Bus Bots for The London Transport Museum Technical Manual

Parker Coady
Samantha Crepeau
James Scherick
Vlad Stelea

May 11, 2020

Contents

Chapter 1

Running The Activity

1.1 General Setup

A mediator can start the activity on the big screen by going to the activity's website and clicking "Create Game". They will be taken to a screen where they can select difficulty and create a game. Currently the difficulty does not change anything, as we were unable to implement it due to time constraints.

Once a game is created, up to 4 players will be able to join. The first, third, and fourth players will be route planners, while the second player will be a traffic light programmer. Note: do not let a user join a game twice from the same computer; if they do, there will be problems if the player tries to refresh the page.

All players must ready up to begin a simulated day. The day ends when all passengers have reached their destination and time has run out. Happier passengers increase the score. The maximum score is 100. To edit bus routes and/or the traffic program, at least one player must unready. Currently, the same day will just restart unless someone unreadies.

Refer to section 3.2 for a list of all known bugs if you encounter any issues with the game.

1.2 Guide to User Interfaces

1.2.1 Bus Route Planner

The players should create a route that overlaps with other routes but has a lot of stops that are unique to themselves. If players are to get a good score, they must think about how long it takes to get from one stop to another and take into account what side of the road those bus stops are on. The route will repeat once a bus completes it.

1.2.2 Traffic Light Programmer

Players should be encouraged to play around with the example program if they have little experience programming. If the game crashed, it is most likely due to an error in the traffic light programmer's code, as we don't have error handling. The most efficient program will be one where the stop light changes based on what side of the road has the most cars.

1.3 Tips for Mediator

1. Always state the goal of the activity: To achieve the highest score, make all passengers as happy as possible by creating the most efficient bus system.

2. You can give more detailed instructions, especially for younger audiences. Here are some hints and ideas for helping players along:

   (a) Bus routes can overlap.

   (b) Work together on a collaborative route rather than alone making a very long route.

   (c) Think about how traffic lights work in real life. For example, a light might turn green if there is a car at that light and not at the opposite light.

   (d) Think out loud and help each other.

   (e) Tinkering: Play around with different variables and see what happens when you change them.

3. Allow children to find solutions themselves, but help them in the right direction if needed. Find a problem and help them think through a solution.

Chapter 2

Technical Overview

This chapter will introduce the Bus Bots activity's technical architecture. This includes what responsibilities each layer will have as well as how they are able to communicate.

Bus Bots has two primary layers that are seperated and must be able to talk to each other. In this document, we will refer to these layers as the client and the server.

2.1 The Client

The client is the layer that players directly interact with. It is responsible for taking player input and converting it into valid game actions. The client also displays the results of player action. There can be multiple active clients at the same time. For more information about the Client, see chapter 3.

2.2 The Server

The server is a centralized communication point for all clients. In order to preserve adequate game state, the server is also responsible for validating that the client has requested valid game actions. Because this activity runs on a website architecture, the server will also serve the client code to compatible browsers. For more information about the Server, see chapter 4.

2.3 Connecting the Layers

In order for the client and server to maintain the same state, it is critical that they are able to communicate. Because the client and server are positioned on the same network, this can be done through a combination of REST requests and websockets. REST requests are the prefered technology for most web applications, but because the Urban Map client screen required instantaneous notification of changes in state, using a REST approach would have resulted in continuous polling of the server.

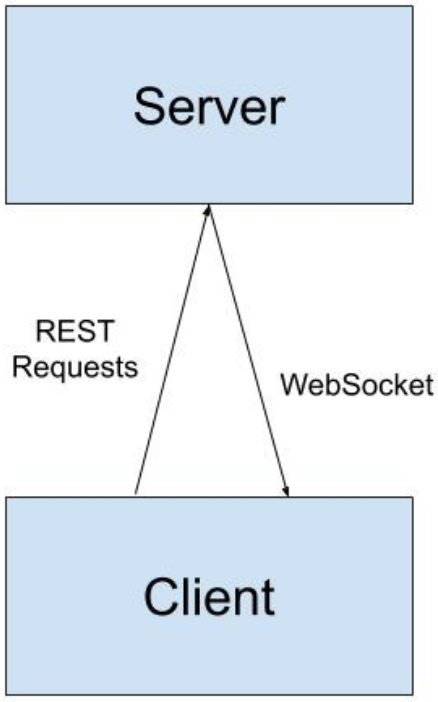

Figure 2.1: High level diagram of how the Client and Server communicate

We decided that both a quick poll and long poll option would waste resources for the creation of each connection. Because of this, we used a websocket

layer that connects the Urban Map screen to the server. A diagram showing the direction of network flow can be seen in Figure 2.1. The arrow pointing from the client to the server POST requests. As you can see, even though websockets support bi-directional communication, the client does not pass data to the server through websockets. This decision was made in order to minimize the number of concurrent active connections that the server would have to mantain.

