LINUX VOICE
CODING TUTORIAL

ANDROID DEVELOPMENT PART 2

GRAHAM MORRISON

We're prettying up last month's simple RSS reader to add more information and we get the whole thing running on real hardware.

WHY DO THIS?

- Android development needed be that difficult, and millions of users can take advantage of your work.
- Help develop the Linux Voice Android application.
- Develop awesome skills and career prospects.

1 KEEPING UP APPEARANCES

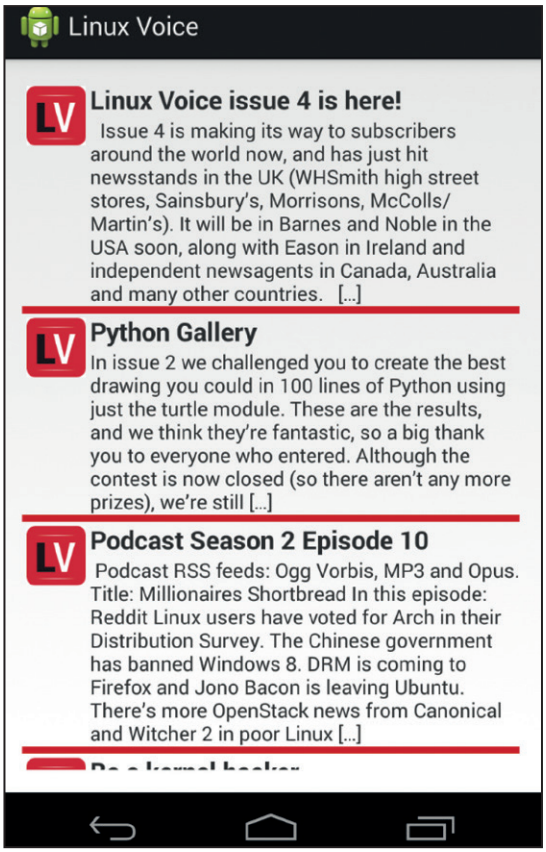
Last month's application was functional but basic. It used a default ListView widget to display the titles of news stories posted onto our website and enabled the user to touch a story to open it in a browser. Changing the default appearance of the ListView in Android, and nearly every other GUI element, is accomplished through the XML files that are used to represent each layout. For our main activity, this is the **activity_main.xml** file found in the **res** sub folder. Open this and add the following inside the ListView element for our application:

```
android:dividerHeight="2dp"
android:divider="#c61a27"
```

What we've done here is add a gap between the elements in the ListView and coloured that gap with an HTML colour notation value of **c61a27**. This just happens to be the red of Linux Voice, and Android Studio will even display a preview of this colour in the border to the left of your code. It appears as a small square, and if you click on this you'll be able to select a different colour from the palette dialog. But what does **dp** represent? This is dimension notation for Android and it's Google's solution to never knowing the resolution or pixel density of the devices you're working with, and working with increasingly large and abstract resolutions. It won't be long until we're dealing with 4k displays, for instance. **dp** is an abbreviation for 'density-independent pixels'. The principle is that you take 160dpi (pixels per inch) as a baseline value, where 1dp = 1 pixel at 160dpi, but that the value is scaled automatically according to the size and resolution of the display. If your display has a density of 320dpi, for example, there would be two pixels for every 1dp.

dp is the most common layout dimension, but it's also possible to use **px** for absolute pixel control. This isn't recommended – your layouts won't be able to adapt to different screen resolutions, but **px** can be useful when dealing with the fixed layouts of bitmaps. **sp** is another dimension format, similar to **dp**, but the number of pixels it represents changes when a user changes the font size. You can often see these changes directly in the design preview window within Android Studio, which is the best way to get a good idea about how your application is going to look.

Colours and images are previewed in the tiny border to the left of your code. Click on a colour to open the colour palette requester.



After this tutorial, your app should look like this, with long story descriptions, bold titles and an icon for each story.

