Chapter 5: Taking Android Out for a Walk

In this chapter, you'll move past the basics, loosen the leash, and let Android stretch its legs a little bit. More and more as the mobile software field progresses, it's become less and less possible to make an application that doesn't rely heavily on the Web. In many ways, fully featured Internet access has become one of the essentials in the mobile world. The depth and breadth of Android's network layer makes it impossible to cram into one small chapter of one small book. With this in mind, I'll try to, as I have before, arm you with the basics you'll need to make production-level applications. Along the way, as in previous examples, you'll explore a few tangential pieces of Android's technology.

Starting with the essentials, you'll learn how to use a simple HTTP connection to download, parse, and list the elements of a remote XML file. These elements, in your sample application, will be Internet radio stations contained in a basic XML file. Indeed, your entire sample application will be focused on building a simple Internet radio player. Sadly, the state of Android's streaming audio does not live up to its documentation. So, this chapter will be more of an exercise and less of a fully functioning application.

Loading a List from the Web

To make your snazzy example application, you'll have to fetch, parse, and display a simple list of radio stations. Doing this will require me to cover a range of subjects from HTTP transactions to `ListView`s. Pulling down a list down from the Web and displaying them onscreen is something I find myself, as an engineer on many mobile projects, doing nearly constantly. From pulling down a "friends list" on a social network to a "high score list" from an online game element, there's something universal about downloading, parsing, and displaying a list.

Although I realize you might not be making a streaming music application anytime soon, this example is general enough that it should serve as a guide for both basic network operations and handling selection menus. Also of note is that you're performing almost the same task you did with the custom widget work back in Chapter 3. Frankly, I'm not sure this method is a whole lot easier, but perhaps that's because I'm using it in a fairly rudimentary way and have missed some of the more complicated bits it would help with. In any case, enough gabbing—let's get into the basics of network connectivity.

First Things…First?

Your first task will be to pull the XML file off the server. I've made a simple XML example file (which, in your final application, would be provided by a PHP script or a Java servlet) and hosted it on my web site. Before we get any further, I'll list a few variable declarations you're going to need later. Both for network connectivity and for our eventual selectable list, Code Listing 5-1 shows the class declaration and variable dump.

Code Listing 5-1. Essential Class Variables

```
public class StationPicker extendsActivity {

//Uncomment the next lines after we've
//defined the StationData class
//Vector<StationData> stationListVector =
//new Vector<StationData>();
SAXParser parser = null;
XMLReader reader = null;
//You'll have to check the code for the following line.
//XMLHandler handler = new XMLHandler();
ArrayAdapter<StationData> adapter = null;
```

What you see in the previous listing is the buffet of objects you'll need to complete your little sample application. You have the `Vector` to hold the station list, a SAX parser, a reader, and a handler for XML parsing. Last you have the `ArrayAdapter`, which, at some point, you'll fill with elements to render for your onscreen menu.

Additionally, Code Listing 5-2 shows how things are initialized within your `onCreate` function.

Code Listing 5-2. Setting Up for XML Parsing

```
{
    super.onCreate(icicle);
    try
    {
        //This can be just about
        //anything at this point
        setContentView(R.layout.main);
        SAXParserFactory f =
            SAXParserFactory.newInstance();
        parser = f.newSAXParser();
        reader = parser.getXMLReader();
        reader.setContentHandler(handler);
        //We'll get to the contents of the
        //following function later.
        // If you're following along
        //just stub it to return null
        initList();
    }
    catch (Exception e)
    {
        Log.e("StationPicker", "Parser FAIL!");
    }
```

}Again, because I'm assuming you're comfortable with Java, I'm not going to walk through all the steps required. If you want the full context and associated code, feel free to grab the project online. As for the `initList` method, I'll define that in a later section. For now, if you're following along, you can follow the comment's advice and stub it to return `null`.

Getting the Network in Gear

I've elected to kick off the network connection during the `onStart` method inside the `ListActivity`. Normally you would probably do this once on startup and then, using an intent, move to a new activity for showing the list. But for the sake of keeping this example as simple as possible, I'm going to do as much as I can within the single activity. It'll keep you from having to deal with intent management, and it'll give me a chance to show you how to use the UI thread. You'll learn more about that later; for now, the Web! See Code Listing 5-3.

