BusGo: Implementing an Optimal System of Transit with D.C. Buses

Grace Jagga

Abstract

This project creates an android application, BusGo, that provides route trip planning using city transportation in D.C. The application is built on Android Studio and uses OpenTripPlanner (OTP), an open source software that provides transportation network analysis and route generation. The project uses Volley in Android Studio to send HTTP Requests to an OTP server hosted on the system and receives output from the server. The application presents the most efficient routes using public transportation data from the OTP server into an efficient interface.

1 Introduction

Public transportation is an important resource for many commuters, tourists, and regular citizens every day. Public buses, specifically, are provided in locations to allow transport within cities or interconnected locales. In regulated areas, each bus follows a specific schedule throughout the day in a repeated and confirmed cycle. The same system exists in universities where campus buses navigate around the university and neighboring city in a cycle. While these buses provide useful transport from students to access all parts of the campus, there is an inefficiency of the design that makes it hard for students to figure out the fastest and most efficient way to get somewhere based on their position. While public bus transportation provides effective transit, the apparent conflict lies in manually determining exactly when, from where, and which set of subsequent buses to take to get to a desired location at a desired time in the most efficient way. This is because buses are always in motion of their regulated route and current position and current time of the user is constantly updated, making it hard to calculate the fastest route. Also, different parameters of varying priorities exist specific to each user. For example, rather than just time efficiency as a priority, one user may require wheelchair accessibility options or another may consider a minimum walking distance as priority.

To target the ambiguity associated with finding the best route, the mobile application BusGo provides efficient and accessible route trip planning in D.C. based on the provided public bus transportation. BusGo takes a user's starting and desired ending location and calculates using graph-based algorithms the most efficient route of subsequent buses to take. The application is based on current time and location so it uses multiple nodes of different weighting to determine calculations. Furthermore, BusGo provides personalization of priorities where users can manually input important factors affecting the route such as walking distance, time between bus

stops, total distance, and ride accommodations. In doing so, BusGo allows a simplistic approach to route trip planning in D.C. using already existing forms of public transportation. While other transportation planning applications exist such as Transit, BusGo serves a specific locale of users within D.C. and only uses bus services from the D.C. Circulator transportation agency. As D.C. is a location for many government jobs and home to tourists throughout the year, many people in the city rely on public forms of public transportation alone. Therefore, BusGo effectively targets a prominent issue in manually regulating trip planning using public transportation in the form of an easily accessible mobile application.

2 Background

2.1 Android Studio

Android Studio is used to host the mobile application for this project. Android Studio is an IDE for app development that uses Java and XML files in order to build an application using a gradle-based build system. Gradle is a build toolkit that allows for the automation of the build process based on set build configurations. Applications are made up of a UI (user interface) layer and data layer. The UI layer displays the visual elements and reflects changes in appearance based on the user interaction. The data layer represents the data and values of information hosted on the application.

Android applications use activities to navigate between different sessions of use within the application. The Activity class handles the flow of activities launched. Activity instances within the application transition through states throughout its lifecycle. Activities cycle through its stages through call backs and can cycle with other activities cohesively to perform different actions simultaneously. The MainActivity.java specifies the creation of the first activity, and other activities can be built on it.

The project will employ fragments within the app to navigate through different pages of the app. Fragments contain portions of the project's setup and manage their own lifecycle and input events. They are hosted by another activity or fragment. The FragmentManager class manages the cycle of fragments throughout an application. References to the FragmentManager enable the host activity to manipulate through child fragments being displayed. The FragmentTransaction class provides the APIs necessary to commit transactions between different fragments. Transactions include adding, removing, and replacing fragments and are processed through the FragmentManager in one single commit called the FragmentTransaction. Committing a transaction asynchronously schedules the transaction once the UI thread is able to do so, creating a set order of transaction events for child fragments.

App resources contain additional content including layout constructions, user interface values, bitmaps, and other structures. This project uses XML files to define UI resources like drawable and layout resources to provide the structure and appearance of the app. Widgets are user interface elements that are used to add personalization and smaller views within an xml layout. A SearchView widget allows users to input search queries and provides an effective interface for searching within an app. An EditText widget serves as an input field that allows users to type text and store the value in the widget.

