**2. Android Resources and Components**

# Android Resources Organizing & Accessing

There are many more items which you use to build a good Android application. Apart from coding for the application, you take care of various other **resources** like static content that your code uses, such as bitmaps, colors, layout definitions, user interface strings, animation instructions, and more. These resources are always maintained separately in various sub-directories under **res/** directory of the project.

This tutorial will explain you how you can organize your application resources, specify alternative resources and access them in your applications.

## Organize resource in Android Studio

MyProject/

app/

manifest/

AndroidManifest.xml

java/

MyActivity.java

**res/**

drawable/

icon.png

layout/

activity\_main.xml

info.xml

values/

strings.xml

|  |  |
| --- | --- |
| **Sr.No.** | **Directory & Resource Type** |
| 1 | **anim/**  XML files that define property animations. They are saved in res/anim/ folder and accessed from the **R.anim** class. |
| 2 | **color/**  XML files that define a state list of colors. They are saved in res/color/ and accessed from the **R.color** class. |
| 3 | **drawable/**  Image files like .png, .jpg, .gif or XML files that are compiled into bitmaps, state lists, shapes, animation drawable. They are saved in res/drawable/ and accessed from the **R.drawable** class. |
| 4 | **layout/**  XML files that define a user interface layout. They are saved in res/layout/ and accessed from the **R.layout** class. |
| 5 | **menu/**  XML files that define application menus, such as an Options Menu, Context Menu, or Sub Menu. They are saved in res/menu/ and accessed from the **R.menu** class. |
| 6 | **raw/**  Arbitrary files to save in their raw form. You need to call *Resources.openRawResource()* with the resource ID, which is *R.raw.filename* to open such raw files. |
| 7 | **values/**  XML files that contain simple values, such as strings, integers, and colors. For example, here are some filename conventions for resources you can create in this directory −   * arrays.xml for resource arrays, and accessed from the **R.array** class. * integers.xml for resource integers, and accessed from the **R.integer** class. * bools.xml for resource boolean, and accessed from the **R.bool** class. * colors.xml for color values, and accessed from the **R.color**class. * dimens.xml for dimension values, and accessed from the **R.dimen** class. * strings.xml for string values, and accessed from the **R.string** class. * styles.xml for styles, and accessed from the **R.style** class. |
| 8 | **xml/**  Arbitrary XML files that can be read at runtime by calling *Resources.getXML()*. You can save various configuration files here which will be used at run time. |

## Alternative Resources

Your application should provide alternative resources to support specific device configurations. For example, you should include alternative drawable resources ( i.e.images ) for different screen resolution and alternative string resources for different languages. At runtime, Android detects the current device configuration and loads the appropriate resources for your application.

To specify configuration-specific alternatives for a set of resources, follow the following steps −

* Create a new directory in res/ named in the form **<resources\_name>-<config\_qualifier>**. Here **resources\_name**will be any of the resources mentioned in the above table, like layout, drawable etc. The **qualifier** will specify an individual configuration for which these resources are to be used. You can check official documentation for a complete list of qualifiers for different type of resources.
* Save the respective alternative resources in this new directory. The resource files must be named exactly the same as the default resource files as shown in the below example, but these files will have content specific to the alternative. For example though image file name will be same but for high resolution screen, its resolution will be high.

Below is an example which specifies images for a default screen and alternative images for high resolution screen.

MyProject/

app/

manifest/

AndroidManifest.xml

java/

MyActivity.java

**res/**

drawable/

icon.png

background.png

**drawable-hdpi/**

icon.png

background.png

layout/

activity\_main.xml

info.xml

values/

strings.xml

Below is another example which specifies layout for a default language and alternative layout for Arabic language.

MyProject/

app/

manifest/

AndroidManifest.xml

java/

MyActivity.java

**res/**

drawable/

icon.png

background.png

**drawable-hdpi/**

icon.png

background.png

layout/

activity\_main.xml

info.xml

**layout-ar/**

main.xml

values/

strings.xml

## Accessing Resources

During your application development you will need to access defined resources either in your code, or in your layout XML files. Following section explains how to access your resources in both the scenarios −

### *Accessing Resources in Code*

When your Android application is compiled, a **R** class gets generated, which contains resource IDs for all the resources available in your **res/** directory. You can use R class to access that resource using sub-directory and resource name or directly resource ID.

