Github

.env

The env file is a plain text configuration that is used to store environment specific variables, like APIs. It keeps sensitive information out of the codebase.

In Index.js which is primarily the back end, this line loads the file into it

index.js

```
require('dotenv').config();
```

In our env file it is structured like this.
ACESr="https://api-endpoint.mta.info/Dataservice/mtagtfsfeeds/nyct%2Fgtfs-ace"

ACESr is the name of our variable, we initialize it as the link because it gets a name and we can call back to the link when we call the API.

.env

```
ACESr=https://api-endpoint.mta.info/Dataservice/mtagtfsfeeds/nyct%2Fgtfs-ace
BDFMSf=https://api-endpoint.mta.info/Dataservice/mtagtfsfeeds/nyct%2Fgtfs-bdfm
G=https://api-endpoint.mta.info/Dataservice/mtagtfsfeeds/nyct%2Fgtfs-g
JZ=https://api-endpoint.mta.info/Dataservice/mtagtfsfeeds/nyct%2Fgtfs-jz
NQRW=https://api-endpoint.mta.info/Dataservice/mtagtfsfeeds/nyct%2Fgtfs-nqrw
L=https://api-endpoint.mta.info/Dataservice/mtagtfsfeeds/nyct%2Fgtfs-l
1234567S=https://api-endpoint.mta.info/Dataservice/mtagtfsfeeds/nyct%2Fgtfs
SIR=https://api-endpoint.mta.info/Dataservice/mtagtfsfeeds/nyct%2Fgtfs-si
```

Index.js The Backend

First we want to install the dependencies and the libraries, for this project we are using

- Express
- Axios
- GtfsRealTimeBindings
- Dotenv
- Cors

What these do

Express:
Helps create the web server and define the routes easily, It helps our requests like the get/all train routes function.

```
const express = require('express');
```

Axios:
It is a “promise based http client for node.js and the browser. It retrieves the binary train data from the feed

```
const axios = require('axios');
```

Gtfs-realtime-bindings:
It decodes data to put into real time. It is used to get tripID, arrivalTime, routeID and others.

```
const GtfsRealtimeBindings = require('gtfs-realtime-bindings').transit_realtime;
```

Dotenv:
It lods are variables from the env file and processes them

```
require('dotenv').config();
```

Cors:
CORS= Cross-Origin Resource Sharing
It allows the front end to make requests to my backend.

```
const cors = require('cors');
app.use(cors());
```

Step 2:
Creating the Express App
Since Express create the web server we want to create a port for it to connect to.

```
const app = express();
const PORT = 3000;
```

Cost app = express();
Creates the express application and the port variable defines what port the server will run on locally. With the url http://localhost:3000 our server will run there when activated.

Step 3:
Front End portion

The Array of train objects each represents one train line, the url pulls the feed from the env file via the process.ev

```
const TRAINS = [
   { name: 'ACESr', url: process.env.ACESr },
   { name: 'BDFMSf', url: process.env.BDFMsf },
   { name: 'G', url: process.env.G },
   { name: 'JZ', url: process.env.JZ },
   { name: 'NQRW', url: process.env.NQRW },
   { name: 'L', url: process.env.L },
   { name: '1234567S', url: process.env['1234567S'] },
   { name: 'SIR', url: process.env.SIR }
];
```

We use the array to loop through the request of app.get

We start by defining the routes of where we go

```
app.get('/all-train-data', async (req, res) => {
   console.log('Received request for /all-train-data');
   const results = [];
```

This creates a GET route at /all-train-data.

Whenever the front end sends a request to this route, this function runs. It stores the data we retrieve into an array.

We loop through each trainline using

```
   const fetchPromises = TRAINS.map(async (line) => {
      console.log(` Fetching data from: ${line.url}`);
```

It loops through each line that is stored in the train array, for every entry it sends a request to the URL that is attached to it, this being the env urls. The .map() function builds an array of these “promises” to get the data. We use this because we want to ask for 8 requests all at once to store the data.

Once we have the data we decode it using our GTFS data.

```
        try {
            const response = await axios.get(line.url, { responseType: 'arraybuffer' });

            const feed = GtfsRealtimeBindings.FeedMessage.decode(response.data);
            console.log(`Successfully decoded data for ${line.name}`);
```

This section sends a get request to the API url, after it uses the gtfs realtime binding library to decode the binary into readable JSON notation. The gtfs readings aren't readable so this decodes them into something we can display and read to the user.

Cleaning up the data

```
results.push({
    line: line.name,
    data: feed.entity
        .map(entity => ({
            tripId: entity.tripUpdate?.trip?.tripId || 'N/A',
            routeId: entity.tripUpdate?.trip?.routeId || 'N/A',
            arrivalTime: entity.tripUpdate?.stopTimeUpdate?.[0]?.arrival?.time?.low
                ? new Date(entity.tripUpdate.stopTimeUpdate[0].arrival.time.low *
1000).toLocaleTimeString()
                : 'N/A',
            vehiclePosition: entity.vehicle?.position || {}
        }))
        .filter(entry => entry.tripId !== 'N/A' && entry.arrivalTime !== 'N/A')
});
```

This function loops through the info and extracts what we need to use for our site, We single out the TripId, the RouteID, the arrivalTime, and the current location of the train using longitude and latitude.

The filter area removes the entries that don't have a valid tripId or arrivalTime.

Errors:

```
        } catch (error) {
            console.error(`Error fetching data for ${line.name}: ${error.message}`);

            if (error.response) {
                console.error(" Status Code: ${error.response.status}`);
                console.error(`Error Data:`, error.response.data);
            } else if (error.request) {
                console.error(`No response received for ${line.name}`);
            } else {
                console.error(`Request Error:`, error.message);
            }
            results.push({ line: line.name, error: `Failed to fetch data for ${line.name}` });
        }
    });
```

If anything fails these errors will print to the console, We want to use errors to prevent broken feeds from occurring.

Waiting and responding:

```
    await Promise.all(fetchPromises);
```

Waits until all the train feeds are done, this line of code ensures that the next step will run perfectly

```
    console.log('Sending decoded data back to frontend:', results);
    res.json(results);
});
```

It logs the data, sends a JSON back to the frontend
Each item in the array will constrain
Valid train data or an error message.