Mobile Application Development (COMP2008)

Lecture 6: Remote Data

Updated: 11th October, 2018

Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

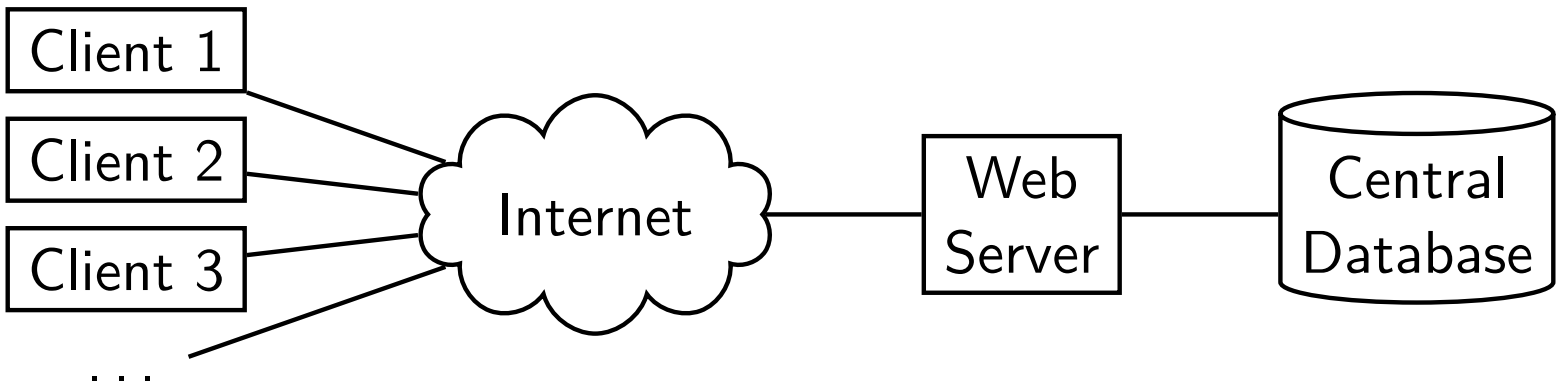
Outline

Apps and Online Services

- Many popular apps are just the "front-end" to a popular online service.
  - Social media.
  - Other media – news, music, video, photos.
  - "Productivity" – email, calendar, messaging, etc.
- The philosophy of a mobile device:
  - Even if you don't make calls, you're still *connected* to the world.
- So, how can we connect our apps to online services?
  - And *what actually are* these services, at a software level?

Online Services

- The design of server software is beyond the scope of this unit.
- *However*. . . we still need to know the general architecture, which is fairly straightforward:



- The "clients" are the mobile devices, desktop PCs, and anything else that might *use* the service.
- The web server provides the service.
  - It sends/receives data to/from the clients.
  - It controls how the data is structured, and who has access to it.
  - It stores it all in a (typically gigantic) central database.

Web Services

- There is a standard-ish way for apps to connect to online services.
  - In fact, there are several, but this is (perhaps) the most common.
- The web server will listen at a particular "endpoint" URL:

```
https://example.com/thewebservice/rest/
```

  - Sometimes there will be several URLs, each representing different functionality.
- Your app sends an HTTP "GET" request to this address.
  - Retrieves structured data, a bit like a database query.
  - Other possible operations include POST (add), PUT (update) and DELETE.
  - Entirely up to the server to decide how to handle these though.
- Note: HTTP (HyperText Transfer Protocol) underlies the whole web, of course.
  - Mostly used to download web pages ("hypertext" documents), but that's *not* what we're doing here.

Web Service Query Parameters

- We also need to pass parameters to the server.
  - A set of key-value pairs, all strings.
- What they are is entirely up to the server.
  - Any number of them. Some optional, some mandatory.
- Generally, one will be an authorisation token/key.
  - Your app must prove it "has the right" to access the server.
  - You must already have acquired this token/key beforehand, and keep it secret.
  - Extremely important to use an encrypted channel like HTTPS (HTTP over TLS/SSL).
- Often, one parameter will identify an API function to be invoked.
  - i.e. what do you actually want the server to do?
- Often, one parameter will indicate the desired *format* of the returned data.
  - i.e. how would you like your data? In XML, JSON, etc.?

