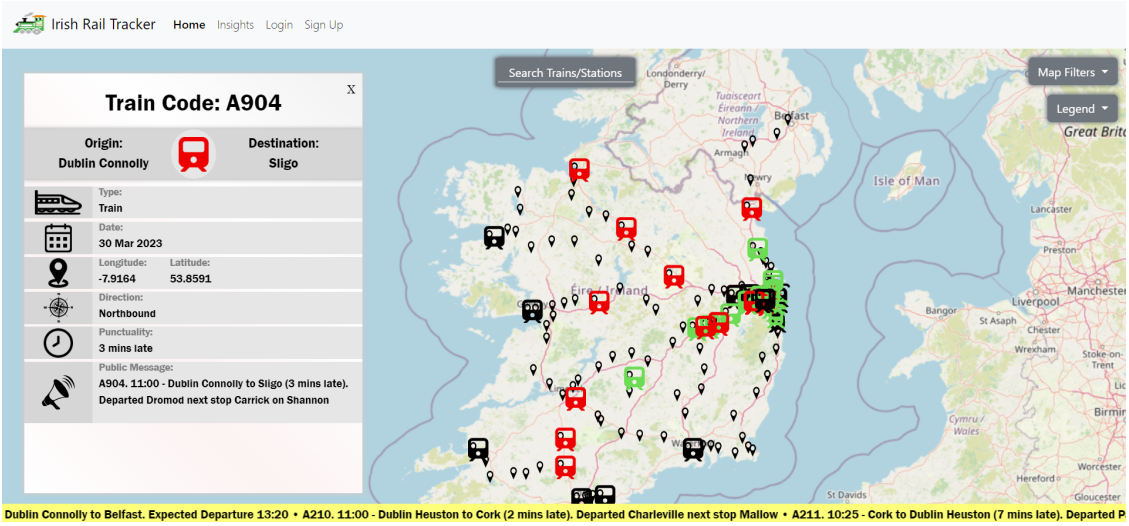

Figure 19: Sidebar on Desktop

Figure 20: Sidebar on Mobile

The insights page changes entirely on screen sizes lower than 850px. We initially considered not showing the graphs on mobile, but we eventually realised that they are the salient features of the page. We then tried keeping them horizontally in line with each other and reducing their size, but this just made them harder to read and interpret. As a result we decided to stack them on top of each other on mobile and allow the users to scroll down through them.

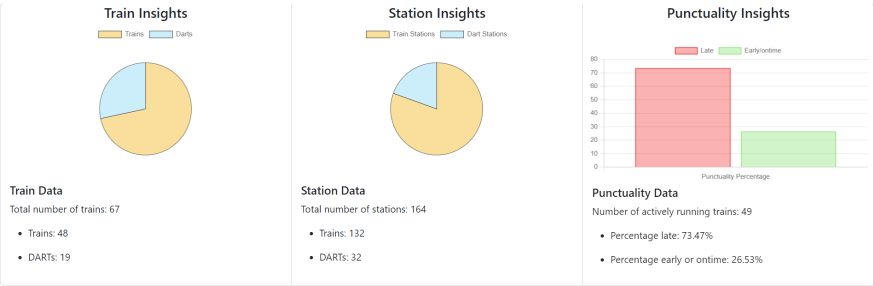

Figure 21: Graphs on Desktop

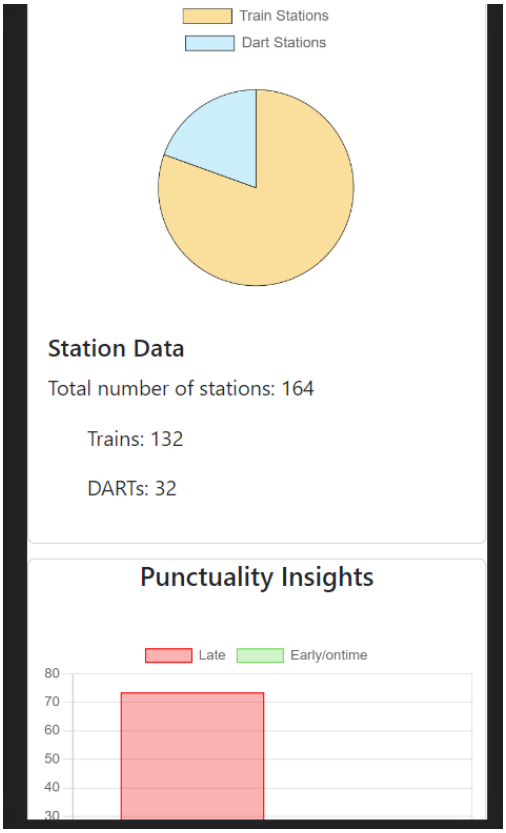

Figure 22: Graphs on Mobile

Along with the graphs, we also had the leaderboard to address. We really liked the design of it on desktop and tried our best to keep it congruent on mobile, but sadly it was once again hard to read when made smaller. To combat this, we removed the table design on mobile and instead put the headers in line with the data for each item in the leaderboard.