Appearance is augmented further through XML files and classes that are linked from the **activity_main.xml** file. For example, you can colour items pressed and unpressed in your list purely from XML. If you add the following line, Android will look for a resource file called **selected** that it can use to illustrate selected items in the list view:

```
android:listSelector="@drawable/selected"
```

2 GRADIENTS

To create the resource file we just linked to, right-click on the **drawable-hdpi** folder in your project hierarchy and select New > Drawable Resource. Give this the name **selected** and make sure the root element type is **selector**. This will create the new file and solve the dependency error in **activity_main.xml**. There are three states we can define here – one where an item is selected, one where an item is selected and still pressed and one where an item is pressed but not selected. The shading for each of these states should also be farmed out to separate files so that any other lists or items can have the same appearance, and to define how each item is handled, you will need the following code inserted between the **selector** start and end elements:

```
<item
 android:state_selected="false"
 android:state_pressed="false"
 android:drawable="@drawable/gradient1" />
<item android:state_pressed="true"
 android:drawable="@drawable/gradient2" />
<item android:state_selected="true"
 android:state_pressed="false"
 android:drawable="@drawable/gradient2" />
```

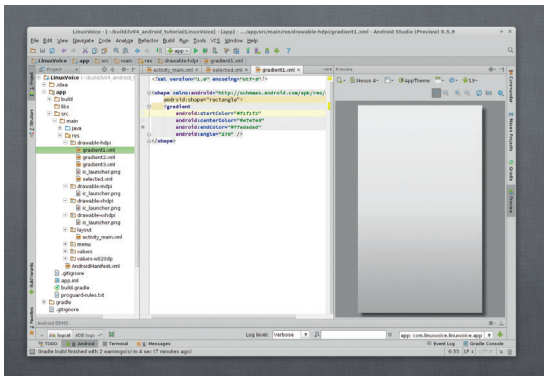
This code is simple enough to understand, and you'll need to create two further drawable resource files to satisfy each of the states. Their root elements will be **shape** rather than **selector**, as **shape** drawable elements can be used to define backgrounds, borders and gradients, which is how we're going to use them. We've called these files **gradient1** and **gradient2**. As you can tell from their names, we simply use these to define different colour gradients for each of the selection states. We won't go through each of the three, but here's how our first gradient looks:

```
android:shape="rectangle">
<gradient
 android:startColor="#f1f1f2"
 android:centerColor="#e7e7e8"
 android:endColor="#ffadadad"
 android:angle="270" />
```

This works in exactly the same way the gradient tool does in Gimp, and after you've added the HTML colour entries you'll be able to click on the small coloured square in the code border to change the colour to something else. You should also see a preview of the gradient to the right of the code so you can make sure it's the effect you think it is. Create the gradients in the same way, making the colours slightly different. This gradient is going to be used when you select an element within the list.

New items

We're now going to take on a bigger job, and that's redefining the appearance of the ListView itself. At the moment, there's very little we can change about the contents of a single item other than the text, but we want to be able to add the title of a story, its brief description and an icon. To do this, we'll need to create a separate layout for this cell and tell the ListView how to use our own layout rather than the one it defaults to. We'll start by creating a new layout for an individual item. Right-click on Layout on the folder view and select Layout Resource File from the drop-down menu that appears. Change the root element from LinearLayout to RelativeLayout and provide a filename – we called ours **singlerow.xml**. Android Studio will now switch to the visual editor layout mode for the new file.



Gradients are a good way of subtly showing the difference between one cell and the next, or when an item is selected.

Android Studio can be very helpful when it comes to layouts and will inform you of any required elements that may be missing. This is important when creating a new user interface, because the way your application looks and behaves is dependent on how you 'hang' the layout within the scaling designer. To get started, drag an ImageView widget from the Tools palette and into your application. As you hold the left button down and drag the destination cursor around the virtual Android device, you'll see Android Studio highlight all kinds of alignment possibilities. But because there's no image attached to the ImageView widget, you won't be able to see any of the alignment effects. To solve this problem, click on the ImageView widget in the component tree (this lists all the various UI components in your layout), then select the **src** data cell in the Properties list below. You should see a … (ellipsis) icon used to denote a file requester. Clicking on this will open a window that lists all the resources currently available to your project. Resources include colours, UI design components and external images. But there's no way from here to add your own images, and to do that, we'll need to go back to Android Studio.