Building the Full URL

The following example comes from the Big Nerd Ranch Guide (3rd ed), page 484:

```
String urlString =
    Uri.parse("https://api.flickr.com/services/rest/")
    .buildUpon()
    .appendQueryParameter("method", "flickr.photos.getRecent")
    .appendQueryParameter("api_key", API_KEY)
    .appendQueryParameter("format", "json")
    .appendQueryParameter("nojsoncallback", "1")
    .appendQueryParameter("extras", "url_s")
    .build().toString();
```

- The `Uri` class lets us build up a complete URL.
- The string ultimately looks like this (one line):

```
https://api.flickr.com/services/rest/?method=flickr.photos.get
Recent&api_key=0123456789abcdef0123456789abcdef&format=json&no
jsoncallback=1&extras=url_s
```

Building the Full URL: Discussion

- The particular example on the previous slide will obtain a list of recent photos uploaded to flickr.com.
- Technically, you could just build the full URL string manually.
  - The parameters begin after ?.
  - They're separated by &.
  - They're each represented as `name=value`.
  - Any "special characters" must be "escaped".
- But `Uri` helps us do this, because it's fiddly and prone to mistakes, which then become security holes.
- Also notice the "`api_key`" parameter.
  - It's effectively a password, and this is why we *must* use encryption (HTTPS).

Opening a Connection

- How do you actually programmatically download something?
- What we *have* is a URL string.
- What we *want* is the string returned by the server.
- First, let's open the connection:

```
String urlString = ...;
URL url = new URL(urlString); // Different from 'Uri'!
HttpURLConnection conn =
    (HttpURLConnection) url.openConnection();
try
{
    ... // Check status.
    ... // Download data.
}
finally
{                       // Whatever happens, we must end the
    conn.disconnect(); // connection when we're done.
}
```

Checking Server Response Code

- Servers and networks are unreliable. You may not always get the response you want.
- HTTP has various response codes, represented as constants in `HttpURLConnection`:
  - 200 (`HttpURLConnection.HTTP_OK`) – everything is fine.
  - 404 (`HttpURLConnection.HTTP_NOT_FOUND`) – the server couldn't find what you wanted.
  - 500 (`HttpURLConnection.HTTP_INTERNAL_ERROR`) – the server has its own internal problems.
  - Many others.
- After we connect, we must check the response:

```
if(conn.getResponseCode() != HttpURLConnection.HTTP_OK)
{
    throw ...; // Throw something. Then catch it
}              // elsewhere and show an error.
```

Downloading

- Assume the connection succeeded. Now we must actually transfer bytes from the server.
- First, we get an `InputStream`:

```
InputStream input = conn.getInputStream();
```

- Then the conventional Java code is fiddly and boilerplate-ish.
  1. Create a `ByteArrayOutputStream` – basically a byte list.
  2. Read a chunk of bytes from `InputStream`, store them in `ByteArrayOutputStream`.
  3. Loop until we're finished.
  4. Convert the contents of `ByteArrayOutputStream` to a string.
- Or we could use Apache Commons (a 3rd-party library):

```
import org.apache.commons.io.IOUtils;
...                                 // Takes InputStream,
String data = IOUtils.toString( // downloads all data,
    conn.getInputStream(),      // converts to string!
    StandardCharsets.UTF_8);
```

Android INTERNET Permission

- In Android, we need to ask for permission to download things.
- So, back to app/src/main/AndroidManifest.xml:

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

Freezing the GUI (or Not!)

- Most of the Android code you've written so far is very fast.
  - We haven't been running code that actually takes noticeable time.
- Downloading breaks this pattern – it can be slow.
  - Especially on mobile devices with unreliable 3G/4G/5G/wifi access.
- Often slow *enough* that the GUI freezes!
  - Why? Because the GUI has exactly one thread.
  - If it gets stuck running a slow download, it can't respond to anything else until that finishes.
- The solution is to run slow tasks (e.g. downloads) in *another* thread.
  - We'll only consider the *simplest* way of doing this!
  - (Software Engineering Concepts will pick up on more general approaches to multithreading.)

AsyncTask

Android introduces the AsyncTask class:

```
public class Xyz
{
    private class MyTask extends AsyncTask<Void,Void,Void>
    {
        @Override
        protected Void doInBackground(Void... params)
        {
            ... // Do complicated (non-GUI) things here!
            return null;
        }
    }
    ...
    // Elsewhere (in a callback?)
    new MyTask().execute(); // Runs asynchronously
}
```

Literally "...”!

[1] https://developer.android.com/reference/android/os/AsyncTask

What is "<Void,Void,Void>"?!

- This is actually fairly unimportant, but may be confusing.
- Basically, it means we *aren't* using AsyncTask's more sophisticated features.
  - If we were, we might instead have:

```
private class X extends AsyncTask<URL,Integer,Long>
```

  - See the the AsyncTask documentation for an example.
- "Void" (with a capital-V) is a bit of a hack.
  - There are situations (like this one) where we'd like to say "void" (small-v) but can't, because the Java language requires a class name.
  - So, the Java API defines the `Void` class. It has no instances, does *absolutely nothing*, and serves only as a placeholder.
  - A `Void` parameter/return must technically have a value, but the only allowable value is `null`.