2.2 Web Servers

Hypertext Transfer Protocol (HTTP) is a network protocol that allows the transfer of documents on the web between a browser and server. A Web Server is an HTTP server that controls how users access files. When a browser needs a file hosted on the web server, the browser sends a request to the file through HTTP. The request reaches the web server and the

HTTP server accesses the file and sends it back to the browser through HTTP. Web server and browser communicate through such HTTP Requests and HTTP Responses. Clients make HTTP requests to servers, and servers respond with HTTP responses.

Web servers can be static or dynamic. Static web servers consist of a computer (hardware) and HTTP server (software) and sends files plainly to the browser. A dynamic web server consists of a static server and additional software, known as an application server. The application server allows files to be updated before being sent to the browser. When servers receive requests, it first checks if the requested criteria exists in a file and then sends the content statically. If the URL does not match an existing file, the server checks if a file can be generated and sent dynamically.

2.3 OpenTripPlanner (OTP)

OpenTripPlanner (OTP) is an open source software that provides passenger and transportation network analysis. It behaves as a backend application that transmits and receives data from a frontend interface. To be used locally, the software can be deployed on a web server over the internet and be capable of receiving requests and providing responses over HTTP. The OTP server downloaded locally acts as a dynamic server and can be accessible over the internet through request URLs that use the IP address of the system. The locally hosted server is opened on the internet through the domain http://localhost:8080, replaced with the IP address of the public server.

OTP uses OpenStreetMap (OSM) and General Transit Feed Specification (GTFS) data to calculate and provide multimodal trip planning by importing data from specified locales. OSM is a freely licensed geographic database that provides street map information including road

networks, buildings, transit, natural landmarks, and other features. GTFS is an open standard that contains transit information from public transit agencies provided in a data format widely accessible to users. The GTFS feed is composed of text files located in one ZIP file specific to transit agencies and a defined location. Each feed is composed of GTFS schedule and GTFS realtime. GTFS schedule text files contain data that remains constant within one transit network including pre-defined routes, schedules, fare, and further transit details. GTFS realtime text files contain data that is dynamically updated including schedule changes, vehicle positions, and current alerts. As an open standard, transportation agencies are able to publicly provide their route data and users can access it through the GTFS feed.

Together, the OTP server uses imported data from OSM and GTFS and builds a transportation network on a graph. The graph contains the visualization of predefined routes and possible created routes in order to employ route calculation algorithms to find optimal results. This allows a client to manually perform trip planning on the graph network by defining parameters to return a route for transportation.

2.3 Volley

Volley is an HTTP library that provides networking specifically for Android apps. It controls the processing and caching of network requests. It serves as a networking class for Android as the Android SDK does not include a specific class for networking requests. Volley performs requests to access data from APIs, internet servers, or web services back to the android application.

Volley can be imported in Android Studio and added to permissions in the Android project. In the dependencies in the build.gradle of the project, the implementation line must be added as follows:

```
dependencies{ implementation 'com.android.volley:volley:1.0.0'}
```

To give access for internet permission, in the AndroidManifest.xml file the uses-permission must be added as follows:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Volley performs the automatic scheduling of network requests and can handle multiple concurrent network connections at once. The two main classes in the Volley library include Request Queue and Request. The RequestQueue class dispatches multiple network requests and behaves as a container for all further requests. The RequestQueue class also handles request prioritization to optimize network requests. The Request class represents a single network in Volley and contains information for the network call like GET or POST method. The Request class contains subclasses such as StringRequest, JsonObjectRequest, and ImageRequest based on the type of data retrieved. A Request object represents a specific network call and each object is added to the RequestQueue to begin the network requests.

Using Volley, a JsonObjectRequest object is created with the HTTP GET method. The object creates a Response Listener parameter that takes in a JSONObject response. Finally, the JSONObject response is received through the Volley transaction as seen in Figure 1.

```
JsonObjectRequest jsonObjectRequest1 = new JsonObjectRequest
        (Request.Method.GET, toPlace, jsonRequest: null, new Response.Listener<JSONObject>() {
            @Override
            public void onResponse(JSONObject response) {
                try {
```

Figure 1: JsonObjectRequest through Volley transaction

REST (Representational State Transfer) is a method of networking in applications and APIs that provides stateless communication of data between client and server. The server does not store any client side data within requests which improves scalability of the application transactions. In Volley, RESTful APIs are used with HTTP methods such as GET and POST when handling requests. This allows for efficient communication and networking capabilities of the Volley requests.