LV PRO TIP

Android Studio can handle more than one project at a time, and can use either the existing application window or open a new environment. Choose your preference when you open a project while another project is already open.

Adding image assets

The various folders beneath **res** are for different resolutions -hdpi, mdpi, and ldpi, as well as designs and menus. Right-click on **res**, select New and then select Image Asset. This opens Asset Studio, which is a tool to make importing and managing images easier. Make sure the asset type is set to Launcher

Icons and use the **...** file requester to the right of the image file location to point at an image location. There are several different kinds of icons in Android. Launcher is the icon you'll see on your device's launch or home screens. Action Bar icons are found in Android's equivalent to a toolbar at the top of the screen, and notification icons are used within the notification view.

We used a download of our 'LV' logo, as this will also be ideal as the launcher icon. When selected, you'll see previews of your resized image for each resolution and you shouldn't need to change any of the other settings. Clicking on 'Next' will show you where each resolution of the image is going to be placed within your project's **res** folder structure, and clicking on Finish will apply those changes. You will immediately be able to see your image as a resource within the resolution folders of the project view, and when you go back to the Resources requester in the designer, you'll see your images listed as a resource – usually near the bottom of the list. Select it and click on OK.

Designing the layout using this view reminds us of the early days of QtDesigner, as you can't always tell what element of the user interface is having the effect you can see. Both the Relative and Linear layouts work like this – relative lets you link the position of one widget to the location of another, while linear places widgets next to each other, either horizontally or vertically. We've found that it's easier to add widgets and make adjustments in the visual editor, while layout is easier to define in the text editor. With that in mind, add two Plain TextView widgets without worrying where they fall in the layout, and switch back to the Text editor. First, make sure that the RelativeLayout, the ImageView and the two TextViews all contain the following:

```
android:layout_width="wrap_content"
android:layout_height="wrap_content"
```

wrap_content will enable a widget to only expand enough to contain whatever values it contains, rather than **fill_parent**, which allows the widget to take up as much space as is available. To stop all our widgets being drawn on top of each other, make sure the ImageView is first and contains the following:

```
android:layout_alignParentTop="true"
android:layout_alignParentLeft="true"
android:layout_alignParentStart="true"/>
```

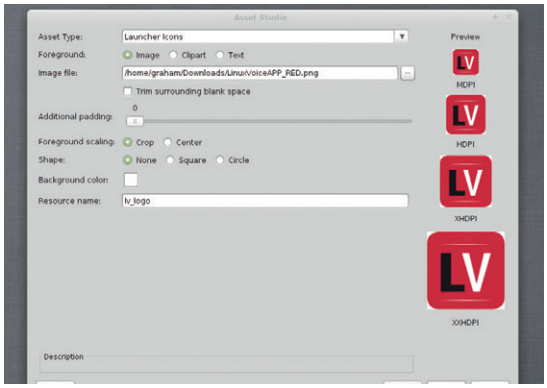
For the first text view, which is going to hold the title of our story, add this to its main element. The ImageView being referenced here is the default **id/name** for the ImageView widget we created, so this should be the same:

```
android:layout_alignTop="@+id/imageView"
android:layout_toRightOf="@+id/imageView"
```

And for the final TextView widget, you'll need to change the above two lines to the following, which places the TextView that will hold the description below the TextView that's going to hold the title:

```
android:layout_toRightOf="@+id/imageView"
android:layout_below="@id/textView"
```

The layout and alignment works in a similar way to object arrangement in an application like Inkscape or Scribus. You can play around with the various options either by letting Android Studio auto-complete the **layout_** keywords, or by dragging widgets around within the visual editor. We'd also suggest changing the font size for the title TextView and perhaps adjusting the colour of the description text, but these can be changed at any time.