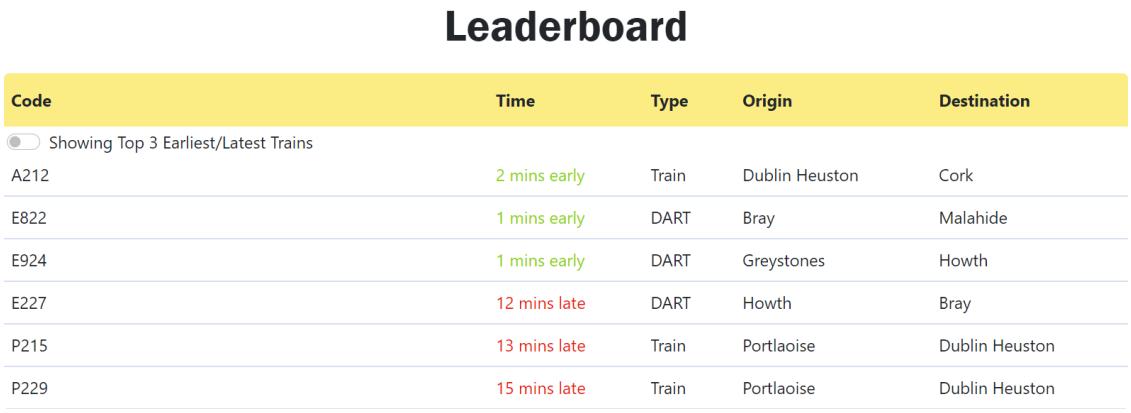

Figure 23: Leaderboard on Desktop

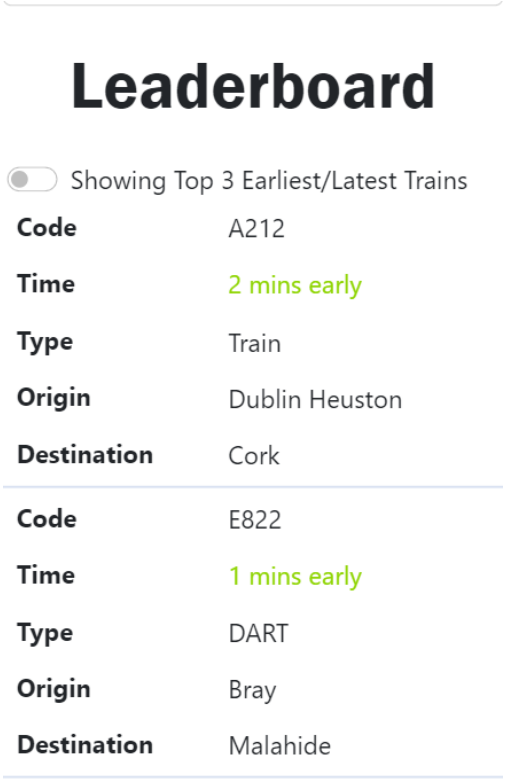

Figure 24: Leaderboard on Mobile

4.2.5 Map Page

OpenLayers

The map is sourced from OpenLayers. At the start of the project, we considered alternatives to OpenLayers including Google Maps, but we decided in favour of OpenLayers for a variety of reasons. One such reason was that acquiring an API key for Google Maps required payment and since we intended to make no money from the project, we favoured the free option. In addition to this, the licensing of the OpenLayers map tiles was far more permissive than Google Maps, and allowed us to do essentially whatever we wanted with them so long as we didn't masquerade as OpenLayers themselves. OpenLayers actually allows anyone to use their Content Delivery Network (CDN) to obtain the map tile images on the fly, free of charge, so long as they are

not putting excessive pressure on the CDN. Taking the scope and the scale of our project into consideration, we decided that our project was not in any way close to a scale that would put any noticeable pressure on OpenLayers' CDN.

OpenLayers also provided a superior plotting library, complete with plentiful and in-depth documentation. OpenLayers allowed us to easily plot the icons using the longitude and latitude provided to us by the Irish Rail API. Furthermore, we also decided that we preferred the aesthetic appearance of OpenLayers, as it showed the trainlines on them, which we felt contributed to an easier to follow user experience.

Train Icons and Sidebar

The icons used on the map are simple PNG or SVG files sourced from the internet under permissive Creative Commons licenses which allowed us to use them without any attribution or copyright issues. The train icons were originally plain black PNG files, but we made a red and a green version ourselves which were to be used to represent late and early trains. We considered the possibility of using SVG files for this, the benefit being that we could have only one icon which we changed the colour of programmatically on the fly, but re-colouring the icon for each train icon was computationally expensive. As we were only using three different colours for each icon, it made far more sense to just generate PNG files of the three different colours once, and then point to these files in the code.

The icons are clickable and increase in size on mouse hover. Clicking an icon reveals the sidebar. The sidebar is comprised of the Train Code (the unique Irish Rail ID for the train) at the top, an appropriately-coloured icon below it, and all the other data on that particular train below that. The train sidebar also shows the Origin and the Destination of the train beside its icon. It also includes an "X" button to close the sidebar. An issue appeared in the sidebar with trains that were not running, as they had no punctuality data. That is to say, a train that isn't running can't be late or early. To combat this we used a `v-if` statement to see if a train was running, if it wasn't then we moved the public message div to where the punctuality one would have been.

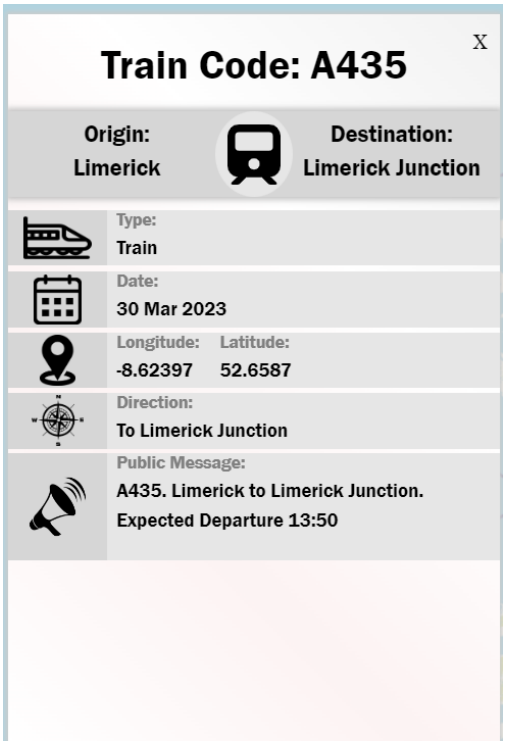

Figure 25: Sidebar for a Non-Running Train

Public Message Ticker

At the bottom of the page is the live ticker. This was a pre-existent component that we imported for use in our project from the `vue-marquee-text-component` library. We coloured it yellow and black to make it visually reminiscent of a news ticker such that you might see on BBC News or similar news stations. The text displayed in it is simply all the public messages from the trains in the Irish Rail API, with a Unicode bullet point character appearing in-between the messages to show where one message ends and a new one begins. Although the text is ordinarily constantly moving, a user can hover their mouse over the ticker to temporarily prevent the text moving from moving, so that they can read it. To restart the text stream, the user can simply stop hovering over the ticker by moving their mouse away. This function also works on mobile, however, as there is no mouse on mobile, the user is required to hold down on the ticker instead.

