

# DEVELOP YOUR FIRST ANDROID APPLICATION

**GRAHAM MORRISON**

Google's Android Studio has made Android development more straightforward than ever. As we prove. Kind of...

**WHY DO THIS?**

- Develop for a hugely successful platform
- Get into the Android app store and make £\$€
- Never miss another Linux Voice podcast

Android's success is colossal. It's the most widely distributed version of a Linux-based operating system, used on everything from smartphones and tablets to cars, fridges and air conditioning units. It's only a matter of time before it reaches your toaster. And while it may have started life outside of Google as an advanced operating system for digital cameras, Google's purchase of Android, Inc. in 2005 slotted Android into Google's plans for mobile world domination, inadvertently making it the operating system for the touchscreen generation – the opposite of Apple's iOS, with all its walled garden and hardware consistency. Android is wild, chaotic, uncountable and outgrowing itself. But it's also open source and still relatively open. You don't need to pay Google to write software for it, and you don't need to root your device to run your applications. Despite an unwieldy API and a huge boilerplate of

requirements for running your own applications, development isn't even that difficult – especially if you just want to hack your own solutions into the chaos. Which is exactly what we're going to do here, in what will be the first part of a series on creating a Linux Voice application for Android.

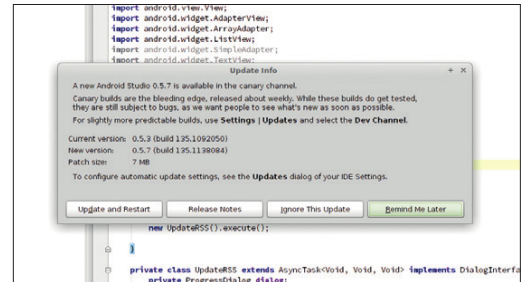
We're going to start modestly. We'll build a very simple RSS reader app that will parse the latest posts from **LinuxVoice.com** and enable you to select a story to open in a browser. As a starting point, we may then expand upon this simple application to create a much more functional application. But it's also a perfect way to acclimatise yourself to Android development, and from there, put your own ideas into code. And because we're embracing all things cutting-edge and chaotic in this tutorial, we're going to use this as an excuse to try Google's own Android development environment – Android Studio.

## 1 INSTALLATION

Android Studio exists solely for Android applications. It ties itself seamlessly with the Android SDK and the emulator tools used to test your code, and also offers cutting-edge shortcuts for writing. There's WYSIWYG realtime app rendering, a UI designer, Android templates and an editor that spots mistakes as they happen. But its cutting edge nature also means it's unstable, and Google doesn't yet recommend it for use in mission-critical development. That's not going to affect us as we take our preliminary steps in Android development, but it's something to consider when your app development ambitions take you further than a simple RSS reader.

Android Studio is based upon another IDE, IntelliJ IDEA, and as you might have inferred by the mention of Eclipse and IntelliJ IDEA – two IDEs written in Java, Android is all about the Java. Java is something of a contentious language even now, but there's no denying its ubiquity. More importantly, as long as you've some previous experience in an object-oriented environment, Java is legible and easy to understand and get along with. It's a good choice for a hacker-friendly tutorial.

We installed Android Studio on both Arch Linux and Linux Mint/Ubuntu. Arch packages are held in the user repository, while manual installation is also relatively straightforward: just download and execute a binary.



It's worth updating Android Studio frequently, as each update makes the environment more stable.

However, we find installation on Mint an easier proposition as it automatically deals with dependencies and the environmental variables required by Android Studio to find the SDK (which also needs to be installed manually). To install from Ubuntu or Mint, just type the following:

```
sudo apt-add-repository ppa:paolorotolo/android-studio
```

```
sudo apt-get update
```

```
sudo apt-get install android-studio
```

This should grab over 650MB of data, as the download includes the SDK alongside the development environment and any dependencies you've yet to install. But it will leave you with a full development environment capable of building

projects that can run on your desktop through an emulator and deployed to your own Android devices. Android Studio can be started by typing **android-studio** on the command line, or through an icon embedded within your desktop's launch menu, and the first thing we'd recommend you do after the splash screen has gone is to ignore the 'Welcome to Android Studio' window, and instead open the small 'Check For Updates' link in the windows bottom

border. After checking against the server, the application will almost certainly announce that there's an update that can be installed, and unlike other applications installed through a distribution, in-place updates to Android Studio work. Just let the application run and update itself. Android Studio is in a rapid state of development, so it's a good idea to keep on top of updates. There are usually one or two per month, and recent updates have been very stable.