Android Studio will automatically scale an image for each different class of resolution.

LV PRO TIP

If you're not using version control, you can zip up an entire project folder to take a snapshot and even move this to another Android Studio installation. Any new SDK is picked up automatically.

3 THE CODE

Now we've got the GUI design sorted, it's time to add the functionality to the code, and this is pretty simple. Most of this is handled by a new class that needs to be created (right-click on the Java folder) and called **lvList**. Here's what it needs to contain:

```
public class lvList extends ArrayAdapter<String>{
  private final Activity context;
  private final String[] title;
  private final String[] description;
  private ImageView imageView;
  public lvList(Activity context, String[] title, String[]
description) {
    super(context, R.layout.singlerow, title);
    this.context = context;
    this.title = title;
    this.description = description;  }

  @Override
  public View getView(int position, View view, ViewGroup
parent) {
    LayoutInflater inflater = context.getLayoutInflater();
    View rowView= inflater.inflate(R.layout.singlerow, null,
true);
    TextView txtTitle = (TextView) rowView.findViewById(R.
id.textView);
    TextView txtDescription = (TextView) rowView.
findViewById(R.id.textView2);
    txtTitle.setText(title[position]);
```

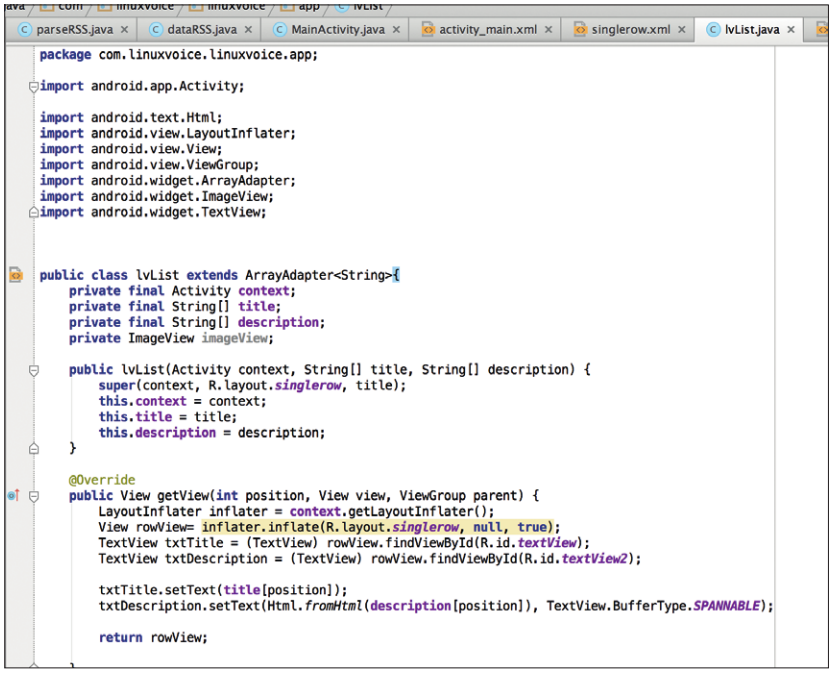
```
    txtDescription.setText(Html.fromHtml(description[positi
on]), TextView.BufferType.SPANNABLE);
    return rowView;
 } }
```

As with last month, add the **import** lines when Android Studio complains about missing modules. This code is creating an adapter to replace the standard adapter used to load ListView items. That's why there's an **@Override** statement, and within the method that follows, we place the title and description passed to the class when initialised to the widgets we created in the designer. Our single ListView item is expanded to a view, and we convert the HTML of our RSS feed using **setText(Html.fromHtml(description [position])** method. By default, this is the entire contents of a post, but you can change this by replacing **getElementsByTagName("content:encod ed")** with **getElementsByTagName("description")**' in our parseRSS class, as they're referring to different elements within the RSS.