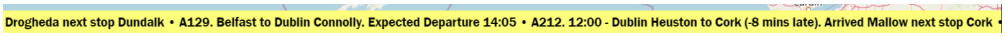

Figure 26: Live Ticker with Text

Map Search and Filtering

We were cognizant that having all the data/icons appear at once on the map page appears very daunting for the user, so we worked on implementing some filtering methods. The first, and most concise, method of filtering is the search bar. The search bar allows a user to input text, the text is then used to search through the public message of all the trains in the database. This means a user could search by origin, destination, or even train code very easily.

Along with the search bar, we also have filter buttons which can work in conjunction with the search bar. For example, if a user typed in “Dublin” and had the buttons set to only show late trains, then the user would see all late trains leaving or going to Dublin along with all stations with “Dublin” in the name (such as Dublin Heuston or Dublin Connolly).

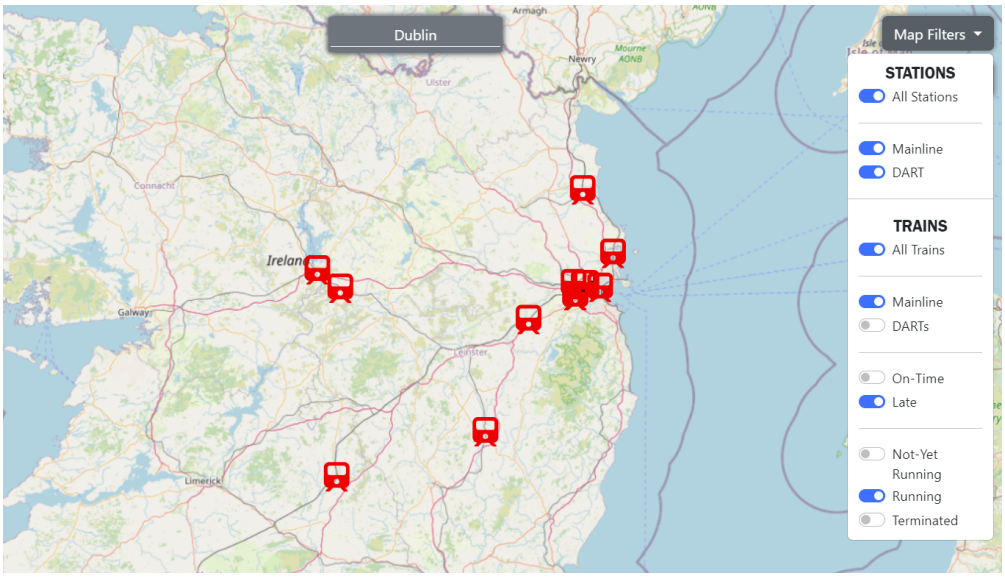

Figure 27: Map with Filters Applied

Finally, there are the dropdown menus that are implemented through the use of `vue-bootstrap`. The filters menu is one of these, and the other is the legend dropdown menu. They are activated when the button is clicked, and then deactivated once the button is clicked, or if another element is clicked.

4.2.6 Insights Page

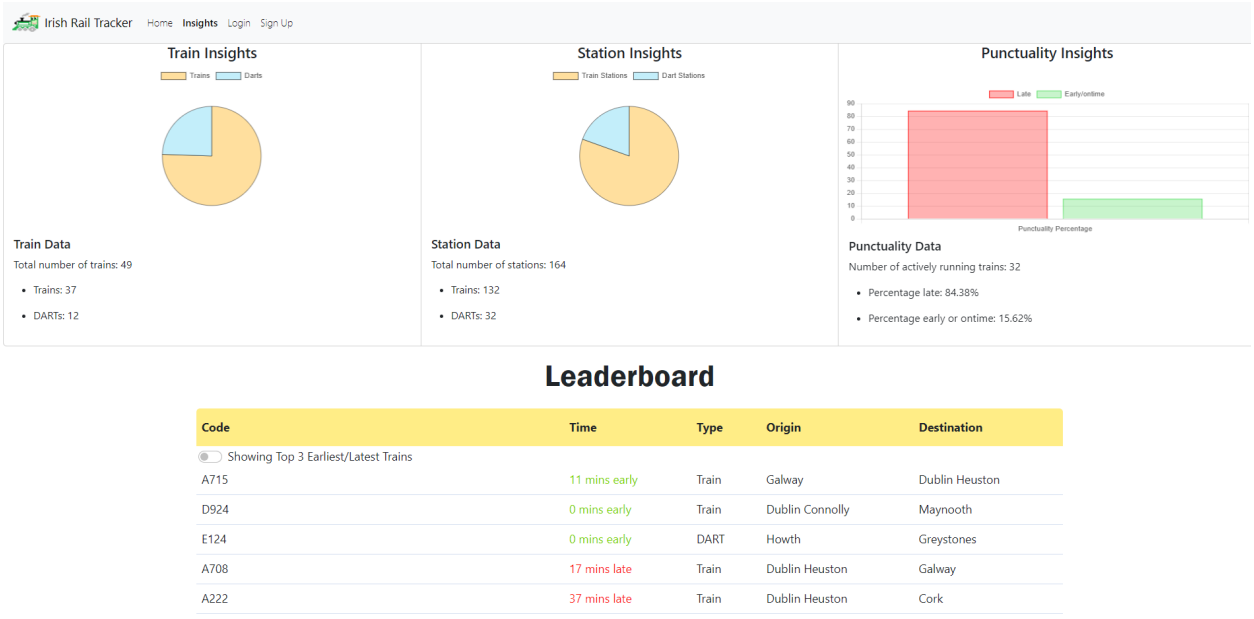

Figure 28: Insights Page (Zoomed Out)

The insights page is shown above. It makes great use of the `vue-chartjs` library. The pie charts and bar chart that take up the top of the page are created with this library. The graphs themselves are their own components. The pie chart component takes a Boolean input to decide whether it is to display the train insight pie chart, or the station insight pie chart. This Boolean input allowed us to use the same component for both pie charts, reducing code repetition. The charts take their data from the "store", which is a feature of VueJS. The train pie chart shows the amount of trains currently in service, this includes not yet running trains but not those that are terminated. On the other hand, the bar chart only includes the running trains. Once again, this updates in real time, changing whenever an ontime/early train becomes late or vice-versa.

