Advanced Mobile Application Development
Week 13: Android and APIs

Network Connections

https://developer.android.com/training/basics/network-ops

Each Android application runs in its own process in a single thread which is called the **Main thread** or **UI thread.** Prior to Ice Cream Sandwich (API 14/15), Android apps could perform any operations from the main UI thread. This led to such situations where the application would be completely unresponsive while waiting for a response. Now there are stricter rules regarding performing operations -- tasks have 100-200ms to complete a task in an event handler. Operations that take longer could be killed by the WindowManager or ActivityManager processes and the user will receive an application not responding (ANR) message.

In scenarios where we're not in control of how long the operation will take we shouldn't perform them on the main thread. Instead we should perform them on a thread that runs in the background so our main UI thread never freezes or errors out.

- File IO
- Database interaction
- Network interaction
- API integration

An asynchronous task is defined by a computation that runs on a background thread and whose result is published on the UI thread.

Network Request

Android includes the **HttpURLConnection** class which supports sending and receiving data over HTTP. The openConnection() method to make a connection to a given URL.

Retrofit and Volley both are the REST client libraries that will handle all of this networking for us.

Volley

https://developer.android.com/training/volley

Volley is an HTTP library that makes networking for Android apps easier and faster.

Volley offers the following benefits:

- Automatic scheduling of network requests
- Multiple concurrent network connections
- Disk and memory response caching
- Support for request prioritization.
- Cancellation request API. You can cancel a single request, or you can set blocks or scopes of requests to cancel.
- Ease of customization, for example, for retry and backoff.
- Strong ordering that makes it easy to correctly populate your UI with data fetched asynchronously from the network.
- Debugging and tracing tools.

Volley uses a `RequestQueue` that manages worker threads for running the network operations, reading from and writing to the cache, and parsing responses. You can use these Volley convenience methods or create a custom RequestQueue.

- The `Volley.newRequestQueue` method sets up a `RequestQueue` using default values and starts the queue

- To send a request you construct a request object and add it to the `RequestQueue` using the `add(stringRequest)` method
- Once you add the request it moves through the pipeline, gets serviced, and has its raw response parsed and delivered.

Volley supports these request types or you can create a custom request

- StringRequest -- receives a raw string from a URL
- JsonObjectRequest (subclass of JsonRequest) – receive a JSON object from a URL
- JsonArrayRequest (subclass of JsonRequest) – receive a JSON array from a URL

Request objects do the parsing of raw responses and Volley takes care of dispatching the parsed response back to the main thread for delivery.

More info on how Volley handles requests https://developer.android.com/training/volley/simple#send

If your application makes constant use of the network, it's probably most efficient to set up a single instance of `RequestQueue` that will last the lifetime of your app. You can achieve this by implementing a singleton class that encapsulates `RequestQueue` and other Volley functionality. This is more modular and the preferred approach over subclassing Application and setting up the queue in `Application.onCreate()`.

A key concept is that the `RequestQueue` must be instantiated with the `Application` context, not an `Activity` context. This ensures that the `RequestQueue` will last for the lifetime of your app, instead of being recreated every time the activity is recreated (for example, when the user rotates the device).

Volley is not suitable for large download or streaming operations, since Volley holds all responses in memory during parsing. For large download operations, consider using an alternative like `DownloadManager`.

Add the following dependency to your app's build.gradle file:

```
implementation 'com.android.volley:volley:1.2.1'
```

Downloading Images

Downloading images from the web introduces some issues around the network request, saving images, and handling caching of the images. There are many image loader libraries available on Android including Picasso, Glide, and Coil. We'll use Picasso. https://square.github.io/picasso/

The Picasso library handles many common pitfalls of image loading on Android:

- Handling ImageView recycling and download cancelation in an adapter
- Complex image transformations with minimal memory use
- Automatic memory and disk caching

It also lets us easily define placeholder and error images (note that these are not automatically resized, so do that before you add them to your project. Mine ended up 75x56).