AsyncTask Results

- AsyncTask has a weird-looking way of dealing with its result:

```
private class MyTask extends AsyncTask<Void,Void,MyResult>
{
    @Override
    protected MyResult doInBackground(Void... params)
    {
        ... // Complicated non-GUI stuff.
        return new MyResult(...);
    }

    @Override
    protected void onPostExecute(MyResult result)
    {
        ... // Show result in GUI. 'result' comes from
    }       // doInBackground().
}
```

Literally "..."

AsyncTask Results – Discussion

- `doInBackground()` performs the downloading (or other long-running task).
  - And returns a result – whatever object you're going to produce.
- `onPostExecute()` takes in that same result, and displays it.
  - It should probably call some other code in the fragment, recycler-view, etc.
  - GUI things aren't really `AsyncTask`'s responsibility.
- Seem unnecessary? Why not just merge the two?
  - Because we're *forbidden* from accessing the GUI directly from another thread.
  - `doInBackground()` runs in another thread.
  - `onPostExecute()` runs in the GUI's own thread.
  - Practically all GUIs are single-threaded, and not thread-safe.
  - Behind the scenes, `AsyncTask` performs some complicated thread communication to pass the result.

AsyncTask Parameters

- While we're looking at AsyncTask's "type parameters"...
- The first can be used to provide parameters to doInBackground():

```
private class MyTask extends AsyncTask<URL,Void,Void>
{
    @Override
    protected Void doInBackground(URL... params)
    {
        for(URL url : params) {...} // For each URL
        return null;
    }
}
```

Literally "..."!

- Then we can pass parameters when executing the task:

```
new MyTask().execute(url1, url2, url3);
```

- The "**URL...**" syntax is just **URL[]** with less clutter.
  - Called a "vararg" parameter.

AsyncTask Progress

- Your download may take a noticeable amount of time.
- So, it would also be show the progress visually.
- Say our code calculates its progress like this:

```
private class MyTask extends AsyncTask<Void,Void,Void>
{
    private int totalBytes;
    @Override
    protected Void doInBackground(Void... params)
    {
        totalBytes = ...; // Get download size
        int nBytes = 0;
        while(...) { ...; nBytes += n; } // Download
        return null;
    }
}
```

Literally "...”!

- nBytes represents the progress. How to show it?

AsyncTask Progress

- Define the progress datatype:

```
private class MyTask extends AsyncTask<Void,Integer,Void>
```

- While downloading, "publish" the progress periodically:

```
while(...) // Download
{
    ...;
    nBytes += n;
    publishProgress(nBytes);
}
```

- Override onPublishProgress() to update the GUI:

```
@Override
public void onPublishProgress(Integer... params)
{
    showProgress(params[0], totalBytes); // Your method
}
```

Server Response Format

- When the downloading finishes, we have a string.
- But what we *want* is almost certainly structured data.
  - Remember: this is a bit like a database query.
- Web services typically provide data in one (or more) of several formats:

  XML – you know what this looks like already!
  JSON – "JavaScript Object Notation", pronounced "Jason".
  Others – We won't worry too much about them.
- We'll focus on JSON.
  - Inspired by JavaScript's syntax for representing "objects".
  - "Objects" in JavaScript/JSON are just collections of key-value pairs, conceptually like "`Map<String,...>`" in Java.
  - Values can be numbers, strings, booleans, arrays, other objects and null.

JSON Syntax

- Remember that JSON is just a way of representing structured data as text.
- An object has:
  - Braces {} around it.
  - Zero or more key-value pairs, of the form *"key": value*.
  - Commas separating the pairs.
- A simple object might look like this:

```
{
    "item":     "burger",
    "quantity": 6,
    "cost":     65.4
}
```

- Whitespace and line breaks are irrelevant.
  - (Useful in examples, but in practice humans don't usually want to read JSON.)

JSON Syntax (2)

- Arrays in JSON are comma-separated lists of values enclosed in square brackets []:

```
["burger", "fries", "nuggets", "salad", "ice cream"]
```

- Objects and arrays can be nested arbitrarily:

```
{
    "postcodes": [ 9123, 9933, 9445 ],
    "items":
    [
        { "item": "burger",  "cost": 10.9 },
        { "item": "fries",   "cost": 3.95 },
        { "item": "nuggets", "cost", 6.50 }
    ],
    "delicious": true
}
```

JSON Parsing

- It's good to know what JSON *is*...
- ...but you don't need to write your own parsing code.
- Android comes with the `JSONObject` and `JSONArray` classes:

```
String theDownload = ...;
JSONObject jBase = new JSONObject(theDownload); // Parse

JSONArray jPostCodes = jBase.getJSONArray("postCodes");
JSONArray jItemList  = jBase.getJSONArray("items");
boolean delicions    = jBase.getBoolean("delicious");

for(int i = 0; i < jPostCodes.length(); i++)
{
    int postCode = jPostCodes.getInt(i);
    ...
}
```

