Broad DSP Engineering Interview Take-Home Documentation

This document's purpose is to detail the reasoning behind the choices made during project setup and answering each of the completed take-home questions. It will also explain the thought process for how to possibly complete question three.

Project Setup Choices:

The project setup uses a relatively simple npm build that compiles ES6, bundles the JavaScript and runs tests. Due not having a great dev environment on my home PC (also new to dev on this operating system), I chose a path that would get me started easily and quickly. I choose vanilla JavaScript due to the ease of getting started and my familiarity with the language. I would have liked to get a better gulp or grunt build running or use Ember, but due to time constraints I went this route. I would have liked to use TypeScript and SCSS as well.

The results of this project are presented in the index.html page body and also logged to the web console via app/js/answers-output.js. Please see the README.md file in github for how to verify the answers, build the project and run tests.

All code can be viewed on Github.
Link to the MBTA API-V3 swagger doc: https://api-v3.mbta.com/docs/swagger/index.html

Question One:

The first step was to look over the api swagger docs and see what data was being returned. After looking over the doc it was decided that using the filtered api call made the most sense. It would require less code and likely be faster to use this api.

The following functions were written to fetch the data and manipulate it into a form that would answer the question.

fetchSubwayRoutes(): Returns a promise object with a “data” array of objects containing individual routes and their details.

It was decided that fetch api would be the best way to hit the mbta apis. Fetch was used in the function fetchSubwayRoutes() to return a promise of all subway routes from the /routes endpoint. It was later determined that async await would be cleaner than promises but it was not done due to time constraints.

Sample return value: The required attributes “id” and “long_name” are highlighted below.


```
{
    "data": [
       {
           "attributes": {
            "color": "DA291C",
            "description": "Rapid Transit",
            "direction_destinations": [
                "Ashmont/Braintree",
                "Alewife"
            ],
            "direction_names": [
                "South",
                "North"
            ],
            "fare_class": "Rapid Transit",
            "long_name": "Red Line",
            "short_name": "",
            "sort_order": 10010,
            "text_color": "FFFFFF",
            "type": 1
        },
        "id": "Red",
        "links": {
            "self": "/routes/Red"
        },
        "relationships": {
            "line": {
                "data": {
                "id": "line-Red",
                "type": "line"
            }
        }
      },
      "type": "route"
    },
  ]
}
```

getRoutesDetails(routesByType) Takes in the promise response from fetchSubwayRoutes and returns a map of route longNames and ids.

Initially the fetchSubwayRoutes() response was used as is but while working on question two, it became apparent that a little manipulation of the data would make it more useful. This function was written to map over the response and return an array of objects containing route long_name and id. The id is used in question two for the /stops api call.

Once the data was in the necessary form, tests were written to verify the result and then the answer was added to the html and logged to the console.

Sample return value:

```
[
    {
        'longName': 'Foo',
        'id': 'foo'
    },
    {
        'longName': 'Bar',
        'id': 'bar'
    },
    {
        'longName': 'FooBar',
        'id': 'foobar'
    }
];
```

Question Two:

(TODO: Break this up in parts one, two, three for better organization)

Question two uses the response from question one to fetch all the stops for the subway routes. Initially it was thought that the /stops endpoint could filter for the “subway” stops like the /routes endpoint. After some research it was not obvious that this was possible so it was decided to modify the api call from question one (in getRoutesDetails) to be more useful here.

The following functions were written to fetch the data and manipulate it into a form that would answer the question.

fetchStopsByRoute(routes) Takes in a list of subway route details and returns a promise map of all stops per route.

This function maps over the route data returned in getRoutesDetails() and fetches the stops for each subway route using the endpoint /stops with the query param “filter[route]=${route.id}&include=route”. The include ensures the relationship route is added to each stop (needed in part three of this question).

Sample return: “route.id” (highlighted below)


```
[{
  "data": [
    {
      "attributes": {
        "address": "Alewife Brook Pkwy and Cambridge Park Dr, Cambridge, MA
02140",
        "at_street": null,
        "description": null,
        "latitude": 42.395428,
        "location_type": 1,
        "longitude": -71.142483,
        "municipality": "Cambridge",
        "name": "Alewife",
        "on_street": null,
        "platform_code": null,
        "platform_name": null,
        "vehicle_type": null,
        "wheelchair_boarding": 1
      },
      "id": "place-alfcl",
      "links": {
        "self": "/stops/place-alfcl"
      },
      "relationships": {
        "facilities": {
          "links": {
            "related": "/facilities/?filter[stop]=place-alfcl"
          }
        },
        "parent_station": {
          "data": null
        },
        "route": {
```

```
        "data": {
          "id": "Red",
          "type": "route"
        }
      },
      "zone": {
        "data": null
      }
    },
    "type": "stop"
  }
],
"included": [
  {
    "attributes": {
      "color": "DA291C",
      "description": "Rapid Transit",
      "direction_destinations": [
        "Ashmont/Braintree",
        "Alewife"
      ],
      "direction_names": [
        "South",
        "North"
      ],
      "fare_class": "Rapid Transit",
      "long_name": "Red Line",
      "short_name": "",
      "sort_order": 10010,
      "text_color": "FFFFFF",
      "type": 1
    },
    "id": "Red",
    "links": {
      "self": "/routes/Red"
    },
    "relationships": {
      "line": {
        "data": {
```

```
            "id": "line-Red",
            "type": "line"
          }
        }
      },
      "type": "route"
    }
  ],
  "jsonapi": {
    "version": "1.0"
  }
}...]
```

fetchResponse(url): Takes in a url and fetches or catches based on the response

Once fetchStopsByRoute() was written it became apparent that the fetch code could be reused so this function was written. This takes a url as a param and performs a fetch/catch. The catch is hit when the fetch response is anything other than a 200-299 response and if this happens a console.warn is logged containing the error.

getStopsDetails(routes) Takes in a list of routes and their stops and returns an object containing the answer details for part two and three of question two.