3 Methodology

The mobile application, when successfully completed, should take in user input of starting and ending locations and provide different route options based on efficiency preferences. Then, the user may select their desired route accordingly and access thorough steps for the specific route option.

3.1 Creating the interface

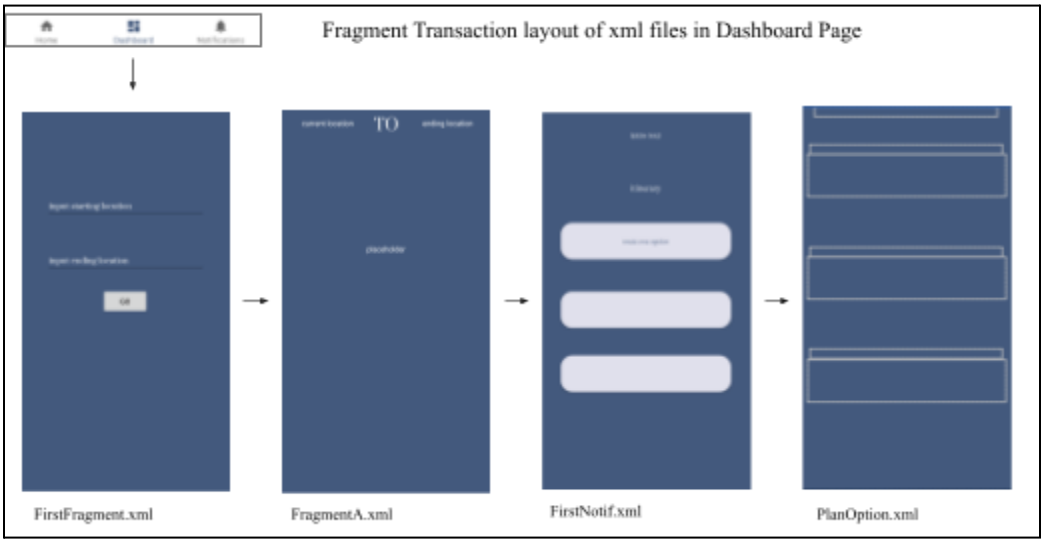

Figure 2: Visualization of Fragment transactions in Dashboard Page.

The application was built on Android Studio using a bottom navigation activity to allow multiple pages. The bottom navigation activity consists of the MainActivity.xml file and three fragment pages hosted under the parent MainActivity class in its onCreate method. Each Fragment page includes the Fragment class and fragment_name.xml file that regulates the visual appearance. The Fragment classes include the Destination Fragment and the Notifications Fragment. Inside the Destinations Fragment, the xml file contains a fragment container to hold a new child Fragment called FirstFragment.

In order to get user input of desired starting location and ending location, one solution consisted of using a SearchView element for users to manually search a location from a database of locales. However, this solution was not efficient because there was no database of locations of D.C. specific enough to include specific street names, positions on streets, and different parts of establishments. A second solution consisted of automatically accessing and using the current location of the user from their mobile phone. While this seemed to provide the most exact location in longitude and latitude, it was difficult to do so without accessing privacy data of the mobile phone and using permissions in the android application. The final solution was to use EditText elements for users to manually input a public location they are in to get the most specific location possible while also being in a format easy to access. To implement the EditText inputs, the prompts were placed in the first viewable fragment in the Destination page.

Within the FirstFragment class, the xml file contained two EditTexts for the user to input desired starting and ending location. Once responses are completed by the user, an onClick method of a button begins the Fragment transaction of FragmentFirst with a new child Fragment, FragmentA, that replaces the FragmentFirst in the container. The responses from FragmentFirst

were now sent to FragmentA (see Figure 2 for a visual representation of the fragment transactions).

3.2 Obtaining latitude and longitude values

The next step consisted of obtaining the longitude and latitude of the inputted locations. One option was to use the Google Geocoding API that takes a place as an address or latitude and longitude and converts it to its other form. While this served the purpose of converting places into coordinates, there seemed to be a simpler approach. Another option was to use a GPS coordinates converter website: https://www.gps-coordinates.net/gps-coordinates-converter.