## 2 DEVICE CONSIDERATIONS

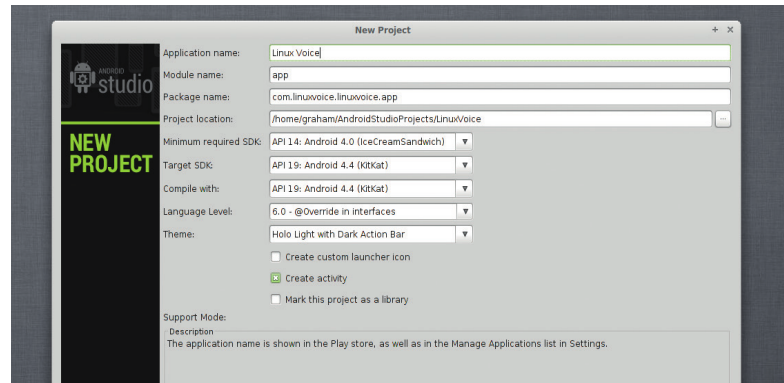
From the 'Welcome to Android Studio' window, click on 'New Project' to start our adventure. We've now got to change a few values in the next window that appears. The app needs a name, and we called ours simply 'Linux Voice'. We'll also need a package name, which needs to adhere to Java's package naming convention. Recent versions of Android Studio will automatically create this from your application name, but if you need to change this to something that closer matches your own configuration, make sure it uses the reverse hierarchical naming pattern, normally (but not always) linked to the top level domain of your organisation plus the name of your app.

The next three fields ask you which version of the SDK you want to build your application against. Android has changed a lot over the years, and many of its more advanced features are only available in the 4.x SDK. But while choosing the latest SDK is the easiest option, the best answer is going to depend on what capabilities you want your application to have and where you want it to be used. Unlike iOS, many Android users can't simply update their devices to the latest releases. Many manufacturers don't even provide updates, and even when they do offer updates, they've usually made so many modifications to Android themselves that it takes many months for an update to be distributed. The result is that there are many different versions of Android in active use.

### Choose your platform

Google uses statistics gathered from its Google Play app to report on the devices accessing the store over the previous seven days, so that developers can make their own judgment on which platforms to target. We've printed the table from early April, and you can see that while a significant percentage of devices are

Current Android versions			
Version	Codename	API	Distribution
2.2	Froyo	8	1.1%
2.3.3 - 2.3.7	Gingerbread	10	17.8%
3.2	Honeycomb	13	0.1%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	14.3%
4.1.x	Jelly Bean	16	34.4%
4.2.x		17	18.1%
4.3		18	8.9%
4.4	KitKat	19	5.3%



using a 4.x version of Android, there's still almost a fifth of all Android devices running Gingerbread. We decided with setting a minimum required SDK of 14 (Ice Cream Sandwich), while setting the target and compile SDK at the very latest version, 19 (Kit Kat).

You can leave the other options at their default values – just make sure that 'Create Activity' is enabled, as this adds the next page to the New Project wizard, which enables you to select which kind of template to use for your project. These are worth exploring, as each example will include the best practice for different elements of a typical Android app, and each new version of Android Studio includes more. But for our first application, we're going to keep things simple and go with the 'Blank Activity'. This lets us build things up from scratch but also avoids too many complex concepts that come with more complex templates. After selecting the blank template, the final page is where we give our activity a name.

An activity is what Android calls the application component that expects something of the user. For nearly every instance, that means putting something on the screen you can interact with. Android likes to break down the functional components within an application, and an activity is one such component, usually with one purpose. It could be a web browser, or a music player, but equally, it could be a stream of RSS stories, which is the only activity our application is going to provide. An application is then a combination of multiple activities tied together to create a fully functional tool. Having said that, you can leave the Activity and Layout names at their default values, as they make only a cosmetic difference.

The Android Studio New Project wizard creates sensible defaults for nearly all values.

