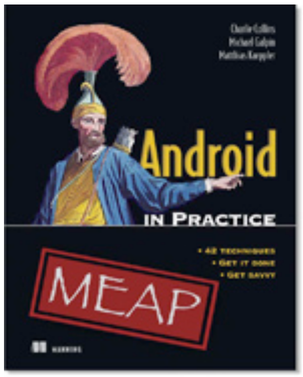
MANNING PUBLICATIONS

Technique: HTTP the Java way

An article from



Android in Practice

EARLY ACCESS EDITION

Charlie Collins, Michael D. Galpin, and Matthias Kaeppler
MEAP Release: July 2010
Softbound print: Spring 2011 | 500 pages
ISBN: 9781935182924

The standard Java class library already has a solution for HTTP messaging. An open-source implementation of these classes is bundled with Android's class library, which is based on Apache Harmony. It's simple and bare-bones in its structure and, while it supports features like proxy servers, cookies (to some degree), and SSL, the one thing that it lacks more than anything else is a class interface and component structure that doesn't leave you bathed in tears. Still, more elaborate HTTP solutions are often wrappers around the standard Java interfaces and, if you don't need all the abstraction provided, for example, by Apache HttpClient interfaces, the stock Java classes may not only be sufficient, they also perform much better thanks to a much slimmer, more low-level implementation.

Problem

You must perform simple networking tasks via HTTP (such as downloading a file) and you want to avoid the performance penalty imposed by the higher-level, much larger, and more complex Apache HttpClient implementation.

Solution

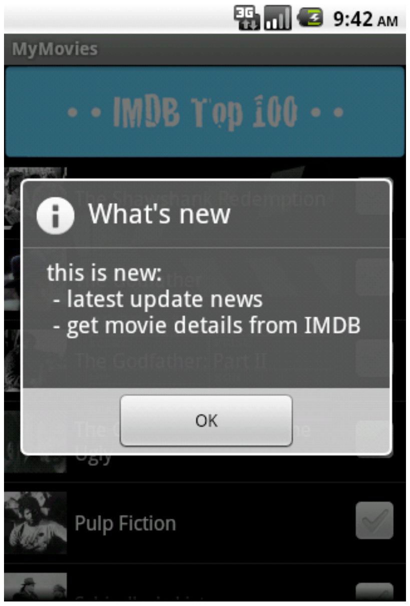
If you ever find yourself in this situation, you probably want to do HTTP conversations through a `java.net.HttpURLConnection`. `HttpURLConnection` is a subtype of the more generic `URLConnection`, which represents a general purpose data connection to a network endpoint specified by a `java.net.URL`. A

URLConnection is never instantiated directly; instead, you construct a URL instance from a string and, based on the URL's scheme, a proper implementation of URLConnection is returned:

```
URL url = new URL("http://www.example.com/");
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
conn.connect();
…
conn.disconnect();
```

This connection implementation lookup by URL scheme is performed by a protocol handler object. The Java class library and Android already provide protocol handlers for all common schemes such as HTTP(S), FTP, MAILTO, FILE, and so on, so typically you don't have to worry about that. This, of course, also means that you are free to create your own protocol handlers that instantiate your own custom URLConnection. URLConnection is based on TCP sockets and the standard java.io stream classes. That means I/O is blocking, so remember to never run them on the main UI thread.

Let's see how it works in a practical example. We want to extend the MyMovies application to display a simple message dialog with the latest update news downloaded from a Web server so the user is always up to date about what has changed in the latest release. For this to work, we just have to place a text file containing the update notes somewhere on a Web server, download and read the file, and display its text in a message dialog. Figure 1 shows what that will look like.



this is new:
 - latest update news
 - get movie details from IMDB

Figure 1 On every application start, we show a message dialog to the user with the latest update notes. The text in the dialog is fetched from a Web server instead of being bundled with the APK.

For simplicity, we will show the dialog on every application start, a "detail" that would probably annoy the heck out of your users if this were a production release, but the example serves our purpose well enough. Implementation wise, the plan is to write an AsyncTask that establishes a connection to an HTTP server via HttpURLConnection and downloads the file containing the update notes text. We then send this text via a Handler object to our main activity so we can show an AlertDialog with that text. Let's look at the MyMovies activity class first, which contains the callback for the handler to show the pop-up dialog (listing 1).

Listing 1 Showing an update notes pop-up dialog in MyMovies

```
public class MyMovies extends ListActivity implements Callback {

   private MovieAdapter adapter;


   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

      setContentView(R.layout.main);

      ...

      new UpdateNoticeTask(new Handler(this)).execute();                  #A
   }

   ...

   public boolean handleMessage(Message msg) {
      String updateNotice = msg.getData().getString("text");              #B
      AlertDialog.Builder dialog = new AlertDialog.Builder(this);
      dialog.setTitle("What's new");
      dialog.setMessage(updateNotice);                                    #C
      dialog.setIcon(android.R.drawable.ic_dialog_info);
      dialog.setPositiveButton(getString(android.R.string.ok),
            new OnClickListener() {
               public void onClick(DialogInterface dialog, int which) {
                  dialog.dismiss();
               }
            });
      dialog.show();
      return false;
   }
}
```

#A Starts a new task to download the file
#B Reads update text from the message
#C Sets update text on the dialog

We launch the `UpdateNoticeTask` in the last line of `onCreate()` since that's where the download proceeds. Listing 2 has the source code.