Each route is looped over and if the length of stops is greater than or less than the initially set value in the "details" object, the object is updated with the new most or least stops and the name of the route.

During each loop of the routes array, a new array of all stops is built up and used while determining what stops connect routes (for part three) in getStopsThatConnectRoutes().

Initial details object:

```
const details = {
    mostStopsCount: 0,
    mostStopsName: '',
    leastStopsCount: 0,
    leastStopsName: ''
};
```

Sample return value:

```
{
    leastStopsCount: 2,
```

```
    leastStopsName: 'RouteTwo',
    mostStopsCount: 3,
    mostStopsName: 'RouteOne',
    stopsThatConnectRoutes: {
        bar: {
            stopName: 'Bar',
            routes: ['RouteTwo', 'RouteOne']
        }
    }
};
```

getStopsThatConnectRoutes(allStops): Takes in a map of all routes and their stops and returns an object containing only stops that connect routes and the associated route names

This function is used in the getStopsDetails() and relies on the allStops param which contains arrays of stops per route. This two dimensional array is flattened so we have one long list of all stops. The thought process here is if a stop exists more than once then it must be on more than one route, thus connecting routes.

The way this works is the flattened array of stops is mapped over and the current stop is returned the first time a stop matches the current stop.id and is not the current stop. A filter is then run to remove the undefined returns from when a matching stop.id was not found. This gives us a list of only stops that have duplicates. This idea for this concept was found on stack overflow but of course I can't find the link to add it here.

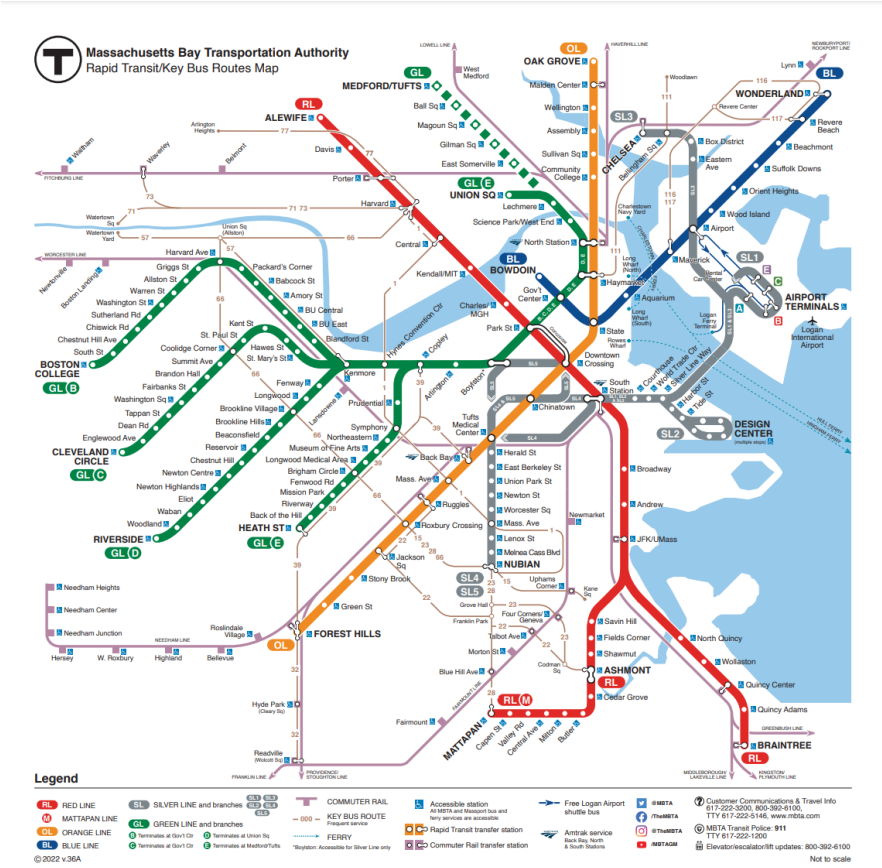
Now that we have all the duplicate stops we loop over them and build up a connectsMultipleRoutes object with unique stop objects containing the stop id as a key name and the relevant stop name and an array of the associated routes as properties. In the loop if the stop id is not already a property of connectsMultipleRoutes then we add a new property containing the relevant data. If the stop id property is already found we just push the route name to the connectsMultipleRoutes.routes array.

Sample return value:

```
{
    foo: {
        stopName: "Foo Stop",
        routes: ["Foo Route One", "Foo Route Two"]
    },
    bar: {
        stopName: "Stop Bar",
        routes: ["Bar Route One", "Bar Route Two"]
```

```
    }
}
```

This [MBTA map] was used to help ensure the connected stops were correct:



Once all data is built up in the details object of getStopsDetails(), it is logged to the console and presented on index.html.

Question Three Thoughts:

- Initially use the project examples start and end stops and loop over the function to find those routes.
    - 1. Davis to Kendall/MIT -> Red Line
    - 2. Ashmont to Arlington -> Red Line, Green Line B
- Determine if the stops are on the same route
    - We can use fetchStopsByRoute return value for this
- If not determine if the routes share a connected stop
    - We can check the connected stops list we built up in question two to verify
- If neither of these are true, log an error saying “You can't get there from here”

- If the stops are on the same route, return the stops and all between using array slice
- If they are on different routes then list the first stop and the stops between it and the connecting stop and then the connecting stop and all stops between it and stop two.

Possible issues:

- We do not know where the stops exist in the stops array. If we use slice we need to get the lowest index item first.
- Some routes branch and it's not clear how this manifests in the data.