### Example

To access *res/drawable/myimage.png* and set an ImageView you will use following code −

ImageView imageView = (ImageView) findViewById(R.id.myimageview);

imageView.setImageResource(R.drawable.myimage);

Here first line of the code make use of *R.id.myimageview* to get ImageView defined with id *myimageview* in a Layout file. Second line of code makes use of *R.drawable.myimage* to get an image with name **myimage** available in drawable sub-directory under **/res**.

### Example

Consider next example where *res/values/strings.xml* has following definition −

<?xml version="1.0" encoding="utf-8"?>

<resources>

<string name="hello">Hello, World!</string>

</resources>

Now you can set the text on a TextView object with ID msg using a resource ID as follows −

TextView msgTextView = (TextView) findViewById(R.id.msg);

msgTextView.setText(R.string.hello);

### Example

Consider a layout *res/layout/activity\_main.xml* with the following definition −

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:layout\_width="fill\_parent"

android:layout\_height="fill\_parent"

android:orientation="vertical" >

<TextView android:id="@+id/text"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Hello, I am a TextView" />

<Button android:id="@+id/button"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Hello, I am a Button" />

</LinearLayout>

This application code will load this layout for an Activity, in the onCreate() method as follows −

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

}

### Accessing Resources in XML

Consider the following resource XML *res/values/strings.xml* file that includes a color resource and a string resource −

<?xml version="1.0" encoding="utf-8"?>

<resources>

<color name="opaque\_red">#f00</color>

<string name="hello">Hello!</string>

</resources>

Now you can use these resources in the following layout file to set the text color and text string as follows −

<?xml version="1.0" encoding="utf-8"?>

<EditText xmlns:android="http://schemas.android.com/apk/res/android"

android:layout\_width="fill\_parent"

android:layout\_height="fill\_parent"

android:textColor=**"@color/opaque\_red"**

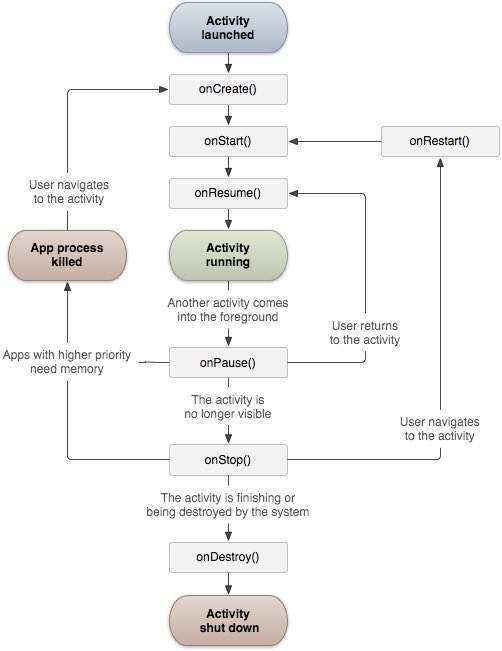
android:text=**"@string/hello" />**

Now if you will go through previous chapter once again where I have explained **Hello World!** example, and I'm sure you will have better understanding on all the concepts explained in this chapter. So I highly recommend to check previous chapter for working example and check how I have used various resources at very basic level.

# Android - Activities

An activity represents a single screen with a user interface just like window or frame of Java.Android activity is the subclass of ContextThemeWrapper class.

