Worksheet 6: Remote Data

Updated: 14th October, 2018

In this worksheet, you'll implement a test app that connects to a web service. To do this, you'll need to run the web server itself, and build an app (making use of some pre-existing code) to connect to it.

Note: There are many existing public web services (and you're welcome to experiment with them!). While it's perhaps a minor concern, I don't wish to require you to sign up for a 3rd-party account (even if free) in order to complete this unit.

The Server

Obtain a copy of `testserver.zip`. This contains a simple, mock web server, designed expressly for this worksheet (certainly not designed to be general-purpose!).

To use it, we need to understand some things about it:

- It requires NodeJS (nodejs.org) to run, which should already exist on the lab machines. (It should be reasonably easy to install this on your own PC too.)
- The main server code is in `testserver.js`, which has a series of editable constants at the top. We'll come back to these later!
- The server uses HTTPS. It is important that you be exposed to encryption, because it's necessary in the real world. However, it does create extra work, both on the server and client (Android app).

  Notice that there are two other files: `key.pem`, which is the server's private key, and `cert.crt`, the server's public certificate. Normally, one must pay a fee to obtain a public certificate, since this involves having an organisation ("certificate authority") verifying that you are who you say you are. Your web server's certificate is therefore a guarantee to the client that it's talking to the right server (and not some malicious 3rd-party trying to access the client's data).

  However, it's impractical to obtain a certificate this way if we just want to experiment with and test some code. Instead, we can create a "self-signed certificate", `cert.crt` in this case. This *significantly compromises* the security of the server, but it is acceptable for testing purposes.

To run the server, you must extract all three files in `testserver.zip`, and run:

```
[user@pc]$ node testserver.js
```

The server will log important messages to the console. You can briefly check that it's working by navigating to `https://127.0.0.1:8000` in your web browser (running on the same machine as the server). Your browser should immediately warn you about security problems (due to our self-signed certificate), but will ultimately let you load the site anyway.

You should now see some JSON code.

1. Downloading with `AsyncTask`

Our app isn't going to do anything especially useful. It will just serve as a way to experiment with the code and the concepts.

Obtain a copy of `appfiles.zip`. This contains some existing code to get you started. Notice that the UI has three elements: a `Button`, a `ProgressBar`, and a `TextView`. Making the `ProgressBar` work is optional, and shouldn't be attempted in any case until you've made the downloading itself work.

Your task for the moment is as follows:

(a) Determine what URL you should connect to.

You can already go to `https://127.0.0.1:8000` in a web browser, but this is the "loopback" address, and only works if the client and server are running on the same machine. That's *not true* when Android is involved, even if it's the Android emulator.

Instead, observe the server's actual IP address(es) from its console messages. For instance, if it says "134.7.45.95", then the URL is "`https://134.7.45.95:8000`". (If you *are* running the Android emulator, and your server is running on the host machine, you can also use the special IP address "10.0.2.2".)

Normally we'd want to provide a path and query parameters to, making the URL more like `https://134.7.45.95:8000/webservice/rest?abc=1&xyz=2`. However, we'll skip this part to begin with. Fortunately, the web server is initially configured to ignore the path and query parameters, and just return the same JSON data no matter what.

(b) Add the server's certificate into the app. Specifically, make a new directory called *yourappproject*`/app/src/main/res/raw/`, and copy `cert.crt` into it (just that one file).

> **Note:** This directory stores miscellaneous files that your app needs to access as raw file data.

(c) Write the outline of the `AsyncTask` subclass within `MainActivity`. Specifically, you should extend `AsyncTask<Void,Void,String>`, as we don't need to set the parameters or update progress, but we *do* need to know that the result is a string. Override the `doInBackground()` and `onPostExecute()` methods. For now, just make them both stubs.

(d) Make the button start the task when tapped/clicked.

(e) Make the `TextView` display the return result (in `onPostExecute()`.

(f) Implement the downloading code.