Add the following dependency to your app's build.gradle file:

```
implementation 'com.squareup.picasso:picasso:2.71828'
```

API

I created an app that uses the Movie Database (TMDb) API
https://developers.themoviedb.org/3/getting-started/introduction

Sign up https://www.themoviedb.org/account/signup

Documentation/Getting started https://developers.themoviedb.org/3/getting-started/authentication

Register for an API key, click the API link from within your account settings page.
https://www.themoviedb.org/settings/api/request individual developer, accept the agreement.
All fields are required.

Now you have an API key and a sample URL
https://api.themoviedb.org/3/movie/550?api_key=9bc41deb95194da5e8865be1fe7750a4

We'll be using the top rated movies API https://developers.themoviedb.org/3/movies/get-top-rated-movies
Use the try it out tab to send a sample request and see the JSON response. You can also do this directly in your browser.
https://api.themoviedb.org/3/movie/top_rated?api_key=9bc41deb95194da5e8865be1fe7750a4&language=en-US&page=1

When using an API you have to look at the JSON to understand the structure of the data. Each API will have a different structure. This one is pretty straight forward.
Keys: page, total_results, total_pages, and results. The value for results is an array. Top rated returns 20 results, which is plenty for the sample app.
Look at one of the results to see what data is sent back.

The next step is to figure out what data we want to use from the API.
id: value is an int
title: value is a String
vote_average: value is a Double
poster_path: value is the name of the jpg (note it is not the full path)

Movies
Create a new project
Empty Activity template
Name: Movies
Package name: the fully qualified name for the project
Language: Kotlin
Minimum SDK: API 21 (21 is the minimum API for Material Design)
Leave legacy libraries unchecked (using these restricts usage of some newer capabilities)
Finish

Our application requires internet access, so we must request the INTERNET permission in the AndroidManifest.xml file. Add the following before the application tag.

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

Add the dependencies for Kotlin extensions (includes support for by viewModels), viewmodel, livedata, Volley, and Picasso into the app's grade file.

```
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.4.1"
```

```
implementation "androidx.lifecycle:lifecycle-livedata-ktx:2.4.1"
implementation 'androidx.activity:activity-ktx:1.4.0'
implementation 'com.android.volley:volley:1.2.1'
implementation 'com.squareup.picasso:picasso:2.71828'
```

Add colors, create a theme with a Material Components parent theme, and apply it to your app.

Layout
In the layout file delete the default textview.
With Autoconnect on in design view drag out a RecyclerView to display the list (can be found in common or containers).
Give the RecyclerView an id `android:id="@+id/recyclerView"`
Add start, end, top, and bottom constraints of 0.
Make sure the RecyclerView has layout_width and layout_height set to "match_constraint" (0dp).

List Item Layout
We also need a layout file to determine what each row in our list will look like. I decided to use a cardView.
File | New | Layout resource file
File name: card_list_item
Root element: androidx.cardview.widget.CardView
Source set: main
Directory name: layout

Then I added a constraint layout as a child of the CardView and set up my layout in that (in layouts)

For each movie we'll show the poster in an ImageView, the title in a TextView and the vote average in a TextView. Add those views, give them each ids, constraints, and any padding or styling you want. Once you don't need the default text to help with layout, remove it.

Change the card view layout's height to "wrap_content" and the constraint layout's height to "wrap_content" as well. If the height is match_constraint each card will have the height of a whole screen.

Kotlin class
We're going to create a custom Kotlin class to represent the data we'll be using.
First create a new package for our model. In the java folder select the movies folder (not androidTest or test)
File | New | Package
Name: model

Then select the new model package and add a new class called Movie.
File | New | Kotlin class
Name: Movie
Kind: Class

The Movie class will be a data class that includes the movie id, title, posterURL, and rating. Instead of saving the vote average as a Double and converting it to a String every time I needed to display it in a

TextView, I decided to save it as a String and I'll just do the conversion once. This works since I won't be doing any mathematical operations on it, I will always need it as a String.
The id is an Int because the API provides an Int id so we can use that.
We'll define all these properties in the primary constructor in the class header.

```
data class Movie(val id:Int, val title:String, val posterURL: String, val
rating: String){}
```

ViewModel
Now we'll create our view model class.