The final chunk of code modifies the **MainActivity** replacing the old use of an adapter with our new one. To do this, comment out the line starting with **storylist.setAdapter** in the **populate_list()** method, and add the following two lines:

```
lvList adapter = new lvList (MainActivity.this, dataRSS.
postTitle, dataRSS.postContent);
storylist.setAdapter(adapter);
```

All we're doing here is creating a new adaptor using the **lvList** class we just created, passing the RSS values from the function we wrote last month. Those values will be used to populate the list and we use this within the **storylist** ListView used by **MainActivity**. After all this has been set and defined, all that's left to do is run your new application. The latest stories will download, complete with either the entire text or the overview, and tapping any story will still open the page in the browser.



The code to pull all the GUI elements together is probably simpler than the XML.

Graham Morrison is the author of Kalbum, a photo collection manager that, in its heyday, was in the Mandriva repositories. Nowaday's he's best known as Linux Voice's Editor In Chief.

Run your app on your phone You're not a real developer until you can touch your own code.

The emulator that's bundled with Android Studio is very convenient, and for most development, it's the best way to test your code. But there's no substitute for running your project on a real device as soon as possible, or even better, on a variety of real devices. There are several good reasons for this. The most important is usability, because unless you're running Android Studio on a touchscreen laptop, you'll only be able to interact with the emulator through your mouse. Your user, on the other hand, is only going to interact with your work through their fingers, so it's important to make sure things are working in the way you expected them to.

Secondly, real hardware can raise bugs or issues that aren't apparent in the emulator. This might be because of different versions of the operating system, but it could equally be because of other applications running alongside, or user-specific Android alternations that affect your application.

Finally, there's performance. The emulator gives zero indication of how your app might perform, and CPU and graphics constraints aren't the only considerations. Bandwidth is vital, especially with our project, and you need to be sure you're not asking too much of a cellular data connection, or that your application can fail gracefully if it loses the connection or doesn't have enough speed.

The official Android documentation asks that you add **android:debuggable="true"** to your **AndroidManifest.xml** file, placing it beneath **android:label="@string/app_name"** within the application element, but this isn't necessary with Android Studio, as you can switch to debug mode dynamically without having to add the line first. The next step is to put your Android device into development mode. On Android 4.0 and 4.1, this option can be found in the Settings> Developer Options page. But with more recent versions, the option has been obfuscated. Just go to Settings > About Phone and tap 7 times on the Build Number field at the bottom of the list. As you approach the 7th tap, you'll see a pop-text message say you're so many taps from being a developer, before the last tap saying 'You are now a developer'. 'Developer options' will now be visible on the previous setting screen, and you need to enable 'USB debugging' and accept the caveat that appears.

The next step will depend on your distribution. With a recent Ubuntu derivative (we're using Mint 16) you can skip this and simply connect your Android device directly. Older versions or other distributions may need to add a udev rule. To do this, type **sudo nano /etc/udev/rules.d/51-android.rules** and add the following line:

```
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1",
MODE="0666", GROUP="plugdev"
```

You will need to change the **idVendor** hexadecimal value to the one that matches your Android device, and you can get this value by typing **tail -f /var/log/syslog** on the command line and plugging in your phone. You should see a line similar to 'New USB device found, idVendor=18d1, idProduct=4ee2', which is what we see when we connect our Nexus 5. Extract the **idVendor** value and place this into the configuration file.

When you connect your Android device, the device itself will ask whether you want to allow USB debugging and to accept a RSA fingerprint that identifies your development PC. You need to accept this, and for convenience, we'd suggest also enabling the 'Always Allow From This Computer' option too. If you want to make sure your device has been detected correctly, type the following:

```
/usr/share/android-studio/data/sdk/platform-tools/
adb devices
```

The output should include the serial number of your Android unit, followed by the word 'device'. If it says 'unauthorized', this means you've not accepted the RSA key on the device. Otherwise, you're ready to go. When you next run your app from Android Studio, the device choice should list your external device, just as it does instances launched from the emulator, and you'll be able to select this as a destination for your app and choose to always use this device. Clicking on 'OK' will launch the app on your device in exactly the same way it does from the emulator.