### PRO TIP

When using 'try' and 'catch', debugging is much easier if you isolate each exception rather than grouping them together (as we have).

### 3 ANDROID EMULATION

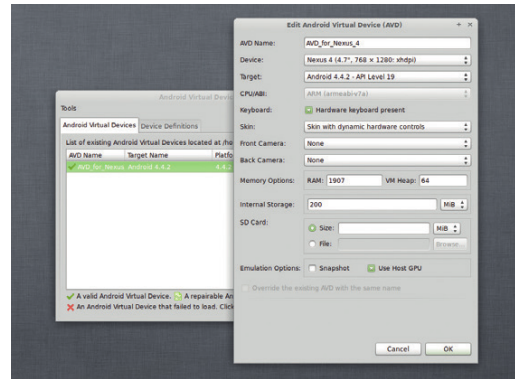
This is where we hit some cutting-edge turbulence. With Android Studio 0.5.7, the template generated an error before we'd even got to the code. The error console complains that the Android Support Repository isn't installed. And while this was indeed installed, any update required another update to push the Support Repository version up from 4 to 5. To fix this, we needed to open one of several support tools used to augment the Android development environment – the Android SDK Manager. This can be opened from the Tools > Android menu, and when opened it lists the various SDKs and tools installed alongside Android Studio. We needed to allow this to update two packages, which it did automatically after accepting a couple of licences from Google. Hopefully, you won't have to go through a similar process when you first start Android Studio.

Before we spend a little time going through Studio's GUI components, first click on the Play button in the toolbar at the top of the window. This will build and attempt to launch the template, which is itself a fully executable application. Most of the few seconds it takes is spent launching the Android Debug Bridge, ADB, which is a tool that grabs runtime and debugging information from a runtime instance of your application, whether it's running on an emulator or on a real device.

**PRO TIP**  
 Android Studio's editor will grey-out lines, such as imports, if they're redundant, and even make suggestions for spelling corrections.

#### Android emulator

The next window that appears lets you select which device you want to run your application on. The Android Emulator is one of the most useful tools for both Eclipse and Android Studio development, as it enables you to run your code on a fully fledged Android device, albeit one being emulated in software. This does affect performance, but it also means you can spin up any kind of device you want, from old models using an ancient version of the SDK, to models with resolutions and hardware combinations that don't get exist, and you don't have to spend any money getting the latest devices. From the 'Choose Device' window, make sure 'Launch Emulator' is



You can run almost any real and functional Android device from the Android Virtual Device manager.

selected and click on the 'Expand' button to the right of the drop-down list. This will open AVD – the Android Virtual Device manager.

This is the tool that manages your real and fictitious virtual Android devices, and to get started you need to click on the 'Device Definitions' tab. This page lists common definitions for a device, and clicking on 'AVD' will create an instance for you to run. If you want to edit any of these settings, you'll need to clone a definition first. This will allow you to adjust things such as the native resolution of a device, or whether both portrait and landscape modes are available. After you've added a device to create an AVD, you can edit runtime options such as the amount of internal story available to the emulation, whether an instance is persistent (storing the data between sessions), or whether graphics calls use your real machine's GPU, as well as the target level of the SDK.

If you've got the correct packages installed, you can also switch between the much slower ARM CPU emulation to the x86 instruction set of your native system. This gives a big performance boost and won't cause any compatibility issues unless you're performing more complex or low-level operations. We opted for a clone of the Nexus 4 profile with no other modifications.

### 4 TESTING YOUR ENVIRONMENT

With a device added to the device list, you just need to click on Launch to run the instance. However... stop! On most displays, a running Android instance takes up nearly all your screen space, because of the high-DPI values on most Android devices. A Nexus 5, for example, has a screen resolution of 1920x1080 pixels, which is more pixels that many screens are capable of displaying. We've not found a way of scaling the emulator from the launcher, but you can scale the emulator from the command line. For that reason, we'd recommend running it from there. If the

Android tools have been added to your path, it can be run with the **emulator** command; if not, you'll need to find it yourself. Arch installs the SDK into the **/opt** folder, while the PPA packages for Ubuntu and Mint install themselves into **/usr/share/android-studio/data/sdk/**. And because your own copies of the AVD profiles are copied to your home folder, you can reference them directly from the **emulator** command. To run the emulator at 50% resolution, which we've found perfectly acceptable, type the following, replacing the name of the virtual device with the AVD

name listed in the AVD manager:

**emulator -scale 0.5 @AVD\_for\_Nexus\_4**

The emulator window will appear at the proper scale and proceed to load Android. This can take a minute or two if you're emulating an ARM CPU, but speed obviously depends on your system. Before too long, you should have a working Android system that you can unlock by sliding the padlock to the right with a click of the mouse.

