KCL Tech Build X: Android

Lecture Seven: Networking and APIs

- What is an API?
- HTTP Requests
- Threads
- Using a Web-based API in Android

What is an API?

An API (*application-programming interface*) is a defined set of **programming instructions and standards for accessing a software application or tool**. The term API normally refers to web-based applications/tools.

APIs are used to **utilise the services of one piece of software, in another**. For example, an app that gives users access to news stories might use the public APIs from the BBC, CNN, etc; a website that sends a confirmation to your phone might use the API from a telephony provider (like Twilio).

We will be considering only **web-based APIs** - these are ones that are dealt with by sending requests (and receiving replies) over the Internet. We will be using the **World Bank API** as an example; this API provides realtime and historical access to over 8,000 economical and environmental data points.

HTTP Requests

To use an API, you need a basic understanding of **HTTP requests**. HTTP (*Hyper-Text Transfer Protocol*) is a system for sending requests and receiving replies over the Internet (and other networked resources). Every time you've loaded a regular web page, you've used HTTP (or HTTPS); you can see it at the start of the URL: `http://android.kcl.tech` **or** `https://secure-bank.com`**.**

An entire lecture could be given to HTTP requests, but the basic concepts you need to understand are as follows:

- **An application sends a request** to a web-based resource, and the resource should send a **response** back to the application.

- A **request** has five main components:
  - A **verb**, telling the resource what kind of request this is. These are the most common verbs:
    - `GET` "I want to read some information"
    - `POST` "I want to create something"
    - `PUT` "I want to update something"
    - `DELETE` "I want to delete something"
  - A **path**, which is the address of the resource.
  - A **protocol**, which is almost always just `HTTP/1.1`.
  - A set of **headers**, which provide extra information about the request, such as what format of data is being sent.
  - A **body**, which will contain the content you are sending if you're making a `POST`, `PUT`, etc. request.

- A **response** has three main components:
  - A **status code**, describing whether the request was successful and what its current state is. These are the most common status codes:
    - `200` - OK
    - `403` - Forbidden
    - `404` - Not Found
    - `500` - Internal Server Error
  - A set of **headers**, which contain extra information about the response (what format it is in, for example).
  - A **body**, which is the content the resource is replying with.

The bodies of requests and replies can be in a range of formats: plain text, HTML, XML, JSON, etc. We're going to consider **JSON**, because it is the easiest format to work with.

JSON stands for *JavaScript Object Notation*, and it is a lightweight format for expressing objects with key/value properties. A JSON object to describe a person could look like this:

```
{
    "name": "Mark",
    "age": 22,
    "is_alive": true,
    "job": {
        "company": "Unitu Ltd.",
        "role": "Software Engineer"
     },
     "friends": ["Alan", "Steve"]
}
```

JSON is easy to write, is human-readable, doesn't take up as much bandwidth, and is very easy for computers to deal with.

Threads

Threads are used by software to **work on multiple tasks at once**. Imagine a single person attempting a project, versus a group of people doing the same project: the single person has to do every task one after another; the group can work on multiple tasks at once. In the group, each person can be thought of as a thread. *(This is the very, very shortened version).*

In Android, threads allow your app to do multiple tasks at once. One very important thread is the **UI Thread** - this thread handles your app's user interface, and by default this is where all of your app's computation happens as well. When you see an app's interface "stutter" or freeze for a moment, that's usually because it's **doing too much work on the UI Thread**, and the processor couldn't keep up.

Why is any of this relevant? Requesting something from the web is a "blocking" action - the thread will stop doing any work until the request completes, which could take several seconds or more. If this was done on the UI Thread, **your app would freeze** until the request finishes. (In fact, if you try to perform a network action on the UI Thread **Android will throw an exception** immediately; this ensures developers don't do this by accident).

Long story short: you need to do networking on a separate thread.

Using a Web-Based API

There are two distinct steps involved in handling a single API transaction: **making the request** (on a new thread) and **parsing the result**.

Making a Request

As we now know, a network request **must** happen on a new thread. You can handle threading manually, or you can use something called an `AsyncTask` to let Android handle it for you. We're going to focus on `AsyncTask`-based threading, because that is what you will most likely be using.

An `AsyncTask` allows you to perform a task **asynchronously (i.e. "in the background") and then handle a result on the UI Thread, without having to manually deal with threads in your code. Just like adapters or database helpers, you can create your own background task by creating a class that extends `AsyncTask`, for example:**

```
public class DownloadFileTask extends AsyncTask {

    // ...

}
```

However, **this won't quite work**. An `AsyncTask` requires you to specify three **generic types**: the type of **parameter** your task will take in, the type of **progress** update it will give during the task, and the type of **result** it will give out.