The leaderboard utilises a JavaScript function which takes all trains sorted in descending order from earliest to latest. By default it will only show the top 3 latest and earliest trains. However, the user can toggle it to show all trains that are actively running. We also decided it would be beneficial to colour the text green for early and red for late to keep with our vision for as much information to be interpreted as easily as possible.

4.2.7 Login and Sign Up Pages, and JWT Tokens

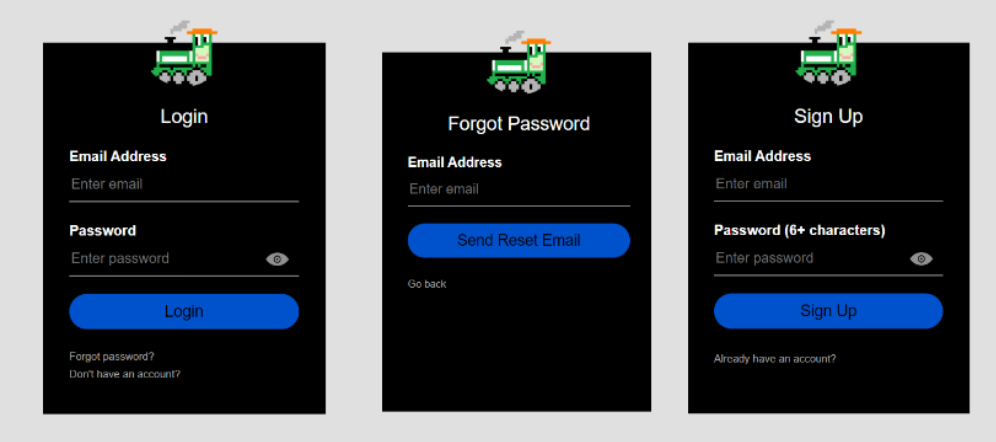

Figure 29: Login and Sign-Up Components

The above image showcases login and sign up components. The forgot password panel is part of the login component and becomes visible if the client clicks "Forgot password?". These components make use of Firebase methods, such as createUserWithEmailAndPassword(), resetPasswordEmail() and signInWithEmailAndPassword(). Using these pre-built methods spec up our development process. We just needed to pass the necessary parameters, such as the client's entered email or password for example. The image below shows the password reset email sent by Firebase to a registered user if they requested it.

Hello,

Follow this link to reset your project-753565032771 password for your a.hayes18@nuigalway.ie account.

https://irishrailtracker.firebaseapp.com/__/auth/action?mode=resetPassword&oobCode=kMe5Qp_hswjsChMb1G-cR3GTKCna8W6mvOh1IgWkjl0AAAGHPqC2zA&apiKey=AIzaSyBsP9SNH1FuRBpbhcvo9flXbAQWydpctd‹

If you didn't ask to reset your password, you can ignore this email.

Thanks,

Your project-753565032771 team

Figure 30: Password Reset Email

When a client logs in or signs up, Firebase will generate and send the client a JSON Web Token (JWT), which is used for identification. Structurally, this token is hashed and has a header, payload and signature. After this JWT is returned to the client's browser and stored from Firebase, their browser would send this token to Firebase if they were interacting with a part of our site that required it, such as changing your password. At which point, Firebase would send the appropriate response back to the user, where this is handled.

4.2.8 Account Settings Page

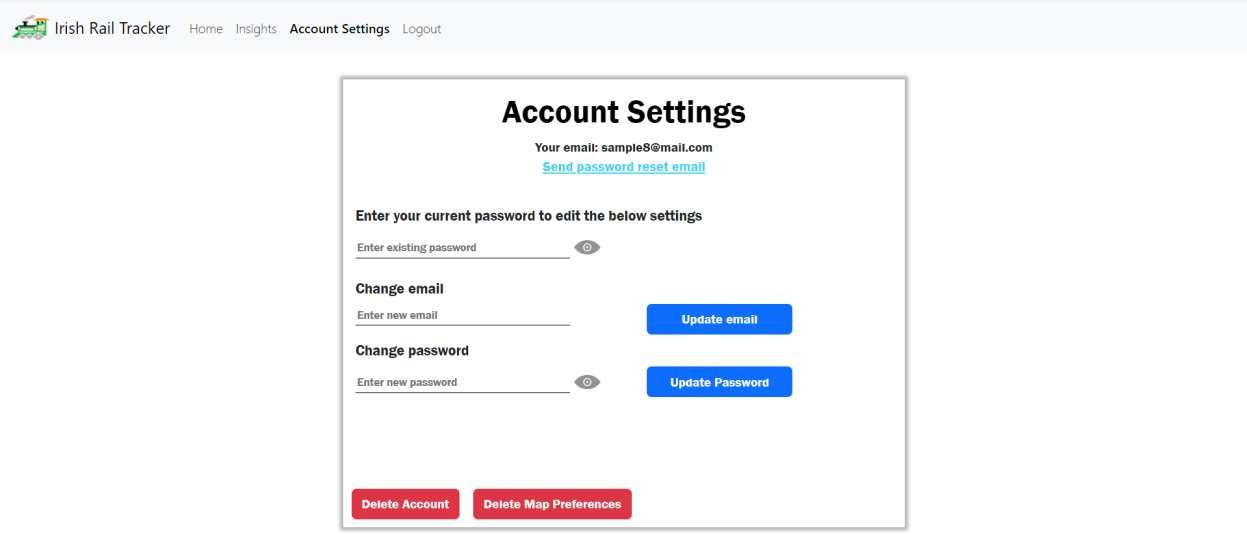

Figure 31: Account settings page

The above images shows the account settings page. A core feature of this page is that a user should not be able to instantly delete their account or change their password without some re-authentication. To solve this problem, we made use of the reauthenticateWithCredential() Firebase method, which requires a user object and a credential object. The credential object can be obtained asynchronously by calling EmailAuthProvider.credential() and passing the user's email and in our case the result of the current password input box. Other features that users have on this page is the ability to change their email or password, and to delete their map preferences.

