Lab 9 - REST API Querying

Android supports an extensive choice of data transfer and connection management options. Data can be transferred using Java Sockets, upon which we can make HTTP calls. REST APIs provided by remote Web services can be accessed using HTTP calls, and for that we use HTTP clients, such as HttpsURLConnection or OkHttp. However, converting data from these Web services (usually structured as JSON or XML) to Java objects that our Android app uses can be tedious if done manually. Libraries such as Retrofit and Volley make this process easier.

General idea - weather map

We will make an app for fetching and displaying current temperature data for different cities on a map. OpenWeatherMap is one of the free sources of such data, we will use API calls explained here https://openweathermap.org/current When a user moves the map, we will download and display a new set of temperatures corresponding to the cities currently shown on the map.

Creating a Map

Create a new Android Google Maps project with the package name `si.uni_lj.fri.pbd.lab9` and the minimum api set to 24.

Follow the instructions to set the Google Maps API key, as we did in one of the previous labs. Go to the **MapsActivity** code, and ensure that **MapsActivity** implements **GoogleMap.OnCameraIdleListener**. This will call `onCameraIdle` every time a user moves the map. However, we have to subscribe to these "move" events. In `onMapReady` make sure that the listener (i.e. `this` MapsActivity class instance) is connected to `mMap` via `setOnCameraIdleListener`, where `mMap` represents a GoogleMap.

By default the map is centered to Sidney, Australia. To fix the starting position of the map to somewhere around Ljubljana, put:
`mMap.moveCamera( CameraUpdateFactory.newLatLngZoom(new LatLng(46, 14.5), 8.0f) );` in `onMapReady`

Interacting with a remote REST API

First, register an account with OpenWeatherMap, log in, and then go to "API Keys" to get an API key you will use in your app. Create Constants class in your project and store the string key as `public static final String API_KEY`. Also, define a constant `String BASE_URL` and set it to "https://api.openweathermap.org/".

We will use "Cities within rectangle zone" (https://openweathermap.org/current#rectangle) to get a list of weather information at cities shown on the map that a user is currently viewing. When a user moves the map, we will fetch new data, relevant for the newly-shown part of the world. In this example you can see what to expect from such a call (examine the raw JSON output).

Setting up Retrofit and OkHttp client

To use Retrofit (for REST API interaction) and OkHttp (as an HTTP client) you need to add the following to your module's dependencies in the build.gradle file:

```
implementation 'com.google.code.gson:gson:2.8.6'
implementation 'com.squareup.retrofit2:retrofit:2.7.1'
implementation 'com.squareup.retrofit2:converter-gson:2.6.2'
implementation 'com.squareup.okhttp3:logging-interceptor:4.4.0'
```

Furthermore, Java 1.8 support is needed for OkHttp, so add the following to the "android" part of the same gradle file:

```
compileOptions {
      sourceCompatibility = 1.8
      targetCompatibility = 1.8
}
```

Data Transfer Objects

We are going to fetch JSON-formatted data from the server and convert it into Java objects. What's the "shape" of these objects? We need to define them and ensure that they match the format of the downloaded JSON. Go back to the example and examine the structure. You should note that:

- The first level (let's call it WeatherResponses) contains variables
  - "cod" - String
  - "calctime" - float
  - "cnt" - int
  - "list" - a list of complex objects (let's call individual objects WeatherResponse)
- WeatherResponse object contains:
  - "id" - int
  - "name" - String
  - "coord" - a complex object that contains coordinates of the city (let's call this one Coord)
  - "main" - complex object that contains the temperature in the city (let's call this one Main)
  - "dt" - int
  - "wind" - a complex object with wind info (let's call this one Wind)
  - "rain" - a complex object with rain info (let's call this one Rain)
  - "clouds" - a complex object with clouds info (let's call this one Clouds)
  - "weather" - a list of complex object with weather info (let's call individual object Weather)
- Going further down the hierarchy:
  - Coord contains:
    - "Lon" - float (longitude)
    - "Lat" - float (latitude)
  - Main contains:
    - "temp" - float
    - "temp_min" - float
    - "temp_max" - float
    - "pressure" - float
    - "sea_level" - float
    - "grnd_level" - float

        ■ “humidity” - float
    ○ and so on…

We need to define Java objects that correspond to the above hierarchy. However, since we are going to use these objects only to “unpack” the received JSON data they do not have to contain all the fields that are there in the JSON, just the ones we are interested in.