This website also took an address and converted it to latitude and longitude. Once addresses are inputted, the longitude and latitude values need to be extracted from the website and sent to the application. To extract the values from the website, the page is inspected in order to find the url request sent in the network tab.

https://api.opencagedata.com/geocode/v1/json?q=lincoln%20memorial&key=03c48dae07364cabb7f121d8c1519492&no_annotations=1&language=en

The HTTP url when opened, presents the output values in a Json array where important values regarding route options are stored under specified indexes. For example, the longitude and latitude values can be found in the Json array output. To navigate to them, the values are within the “results” index, then “geometry”, then “lat" and “lng” respectively. To access these results from the mobile application, a Volley request is sent from the application with a URL that includes the street address inputs.

First, the EditText values containing the addresses were split into each individual word and then combined into a format readable by the website. Here, the words are combined with either a “%23”, “%2C%20”, or “%20” string depending on the format being used between

words. Then the value is put into a string format in startSTRING for the initial location and endSTRING for the final location (see Figure 3).

```
if(curr.substring(0,1).equals("#")){
    startSTRING += "%23";
    curr = curr.substring(1,curr.length()); //get rid of first element
}
if(curr.substring(curr.length() - 1).equals(",")) {//last is comma
    startSTRING += curr.substring(0,curr.length()-1); //all but comma
    startSTRING += "%2C%20";
}
else{
    startSTRING += curr;
    startSTRING += "%20";
}
```

Figure 3: Values extracted from url.

Then, the startSTRING and endSTRING are used to form a new URL of the GPS converter website in order to create a link that provides a json format of the new longitude and latitude as seen in Figure 4.

```
String fromPlace="https://api.opencagedata.com/geocode/v1/json?q="+startSTRING+
        "&key=03c48dae07364cabb7f121d8c1519492&no_annotations=1&language=en";
String toPlace="https://api.opencagedata.com/geocode/v1/json?q="+endSTRING+
        "&key=03c48dae07364cabb7f121d8c1519492&no_annotations=1&language=en";
```

Figure 4: Construction of new url with start and end strings.

Once the link is formed, a JsonObjectRequest is created using an HTTP GET method in Volley (see Figure 5). The JSONObject is retrieved into response, and new objects are created from the JSONArray to arrive at the latitude and longitude values stored in “lat” and “lng”. Now the longitude and latitude values of the address are stored and can be used in OTP calculations for route generation.

```
JsonObjectRequest jsonObjectRequest = new JsonObjectRequest
        (Request.Method.GET, fromPlace,  jsonRequest: null, new Response.Listener<JSONObject>() {
            @Override
            public void onResponse(JSONObject response) {
                try {
                    JSONArray val = response.getJSONArray( name: "results");
                    JSONObject v = val.getJSONObject( index: 0);
                    JSONObject vv = v.getJSONObject("geometry");
                    start[0] = vv.getString( name: "lat");
                    start[1] = vv.getString( name: "lng");
                    Log.d( tag: "tag",  msg: "sent request for FROMplace");
                    putPlace(view, start,  end: false);
                    Log.d( tag: "tag",  msg: "finished FROMplace request");
                } catch (JSONException e) {
                    Log.e( tag: "went wrong?", String.valueOf(e));
                    e.printStackTrace();
                }}} , new Response.ErrorListener() {
```

Figure 5: JsonObjectRequest to extract values from website url.

3.3 Using OpenTripPlanner to perform trip planning

OTP takes the input of starting location and ending location in lat and long formats and presents a variety of the most efficient routes to choose from based on preferences. In order to integrate the software to be used by the application, it must be downloaded onto the system. To do so, Java, Git, Maven, and an IDE must be installed onto the computer before OTP can be downloaded.