4.2.9 Custom Error Handling with Toasts

The `mosha-vue-toastify` library was used to show user friendly error and success messages on the screen for some time interval. Using this library, we could specify the amount of time a message should be shown for, as well the text it displays and its background colour. Errors specific to Firebase authentication were caught, parsed and then displayed. In other situations, we just wrote our own toast messages. Sample toasts are shown below:

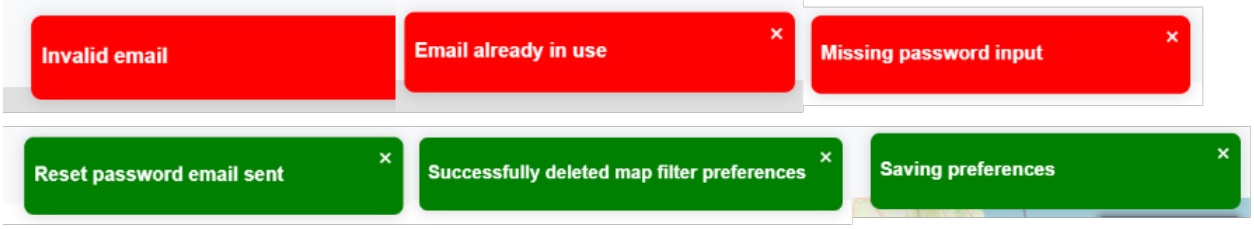

Figure 32: Sample Toast Error & Success Messages

4.3 Unit and Integration Testing

Mocha is a JavaScript-based test runner and Chai is a JavaScript-based assertion library. These were used for both Vue.js unit tests and Firebase function integration tests.

Unit tests are a testing method that checks individual units or components of code. In our case, we created unit tests for Vue.js components.

```
it('Not logged in test', () => {
    expect(wrapper.text()).to.include('Irish Rail Tracker');
    expect(wrapper.text()).to.include('Home');
    expect(wrapper.text()).to.include('Insights');
    expect(wrapper.text()).to.include('Login');
    expect(wrapper.text()).to.include('Sign Up');
}),

it('Logged in test', () => {
    // re-render the component
    wrapper.setData({isLoggedIn: true})
    nextTick(() => {
      expect(wrapper.text()).to.include('Irish Rail Tracker');
      expect(wrapper.text()).to.include('Home');
      expect(wrapper.text()).to.include('Insights');
      expect(wrapper.text()).to.include('Account Settings');
      expect(wrapper.text()).to.include('Logout');
    })
})
```

Shown above is sample of code taken from the Navbar.vue component unit test. The first test makes assertions when the client is logged out. Then, the second test makes assertions when the client is logged in. If these assertions fail, then the test fails. The `wrappper` object refers to a "mount" of the component for testing purposes. Unit tests were created for thr 404Page, LoginPage, Navbar, and SignUpPage components. As a future development, more unit tests would be created.

Integration tests are a testing method that checks multiple units acting together. In our case, we created integration tests for Firebase functions.

```
it('Test /getStationData', async() => {
    const result = await chai.request('https://us-central1-irishrailtracker.
    cloudfunctions.net') .get('/getStationData')
    expect(result.statusCode).to.equal(200);
    expect(result.body.data).to.be.an('Array');
    expect(result.body.data[0]).haveOwnProperty('StationDesc');
    expect(result.body.data[0]).haveOwnProperty('StationLatitude');
    expect(result.body.data[0]).haveOwnProperty('StationLongitude');
    expect(result.body.data[0]).haveOwnProperty('StationCode');
    expect(result.body.data[0]).haveOwnProperty('StationId');
    expect(result.body.data[0]).haveOwnProperty('StationType');
})
```

Shown above is sample code taken from the integration test that checks the result of making a GET request to the /getStationData cloud function. If the result does not pass all these assertions, then the test fails.

These are white box tests as they involve the test writer having a strong understanding of the underlying code to create them, whereby they write assertions that these tests must pass. Testing is important as it ensures the quality and robustness of our system whenever we make a change to it. We integrated these tests into our CI/CD deployment.

4.4 Continuous Integration and Deployment (CI/CD)

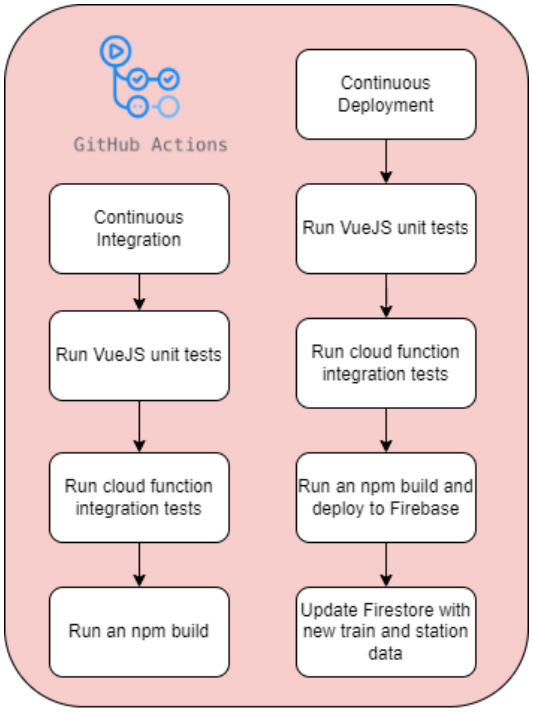

Figure 33: GitHub Actions Flowchart

GitHub Actions was used to create an automated CI/CD pipeline. Two YML scripts were created and placed into a `/workflows` directory, which GitHub ran depending on their configuration. The above image showcases the flow that the each of the CI and CD builds took respectively.