You can mostly piece this together from the lecture notes, except for one crucial extra step: we need to tell Android to trust our self-signed certificate. Fortunately, some existing code – `DownloadUtils.addCertificate()` has been provided to take care of that. You just need to call it as follows:

```
HttpsURLConnection conn = ...
DownloadUtils.addCertificate(MainActivity.this, conn);
```

> **Note:** I'm not getting you to write the `addCertificate()` code, because it's actually just a workaround for the fact that we're using a self-signed certificate. It wouldn't be needed in a real-world context, where a certificate's authenticity can instead be established via a "chain of trust".

Also note that we specifically need "`HttpsURLConnection`" (note the "s") in order to do this. There's no trick to getting it. If you open a connection to an `https://...` URL, Java will give you an `HttpsURLConnection` object.

Also, for convenience, here is an implementation of the algorithm for copying data from an `InputStream`:

```
InputStream is = ...;
ByteArrayOutputStream baos = new ByteArrayOutputStream();

byte[] buffer = new byte[1024];
int bytesRead = is.read(buffer);
while(bytesRead > 0)
{
    baos.write(buffer, 0, bytesRead);
    bytesRead = is.read(buffer);
}
baos.close();

String result = new String(baos.toByteArray());
```

(g) Mind your exception handling! You'll get stuck in debugging hell if you don't handle exceptions sensibly, and exceptions can be thrown at almost any stage of the downloading process. You basically have to deal with:

- `IOException`, caused by connection failures in this context; and
- `GeneralSecurityException`, caused by the failure of one of the steps involved in trusting the self-signed certificate.

(Technically the actual exception types will be *subclasses* of these, but that's unlikely to matter to you here.)

So what should you do?

- Have `Log.e(...)` inside every `catch` block, and pass the exception object as a third parameter. (Note: in other cases `Log.w()` or `Log.d()` may be more appropriate, but you *do* want to log exceptions!)

- Don't let your code carry on as if nothing had happened after the `catch` block.
- Don't catch the `Exception` or `Throwable` superclasses.
- In this case (just for testing/experimentation purposes!), you could add the exception message (`e.getMessage()`) to the `TextView` rather than the server response.

  In general, you should consider displaying a more formal error message.

2. Building URLs

We'll come back to the path and query parameters now. First, stop the server (Ctrl-C), open `testserver.js`, and do two things:

- Change `IGNORE_URL` to `false`;
- Note down the `REQUIRED_PATH` and `REQUIRED_PARAMS`.

In your app, use the `Uri` class to build a more complete URL that incorporates the path and query parameters required by the server. Use the lecture notes as a guide.

Restart the server, and check whether your app can still connect to it. (And make a couple of deliberate, temporary mistakes so that you can see the server is verifying the request.)

3. Parsing JSON

We've just been displaying the raw JSON code so far, but to complete the exercise we need to parse it using `JSONObject`, and extract the parsed data.

For our purposes here, it will suffice if you can simply construct and display a more nicely/concisely-formatted version of the JSON data, such as the following:

```
Covenant: stength=100, enemy
Sirius Cybernetics Corporation, Complaints Division: strength=50, neutral
Twelve Colonies: strength=25, ally
```

4. Optional: Implement the Progress Bar

Some of the basic details are given in the lecture notes. The `ProgressBar` view has two key properties of interest: the progress (`setProgress()`), and the maximum progress (`setMax()`). In our case, these will correspond to the number of bytes downloaded so far, and content length, which can be retrieved from the `HttpsURLConnection` object (`getContentLength()`).

Note: The server is configured to send across the content length at the start of the connection. Technically, HTTP(S) servers are not absolutely required to do this, and if they don't then showing the progress can't really be done. However, `ProgressBar` can be set to "indeterminate" mode, which instead simply shows an ongoing animation.

It may also be nice to make the `ProgressBar` invisible unless a download is actually happening (`setVisibility(...)`).

5. Optional: AlliesAndEnemies

Try incorporating your downloading and parsing logic into the AlliesAndEnemies app from worksheet 4 (either the original or your modified version of it).

End of Worksheet