OTP is available to download on Github by accessing the shaded.jar file in the directory. Once the file is downloaded onto the computer, a GTFS feed must be downloaded as well to integrate into OTP. Selecting an agency from TransitFeeds, the D.C. Circulator agency can be downloaded by extracting it from the registry of feeds and into a newly created OTP directory on the computer.

```
$ cd /home/username
$ mkdir otp
$ cd otp
$ wget "http://developer.trimet.org/schedule/gtfs.zip" -O trimet.gtfs.zip
```

The OTP can now be started by running the java file in the terminal of the computer. Also run commands buildStreet and loadStreet to save the instance all at once instead of building each time.

```
$ java -Xmx2G -jar otp-2.4.0-shaded.jar --build --serve /home/gjagga/otp
java -Xmx2G -jar otp-2.4.0-shaded.jar --build --save /users/gracejagga/otp
java -Xmx2G -jar otp-2.4.0-shaded.jar --buildStreet /users/gracejagga/otp
java -Xmx2G -jar otp-2.4.0-shaded.jar --loadStreet --save /users/gracejagga/otp
```

The OTP server can now always be opened by running the following command:

```
java -Xmx2G -jar otp-2.4.0-shaded.jar --load /users/gracejagga/otp
```

The server is accessed on the internet through the url: http://localhost:8080/. To access the server on the system, a new url can be used specific to the computer. This is done by replacing localhost with the IP address of the system. The new url now looks like: 10.16.180.199:8080/. Figure 6 shows the OTP instance now successfully saved on the system and outputted in a new web server.

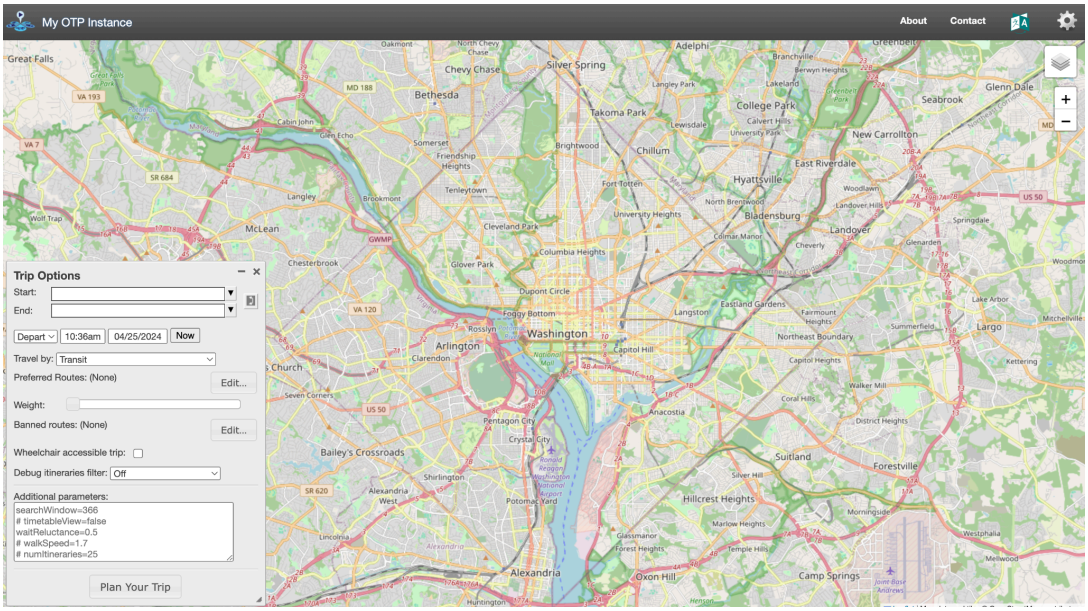

Figure 6: Output of OTP web server.

The server allows the user to manually input longitude and latitude values of starting and ending location and customizable factors regarding the trip. Once values are inputted, itinerary options

are returned in which the user can pick one to view the trip summary. OTP does the calculations itself manually, and presents it in an accessible format (see Figure 7).

