Internet Connectivity: Part 1 JSON Parsing and Background Work

Recap

- We studied how to create new Activities and Fragments.
- We studied Activity and Fragment lifecycles, lifecycle awareness.
- We saw best practices on implementing UI elements like buttons.
- We saw how to efficiently implement lists.

Recap

- We studied Master/Detail flows.
- We learned how to deploy layouts targeted at tablets and phones (using screen width).
- We emphasized some aspects of software engineering: data encapsulation, data flow.
- Thinking about these aspects led to a good software design.

Today

- We’ll connect our app to the cloud to download weather data (very standard example).
- This will require some JSON parsing.
- We’ll discuss how to fetch this data.

JSON: Parsing with JSONObject

- To parse JSON, we need to grab name-value pairs.
- Treat the name as a key, the value goes into variables.
- The code to parse the previous example in Android is straightforward.
- Assume we read that whole JSON code into a String variable from the internet (or a file).
- The parsing code is as follows.

JSON: Parsing with JSONObject

```
{
  “firstName”: “John”,
  “lastName”: “Cena”,
  “address”: {
      “city”: “Awesomeland”,
      “country”: “USA”
      “zip” : 98416
   }
}
```

```
JSONObject jsonObj = new JSONObject(json_string);
String firstName = jsonObj.getString(“firstName”);
String lastName = jsonObj.getString(“lastName”);

JSONObject adrsObj =
jsonObj.getJSONObject(“address”);
String city = adrsObj.getString(“city”);
String country = adrsObj.getString(“country”);
int zip = adrsObj.getInt(“zip”);
```

JSON: Parsing with GSON

- This isn’t too bad. A bit error prone, but okay.
- Always check if there are other widely-accepted solutions before you roll your own.
- Google kindly wrote Gson, a Java library for converting between Java objects and JSON strings.
- To use this, add a Gradle dependency:
  - implementation 'com.google.code.gson:gson:$Version'
- We don’t have any Gson examples, but they’re easy to find.

Internet Connectivity

- Let’s learn how to use a fairly standard workflow.
- We’ll get some JSON objects from an internet source.
- We’ll read it in, parse it, and display it in a RecyclerView.
- A great source is Openweathermap.
- This API gives us detailed weather data for many cities.
- You need an account to access their data.
- An API call for London looks like:

  http://api.openweathermap.org/data/2.5/weather?q=London,uk&appid=yourapikeyhere

Openweathermap: JSON for London

```
{
"coord":{
"lon":-0.13,
"lat":51.51
},
"weather":[
{
"id":803,
"main":"Clouds",
"description":"broken clouds",
"icon":"04n"}
],"base":"stations","main":
{"temp":289.95,"pressure":1018,"humidity":77,"temp_min":289.15,"temp_max":291.15},"visibility"
:10000,"wind":{"speed":7.2,"deg":230},"clouds":{"all":64},"dt":1536643200,"sys":
{"type":1,"id":5091,"message":0.0337,"country":"GB","sunrise":1536643751,"sunset":153669023
7},"id":2643743,"name":"London","cod":200
}
```

Openweathermap: JSON API

- The main tags and their JSON data types are:
  - coord (object)
  - sys (object)
  - weather (array)
  - main (object)
  - wind (object)
  - name (String)
- We can parse the JSON for this without using Gson since it is pretty straightforward.

JSON Parser: Example 25 (Openweathermap API)

- We’re going to write a simple JSON parser for Openweathermap’s API.
- This will be the JSONWeatherUtils class.
- **Software design note:** JSONWeatherUtils is a helper class. You should never have to instantiate it unless there is some benefit in doing so.
- The class must therefore have static member functions.
  - In Kotlin, you simply declare the class using “object” instead of “class”, and it makes all the functions static
- We’ll use this parser to populate our own WeatherData class.
- This is not particularly complicated, just tedious.
- In your codes, I recommend you use Gson instead.

Getting the JSON Data

- Our previous example just sits there and does nothing.
- We’d like to get data from the Openweathermap servers.
- Ideally, we get to pick the location for which we’re retrieving data.
- We’ll need the URL and the api key associated with my Openweathermap account.
- Let’s look carefully at constructing the URL.

Getting the JSON Data: URL

- The URL must be constructed piece by piece.
- First piece is fixed:
  - http://api.openweathermap.org/data/2.5/weather?q=
- Second piece is user-defined:
  - {city},{country}
- Third piece is fixed:
  - &appid=
- Fourth piece is my app id:
  - Some long string here

Getting JSON Data: URL from String

- You can use the URL class to do this, or use Uri builder and convert it to URL.
- The URL class can be used as follows:

```
private static String piece1 = “http://api.openweathermap.org/data/2.5/weather?q=”;

private static String piece3 = “&app_id=”;

private static final String app_id = “somekey”;

public URL BuildURLFromString(String loc){

        return new URL( piece1 + loc + piece3+app_id);

 }
```

Getting JSON Data: String->URI->URL

- The previous approach is not considered very safe or portable.
- Google recommends building URLs from URIs.
- URIs should in turn be built piece by piece using URI.Builder.
- We won’t cover this, but keep it in mind.

Getting JSON Data: Connecting to the Internet

- Now that we have the URL, we are ready to get our JSON data.
- There are many ways to do this.
- Many programmers use the library Volley.
- We will write code so we can explore some of the engineering decisions.
- Let’s see an example.