> **Remember!** Generic types tell Java what kind of object you're going to be using with a certain class. You've used them before to create lists:
>
> `private ArrayList<String> names = new ArrayList<String>();`
>
> The **`<String>`** blocks tell Java that you're making an `ArrayList` of `String` objects.

In our example, we will be downloading a text file, so our generic types might look like this:

- **Parameter**: a `String` URL of the file to download
- **Progress**: a `Double` progress percentage (i.e. a decimal from 0 to 1)
- **Result**: the `String` contents of the file

This causes our correct class signature to look like this:

```
public class DownloadFileTask extends AsyncTask<String, Double, String>  {

    // ...

}
```

Once we have our class signature defined, we need to start **overriding some methods** to allow the class to actually do something.

The most important method, and the only one that is mandatory for overriding, is `doInBackground(...)`. As the name suggests, this is the method that runs on a background thread, because it may take several seconds or longer for your download request to complete.

This method will take in arguments, allowing information to be passed in from elsewhere in your code. These arguments will be whatever type you specified for **parameters** (the first generic); in our case, that means `String`s. It will return an object of whatever type you specified for **result** (the third generic); in our case, that means another `String`.

Our almost-working code now looks like this:

```
public class DownloadFileTask extends AsyncTask<String, Double, String>  {

    protected String doInBackground(String... params) {
        // do some long-running task, like download a file
        return "This must return a String, or null";
    }

}
```

The code to actually download a file will be covered live during the lecture, but for reference it is included at the end of this document in Appendix One.

Great! This `AsyncTask` will work, and will execute whatever code you put inside `doInBackground(...)` on a background thread, stopping the UI Thread from stuttering. But how do we **get access to the result** to actually do something with it?

To access the result we need to override another method: `onPostExecute(...)`. This is called after the long running-task finishes, and will receive one argument of whatever type you specified for **result**. In this method you have access to the result and you can use it as you see fit.

Our final code example looks like this:

```
public class DownloadFileTask extends AsyncTask<String, Double, String>  {

    protected String doInBackground(String... params) {
        // do some long-running task, like download a file
        return "This must return a String, or null";
    }

    protected void onPostExecute(String result) {
        // here we could show the result to the user
        // or use it in our UI
        // or do anything else we want!
    }

}
```

Parsing the Result

Now that we've made a request and received a basic `String`, we can step things up and **receive some JSON**. We're going to use a basic API provided by the **World Bank**: a list of countries. The URL for this request can be found in Appendix Two, and a sample response in Appendix Three.

This final section will be covered in live code, but these are the steps we will be covering:

- Changing the return type to `JSONArray`
- Converting the String response into a `JSONArray`
- Looping through the array to output a list of countries

Appendix One: File Download Code

```
public class DownloadFileTask extends AsyncTask<String, Double, String>  {

    protected String doInBackground(String... params) {
        // check input
        if (params.length != 1) return null;

        try {
            // open connection
            URL url = new URL(params[0]);
            InputStream inStream = u.openStream();
            DataInputStream dataInStream = new DataInputStream(inStream);

            // buffer to hold chunks as they are read
            byte[] buffer = new byte[1024];
            int bufferLength;

            // string builder to hold output
            ByteArrayOutputStream output = new ByteArrayOutputStream();

            // read from stream
            while ((bufferLength = dataInStream.read(buffer)) > 0) {
                output.write(buffer, 0, bufferLength);
            }

            // return output
            return output.toString("UTF-8");
        } catch (Exception e) {
            // something went wrong
            return null;
        }
    }

}
```

Appendix Two: Request URL

http://api.worldbank.org/countries/?format=json&per_page=300

Appendix Three: Sample Response


```
[
    {
        "page": 1,
        "pages": 1,
        "per_page": "300",
        "total": 264
    },
    [
        {
            "id": "ABW",
            "iso2Code": "AW",
            "name": "Aruba",
            "region": {
                "id": "LCN",
                "value": "Latin America & Caribbean (all income levels)"
            },
            "adminregion": {
                "id": "",
                "value": ""
            },
            "incomeLevel": {
                "id": "NOC",
                "value": "High income: nonOECD"
            },
            "lendingType": {
                "id": "LNX",
                "value": "Not classified"
            },
            "capitalCity": "Oranjestad",
            "longitude": "-70.0167",
            "latitude": "12.5167"
        },

        ...

        each country has a block like above

        ...

    ]
]
```