If you have worked with C, C++ or Java programming language then you must have seen that your program starts from **main()** function. Very similar way, Android system initiates its program with in an **Activity** starting with a call on *onCreate()* callback method. There is a sequence of callback methods that start up an activity and a sequence of callback methods that tear down an activity as shown in the below Activity life cycle diagram: (*image courtesy : android.com* )



The Activity class defines the following call backs i.e. events. You don't need to implement all the callbacks methods. However, it's important that you understand each one and implement those that ensure your app behaves the way users expect.

|  |  |
| --- | --- |
| **Sr.No** | **Callback & Description** |
| 1 | **onCreate()**  This is the first callback and called when the activity is first created. |
| 2 | **onStart()**  This callback is called when the activity becomes visible to the user. |
| 3 | **onResume()**  This is called when the user starts interacting with the application. |
| 4 | **onPause()**  The paused activity does not receive user input and cannot execute any code and called when the current activity is being paused and the previous activity is being resumed. |
| 5 | **onStop()**  This callback is called when the activity is no longer visible. |
| 6 | **onDestroy()**  This callback is called before the activity is destroyed by the system. |
| 7 | **onRestart()**  This callback is called when the activity restarts after stopping it. |

## Example

This example will take you through simple steps to show Android application activity life cycle. Follow the following steps to modify the Android application we created in *Hello World Example* chapter −

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | You will use Android studio to create an Android application and name it as *HelloWorld* under a package *com.example.helloworld* as explained in the *Hello World Example* chapter. |
| 2 | Modify main activity file *MainActivity.java* as explained below. Keep rest of the files unchanged. |
| 3 | Run the application to launch Android emulator and verify the result of the changes done in the application. |

Following is the content of the modified main activity file **src/com.example.helloworld/MainActivity.java**. This file includes each of the fundamental life cycle methods. The **Log.d()** method has been used to generate log messages −

package com.example.helloworld;

import android.os.Bundle;

import android.app.Activity;

import android.util.Log;

public class MainActivity extends Activity {

String msg = "Android : ";

/\*\* Called when the activity is first created. \*/

@Override

public void **onCreate**(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

Log.d(msg, "The onCreate() event");

}

/\*\* Called when the activity is about to become visible. \*/

@Override

protected void **onStart()** {

super.onStart();

Log.d(msg, "The onStart() event");

}

/\*\* Called when the activity has become visible. \*/

@Override

protected void **onResume()** {

super.onResume();

Log.d(msg, "The onResume() event");

}

/\*\* Called when another activity is taking focus. \*/

@Override

protected void **onPause()** {

super.onPause();

Log.d(msg, "The onPause() event");

}

/\*\* Called when the activity is no longer visible. \*/

@Override

protected void **onStop()** {

super.onStop();

Log.d(msg, "The onStop() event");

}

/\*\* Called just before the activity is destroyed. \*/

@Override

public void **onDestroy()** {

super.onDestroy();

Log.d(msg, "The onDestroy() event");

}

}

An activity class loads all the UI component using the XML file available in *res/layout* folder of the project. Following statement loads UI components from *res/layout/activity\_main.xml file*:

setContentView(R.layout.activity\_main);

An application can have one or more activities without any restrictions. Every activity you define for your application must be declared in your *AndroidManifest.xml* file and the main activity for your app must be declared in the manifest with an <intent-filter> that includes the MAIN action and LAUNCHER category as follows:

<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.tutorialspoint7.myapplication">

<application

android:allowBackup="true"

android:icon="@mipmap/ic\_launcher"

android:label="@string/app\_name"

android:supportsRtl="true"

android:theme="@style/AppTheme">

<activity android:name=".MainActivity">

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

</intent-filter>

</activity>

</application>

</manifest>

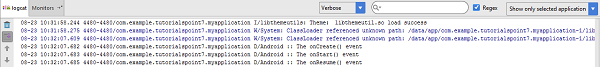
If either the MAIN action or LAUNCHER category are not declared for one of your activities, then your app icon will not appear in the Home screen's list of apps.