You should also be able to go back to the 'Choose Device' window from Android Studio and see the running instance listed beneath the 'Choose A Running Device' button, which should be enabled. To run your application, select the running device from the list and click on OK. We'd also suggest you

enable the 'Use Same Device For Future Launches' so that you don't have to go through this step again - although this only works for the current session.

A few moments later, your virtual Android device should burst into life and your application will run automatically. This is the default testing environment, and if you take a look back in Android Studio, you should see that it's in constant contact with the emulator. When you start debugging or monitoring your application, you'll be able to break execution and watch variables across the Android Debug Bridge exactly as you could if the application was running natively on your desktop. If everything is working correctly, you should see the ubiquitous "Hello world" staring back at you, and it's now time to start coding.

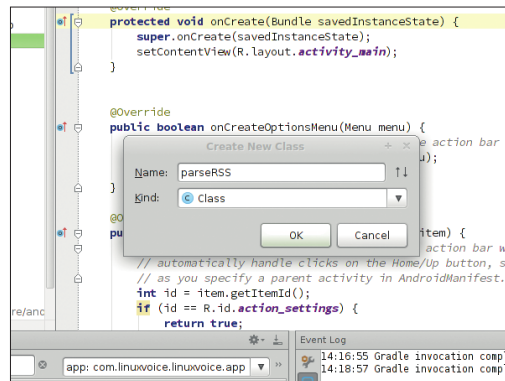
## 5 LET'S START CODING!

Before we start with the code specifics, we need to allow our application to access the internet (on the device you eventually run it on). Think of this being the Android equivalent to SELinux, and all such control is governed by a single file found in the root directory of your project – **AndroidManifest.xml**. This file controls access and permissions as well as what can and can't be received by the various activities that make up your application. You can edit it in Android Studio from the project hierarchy view on the left, by clicking on the project name, 'app' then 'src'. You need to add the following before the '<application' tag, as this ensures accesses to the entire package:

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

With that line edited, we can now start with our own code. The first thing we want is a method to grab and parse the RSS provided by our site, and as this is such a common requirement, many people have already solved this problem in lots of different ways. The most straightforward we found was written by Tony Owen (<http://droidapp.co.uk>) and used in several open source projects, including the now rather old rpdroid podcasting client. Tony offloads RSS handling into its own Java class, and we'll do a similar thing. To create a new class, follow the path hierarchy in Android Studio down through src, main, Java and your package name. You should already have a single activity called 'MainActivity', unless you changed this in the startup wizard. MainActivity is where all the action is happening, and where you can quickly augment the "Hello World" app with your own features.

To create a new class (which will be contained within its own 'Java' file), right click on MainActivity and select 'New > Java Class'. In the small window that appears, enter a new class name – we used 'dataRSS', but as long as you're consistent within the application, it won't matter which name you choose. You'll then see the new class listed beneath MainActivity, with the text editor waiting for your



Create a new class within Studio by right-clicking on the main activity source file.

inspiration. This class is going to be very simple, with no methods to do calculations and only four variables – three to hold the string parts of the RSS feed we want to pass back to MainActivity, and a single integer to keep the list location of the eventual post when it appears in our application. The only peculiarity, if you're not used to Java, is that our strings are arrays that we don't have to give a size to until we initialise them later:

```
public class dataRSS {
    public static String[] postTitle;
    public static String[] postURL;
    public static String[] postContent;
    public static int position;
}
```

We're going to follow this with the only other class we're going to write, a class we've called parseRSS, which should be created the same way we created dataRSS. This will contain the method that will download and parse the bits we want from the RSS file, so it's going to be slightly more complex. Let's go through the code, starting with the first few lines:

```
public class parseRSS {
    public static void parse() {
        URL url;
```

```
try {
```

