Networking, Asynchrony

Overview

- Persistent Storage (DataStore, Files)
- Room, Repository
- Jetpack compose
- Lab 4
- Project Phase 2

Today

- We'll look at making HTTP Requests as a case study in asynchronous tasks in Android
- We'll see a few of the approaches for managing background work/scheduling, etc

HTTP request/response steps

- Create request (string processing, but sort of tricky)(fast)
- Send request via network (maybe slow)
- Wait for response via network (definitely slow)
- Parse response (string processing, tricky, fast)

Request preparation

- We need to specify the URL we want to access
- This can contain many parts:
  - Scheme (http or https)
  - User info (unlikely but possible)
  - Host, port
  - Path
  - Query string
  - Fragment (unlikely)
- Also, potentially header info, especially if we need to be authorized to access the data

URI.Builder

- Class for creating a URI
- We specify the parts of the URI and it will handle stuff like string encoding (if a query parameter has a space, etc) for us
- Example of the “builder pattern”

```
val builder = Uri.Builder();
builder.scheme("https")
    .authority("www.myawesomesite.com")
    .appendPath("turtles").appendPath("types")
    .appendQueryParameter("sort", "relevance")
    .fragment("section-name");
val myUrl = builder.build().toString();
```

Design Pattern: Builder

- Since URIs have a ton of optional configuration parameters, you can create them using a “Builder”
- You create a Builder object with (for example) `someClass.Builder()`
- The builder has methods for setting cofig options such as `.addTag(SOME_TAG) or setInputData(...)`
- Once you set all the options, you call `.build()` to get the actual object
- This pattern is useful whenever you have a complex object with lots of options that can be optionally specified. An alternative in a dynamic language would be to take a map of options as a parameter

JS example avoiding builder

```
var myObj = myLibrary.createObject({
    goFast: true,
    magicString: "hello world",
    fudgeFactor: 3.0", secretOption:
    { very: "special"}
    });
```

Sending the request, waiting for the request

- Java's `URL` class has an `openConnection` method that returns a `HTTPUrlConnection` whose `connect()` method will make the request
- It has a `getInputStream()` method that lets us read the response
- The stream skips the headers which are accessible via some other methods
- Sounds easy, but it's a bit more complicated

Permissions

- Apps by default are very limited in what they can do
- We need to add required permissions to the app's `AndroidManifest.xml` file:

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

- Also, the API we'll be hitting doesn't support HTTPS, so we'll need to add another line to allow our app to use unencrypted HTTP

More important: Blocking

- Our request might take a long time to complete, so if we do it on UI thread and wait for a response, we'll have a bad time!
- We need to make sure (and Android will enforce!) that we don't do something slow like networking on the main thread
- There's a few ways around that

Approach 1: Threading

- It's easy to run code in `Thread({ my code here}).start()`
- But there's a few issues there...
- UI updates (or VM updates) need to happen on particular threads, so we'll have to use someting like the `post` method to make sure we run that part of our code in the right thread, even if we start it in our own `Thread`. This isn't a big deal, but might be surprising at first
- The bigger issue is that threads are difficult to control. In general, they can't be cancelled!
- Their lifetime is not tied to the lifetime of where we created it. Maybe the thread will keep running after our activity has been destroyed!

Much better approach: Coroutines!

- Most of the time when we make an HTTP request, we'll initiate it on some screen, and only need the result while that screen is displayed
- A coroutine scope that's tied to the lifecycle of something visible will cancel the request when it's no longer needed... perfect
- We can also manually cancel coroutines, as we'll see in an example
- Coroutines are also cheaper than starting a new thread, so it's win win win

Useful coroutine trick: withContext

- This method runs a coroutine in a given scope, and there are several useful ones to choose from
- For our purposes, we'll use `Dispatchers.IO` which is a scope designed for long running IO jobs, like our HTTP request
- It's also a suspend function, so itself must be run in a coroutine scope, but that allows it to return a value, which is pretty convenient

Handling the reponse: GSON

- GSON is a lib from google for serializing/deserializing JSON
- Include it as a gradle dependency: `implementation 'com.google.code.gson:gson:2.10.1'`

```
val gson = Gson()
val Obj = object{
    val x = 1234;   val y = "hello";   val z = arrayOf(1, 2, 3, 4)
}
val str = gson.toJson(Obj)
val result = gson.fromJson(Reader, Obj::class.java)
```

- Deserializing works similarly for other objects, you just get a kotlin object back. Tip: write data classes for objects you'll need to serialize/deserialize

Alternatives to Coroutines

- Using coroutines to run tasks in the background is pretty nice, but can require a fair amount of bookkeeping to track all the tasks you're performing
- We'll look at some libraries that handle that for us

Volley

- Manages scheduling/resource management of network requests
- Uses multiple concurrent connections
- Handles prioritization, cancelling
- Optimized for small amounts of data (JSON formatted data, not Linux ISOs)
- Some good software engineering ideas that we'll examine and can steal in our own projects

Main ideas

- You use volley by creating a `RequestQueue` and adding `Request` objects to it
- Volley manages threading for all requests in the queue, including network access, parsing, etc
- Your request includes callbacks that Volley will run on success or error
- The callbacks run on the UI thread so you can modify views

Details: Singletons (Review)

- Our app should only have a single RequestQueue/ImageLoader since it's designed to manage multiple simulateous requests
- It's common to want exactly one instance of a class, and the software engineering term for this is the “Singleton Pattern”
- There's some trickyness about efficiently getting exactly one initialized instance in multithreaded programs. We need to make sure that if 2 threads try to access the instance for the first time together, that only one of them actually creates it
- In kotlin an `object` handles all this synchronization for us automatically.

  Unless...

Pain point: Singletons with constructor params

- If we need to pass constructor params, we have to implement the initialization stuff manually
- It's still nicer in Kotlin than in Java
- We have a static reference to the one object, and we make a getter method for it
- In that method we safely create it if necessary, and otherwise return the existing instance

Services, Download Manager

- Next we'll look at Download Manager for handling slower network requests
- Download Manager is an example of a `Service`
- Services are long running background operations with no UI elements
- They can keep running even when users leave their app
- They come in 3 flavors: Foreground, Background, and Bound

Broadcast Receivers

- One more Application component: Broadcast Receivers
- A broadcast receiver can subscribe to messages
- A publisher broadcasts those messages and all receivers can react to those messages

Download Manger

- DownloadManger is a service
- It sends a broadcast when a download completes
- In order to use this service, your app must be a broadcast receiver to react to that broadcast
- It handles retries and shows results in the Downloads app
- It only downloads one item at a time, but can queue multiple requests (via simple URLs)

Requesting a download

- Use `getSystemService` to get a reference to the Download Manager service
- Create a `DownloadReuest` object and enqueue it with the manager which gives you an ID to track your download with
- Register a handler with the system and specify which types of broadcasts you want to handler via an `IntentFilter`
- In your handler, if the ID matches, your download completed

Recap

- We saw a few more ways to handle asynchronous background work: Coroutines, Volley, DownloadManager
- In any type of application development, you'll be using tools like these to manage background work
- Based on the type of background work, the libraries provide different tradeoffs in terms of latency or resource usage
- Specific to Android, we saw how `Activities`, `Services`, and `Broadcast Receivers` work together