Code Listing 5-3. Creating and Using a Simple HTTP Connection

```
public void onStart()
{
super.onStart();
Thread t = new Thread()
    {
        public void run()
        {
            HttpUriRequest request = null;
            HttpResponse resp = null;
            InputStream is = null;

            DefaultHttpClient client =
               new DefaultHttpClient();

            try{
                //Build the request
                request =
                  new HttpGet(
             "http://www.wanderingoak.net/stations.xml");
                //Execute it using the default
                  //HTTP Client settings;
                resp = client.execute(request);
                //Pull out the entity
                HttpEntity entity= resp.getEntity();
```

```
                    //Snag the response stream from the entity
                    is = entity.getContent();
                    //Parse the incoming data
                    reader.parse(new InputSource(is));
                } catch (Exception e)
                 {
                   Log.e("LoadStations","FAIL!");
                 }
            }
        };
        t.start();
    }
```

You first need an instance of the `DefaultHttpClient`. You can obtain this by simply creating it with `new`. Next, you'll create a new `HttpGet` object, passing in the location of your XML feed. You can then execute the HTTP request on the default client with your new request object. This is a blocking operation (hence the new thread), and once the execute method returns, you can get the `HttpEntity`. Out of this object you can retrieve an `InputStream` containing the body of the response.

If that final reader call makes no sense to you at all, that's because it shouldn't, because I haven't told you what it does yet. Yes, I know your sample code won't compile without it. Hold on a second, and I'll get to that.

Note The `DefaultHttpConnection` object seems to spool up and run, at least with the current version of the emulator, hideously slowly. You can probably get better performance by tinkering with the various subclasses of the `HttpClient` class. Your mileage may vary, but if you need a fast and easy proof-of-concept demonstration, the default one may be the way to go.

Putting the Data in Its Place

As you can see in the previous code, pulling down a bit of XML data from a server or any data, for that matter, is a pretty simple process. That `reader.parse` line you've been pestering me about is a simple call to a SAX parser. Android rolls out the door with a few XML parsers to choose from, and since I'm assuming you're comfortable with Java and for the sake of time, I'm not going to spell it out for you.

If you absolutely must know what's going on, you're welcome to grab the sample code and check it out. For now, however, it's enough to know that the parser fills a `Vector` full of `StationData` objects. Code Listing 5-4 shows the definition.

Code Listing 5-4. Defining the Data-Housing Class

```
class StationData
{
    public String title = "";
    public String url = "";
    public String toString()
    {
        return title;
    }
}
```

For simplicity's sake, I've avoided the common encapsulation practice of defining getters and setters on private `String` elements. Instead, you'll just access the elements within the class directly. If you're a C/C++ programmer, this looks more like a "struct" than a "class." Take special note of that `toString` method. It may look useless at this moment, mostly because it is right now, but its function will become much more apparent in a few paragraphs. Each station from the XML file will get its own `StationData` object. Again, just for the sake of example, Code Listing 5-5 shows what a single station element in the XML looks like.

Code Listing 5-5. Sample Network Data

```
<xml>
<stationList>
<station>
<title>Pop Rock Top 40</title>
<audioUrl>
http://scfire-nyk-aa02.stream.aol.com:80/stream/1074
</audioUrl>
</station>
</stationList>
</xml>
```

Since, at this point, you'll let the XML parser take care of things with that `reader.parse` line, you can get on with making your list of selectable elements. Your parser will fill the `StationData` vector with a few elements. Your next few tasks are to pull them out of the vector and place them on the screen in a way the user can interact with.

Making a List and Checking It...

Making your list menu function correctly will require a few steps. You'll first have to convert your activity to a `ListActivity` and do all the housework that switch demands. Next, you'll actually insert the elements from the vector you built previously. Last, you'll react to select events and begin streaming some theoretical audio. Again, in a production version of this application, you would probably use more than one activity, but for the sake of simplicity, you'll just to cram it all into one.

The Setup: Embracing the List

Your first task, if you're going to display the selectable list of stations, is to switch your humdrum activity to a shiny new `ListActivity`. Here's the class declaration in its new and pristine form:

```
public class StationPicker extends ListActivity
```

This conversion carries with it a few notable responsibilities. If you don't fulfill these obligations, Android will throw a bunch of exceptions at you. First, you'll need to add a `ListView` to that default layout file because each `ListActivity` must have an associated `ListView`. Here's what the example `main.xml` looks like:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
>
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Loading Stations..."
    android:id="@+id/loadingStatus"
    />
<ListView android:id="@+id/android:list"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        />
</LinearLayout>
```

Adding the Adapter

Second, you'll need to add an adapter to the list widget. You need to define what each element is going to look like. You'll make a simple XML file containing a single text element. Call it `list_element.xml`; it should look like Code Listing 5-6.

Code Listing 5-6. `res/layout/list_element.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<TextView id="@+id/textElement"
xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="fill_parent"
android:layout_height="wrap_content"/>
```

This `TextView` describes, to Android, what each element in the list should look like. This is the place for fonts, colored texts, and background resources. More complicated list elements are possible, but I'll get into that variation a little bit later.