Define the following Java public classes in `si.uni_lj.fri.pbd.lab9.models.dto` package:

- **WeatherResponsesDTO**
- **WeatherResponseDTO**
- **CoordDTO**
- **MainDTO**

We are not going to bother with other classes (e.g. for Wind, Rain, Clouds, etc.) as we are not going to use them. Similarly, in the **MainDTO** we are only interested in the “`temp`” field. Below is the full implementation of **WeatherResponsesDTO**. Implement the rest based on this class:

```
public class WeatherResponsesDTO {

    @SerializedName("cod")
    private String cod;

    @SerializedName("calctime")
    private float calctime;

    @SerializedName("cnt")
    private int cnt;

    @SerializedName("list")
    @Expose
    ArrayList<WeatherResponseDTO> list;


    // Getter Methods
    public String getCod() {
    return cod;
    }

    public float getCalctime() {
    return calctime;
    }

    public int getCnt() {
    return cnt;
    }

    // Setter Methods
    public void setCod( String cod ) {
    this.cod = cod;
    }
```

```
        public void setCalctime( float calctime ) {
        this.calctime = calctime;
        }

        public void setCnt(int cnt ) {
        this.cnt = cnt;
        }

        public ArrayList<WeatherResponseDTO> getWeatherResponses() {
        return list;
        }
}
```

NOTE: the "`coord`" object differs between the example call response and what you will actually get when you call the API with your key. In **CoordDTO** should use "Lat" and "Lon" SerializedNames (pay attention to capitalization).

REST API Client definition and instantiation

In package `si.uni_lj.fri.pbd.lab9.rest` create a new public interface called **RestAPI**. Examining the URL from the example you will notice that it calls the "data/2.5/box/city?" endpoint with parameters "`bbox`" and "`appid`". The `bbox` parameter defining a rectangle area of the world we are interested in. The `appid` is the developer's API key to the OpenWeatherMap.

In this interface we define the API calls as java function calls. This is the form our call:

```
@GET("data/2.5/box/city?")
Call<WeatherResponsesDTO> getCurrentWeatherData(@Query("bbox") String bbox,
@Query("APPID") String app_id);
```

Examine the above lines, make sure you understand what they mean.

Next, create a public Java class **ServiceGenerator** in `si.uni_lj.fri.pbd.lab9.rest`. This helper class will for a given REST API interface create a client. The class is the same as the one from your Mini App 3: https://github.com/vpejovic/MiniApp3/blob/master/app/src/main/java/si/uni_lj/fri/pbd/miniapp3/rest/ServiceGenerator.java (please delete the Timber lines, if you don't use Timber for logging). Again, you just need to add the following interceptor:

```
HttpLoggingInterceptor httpLoggingInterceptor = new
HttpLoggingInterceptor();
httpLoggingInterceptor.level(HttpLoggingInterceptor.Level.BODY);
```

...to the OkHttpClient.

Calling the REST API

When a user moves to map we want to load the weather info for the selected area. We will show markers for each of the cities. Clicking on a marker will show the city name and the temperature (see figure on the right).

First, add a field `mRestClient` of type RestAPI to the activity. Instantiate it with `ServiceGenerator.createService(RestAPI.class);`

In “`onCameraIdle`”, which is a function that gets called when a user moves the map area, you need to call `getCurrentWeatherData` of the REST API interface. Data downloads should be done on a background thread to prevent app stalling. In Retrofit you can do that with “`enqueue`” function to which you provide a **Callback** object that will be called when the request is serviced. Your call should look like this:

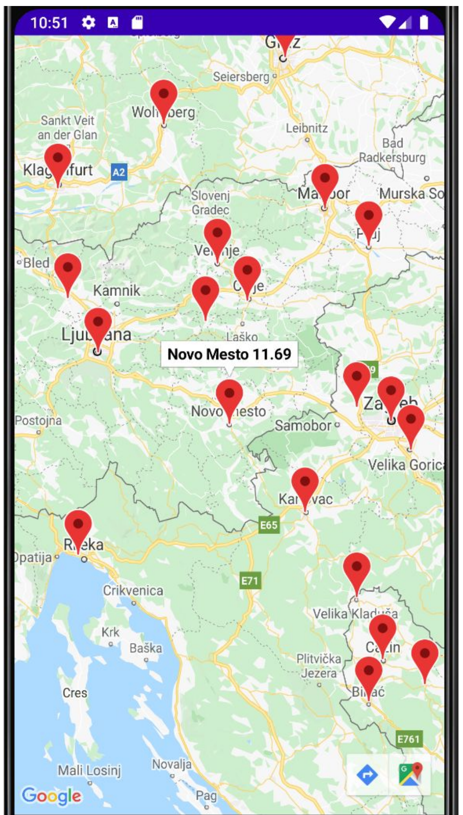
```
mRestClient.getCurrentWeatherData(bbox, Constants.API_KEY)
.enqueue(new Callback<WeatherResponsesDTO>() {

// override onResponse and onFailure

}
```

The `bbox` is a String parameter and it should be composed of “longitude-left,latitude-bottom,longitude-right,latitude-top,zoom”. “zoom” can be fixed to 10. The other parameters you can get from a **LatLngBounds** object obtained from `mMap.getProjection().getVisibleRegion().latLngBounds;` Read about **LatLngBounds** to figure out how to extract the latitude and longitude values of the bounding box.

Showing the markers

If the response you receive from the above call is successful (hint: `response.isSuccessful()`) it will return a **WeatherResponsesDTO** object. On this object you can call `getWeatherResponses()` to get a list of **WeatherResponseDTO** objects. Iterate over these objects and for each extract name, latitude, longitude, and temperature from the corresponding DTOs and create a marker for each:

```
mMap.addMarker(new MarkerOptions()
.position(new LatLng(lat, lon))
.title(name + " " + temp)
.icon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_RED
)));
```



A great thing about Retrofit is that you don't have to do any JSON parsing yourself and the interface ensures that what you do is type-safe, i.e. you are not misinterpreting the JSON result.

Connectivity Info

Our users might have limited data plans, so let's fetch data from the server only if they are connected to a network that is not metered (such as WiFi). In recent versions of Android you don't check the network type explicitly. Rather, you register to receive a callback when the network capabilities change[1].

Please add the following permission to the Manifest:

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

We need the following member fields in our MapsActivity:

Connectivity manager: `private ConnectivityManager mNwManager;`
Callback called on state changes: `private ConnectivityManager.NetworkCallback mNwCallback;`
Indicator of network state (metered/unmetered): `private boolean mOnUnmetered;`

In `onCreate` set **mOnUnmetered** by default to **false**. Then, instantiate `mNwCallback` with new `ConnectivityManager.NetworkCallback()`, and there override `onCapabilitiesChanged`. Check whether the new capabilities contain **NetworkCapabilities.NET_CAPABILITY_NOT_METERED**, and if so set **mOnUnmetered** to **true** (otherwise set to **false**).

The only thing left is to register the callback. Simply finish of `onCreate` with the following lines:

```
mNwManager = (ConnectivityManager)

getSystemService(Context.CONNECTIVITY_SERVICE);
mNwManager.registerDefaultNetworkCallback(mNwCallback);
```

You should unregister the callback in `onDestroy()`.

Finally, before calling the REST API client in onCameraIdle, check whether you are on an unmetered network. If not, don't execute the task, but show a **Toast** stating something like `"Connect to an unmetered network, please"`.

You can experiment with the app connected to different networks by enabling/disabling WiFi in the emulator, and/or turning the Airplane mode on/off.

Submitting Your Work

Submit your assignment in a private Bitbucket repository called PBD2020-LAB-9 (the last commit before the deadline will be graded) and add pbdfrita account (pbdfrita@gmail.com) as a read-only member to your repo.

You should **not submit** your Google Maps API key nor your OpenWeatherMap API keys. Instead, before you commit your project, clear these fields. However, you should add a directory "screenshot" to your repository's root where you should place a screenshot of the app running.

Happy coding!

[1] See here for the list of capabilities, including roaming, metered, etc.
https://developer.android.com/reference/android/net/NetworkCapabilities