Listing 2 Downloading a text file using HttpURLConnection and AsyncTask

```
public class UpdateNoticeTask extends AsyncTask<Void, Void, String> {

   private String updateUrl =
      "http://android-in-practice.googlecode.com/files/update_notice.txt";

   private HttpURLConnection connection;

   private Handler handler;

   public UpdateNoticeTask(Handler handler) {
      this.handler = handler;
   }

   @Override
   protected String doInBackground(Void... params) {
      try {
         URL url = new URL(updateUrl);
         connection = (HttpURLConnection) url.openConnection();           #1
         connection.setRequestMethod("GET");                              #2
         connection.setRequestProperty("Accept", "text/plain");           #2
         connection.connect();                                            #3
         int statusCode = connection.getResponseCode();
         if (statusCode != HttpURLConnection.HTTP_OK) {                    #4
            return "Error: Failed getting update notes";
         }
         String text = readTextFromServer();                               #5
         connection.disconnect();                                          #6
         return text;
      } catch (Exception e) {
         return "Error: " + e.getMessage();
      }
   }

   private String readTextFromServer() throws IOException {
      InputStreamReader isr =
               new InputStreamReader(connection.getInputStream());
      BufferedReader br = new BufferedReader(isr);

      StringBuilder sb = new StringBuilder();
      String line = br.readLine();
      while (line != null) {
         sb.append(line + "\n");
         line = br.readLine();
      }
      return sb.toString();
   }

   @Override
   protected void onPostExecute(String updateNotice) {                     #7
      Message message = new Message();
      Bundle data = new Bundle();
      data.putString("text", updateNotice);
      message.setData(data);
      handler.sendMessage(message);
   }
}
```

#1 Get an instance of HttpURLConnection
#2 Configure the HTTP request
#3 Establish the connection
#4 Handle non-200 server reply
#5 Read text from response body
#6 Close the connection
#7 Pass retrieved text to the activity

After reading the URL from the parameters, the first thing we have to do is use that `URL` object to retrieve an instance of a fitting `URLConnection` instance (#1)—an `HttpURLConnection` in this case, since our URL has the `http://` scheme). Note that the call to `openConnection()` does not yet establish a connection to the server, it merely instantiates a connection object. We then configure our HTTP request (#2): we first tell it that it should use the `GET` method to request the file (we could have also omitted this call, since GET is the default) and also set an HTTP Accept header to tell the server what kind of document type we expect it to return—plain text in this case. The request is now configured and can be sent to the server by a call to `connect()` (#3). Depending on the server reply, we either return an error message if we received a status message that was not 200/OK (#4) or proceed to read the text from the response body (#5). Don't forget to close the connection when you are done processing the response (#6). Finally, we send the text we received from the server to our main activity using the `Handler` (#7).

Discussion

The example we showed here was extremely simple—the simplest kind of request you can send, really. For these scenarios, `HttpURLConnection` does the job quite well. The biggest problem with it is its class architecture. `HttpURLConnection` shares a large part of its interface with the general purpose `URLConnection` (obviously, since it inherits from it), which means that some abstraction is required for method names. If you've never used `HttpURLConnection` before, you have probably pondered the call to `setRequestProperty()`, which is the way to set HTTP headers—not very intuitive. This is simply because implementations for other protocols may not even have the concept of header fields but would still share the same interface, so the methods in this class all have rather generic names.

While this is just a cosmetic thing, the actual problem with `URLConnection` is its entire lack of a proper separation of concerns. The request, the response, and the mechanisms to send and receive them are merged into a single class, often leaving you wondering which methods to use to process which part of this triplet. This is a bit like creating a five-course meal just to stick it in the blender: you can still serve it, but it's just disgusting. It also makes each part difficult to customize and even more difficult to mock out when writing unit tests. It's simply not a beaming example of good object-oriented class design.

There are also some more practical problems with this class. If you find yourself in a situation where you need to intercept requests to preprocess and modify them (a good example is message signing in secure communication environments, where the sender needs to compute a signature over a request's properties and then modify the request to include the signature), then `HttpURLConnection` is not a good choice for sending HTTP requests. That's because request payload is sent unbuffered, so there is no way to get your hands on it in a non-intrusive way. Last but not least, `HttpURLConnection` in Apache Harmony has bugs—serious bugs. One of the major ones is detailed in the sidebar.

Android, HttpURLConnection, and HTTP header fields

As you already know, the Java class library bundled with Android is based on Apache Harmony, the open-source Java implementation driven by the Apache foundation. At the time of this writing, there is a serious bug that affects HTTP messaging using HttpURLConnection: it sends HTTP header field names in lowercase. This is nonconformant to the HTTP specification and, in fact, breaks many HTTP servers since they will simply drop these header fields. This can have a wide array of effects, with documents served to you in the format that doesn't match your request (for example, the Accept header field was ignored) and failed requests to protect resources due to the server's failure to recognize the Authorization header field. A workaround is simply to not use HttpURLConnection until these problems have been fixed and use Apache HttpClient instead. If you want to follow the progress of fixing this bug, you can find the official issue report at this Web address: http://code.google.com/p/android/issues/detail?id=6684.

Summary

Overall, `HttpURLConnection` is a simply structured but rather low-level way of doing HTTP messaging, and its clumsy interface, lack of object-orientation, and proper abstractions make it difficult to use. For simple tasks like the file download shown here it's absolutely fine and comes with the least overhead (it doesn't take a sledgehammer to crack a nut) but, if you want to do more complex things like request interception, connection pooling, or multipart file uploads, then don't bother with it—there's a much better way to do this in the Java world and, thanks to the engineers at Google, it's bundled with Android!