Select the model package and add a new class called MovieViewModel.
File | New | Kotlin class
Name: MovieViewModel
Kind: Class

Our MovieViewModel class should subclass the ViewModel class.
The view model will include a MutableLiveData ArrayList that we'll later observe in MainActivity.
We'll also add a method that updates the movie list with a new list.

```
class MovieViewModel: ViewModel() {
    val movieList = MutableLiveData<ArrayList<Movie>>()

    fun updateList(newList: ArrayList<Movie>){
        movieList.value = newList
    }
}
```

Now create another package in movies called util and then a Kotlin class in data called JSONData where we will request the data from the API and parse the JSON data. This will keep the actual data handling separate from our view model class.

File | New | Package
Name: util

Then select the new data package and add a new class called JSONData.
File | New | Kotlin class
Name: JSONdata
Kind: Class

```
class JSONdata {
    fun loadJSON(context: Context, movieViewModel: MovieViewModel){
        val url =
"https://api.themoviedb.org/3/movie/top_rated?api_key=9bc41deb95194da5e8865b
e1fe7750a4&language=en-US&page=1"

        // instantiate the Volley request queue
        val queue = Volley.newRequestQueue(context)

        // Request a string response from the provided URL.
        val request = StringRequest(Request.Method.GET, url,
```

```
            { response ->
                parseJSON(response, movieViewModel)
            },
            {
                Log.e("RESPONSE", error("request failed"))
            }
        )

        // Add the request to the RequestQueue
        queue.add(request)
    }

    fun parseJSON(response: String, movieViewModel: MovieViewModel){
        val dataList = ArrayList<Movie>()
        // Base url for the posters
        val poster_base_url = "https://image.tmdb.org/t/p/w185"

        try {
            //create JSONObject
            val jsonObject = JSONObject(response)

            //create JSONArray with the value from the results key
            val resultsArray = jsonObject.getJSONArray("results")

            //loop through each object in the array
            for (i in 0 until resultsArray.length()) {
                val movieObject = resultsArray.getJSONObject(i)

                //get values
                val id = movieObject.getInt("id")
                val title = movieObject.getString("title")
                //save the fully qualified URL for the poster image
                val posterURL = poster_base_url +
movieObject.getString("poster_path")
                val rating =
movieObject.getDouble("vote_average").toString()

                //create new movie object
                val newMovie = Movie(id, title, posterURL, rating)

                //add character object to our ArrayList
                dataList.add(newMovie)
            }
        } catch (e: JSONException) {
            e.printStackTrace()
        }
        movieViewModel.updateList(dataList)
    }
}
```

Adapter

Now we need to create an adapter which will be used to populate the data into the RecyclerView. The adapter's role is to convert an object at a given position into a list row item to be inserted in the list.

In the java folder create a new class for our adapter.
File | New | Kotlin Class
Name: MyListAdapter

The class will need to subclass RecyclerView.Adapter

```
class MyListAdapter: RecyclerView.Adapter {}
```

AS should add the import for the RecyclerView class:

```
import androidx.recyclerview.widget.RecyclerView
```

With a RecyclerView the adapter requires the existence of a ViewHolder object which describes and provides access to all the views within each item row.
To indicate that our class will be implementing the ViewHolder class add it to the class definition.

```
class MyListAdapter: RecyclerView.Adapter<ViewHolder>() {}
```

You will give you errors until you implement the required methods.
Select the light bulb and choose implement methods to get rid of the error.
Every adapter has three required methods that we need to implement.