Parsing Errors

- Even if you get `HTTP_OK`, *other* things can go wrong!
- The JSON constructors and methods throw `JSONException` if the data isn't in "right" format.
- *Could* indicate a bug in your code.
  - If you're calling `jBase.getInt("delicious")` when it should be `jBase.getBoolean("delicious")`...well, that's your fault!
- Could *also* happen if there's a bug at the server's end.

```
String theDownload = ...;
try
{
    JSONObject jBase = new JSONObject(theDownload); // Parse
    JSONArray jPostCodes = jBase.getJSONArray("postCodes");
    ...
}
catch(JSONException e) {...}
```

What to Do With the JSON Data?

- We want to build up our own "proper" objects to represent the JSON information.
- e.g. We might have this class:

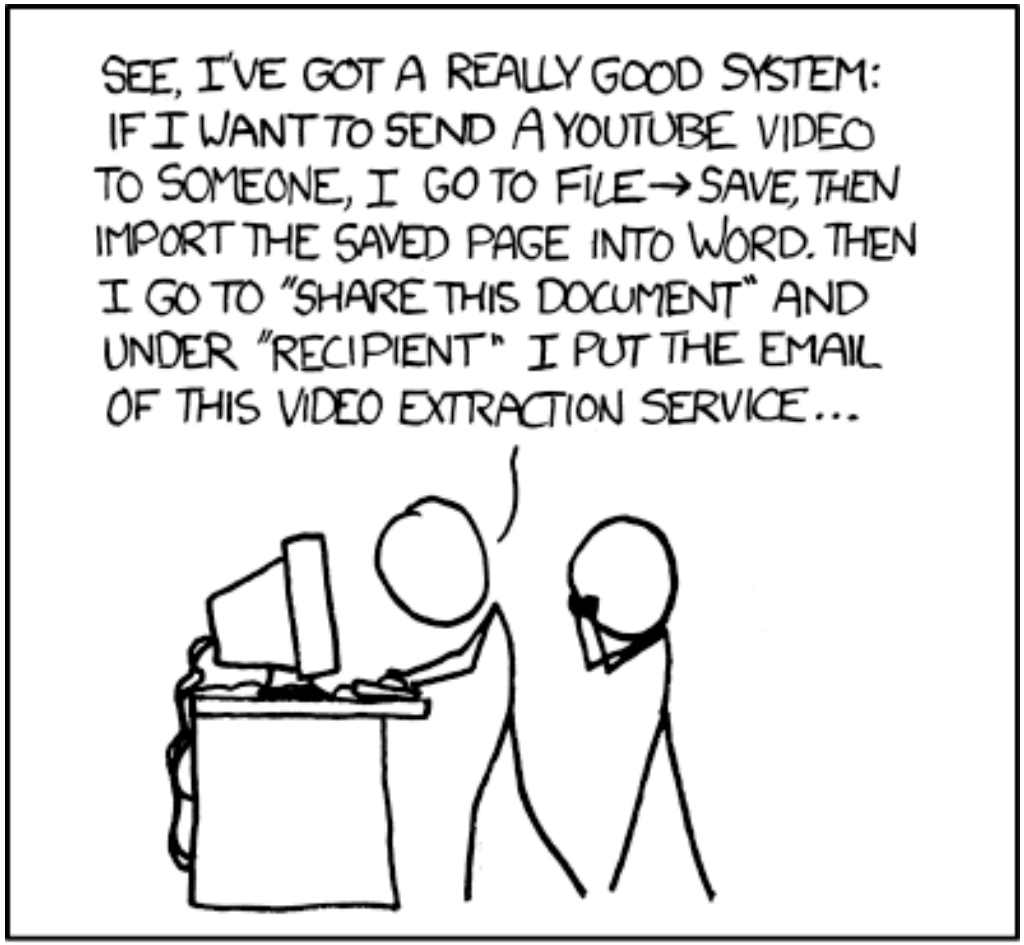
```
public class MenuItem {...} // Contains 'item' and 'cost'
```

- And so we want to make a list of them:

```
JSONArray jList = jBase.getJSONArray("items");
List<MenuItem> items = new ArrayList<>();

for(int i = 0; i < jList.length; i++)
{
    JSONObject jItem = jList.getJSONObject(i);
    items.add(new MenuItem(jItem.getString("item"),
                           jItem.getDouble("cost")));
}
```

- And then we (might) get RecyclerView to display the list!



(https://xkcd.com/763/)