Chapter 3

The Client

3.1 The Screens

3.1.1 CreateGameScreen.html / Index.html / JoinGameScreen.html

These screens all work fairly similarly. Index.html is the starting screen and just has buttons which link to CreateGameScreen.html and JoinGameScreen.html.

JoinGameScreen.html and CreateGameScreen.html both have methods that send get requests to the server and then load up a new page based on that information. Before they send you to a new page, they will store the gameID for the game you are playing (as well as the playerID for which player you are for JoinGameScreen.html) in the cookies. This way you can store the information for what gameID and playerID you are a part of in between screens.

3.1.2 RouteUI.html

The RouteUI screen will onload retrieve the gameID and the playerID from cookies and store them as variables.

This screen mostly consists of an HTML canvas element that displays the current route and the buttons to the right of it that can be used to add bus

stops to the route. The canvas has several functions that handle how a mouse interacts with it. Within those functionalities, a player can drag a bus stop around the screen, or delete them by clicking on them.

The submit button sends a get request to the server telling it to toggle the ready state of the current player as well as send the route to the server. The order of which it does this is based on if it is currently ready or not.

This page will also send a request to the server letting it know what the current route is every 5 seconds.

3.1.3 StreetLightProof.html

This is the screen that allows users to program streetlights. This is done by creating custom blocks for the JavaScript library Blockly. Please read Blockly's documentation for a better understanding on how to make changes to this part. Unlike most Blockly code, this does not actually export code, but exports a string that represents a JSON array of blocks. This way we can run a block one at a time to update traffic lights and dont need to rely on multithreading. Once a player is ready, all this information is sent to the backend.

3.1.4 BigScreen.html

BigScreen is the screen that handles the map. This is supplemented by createNodes.js and Vehicles.js. This main file mostly has to do with creating gamestate variables, constants, and general functions. Instead of going over every variable and function, I am going to go over how this screen works in general.

This screen gets information from the backend by connecting to it with a websocket.

Cars and busses are objects that move around a canvas. They know where they are going because of a path object they have in their variables.

Car paths have an ending whereas bus paths loop. Paths are made up of invisible nodes on the map. For the most part, nodes are just a position with a list of nodes they connect to. This way cars can drive to them and paths can be created using breadth first search. Nodes can also have a pointer to a bus stop and a pointer to a traffic light. If it has a pointer to a bus stop, busses will stop at them to pick up passengers that want to go on that bus. If there is a pointer to a traffic light, busses and cars can only move on to the next node when the light is green.

All of this gets drawn onto the canvas.

3.2 Known Bugs

Bug: Route colors can change when other people create or remove their routes during setup. *Cause:* The color of the route is based on when it appears in a list of routes. If a player has an empty route, their route does not appear in the list. This means that other players' positions can change when other players remove or start a route.

Bug: BusStops can change colors.
Probable cause: Due to how canvas context colors are stored and kept between functions.

Bug: Days restart immediately after the day ends.
Cause: Every five seconds, the route sends an update to the backend. If everyone is still readied up, the backend sends this information as well as the fact that everyone is ready to the BigScreen from the websocket when the update occurs.

Bug: Some traffic light programs crash the game.
Cause: Unknown; only happened once.We did not have the time to look into it. Probably due to the improper use of the Blockly code, but it is still a problem.

Bug: Busses don't spawn.
Cause: Unknown; only happened once during testing. Refreshing the page fixed it.

Chapter 4

The Server

The server is the backbone of the Bus Bots activity. This chapter will go into technical detail about how the different sublayers of the server function. In addition to the responsibilities mentioned in section 2.2, the server also handles a hidden data layer.