All we're doing is creating the class and adding a single method, which is the function we're going to call to process the RSS. You should notice that as you add 'URL', it gets highlighted in red. This is because the Java compiler can't find any other reference. Hold the cursor over the URL characters and the editor will complain it can't resolve the symbol. The editor in Android Studio is smart and can make lots of intelligent calls when spotting and fixing errors. If you now click the 'URL' text, the tooltip should say 'Java.

net.URL? Alt + Enter'. The editor is saying that Java.net.URL satisfies the resource and can be imported by pressing Alt + Enter. If you do this, a popup menu will open and the first option will allow you to import the class, and selecting that will add 'import java.net.URL;' to the top of the file and solve the error. For this reason, we haven't included any of the 'import' statements for our code, as you should be able to add them yourself (and it saves space). If you have problems working them out, download the project source code from [LinuxVoice.com](http://LinuxVoice.com).

## 6 MAKING A CONNECTION

The last line in the previous snippet is **try**, and it's one of Java's most useful functions. It's an exception handler that forces us to deal with the consequences of something not happening rather than allowing the app to crash. The situations dealt with by **try** will be dealt with by a **catch** instruction later on in the code.

In the next two lines, we'll give **url** the address of our feed and ask Java to open a connection:

```
url = new URL("http://www.linuxvoice.com/feed");
```

```
URLConnection conn = (URLConnection) url.  
openConnection();
```

If the connection can be made successfully, our application will get past the following 'if' statement, taking us into the section of code that deals with the nitty gritty of the XML within the RSS feed. There are many ways to deal with RSS/XML, but we've found the easiest to be DocumentBuilder, which provides an API to access the tree layout of an XML file. In the following code you'll see that we set up a few variables before structures for getting to the data within the data stream before **db.parse(url.  
openStream())** passes control of the data to the variable we call **doc**. We'll also initialise the arrays we created in the **dataRSS** class using the size of the list gleaned from the RSS and now held in **itemLst**:

```
if (conn.getResponseCode() == HttpURLConnection.HTTP_OK) {  
    DocumentBuilderFactory dbf = DocumentBuilderFactory.  
newInstance();  
    DocumentBuilder db = dbf.newDocumentBuilder();  
    Document doc;  
    doc = db.parse(url.openStream());  
    doc.getDocumentElement().normalize();  
    NodeList itemLst = doc.getElementsByTagName("item");  
    dataRSS.postTitle = new String[itemLst.getLength()];
```

```
    DocumentBuilderFactory dbf = DocumentBuilderFactory.  
newInstance();  
    DocumentBuilder db = dbf.newDocumentBuilder();  
    Document doc;  
    doc = db.parse(url.openStream());  
    doc.getDocumentElement().normalize();  
    NodeList itemLst = doc.getElementsByTagName("item");  
    dataRSS.postTitle = new String[itemLst.getLength()];
```

```
    DocumentBuilder dbf = DocumentBuilderFactory.  
newInstance();  
    DocumentBuilder db = dbf.newDocumentBuilder();  
    Document doc;  
    doc = db.parse(url.openStream());  
    doc.getDocumentElement().normalize();  
    NodeList itemLst = doc.getElementsByTagName("item");  
    dataRSS.postTitle = new String[itemLst.getLength()];
```

```
    DocumentBuilder dbf = DocumentBuilderFactory.  
newInstance();  
    DocumentBuilder db = dbf.newDocumentBuilder();  
    Document doc;  
    doc = db.parse(url.openStream());  
    doc.getDocumentElement().normalize();  
    NodeList itemLst = doc.getElementsByTagName("item");  
    dataRSS.postTitle = new String[itemLst.getLength()];
```

```
    DocumentBuilder dbf = DocumentBuilderFactory.  
newInstance();  
    DocumentBuilder db = dbf.newDocumentBuilder();  
    Document doc;  
    doc = db.parse(url.openStream());  
    doc.getDocumentElement().normalize();  
    NodeList itemLst = doc.getElementsByTagName("item");  
    dataRSS.postTitle = new String[itemLst.getLength()];
```