The first script, `integrate.yml`, handled continuous integration (CI), and ran whenever a pull request was made to a branch that pointed to `main`. The CI script succeeded if it successfully ran the Vue.js unit tests and Firebase integration tests, and built without error.

The second script, `deploy.yml`, handled continuous deployment (CD), and ran whenever a branch was pushed into `main`. The CD script succeeded if it successfully ran the VueJS unit tests, built without error, deployed to Firebase and ran Firebase integration tests, and updated the Firestore database with new train and station data (this final step was included as sometimes a code change might alter future database schemeas, and if so, the production database needed to also have these changes).

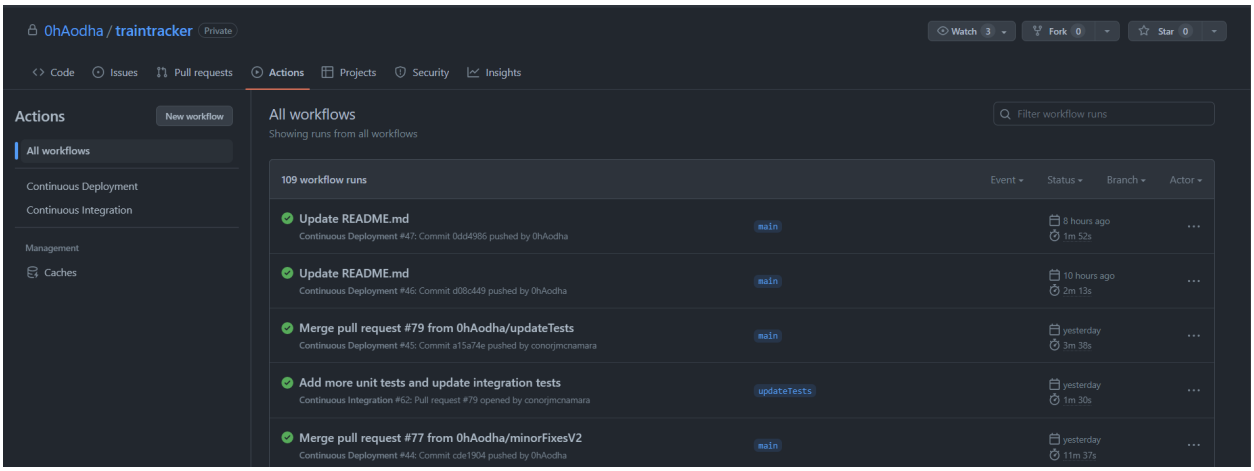

Figure 34: GitHub Actions Workflows

This CI/CD pipeline increased the team's efficiency by automating repetitive tasks handling Firebase deployment and accurately testing our code to catch errors before they became a production issue. The image above showcases sample completed GitHub Actions workflows taken from GitHub. These workflows can be viewed in real-time as they execute that their results are stored in the `Actions` tab in the repository.

4.5 Version Control and Project Management

4.5.1 Git and Github

We made extensive use of Git & Github throughout the project. From the very beginning, we hosted our source code on a Github repository. We had strict rules for committing to the repository.

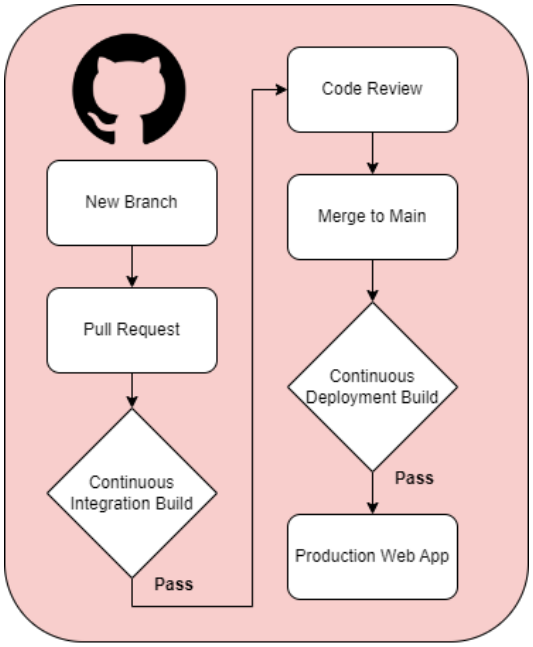

Figure 35: Release of a New Feature Flowchart

The image above illustrates the steps that were taken when adding a new feature to the source code. These steps include:

1. Create a new branch for the new feature, with a descriptive name.
2. Make the changes on this new branch.
3. Commit these changes with a descriptive commit message.
4. Push these changes to the branch.
5. Submit a pull request on Github. This pull request cannot be accepted and merged into main branch by the person who submitted it, instead it must be reviewed and manually tested by at least one other team member before it is accepted and merged into the the main branch.

4.5.2 Jira, Agile, and Scrum

From the very beginning of the project, we endeavoured to follow Agile and Scrum methodologies. We had a weekly stand-up every Thursday, in which we each discussed what we had been working on, what we were going to work on this week, and if there were any issues blocking us. This was extremely helpful for keeping the project on-track, ensuring that everyone got a fair share of work to do, and that everyone got the help that they needed on any issue that was blocking them.