4.1 RESTful Architecture

The RESTful design pattern is an architectural style that utilizes stateless messages to communicate with a central server. It is built on top of http, meaning that each time that data is sent or received, a new connection must be made. The stateless nature of REST means that it is important for a client to send all the data that it wants the server to know in order for the server to properly serve a request. In the Bus Bots activity, this usually entails sending the game id and potentially the player id for each action. More information about the REST endpoints can be found within the swagger doc included with our code. To see more information about how to render this swagger doc see, section 5.2.

In order to develop a functional REST system, we utilized the Flask framework to define endpoints. The entrypoint for this system is contained within the app.py file in the private folder of the source code. An example of defining a REST endpoint can be seen in Figure 4.1.

```
1  from flask import Flask
2
3  @html.route('/new_route')
4  def new_route():
5      # Do Processing
6      # ...
7      # Return a dict that will be the body
8      return {
9          "item1" : value1,
10         "item2" : "value2"
11     }
```

Figure 4.1: Setting up a new endpoint

Often when utilizing a RESTful architecture, it is important to be able to send data within the request. Remember, the server does not automatically know which client is making a request; therefore, it is important for the client to identify itself. This can be done by sending data through the URL parameters or from the body of the request.

```
1  @html.route('/url_parameter/<id>')
2  def new_route(id):
3      # id is now a local variable
```

Figure 4.2: Getting the URL parameters

Of the two options, utilizing a URL parameter is more straightforward. As seen in Figure 4.2, there are two steps one must take before utilizing a URL parameter. Firstly, one must redefine the route to include the URL parameter in the *<parameter>* form. Then the parameter must be added to the function definition exactly as is defined in the URL. This parameter can then be used in code like a normal variable.

Sometimes, URL parameters do not provide enough control over your data. While you can define endless amounts of parameters(within the max URL

```
1 from flask import Flask, request
2
3 @html.route('/bodymethod')
4 def new_route():
5     # Get the request body in json form
6     payload = request.get_json()
7     # Can use payload as normal dict
```

Figure 4.3: Getting the body of a request

length limits), sometimes it is simpler to pass your data through the body of a request. An example of how this is done in Figure 4.3. As shown, this method requires another import from flask: namely, *request*. After you import this, you can use the *request.get_json()* call to get the body. This call will give you a standard python dictionary representing the JSON body in a pythonic format.

4.2 WebSockets

Unlike standard http, WebSockets utilize the same connection to send data between server and client. This provides many benefits, including not needing to open a new connection for each message, as well as allowing the server to send data to clients at its own volition. This document will not explain exactly how WebSockets work, but will explain how they were used for the Bus Bots project.

To set up WebSocket connections, the Bus Bots team utilized the flask_sockets library. This library provides a simple entry point for websocket connections on the server-side.

The *SocketProvider* class is responsible for maintaining WebSockets in an efficient and scalable way. By utilizing Redis's PubSub functionality, we were able to build a Reactive message sending system. While the WebSocket objects are still being stored in memory, we do not have to hang threads.

```
1  from flask_sockets import Sockets
2  import gevent
3
4  @ws.route('/setup_socket/<id>')
5  def setup_socket(socket, id):
6      socket_provider.register_socket(id, socket)
7      data = #data you want to send over socket
8      socket_provider.send_data(id, data)
9      # Keep this context alive
10     while not socket.closed:
11         gevent.sleep(0.1)
```

Figure 4.4: Setting up a WebSocket

An example of how to use *flask_sockets* and the *SocketProvider* can be seen in Figure 4.4. Notice how we were able to utilize the path parameters we talked about in section 4.1. Additionally, notice how we utilize gevent on line 11 to keep the context open. While this might seem wasteful, gevent is actually a lightweight coroutine library. This helps us keep open a connection without blocking all of our server's threads.

4.3 HTML Serving

For this project, we decided that static serving would be the best way to provide client web pages to users. This allowed us to focus more on the overall design of the web page rather than creating and filling templates. This also allots more resources to serving RESTful requests and sending data through WebSockets rather than rendering on the server.

To achieve static serving, we continued to utilize flask and pointed it towards a static server. It is important that flask still maintains the "/" path to serve the index file because it becomes the entrypoint for a user session.

4.4 The Hidden Data Layer

In addition to the two main layers of the Bus Bots project, there is a hidden Data Layer within the server. This is of course the Data Layer. Because our flask server does not maintain state between RESTful operations, it is important to maintain the game state somewhere.