Remember that `initList` method I told you to stub out earlier? Rather than just returning `null`, Code Listing 5-7 is what it should look like.

Code Listing 5-7. Must Do: Adding an Adapter to the ListView

```
privatevoid initList()
{
    adapter = new ArrayAdapter<StationData>(
         StationPicker.this, R.layout.list_element);
    setListAdapter(adapter);
}
```

Every `ListView` must have a corresponding adapter. Adapters come in a few sizes and flavors. Table 5-1 gives a brief description of the most important among them.

Table 5-1. List Adapters

LIST ADAPTER	DESCRIPTION
Cursor adapter	A simple adapter that is perfect for listing the contents of SQL databases, search results, or any other data that is commonly formatted in a cursor. In fact, the Google documentation has an excellent example of using a cursor adapter: `http://code.google.com/android/intro/tutorial-ex1.html`.
Resource cursor adapter	The perfect adapter for building a selectable list from a static XML file. If your menu/list is a list of known elements, such as a main menu, list of help topics, or other well-known catalog of information, this is the adapter for you.

continued

Table 5-1. continued

LIST ADAPTER	DESCRIPTION
Array adapter	The adapter we're using in this example. If you don't know what's going to be in your list at compile time, because you won't know what your station list will be, then this is the easiest way to convert a list of XML elements into a selectable list.

For now, you have a fully functioning, while extremely ugly, menu list ready to go. Now all you need is some data!

Stuffing Data into the Adapter

Placing the data in the adapter is simple but for one thing: it must take place inside the UI thread. What, you may ask, am I talking about? The UI thread is a specific thread of execution, which controls the redraw loop. You'll notice that if you start a new generic Java thread and then try to change the current view, add data elements to a list adapter, or any other UI task, Android will get really grumpy with you. By grumpy, I mean it won't work, or it'll throw a stack of exceptions at you.

Reclaiming the UIThread

Since you started a new inline Java thread to handle your blocking network connection, you'll now have to define another "runnable" to get back into the good graces of the UI gods. Thankfully, activities contain a method for scheduling code for the UI thread. You'll add it to the bottom of your network code (see Code Listing 5-8). I'll repeat the last few lines for context.

Code Listing 5-8. Recovering the UI Thread

```
Thread t = new Thread()
{
    //---------
    //Skipping a bunch of
    //Code here
    reader.parse(new InputSource(is));
    //Run our code on the UI Thread.
    UIThreadUtilities.runOnUIThread(
            StationPicker.this,r);
```

Note Don't try to paste the previous code into your project and compile it just yet. You need to define that runnable `r` object first. Bear with me for just a few minutes, or paragraphs, depending on how fast you read.

The object `UIThreadUtilities` is a mostly static class, which is a member of the `Activity` class. You'll have to pass in a context object to `runOnUIThread`, and since `this` is a pointer to your currently running `Thread` instead of your `ListActivity`, you'll have to grab your `ListActivity` (a subclass of `Context`) from `StationPicker.this`. That `r` reference is a "runnable" that you'll define in just a minute.

At Last, Adding the Data

You're finally ready to start shoveling `StationData` elements into your `ArrayListAdapter`. You'll do that inside that runnable `r` object you heard me talk about earlier (Code Listing 5-9).

Code Listing 5-9. Adding Elements to the Adapter

```
Runnable r = new Runnable()
{
public void run()
{
    TextView t =
         (TextView) findViewById(
             R.id.loadingStatus);
    t.setText("Stations Loaded");
    try{
         for(int i=0;
             i < stationListVector.size();
                i++)
                  adapter.addObject(
                  stationListVector.elementAt(i));
         }catch (Exception e) {}
            getListView().invalidate();
         }
};
```

Because you're now on the UI thread, it's possible to modify the contents of the loading status text. Once you've changed the status message, you can begin adding elements to the `ArrayAdapter`. You'll just loop through the size of the vector and add each item to the adapter. How, you may ask, does the list element know what text to insert into the `TextView` that comprises each visual element in the list? Simple, look back to that `toString` method you overrode in your `StationData` class. When building the list, the `ArrayAdapter` calls `toString` on each element in the array and displays that text onscreen.