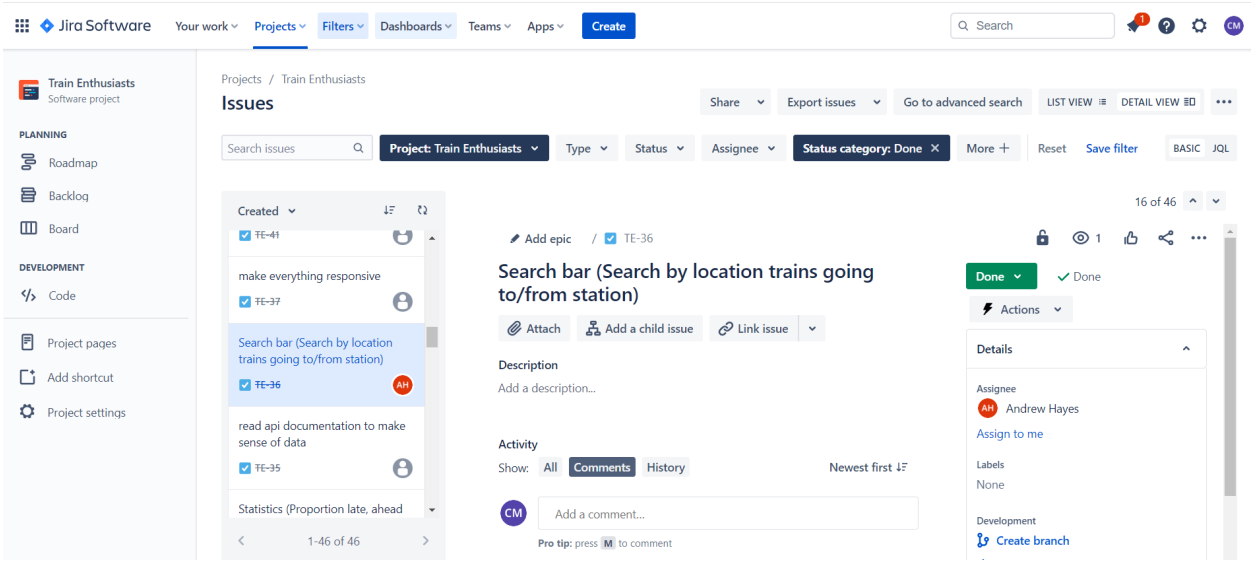

Figure 36: Completed Issue on Jira

As shown above, we used Atlassian's Jira to track any issues that were brought up at the stand-up. These issues would be added to a Kanban board, first being put in the "To Do" section. Each issue would then be assigned to someone, usually by themselves. The person assigned to the issue would move it to the "In Progress" section once they had begun to make progress on the issue. Finally, once the issue had been completed, it would be moved to the "Done" section. We found that this was massively beneficial to us to ensure that no issue that was brought up was ever forgotten, and it helped us to stay accountable for our work. It was also always very satisfying to be able to move an issue to the "Done" section of the Kanban board. In total 46 issues were created and completed.

5 Key Challenges

- One of the key challenges that we faced in this project was our initial migration from the prototype written in pure HTML + JavaScript to VueJS. This proved to be a lot more challenging than expected, as it required that the way in which the original map was implemented to be entirely re-invented to make proper use of VueJS templates, scripts, and data. Initially, we just attempted to insert our existing HTML into a VueJS component, but this didn't work properly. After much trial and error, we finally got a working prototype in VueJS. It was almost immediately worth it, as the new, properly set-up version which utilised all the proper VueJS best practices such as "component-isolation" immediately offered a clearer way to manage the data on the Map Page.
- Another key challenge was managing the order of external web requests that needed to be created when loading the map home page for example. This problem had an extra layer of complexity as since we were using the Firebase Emulator, reading from the database only would fail locally, as the local Firestore instance would first need to request data from Irish Rail. As already mentioned, this is why we have both PubSub and onRequest functions for getting and posting Irish Rail data to/from our database. That being said, using the created() VueJS lifecycle method, the home page will populate the database with new Irish Rail data if the host URL is local-based. Regardless, it will then fetch data from the database, and afterwards fetches the user's preferences if the client's session is logged in.
- Another challenge was if a user had map filter preferences saved, we had to ensure only trains/stations specific to those filters were always shown, regardless if the user logged out and logged back in for example. At the start there was a bug where all trains/stations would show for a second and then

disappear after the user's preferences were fetched. This resulted in an awkward display for the user. To solve this.

- A challenge arose when it came to making the whole website responsive. There are a lot of elements on each page of the website, and we tried to keep as many of them as possible. This became quite difficult as we had less space to work with on smaller devices. To achieve this we had to make extensive work of options such as the z-index of elements to allow them to appear over/under other elements. We had to work with resizing elements, and adjusting their positions based on percentages. As we added more items to the project, we slowly reduced the space we had to work with, this resulted in us having to change the layout of some pages entirely. Overall we believe we dealt with this challenge exceptionally well, and we are pleased with the final look of the project.

6 Future Developments

Although we achieved all of the goals that we set out to achieve at the start of the project, there are still future developments and enhancements that could be made to the project:

- One such future development would be to expand the scope of the project beyond just Irish Rail trains. There is no reason why we need only obtain data from the one API. We could perhaps obtain bus data from public APIs, or indeed expand our scope to other countries as well, plotting, say for example, British or French trains on the map as well.
- We could also expand the search feature for more advanced users. At present, the focus of the search bar is on ease of use, so you can only search for keywords in the data of each train. However, a more advanced user might be able to make use of a search box that allows the user to enter their own regular expression, and only displays the trains that match that regular expression.
- At present, we are using publicly available map tiles provided by OpenLayers. However if this project were to grow to a commercial size, we would no longer be permitted to make use of OpenLayers' Content Delivery Network (CDN), and we would have to host the map tiles ourselves, although we could still use OpenLayers tiles.
- Another feature that could be useful to the users of our website would be to make the train timetables available on our website.
- We could implement the search and filtering features on the Insights page.
- More unit tests could be added to the project to achieve a higher code coverage. Likewise, other forms of testing such as frontend end-to-end tests could be made to simulate user actions in a test environment.

7 Individual Contributions

7.1 Andrew Hayes

- "Scrum Master" of the project, coming up with the initial idea for the project, selecting the Irish Rail API, selecting OpenLayers as the map tiles source, selecting the map projection to be used, overseeing the weekly stand-ups, and creating & managing the "Train Enthusiasts" Jira Board.
- Made the first working prototype which plotted circles on an OpenLayers map, originally without the use of any framework, just HTML & JavaScript, and using a JavaScript function written by Conor to fetch the data to the client.
- Set up the Firebase hosting, including obtaining a unique project name to facilitate a memorable URL.
- Migrated the existing code to use VueJS and a VueJS library called `vue3-openlayers` for the map.
- Implemented the feature that would make the train icons red or green depending on whether or not they were late, including a function that would tell if any given train was late. Also added a feature

that would give a train a black icon if it was not yet running or terminated, to distinguish it from the running trains and implemented the icon colouring feature in the sidebar component.