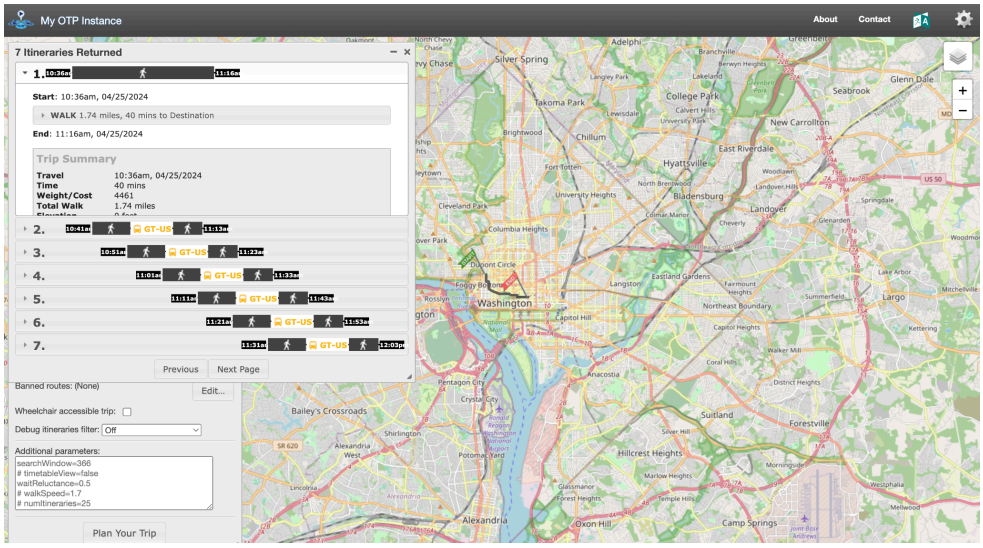

Figure 7: Route options presented in OTP server.

The new server is accessed on the link: http://10.16.189.68:8080/?module=planner&fromPlace=38.90679719352289%2C-77.0464324951172&toPlace=38.89744587262311%2C-77.0241165161133&time=10%3A36am&date=04-25-2024&mode=TRANSIT%2CWALK&arriveBy=false&wheelchair=false&showIntermediateStops=true&locale=en&baseLayer=OSM%20Standard%20Tiles

The itinerary summary can similarly be extracted from the page by inspecting and opening the network tab to see the new JSON url representing the output of the page (Figure 8).

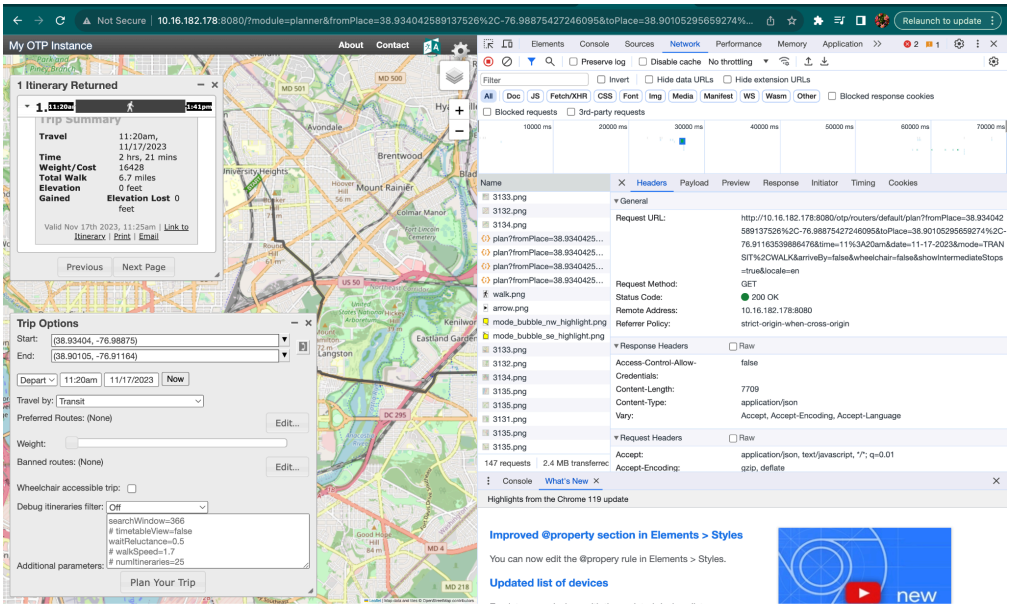

Figure 8: Obtaining JSONRequest url from web server.

The JSON Request url obtained is as follows:

http://10.16.189.68:8080/otp/routers/default/plan?fromPlace=38.9116059646841%2C-77.0467758178711&toPlace=38.900384991903856%2C-77.02068328857422&time=10%3A47am&date=04-25-2024&mode=TRANSIT%2CWALK&arriveBy=false&wheelchair=false&showIntermediateStops=true&locale=en

The url is opened to present a JSON output of values. Values are stored in indices based on the parameter of interest (see Figure 9).