We achieved this by adding an in memory key-value store called Redis to our project. We chose to utilize Redis because it is able to quickly store and retrieve data without us needing to generate complex SQL statements. This is why Redis is commonly used for caching data. This is the same offboard server that we utilized to make our WebSockets operate efficiently.

4.4.1 Data Types

Redis has seven basic data types. For the purpose of the Bus Bots system we utilized three of them: the **string, hash, and list**.

The string is the most basic of the seven types. Strings are useful for encoding a lot of data within one key. In fact, our first iteration of a data model involved simply storing a game's JSON model as a string. We quickly realized some of the drawbacks of this data model, most notably inefficient concurrent modification of a game.

Because of this, we moved to hashes. A hash can be viewed as a condensed map of key value pairs. This makes this data type perfect for representing objects within a redis database. However, this change still prompted the issue of how we would represent nested objects. Our solution was to create a new hash for the nested object and store the new hash's key within the first hash. These stored hash keys can be seen as pointers to other nested hashes.

The last data type from redis that we utilize is the list. Lists behave similarly to their counterparts in most programming languages. Namely, you can push and remove information on them at will (provided that this information is in string form). It is important to note that there is no such thing as an empty list in redis. This means that if you pop off all items in a list it, will

be removed. We solved this issue by keeping a head object at the beginning of lists that might eventually become empty so that we know when it is no longer safe to remove items.

4.4.2 Data Models

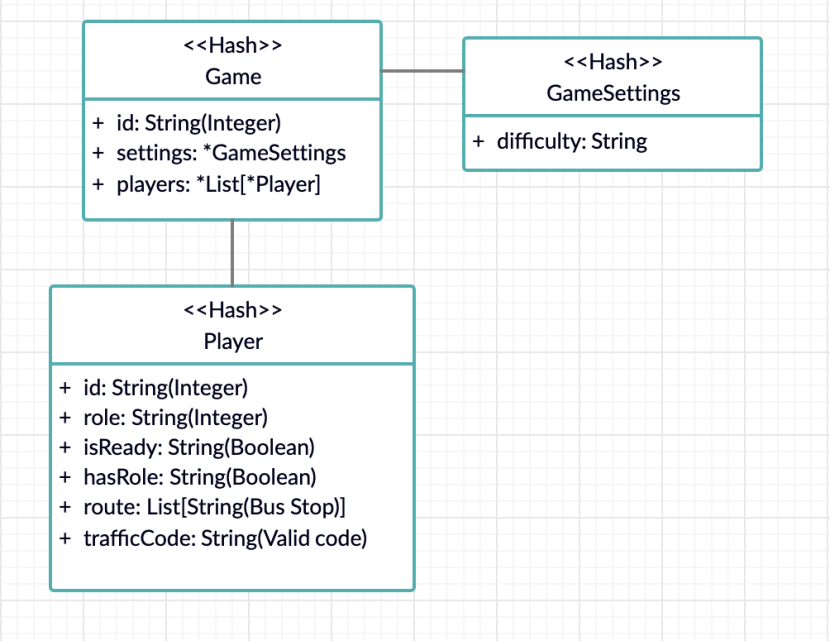

Figure 4.5: UML diagram of the Bus Bots redis data model

This subsection will document how different models are stored within our redis instance. These models use a mixture of Strings, Hashes and Lists to model nested objects.

The diagram shown in Figure 4.5 utilizes pointer notation (*) to show when a field represents an id of a hash or string. When you see the notation String(*data type*), it means that that string encodes that respective type. For example, a String(Boolean) means that that field can take the values "True" or "False".

4.4.3 Interacting With Redis

It is recommended that you read and write data to redis from the provided *GameDAO* class. DAO is a shortening for *Data Access Object*. This class contains useful methods for modifying and reading game states. If the models change, it is recommended that you modify the DAO to reflect these changes. Furthermore, if redis no longer suits your needs and you require a full database, you should view the *GameDAO* class as an interface in order to not break functions that call it.

Chapter 5

The Development Environment

Before editing the Bus Bots project, it is important to be able to set up a dev environment. This chapter will document what we use each tool for at a high level. It will also document how to start a lightweight development environment, and how to view our API in a clean format.