- Implemented the feature that would use distinct icons for DARTs & normal trains.
- Created the green/red versions of the train & DART icons using photo editing software and free-to-use plain black icons sourced under a permissive Creative Commons license as the base.
- Implemented the handling of the `\n` escape sequences to improve readability.
- Implemented a filtering feature that allowed the user to filter which trains appeared on the map by whether the trains were Mainline or DART, On-Time, Late, or Early, Not-Yet Running, Running, or Terminated, including a meta-toggle that would toggle on or off each other toggle.
- Implemented a filtering feature that would allow the user to filter which stations appeared on the map by type, either Mainline or DART, including a meta-toggle that would toggle on or off each other toggle.
- Wrote a variety of utility methods that returned data on a given train or station, primarily used in the Boolean algebra expressions that were evaluated for each train to determine whether or not they should be displayed on the map, including `isTrainLate(i)` (which used logic written by Conor), `getTrainType(i)`, and `getStationType(i)`.
- Implemented a case-insensitive search feature that would allow the user to search for specific trains or stations on the map by keyword such as “Dublin“ or “roscommon *[sic]*”.
- General cleaning up, such as making the tab titlebar display "Irish Rail Tracker", fixing incorrect labelling of latitude as longitude, writing the `README.md`, making the map size responsive to viewport size, fixing the positioning of the train icon, changing the favicon, etc.
- General maintenance of the Git repository, manually testing & approving PRs, resolving merge conflicts in the PRs of others, etc.

7.2 Conor McNamara

- Created the onRequest, onCall and PubSub cloud functions, which included the parsing of data from the Irish Rail API, interacting with Firestore and error handling.
- Managed the storage and retrieval of data from the database from the front and backend.
- Setup the raw data pipelines needed on each page, including the map, insights charts and leaderboard, and train/station sidebars.
- Implemented user authentication on Firebase, the storage of user map preferences in Firestore, and the ability to login, logout, signup, change a password or email, request a password reset email, delete an account or map preferences.
- Setup and implemented the VueJS store state management, router, including the 404 catch all route.
- Setup the Firebase emulator for local development.
- Created the Mocha and Chai unit tests for VueJS components and integration tests for Firebase functions
- Created and wrote the YML scripts for the GitHub Actions CI/CD pipeline.
- Implemented the public message ticker and toast system for error and success messages, as well as custom error handling on every page.
- Got the first uniquely clickable map icons working, and implemented logic to determine how much trains were late by, as well as general parsing including origin and destination extraction amongst others.
- Contributed to the design of the insights and map pages and general responsiveness of the site.
- Created the report diagrams amongst other writings.

7.3 Jack Lennox

- Created the first basic layout for the page in HTML including login page, and basic map page without any features. Created early prototype for sidebar with different data for different buttons. Animated the sidebar prototype to slide in when called and slide out when closed.
- Managed the styling for the login, signup, account settings, and insights pages; keeping the aesthetic consistent for all these pages. Implemented and styled the navbar at the top of every page, including creating the logo and applying effects to active links. Fixed issues where the live ticker wouldn't stay stuck to the bottom of the page. Created the "legend" dropdown and a non-functioning version of the preferences dropdown menu including all the text and buttons.
- Consistently updated stylistically both the station and train sidebar component to where it is in the final project. Fixed layout issues caused by non-running or terminated trains having no punctuality data.
- Implemented small quality-of-life features such as icons increasing in size on hover, eye icons changing to represent hiding/showing the users password, and colouring items on the insights page to make information clearer.
- Implemented the graphs on the insight page. This includes passing in data, the general layout of them, and general styling such as the change on hover etc. Also worked on the layout of the leaderboard on the insights page.
- Overall made everything responsive, changing the site layout where needed for screen sizes smaller than 850px. Sometimes implementing complete design changes for smaller screen sizes as seen on the insights page.

7.4 Owen Guillot

- Implemented a zoom limit feature on the map, constraining the amount a user can zoom in or out based on what would be relevant to the user.
- Implemented some of the early styles on the website, namely on the Accounts Page.
- Implemented a border limit feature on the map also, which disallows the user from scrolling around the entire globe, this type of feature seemed to be missing from similar sites to ours. Fixed layout issues related to this with the map not wanting to be centered where we needed it.

8 Conclusions

We are happy to say that we achieved all of the goals that we had, including both goals that we had from the beginning of the project, and goals that we added in over time. We:

- Successfully created a live train tracker that presents only data that is as accurate as possible, with no conjecturing.
- Created an intuitive, easy to use, and reasonably visually pleasing user interface that can be understood and made use of by almost anyone, regardless of their technical ability.
- Created a map that has clear visual distinctions between normal trains and DARTs, late trains and early trains, running and not running trains, etc.
- Implemented a clear and precise data insights page on the website that gives data in a more objective format than the visual format of the map.
- Created a powerful filtering system for the trains shown on the map so that the user may choose exactly which categories they want to see.
- Implemented a robust user system that allows the user to save their map filter preferences, allowing them to re-use them whenever they visit the site, regardless of the device that they are on and without having to re-select them each time.

- Implemented a search feature so that the user may search for individual trains or stations on the map, only displaying those which contain the searched-for keyword.
- Managed and planned our project using Scrum and Agile methodologies, and made extensive use of software such as Atlassian's Jira.
- Made extensive use of web development frameworks such as VueJS and Bootstrap, and got a lot out of using Firebase hosting and cloud functions.
- Used Git source management, made extensive use of branching and pull requests, and never merged a pull request to the master branch without first getting a review from another teammate.
- Designed and implemented a good testing environment using the Firebase Emulator to ensure that our functions were working as intended.
- Implemented an effective CI/CD pipeline that automatically runs tests and deploys to Firebase if the code passes the unit and integration tests.