Figure 9: Portion of Json output response from OTP server.

To access these results from the mobile application itself, the server must be started on the computer and then a Volley request can be sent from the application with a custom-made URL of the longitude and latitude values gathered.

The URL that will be deployed for the Volley request is constructed as seen in Figure 10 by values storing latitude and longitude of starting and ending location, as well as a value representing the current time and date.

```
String url="http://10.16.188.111:8080/otp/routers/default/plan?fromPlace="+ll[0]+"%2C"+ll[1]+
        "&toPlace="+ll[2]+"%2C"+ll[3]+timeDate+"mode=TRANSIT%2CWALK&arriveBy=false&w" +
        "heelchair=false&showIntermediateStops=true&locale=en\n";
```

Figure 10: Construction of url sent to OTP web server.

Now, a JSONRequest can be made using the new url in order to start the server and read the result of the json url (see Figure 11).

```
JsonObjectRequest jsonObjectRequest = new JsonObjectRequest
        (Request.Method.GET, url, jsonRequest: null, new Response.Listener<JSONObject>() {
            @Override
            public void onResponse(JSONObject response) {
                try {
                    JSONObject val = response.getJSONObject("plan");
                    JSONArray val2 = val.getJSONArray( name: "itineraries");
                    String from3lon = val.getJSONObject("from").getString( name: "lon");
                    String from3lat = val.getJSONObject("from").getString( name: "lat");
                    String to3lon = val.getJSONObject("to").getString( name: "lon");
                    String to3lat = val.getJSONObject("to").getString( name: "lat");
                    //save itinerary values in different JSONObjects
                    o[0]=val2.getJSONObject( index: 0);
                    o2[0]=val2.getJSONObject( index: 1);
                    o3[0]=val2.getJSONObject( index: 2);
```

Figure 11: JsonObjectRequest to read results of personalized url.

Each itinerary option is stored in a JSONObject as seen for the first three in FragmentA. Each itinerary is served in a new button in the xml file that has an onClick method (see Figure 12) for the user to choose the itinerary option after viewing the trip summaries of each. The onClick method of the button pressed sends the respective JSONObjects to a new Fragment, PlanOption.java through a Fragment transaction.

```
private void option1(View view, JSONObject[]b) {
    if(!optionPressed) {
        FragmentTransaction ft2 = getActivity().getSupportFragmentManager().beginTransaction();
        ft2.replace(R.id.fragment_container1, PlanOption.newInstance(b[0]), tag: "PlanOption");
        ft2.commit();
    }
    optionPressed=true;
}
```

Figure 12: onClick method of option.

In the PlanOption Fragment, the JSONObject that was received is iterated through to retrieve each individual step of the journey. Each step is checked with different parameters like mode of transport (walking or bus), time arrival and departure, and name of bus stop or location. Then,

the steps are combined into a new string format that contains all relevant information regarding the step. Within the step, note the mode of transport, agency name, board time, dismount time, and date (see Figure 13).

```
for(int i=0; i<times;i++){ //for each of the steps
    JSONObject step = r.getJSONObject(i);
    String mode = step.getString( name: "mode").toLowerCase(Locale.ROOT);
    if(mode.equals("bus")){
        //have to do differently than walking mode
        String agency = step.getString( name: "agencyName"); //should always be DC Circulator
        String busName = step.getString( name: "routeLongName");
        //cost and timings
        String board = step.getJSONObject("from").getString( name: "name"); //board at
        String dismount = step.getJSONObject("to").getString( name: "name"); //dismount on
        int intermediate = step.getJSONArray( name: "intermediateStops").length();
        int transitTime = (int)(step.getDouble( name: "duration")/60);
        Long arrival = step.getLong( name: "startTime");
        long dv = Long.valueOf(arrival);// its need to be in milisecond
        Date df = new Date(dv);
```

Figure 13: Values extracted from server's results.

Once values are stored in Strings, they are placed into TextViews in the PlanOption.xml file to display the results of the itinerary of one journey in a simplistic format (Figure 14).

```
arrayListThree.add("Board at " + board + " at " + ar + ".");
arrayListThree.add("There are "+intermediate+" intermediate stops.");
arrayListThree.add("Depart at \"+dismount+\" at \"+ de");
adapterThree = new ArrayAdapter<String>(getActivity(), android.R.layout.simple_list_item_1, arrayListThree);
thirdSteps.setAdapter(adapterThree);
```

Figure 14: Steps of itinerary stored in arrayLists.