5.1 Docker

Docker is a program that packages code and all of its dependencies in "containers" based on a Dockerfile. From a simple point of view, containers can be viewed as lightweight virtual machines built for a specific purpose. We chose Docker containers because of their ability to be deployed to a variety of cloud computing platforms including **Amazon Web Services(AWS)**, and **Google Cloud Platform(GCP)**. Additionally, Docker has a multitude of pre-built containers stored in its container registry, DockerHub.

We have a few Dockerfiles within our project structure. For development, the most important one is contained within the private folder. This Dockerfile is responsible for setting up the container used to serve requests to the server.

It is important to note that by itself, this Dockerfile will not run, because our backend relies on a connection to a redis instance. To solve this, in the main project directory we have defined a file called *docker-compose.yml*. Docker-compose files are used to connect multiple containers within a local Docker network. As such, this file is useful for defining environments that require

communication between different containers. Linking between containers can be done with the links command.

5.2 Swagger

Swagger provides a simple way to define rich RESTful APIs through the use of yaml files. For the Bus Bots project, this yaml file can be found in the main directory under the name *endpoint-map.yaml*. In addition to yaml files being easy to read in their raw form, they can be rendered to show API documentation even easier as shown in figure Figure 5.1

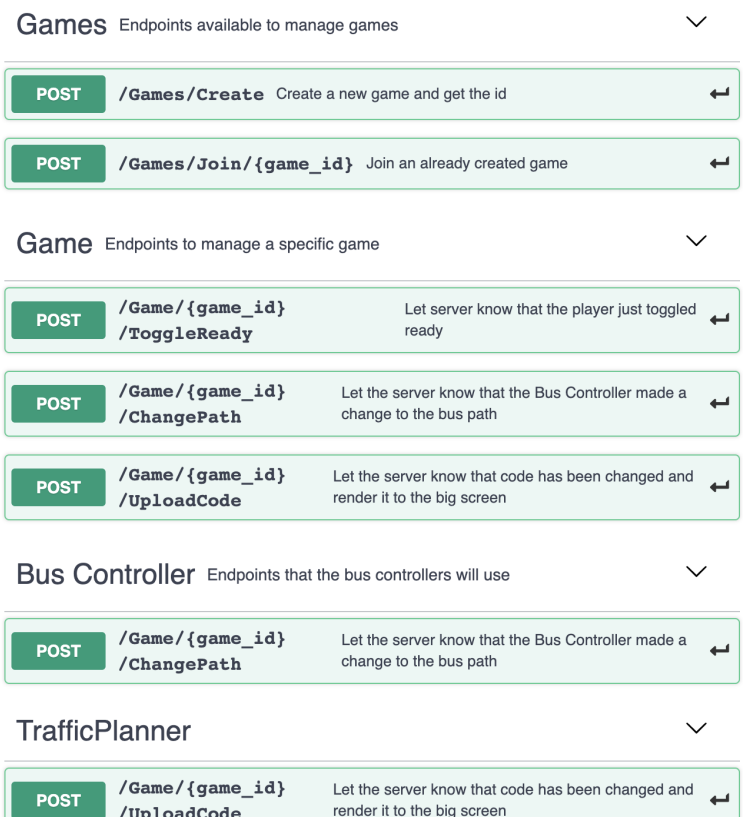

Figure 5.1: A fully rendered swagger doc

To render a swagger doc, one developer should make an account at `https://swagger.io/tools/swaggerhub`. They will then be able to upload the *endpoint-map.yaml* to swaggerhub and share the project with their team members.

5.3 Deploying

In order for Bus Bots to be accessible anywhere, we needed to deploy it to a cloud server. While we would have been able to utilize AWS or GCP for this task, we decided to use Heroku because of how easy it is to transfer a project.

In order to deploy to Heroku you will need to install the **Heroku CLI**. This is a command line interface that allows users to perform actions on their Heroku account and projects directly from a local shell. Follow the instructions on Heroku's website to install this and login to your account. After you complete this, you can run the enclosed *deploy.sh* script.

It is also important to note that our deployment relies on a redis environment variable. As of the writing of this document, it is set to *redis://localhost* because redis is packaged in the deployed container. While this might not be best practice (and we recommend splitting the two as soon as possible), we opted for a simple deploy without external dependencies.