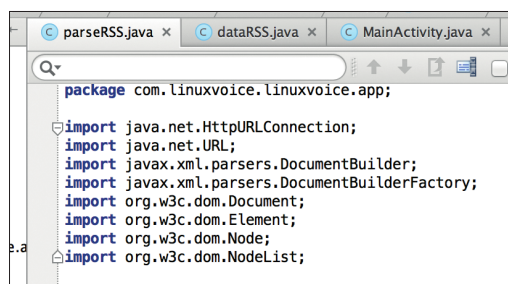
```
    DocumentBuilder dbf = DocumentBuilderFactory.  
newInstance();  
    DocumentBuilder db = dbf.newDocumentBuilder();  
    Document doc;  
    doc = db.parse(url.openStream());  
    doc.getDocumentElement().normalize();  
    NodeList itemLst = doc.getElementsByTagName("item");  
    dataRSS.postTitle = new String[itemLst.getLength()];
```

```
    DocumentBuilder dbf = DocumentBuilderFactory.  
newInstance();  
    DocumentBuilder db = dbf.newDocumentBuilder();  
    Document doc;  
    doc = db.parse(url.openStream());  
    doc.getDocumentElement().normalize();  
    NodeList itemLst = doc.getElementsByTagName("item");  
    dataRSS.postTitle = new String[itemLst.getLength()];
```

```
    DocumentBuilder dbf = DocumentBuilderFactory.  
newInstance();  
    DocumentBuilder db = dbf.newDocumentBuilder();  
    Document doc;  
    doc = db.parse(url.openStream());  
    doc.getDocumentElement().normalize();  
    NodeList itemLst = doc.getElementsByTagName("item");  
    dataRSS.postTitle = new String[itemLst.getLength()];
```

```
    DocumentBuilder dbf = DocumentBuilderFactory.  
newInstance();  
    DocumentBuilder db = dbf.newDocumentBuilder();  
    Document doc;  
    doc = db.parse(url.openStream());  
    doc.getDocumentElement().normalize();  
    NodeList itemLst = doc.getElementsByTagName("item");  
    dataRSS.postTitle = new String[itemLst.getLength()];
```

All our import lines were automatically added via Android Studio's auto-complete features.



```
dataRSS.postURL = new String[itemLst.getLength()];
```

```
dataRSS.postContent = new String[itemLst.getLength()];
```

We're now going to go through **itemLst** using a 'for' loop and attempt to assign the contents of the elements referred to by the tag names 'title', 'link' and 'content:encoded' by passing them to **NodeList** elements, which is simply a collection of nodes we hope will contain the data we're after.

```
for (int i = 0; i < itemLst.getLength(); i++) {
```

```
    Node item = itemLst.item(i);
```

```
    if (item.getNodeType() == Node.ELEMENT_NODE) {
```

```
        Element ielem = (Element) item;
```

```
        NodeList title = ielem.getElementsByTagName("title");
```

```
        NodeList link = ielem.getElementsByTagName("link");
```

```
        NodeList content = ielem.getElementsByTagName("content:en  
coded");
```

Because we can't be certain that the data we're after will be contained within the title, link and content **NodeLists** we've created, we'll need some exception handling again with **try** as we attempt to assign the values from the RSS to the string arrays we created:

```
try {
```

```
    dataRSS.postTitle[i] = title.item(0).getChildNodes().item(0).  
getNodeValue();
```

```
    dataRSS.postURL[i] = link.item(0).getChildNodes().item(0).  
getNodeValue();
```

```
    dataRSS.postContent[i] = content.item(0).getChildNodes().  
item(0).getNodeValue();
```

```
} catch (NullPointerException e) {
```

```
    e.printStackTrace();
```

```
}
```

In the previous piece of code, you'll see that we've included the **catch** block for dealing with the exception. As you might expect, all this does is output the methods (the stack trace) of the application that have led to this point, and we'll finish of the class with closure on the previous **try** as well as close brackets for all the other blocks. We've put these on the same lines for brevity, as you should be properly indenting your code or getting Android Studio to do it for you (Code > Reformat Code in the menu):

```
}}}
```

```
} catch (Exception e) {
```

```
    e.printStackTrace();
```

```
}}}
```