The results of the input are displayed in the Fragment in the Dashboard page. Figure 15 shows the order of most efficient route options from which the user can select one. Figure 16 displays the new page showing the steps of the chosen route the user selected.

Figure 15: Route options displayed in results.

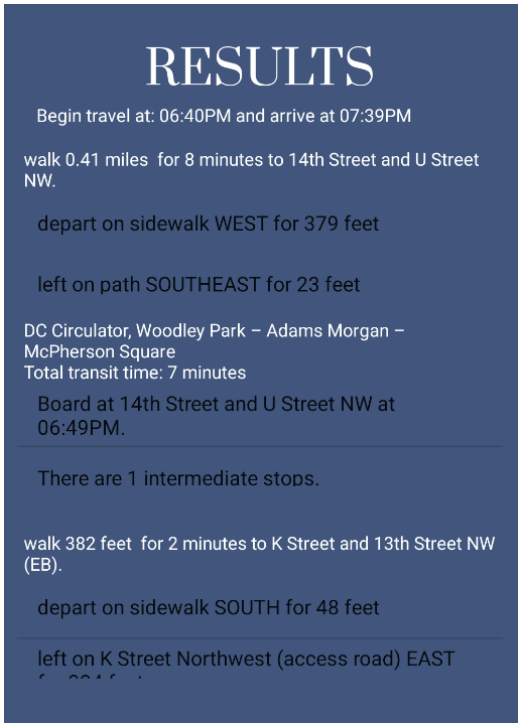

Figure 16: Steps of preferred route displayed.

4 Discussion

After the completion of the application, BusGo, there are still advancements that can be made. The application is not completely functional because it does not include the extent of the capabilities given by OpenTripPlanner in the actual app. For example, the OTP server allows for personalization options like wheelchair accessible trips, banned routes, setting departure and arrival times (different from current time), and other itinerary filters. Since BusGo only sends the longitude and latitude parameters to the OTP server through the custom made url, the other parameters are not applied to the trip options. However, this can be added on later by simply altering the url link to include true and false values for each further parameter.

In the current stage of the project, BusGo runs dependent on the OTP web server run on the system locally. This means the application cannot run independently of manually starting the server on the system's terminal and opening it in a new web browser. For the application to be accessible when fully published, it should later be hosted on an external hosting service in order to make it properly functional on an actual android app.

However, at the current stage, BusGo successfully takes an inputted starting and ending location and provides all efficient options of itineraries. This is what the original goal of the project was so the overall completion was successful. Although it isn't capable of accounting for every parameter and does not run independently of running the server locally, it performs the task successfully.

5 Conclusion

BusGo can be updated to reflect the further advancements that can be made to make it functional. For example, a next step could be to make it accessible independent of a server running on the side. It can also be updated to account for more personal accommodation in

transit. Another further development could be to use the current location through the mobile phone's storage of latitude and longitude. This would allow easier trip planning in the case users are unsure of where they are. In conclusion, BusGo successfully provides the most efficient route of buses to get from one location to another by using OpenTripPlanner on a web server linked to the application.

References

"GTFS: Making Public Transit Data Universally Accessible¶." *General Transit Feed Specifications RSS*, gtfs.org/. Accessed 7 May 2024.

MozDevNet. "What Is a Web Server? - Learn Web Development: MDN." *MDN Web Docs*, developer.mozilla.org/en-US/docs/Learn/Common_questions/Web_mechanics/What_is_a_web_server. Accessed 7 May 2024.

"Volley Library in Android." *GeeksforGeeks*, 27 Oct. 2022, www.geeksforgeeks.org/volley-library-in-android/.

"Volley Library in Android." *Top Website Designers, Developers, Freelancers for Your Next Project*, www.topcoder.com/thrive/articles/volley-library-in-android. Accessed 7 May 2024.

"Volley Overview." *Volley*, google.github.io/volley/#:~:text=Volley%20is%20an%20HTTP%20library,Automatic%20scheduling%20of%20network%20requests. Accessed 7 May 2024.