Selection…

You now have a functioning, selectable list of radio stations! Of course, you don't do anything when an item is selected, so you'll have to do something about that. Thankfully, the `ListView`'s tight integration with the `ListActivity` makes this a breeze. Simply override the protected method:

```
protectedvoid onListItemClick(
      ListView l, View v, int position, long id)
{
    StationData selectedStation =
    stationListVector.elementAt(position);
    MediaPlayer player = new MediaPlayer();
    try
    {
        player.setDataSource(selectedStation.url);
        player.start();
    }
    catch (Exception e)
    {
        Log.e("PlayerException", "SetData");
    }
}
```

I've included the audio code that is, as far as I can tell, correct according to the documentation. Just because the documentation says that it works doesn't, however, mean that it actually will work. In fact, the previous code, which links to a Shoutcast MP3 link, doesn't throw an exception but doesn't play. I can only hope the Android engineers resolve this issue before the application launches.

There's been lively debate and lots of example code running around on the Web. A little work with Google's search engine will unfold the multitude of hacky workarounds.

Caution Nowhere in this example application have I done any useful error handling. Mostly I'll catch exceptions and print something to a log. Your eventual mobile app will have to be better about errors than I'm currently being, because, trust me, networking on the mobile can be a bit touch-and-go.

The Next Step

The final step in this chapter is to give the `ListView` a little panache. You'll want to add a background to the entire screen. Doing this should look a little familiar, because you've done it before in a previous example (see Code Listing 5-10).

Code Listing 5-10. Linear Layout XML Block Inside `Main.xml`

```
<LinearLayout xmlns:android=
  "http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="@drawable/bg"
>
```

`@drawable/bg` refers, of course, to an image inside the `/res/drawable/` directory. You'll also want to adjust the width of the list widget:

```
<ListView android:id="@+id/android:list"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        />
```

This will keep the menu elements from changing size element by element, which, I think you can agree, looks pretty horrendous. Setting a list view layout width or height to `wrap_content` causes it to wrap each menu item individually. Go figure.

Dressing Up the Menu

There is one more major change you can make to the menu that will give you a little more control over rendering it.

Android allows you, when defining the UI elements for the adapter, to specify a large menu item object and then point to a `TextView` inside that you'd like edited. Before, you would point to a single, predefined

`TextView`. Code Listing 5-11 shows what your new list element layout file will look like.

Code Listing 5-11. The New and Improved `list_element.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
   "http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="22dip”
    android:background="@drawable/listbg"
>
<TextView android:id="@+id/textElement"
          xmlns:android=
     "http://schemas.android.com/apk/res/android"
          android:layout_width="fill_parent"
          android:layout_height="fill_parent"
          android:textAlign="center"
          />
</LinearLayout>
```

In this code, you've added a linear layout with some specific dimensions. You've also given it the background `listbg.png`. Interestingly enough, Android will rescale your background image to fit the space of the calculated background size. You may wonder, if you've done your homework, why you're using a linear layout instead of just adding a background and dimensions to the previous text view. You're doing this simply for demonstrative reasons. I want you, when you make an application that's much better than mine, to see how complex lists can be put together. Before I wrap up, there's one more line in the code you need to update to make this change. It's within the `initList` method:

```
private void initList()
{
adapter = new ArrayAdapter<StationData>(
  StationPicker.this, R.layout.list_element,
   R.id.textElement);
 ..............
```

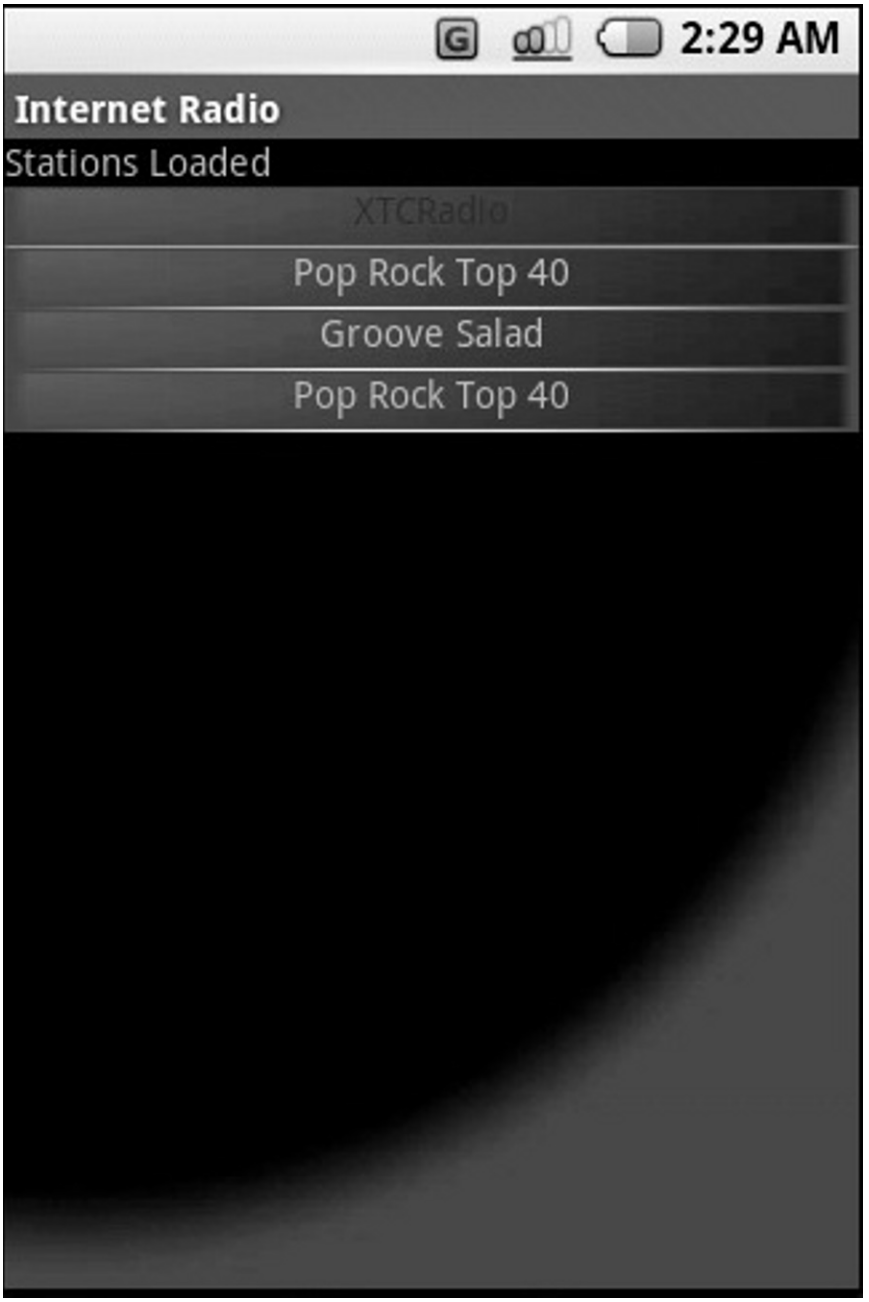
```
setListAdapter(adapter);
}
```

In the previous Adapter initializer, you specified only the layout element. Now you need to point to a file in the `/res/folder/` that contains the more complicated list element as well as a pointer that tells Android where to place the text pulled from the `toString` function of the `StationData` object.

Now, if you've done everything correctly (or you've cheated and downloaded the sample file), you should see the layout looking like Figure 5-1.

Frankly, thanks to my abysmal graphic designer skills, this version of the UI couldn't be described as pretty. It probably also couldn't be described as good. The point is not to make fun of my poor sense of graphical design, although you're more than welcome to do so. The point is that this example should show you how to make your application look better than my example. Now you can use nearly anything to construct this menu.

Figure 5-1. The dressed-up station list



Looking Back

Over the course of this chapter, you've had a chance to let Android stretch its legs a little bit. I covered basic networking, some more in-depth UI layout, and a little XML parsing to boot.

The HTTP layer is straightforward and easy to use at this point, despite being cumbersome and slow (at least on the OS X emulator). Android clearly has the ability to delve into proxies, cookies, socket-level connections, and much more advanced web-fu. You were able to get into downloading data, using XML data, and using a SAX parser to stuff it into a vector. From this `Vector`, you built up a list that, when an item is selected, launched into some theoretical media streamer. Sadly, the media-streaming capabilities don't live up to its documentation, but, over time, this is something that should be remedied.