You will still have an error because we need to define the ViewHolder class which will set up the bindings to the view in the layout file.
In our MyListAdapter class we'll create a ViewHolder class with a default constructor that defines our card with the title, rating, and image.

```
class ViewHolder(view: View): RecyclerView.ViewHolder(view) {
    val titleTextView: TextView = view.findViewById(R.id.textView)
    val ratingTextView: TextView = view.findViewById(R.id.textView2)
    val imageView: ImageView = view.findViewById(R.id.imageView)
}
```

This should have fixed the errors.
With the viewholder defined, let's set up the MyListAdapter class. We'll define a view model and add it to the primary constructor.

```
class MyListAdapter(private val movieViewModel: MovieViewModel):
RecyclerView.Adapter<MyListAdapter.ViewHolder>(){}
```

Create an array with the items from our view model at the class level.

```
private var myMovieList = movieViewModel.movieList.value
```

Now we'll implement the 3 required methods.
OnCreateViewHolder() is called automatically by the adapter each time it needs to display a data item.
When you inflate the xml layout file, you get a view, and then you wrap that in an instance of your ViewHolder class and return that object.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ViewHolder {
    //create an instance of LayoutInflater
```

```
    val layoutInflater = LayoutInflater.from(parent.context)
    //inflate the view
    val itemViewHolder = layoutInflater.inflate(R.layout.card_list_item,
parent, false)
    return ViewHolder(itemViewHolder)
}
```

OnBindViewHolder is called each time the adapter needs to display a new data item. It passes in the reference to the ViewHolder and the position of the data item in the collection. The job of OnBindViewHolder is to take that data object and display its values.
I added an image(image_placeholder75.png) to the drawable directory to act as the placeholder and error image and named it image_placeholder.png.

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    //get data at the position
    val movie = myMovieList?.get(position)
    //set the text of the title textview to the name
    holder.titleTextView.text = movie?.title ?: ""

    //set the text of the rating textview to the rating
    holder.ratingTextView.text = "Rating: " + movie?.rating ?: ""

    //load image using Picasso
    Picasso.get().load(movie?.posterURL ?: "")
        .error(R.drawable.image_placeholder)
        .placeholder(R.drawable.image_placeholder)
        .into(holder.imageView);
}
```

getItemCount() returns the number of items in the collection.

```
override fun getItemCount(): Int {
    if (myMovieList != null) {
        return myMovieList!!.size
    } else return 0
}
```

We'll also need a method that we can call when there's new data that will update the array list using the view model and then notify the adapter that the data has changed.

```
fun update(){
    myMovieList = movieViewModel.movieList.value
    notifyDataSetChanged()
}
```

MainActivity
We need to define an instance of the view model so we can access a single source of truth for our data and we'll do that at the class level.

```
private val viewModel: MovieViewModel by viewModels()
```

In onCreate() we need to

1. Load the JSON data
    a. Because this is an expensive transaction I don't want to do it every time the activity is created, which includes on every rotation. So I check to see if the ArrayList is null and only load the data in that case.
    b. Because the view model has the lifetime of the whole app, the view model and the data will outlive the activity.
    c. We also won't need to worry about saving state because the view model handles that as well
2. set up our recyclerView
3. instantiate an adapter with our view model
4. set the adapter to the RecyclerView
5. set a Layout Manager for our RecyclerView instance
6. create the LiveData observer so that whenever the onChanged event is triggered we notify the recycler view that the data has changed.
7. assign the observer to our view model's list of items. So whenever this list is changed the onChanged event will fire and the observer will be called.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    //only load the JSON data once per app lifetime
    if (viewModel.movieList.value == null){
        val loader = JSONdata()
        loader.loadJSON(this.applicationContext, viewModel)
    }

    //get the recycler view
    val recyclerView: RecyclerView = findViewById(R.id.recyclerView)

    //set a layout manager on the recycler view
    recyclerView.layoutManager = LinearLayoutManager(this,
LinearLayoutManager.VERTICAL, false)

    //define an adapter
    val adapter = MyListAdapter(viewModel)

    //assign the adapter to the recycle view
    recyclerView.adapter = adapter

    //create the observer
    viewModel.movieList.observe(this, Observer {
        adapter.update()
    })
}
```

Try it first without the observer, what happens? Put in some Log statements to figure out what's going on.
Why does the observer fix this?
Your app should now be able to request the JSON data from the API, parse it and display it in the RecyclerView.

You could build on this and have the card expand when the user clicks on it to show the movie overview or other information available through the API.