## 7 ADDING THE FUNCTIONALITY

We're not going to spend much time on the GUI this month. All we're going to do is replace the `TextView` widget with the `ListView` widget. `ListsViews` are now synonymous with smartphones – they're the bread and butter widget used to display everything from email to Tweets. And while Android Studio has a wonderful visual editor for creating your own user interfaces, the best way to accomplish this is to open the GUI resource in your Android project – `app/src/res/activity_main.xml` in the project's hierarchy. When you click on this file, Android Studio will switch to the visual editor, but you'll need to switch from the 'Design' tab to the 'Text' tab to edit the XML file responsible for the layout directly. This view is still excellent, as it updates a virtual Android device of your choice, giving you the best of both worlds. To replace the `TextView`, simply paste over the entire block with the following:

```
<ListView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/listView"
    android:layout_gravity="center_vertical"
    android:layout_weight="1" />
```

The important thing to note here is that we're calling the new widget `ListView`, as referenced by the `android:id` tag. We'll use this to reference the widget within our `MainActivity` code, and that's going to be our next and final target for this tutorial, so double click on `MainActivity.java` and let's make it so!

Before we get onto the complete code for the main class, we've removed both of the methods that deal with the menu, because while they're an essential part of the template for adding functionality, they're only going to add complexity here.

```
public class MainActivity extends Activity {
    parseRSS parserss;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        parserss = new parseRSS();
        new updateRSS().execute(); } }
```

The above section of code is the beginning of our class. We create our own global `parserss` variable from the `parseRSS` method we created earlier – this should be important automatically because the Java class is in the same file as `MainActivity`. We then change the contents of the `onCreate` method that Android Studio creates automatically, adding the `parserss` activation and an `execute()` method call for a background processing class we haven't created yet. The `@Override` annotation is already there, because this is a reimplementation of another method and we want to make sure Java executes our version instead.

The next stage is to write the method to add items to the `ListView` widget, and to handle what happens when one of those items is clicked to add items to the

`list`. This is quite straightforward. We use `findViewById` to link the UI widget we laid out earlier with the `ListView` widget we create in code, and call this 'storylist'. We're then able to add elements via the `setAdapter` method. We're just adding the title at the moment, but we use the URL in the next view lines. We need to create an even listener to deal with user input, and we use `ListView`'s `setOnItemClickListener` and the `AdapterView.OnItemClickListener()` to handle the callback, which then decodes which item has been pressed via our 'position' variable and starts a new activity based on the URL. If we linked to an audio file, Android would launch an audio player, but as we're linking to a web page, it's going to open the web browser. You'll always be able to return to our app using the 'back' button.

```
public void populate_list() {
    ListView storylist;
    storylist = (ListView) findViewById(R.id.listView);
    storylist.setAdapter(new ArrayAdapter<String>(MainActivity.this, android.R.layout.simple_list_item_1, dataRSS.postTitle));
    storylist.setOnItemClickListener(new AdapterView.OnItemClickListener() {
        public void onItemClick(AdapterView<?> parent, View v, int position, long id) {
            dataRSS.position = position;
            Uri uri = Uri.parse(dataRSS.postURL[dataRSS.position]);
            Intent intent = new Intent(Intent.ACTION_VIEW, uri);
            startActivity(intent);
        } }); }
```

The final chunk of code is the `updateRSS` method that we called `execute()` on earlier. It uses `AsyncTask` to show an update dialogue and stop our application from locking up whilst we update the RSS feed.

```
private class updateRSS extends AsyncTask<Void, Void, Void>
    implements DialogInterface.OnCancelListener {
    private ProgressDialog dialog;
    protected void onPreExecute() {
        dialog = ProgressDialog.show(MainActivity.this, "", "Updating RSS. Please wait...", true); }
    protected Void doInBackground(Void... unused) {
        parserss.parse();
        return null; }
    protected void onPostExecute(Void unused) {
        dialog.dismiss();
        populate_list(); }
    public void onCancel(DialogInterface dialog) {
        cancel(true);
        dialog.dismiss();
    } }
```

And that's it! Run the project and it should download our RSS feed. Click on a story and it will open it from your default browser. Let us know how you get on! 📧

Graham Morrison is the author of *Kalburn*, a photo collection manager that, in its heyday, was in the Mandriva repositories. Nowadays he's best known for synthesizer music.