Getting JSON Data: Connecting to the Internet (Example 25 continued)

- Create another helper (object) class called NetworkUtils.
- Two methods: buildURLFromString and getDataFromURL.
- The latter takes a URL, and uses HTTPURLConnection to open a connection.
- Open an InputStream through that connection.
- Use Scanner to parse input stream, return the JSON string.
- NetworkUtils should only have static methods (why?).

Connecting to the Internet

- There’s a serious problem with what we’ve written so far.
- If you run it naively, this could crash your app.
- This is because the network request is being done on the main thread.
- Your app will become unresponsive if the data is large, or the network is slow.
- **Solution**: network request must be on its own thread.

Multithreading in Android

- When you start an Android app, the “main” thread (aka UI thread) is automatically created.
- Must avoid lengthy ops on UI thread.
- Move non-UI operations (data-related) to background/worker threads.

Multithreading in Android

- For small to medium data transfers, Google used to recommend the use of AsyncTask.
- Let’s see how to use AsyncTask.

Multithreading in Android: AsyncTask

- AsyncTask is an abstract class that needs to be extended.
- Ideal for threads that run for a few seconds and stop.
- An AsyncTask is defined by 3 generic types:
  - Params, Progress, and Result.
- An AsyncTask needs four methods:
  - onPreExecute()
  - doInBackground()
  - onProgressUpdate()
  - onPostExecute()

Multithreading in Android: AsyncTask

- Params: what you send in. In our case, we want to send a String or URL from which to read the JSON data.
- Progress: some type of progress update. You can leave this as Void in many cases, or return some calculation of how much data has been downloaded.
- Result: the result of the background thread. In our case, it's the String obtained from the URL, or maybe an instance of WeatherData.

Multithreading in Android: AsyncTask

- onPreExecute(): invoked on UI thread before the task is executed.
- doInBackground(Params...): invoked on background thread. Takes in data for executing task, returns Result.
- onProgressUpdate(Progress...): invoked on the UI thread, can be used for conveying progress (logs, animations).
- onPostExecute(Result): invoked on the UI thread. Good time to store result or display it.
- Finally, call execute(Params...) in the UI thread.
- Sounds neat, right?

Multithreading in Android: AsyncTask

- AsyncTask has been deprecated!
- It would be nice to be able to replicate the functionality of AsyncTask.
- The pieces we need are:
  - Do something before the background thread starts.
  - Do something in the background.
  - Post something to the main (UI) thread once work is done.
- We could use a threading library like Groovy or Kotlin coroutines.
- Or... we could roll our own solution.
  - We’ll return to Kotlin coroutines in more detail later.

ExecutorService

- We’ll create a single background thread using the ExecutorService class:
  - ExecutorService var = Executors.newSingleThreadExecutor()
- Then make the thread do something like so:
  - var.execute{

```
    //work goes here...
    postToMainThread(...)
    }
```

Handler and Looper

- How do we implement postToMainThread()?
- One elegant way on Android is to use a Handler-Looper pair:
  - Handler var2 = HandlerCompat.createAsync(Looper.getMainLooper());
- The above line grabs the “main” thread (aka the UI thread) and holds a reference to it in var2.
- Then, we can use that handle to post to the main thread:
  - var2.post{

    //modify UI or member vars here}

    }

Progress Update?

- How do we replicate AsyncTask’s onProgressUpdate method?
- We won’t explore this in any great detail but...
  - Could use Handler-Looper pair to update to the main thread periodically.
  - Could set up a listener interface using the callback pattern that occasionally posts results back to the main thread.

JSON Parser: Example 26 (Openweathermap API + Background thread)

- Remember Example 25: a parser utility class, a network utility class, a class to hold the parsed data.
- Now, we create a layout that takes in a city,country.
- On hitting a submit button, we fetch temperature, pressure and humidity.
- We’ll create a new subclass of our activity called FetchWeatherTask that does the work in a background thread.
- We’ll engineer this carefully so it’s mostly a drop-in replacement for the now outdated AsyncTask.

JSON Parser: Example 26

- When I ran this, I got an exception from Android saying that cleartext http is not permitted.
- In other words, they’d prefer we use an https:// connection.
- To connect to the internet, we need two permissions:
  - android:permission.INTERNET
  - android:usesCleartextTraffic= “true”
- Warning: the app is not lifecycle-aware!
- Could create zombie threads each time we rotate the screen (dangling references on the app side).
- How do we fix this?

Example 27: Making Example 26 Safe (+ in a fragment)

- Now, if this thread lives in a fragment, we have to do some work to fix things.
- We’ll need to make the FetchWeatherTask static so it doesn’t get recreated each time the fragment gets recreated.
  - In Kotlin, this is done by creating a companion object
  - Static nested classes cannot access private members of outer class without a WeakReference, so we’ll need to add that.
- An alternative is to make FetchWeatherTask a singleton.
- We’ll also make sure in the Activity that we don’t recreate the Fragment when the Activity is recreated.
- We’ll override the fragment’s onSaveInstanceState() to do some UI state backup, and restore in onCreateView().

Fragments and Lifecycle awareness

- Do what we did in Example 27, or...
- Use the older API method fragmentName.setRetainInstance(true). This is now deprecated.
- The **right way is to use a ViewModel**, which actually preserves information across Activity and Fragment lifecycle changes.
- We will see this starting next week.
  - Either use an older API or use lifecycle methods for project 1 (unless you figured out ViewModels).