Let's try to run our modified **Hello World!** application we just modified. I assume you had created your **AVD** while doing environment setup. To run the app from Android studio, open one of your project's activity files and click Run Eclipse Run Icon icon from the toolbar. Android studio installs the app on your AVD and starts it and if everything is fine with your setup and application, it will display Emulator window and you should see following log messages in **LogCat**window in Android studio −

08-23 10:32:07.682 4480-4480/com.example.helloworld D/Android :: The onCreate() event

08-23 10:32:07.683 4480-4480/com.example.helloworld D/Android :: The onStart() event

08-23 10:32:07.685 4480-4480/com.example.helloworld D/Android :: The onResume() event



Let us try to click lock screen button on the Android emulator and it will generate following events messages in **LogCat** window in android studio:

08-23 10:32:53.230 4480-4480/com.example.helloworld D/Android :: The onPause() event

08-23 10:32:53.294 4480-4480/com.example.helloworld D/Android :: The onStop() event

Let us again try to unlock your screen on the Android emulator and it will generate following events messages in **LogCat** window in Android studio:

08-23 10:34:41.390 4480-4480/com.example.helloworld D/Android :: The onStart() event

08-23 10:34:41.392 4480-4480/com.example.helloworld D/Android :: The onResume() event

Next, let us again try to click Back button Android Back Button on the Android emulator and it will generate following events messages in **LogCat** window in Android studio and this completes the Activity Life Cycle for an Android Application.

08-23 10:37:24.806 4480-4480/com.example.helloworld D/Android :: The onPause() event

08-23 10:37:25.668 4480-4480/com.example.helloworld D/Android :: The onStop() event

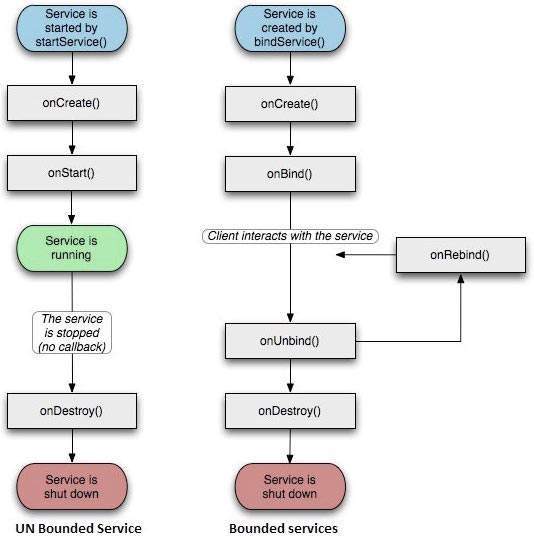
08-23 10:37:25.669 4480-4480/com.example.helloworld D/Android :: The onDestroy() event

# Android - Services

A **service** is a component that runs in the background to perform long-running operations without needing to interact with the user and it works even if application is destroyed. A service can essentially take two states −

|  |  |
| --- | --- |
| **Sr.No.** | **State & Description** |
| 1 | **Started**  A service is **started** when an application component, such as an activity, starts it by calling *startService()*. Once started, a service can run in the background indefinitely, even if the component that started it is destroyed. |
| 2 | **Bound**  A service is **bound** when an application component binds to it by calling *bindService()*. A bound service offers a client-server interface that allows components to interact with the service, send requests, get results, and even do so across processes with interprocess communication (IPC). |

A service has life cycle callback methods that you can implement to monitor changes in the service's state and you can perform work at the appropriate stage. The following diagram on the left shows the life cycle when the service is created with startService() and the diagram on the right shows the life cycle when the service is created with bindService(): *(image courtesy : android.com )*



To create a service, you create a Java class that extends the Service base class or one of its existing subclasses. The **Service** base class defines various callback methods and the most important are given below. You don't need to implement all the callbacks methods. However, it's important that you understand each one and implement those that ensure your app behaves the way users expect.

|  |  |
| --- | --- |
| **Sr.No.** | **Callback & Description** |
| 1 | **onStartCommand()**  The system calls this method when another component, such as an activity, requests that the service be started, by calling *startService()*. If you implement this method, it is your responsibility to stop the service when its work is done, by calling *stopSelf()* or *stopService()* methods. |
| 2 | **onBind()**  The system calls this method when another component wants to bind with the service by calling *bindService()*. If you implement this method, you must provide an interface that clients use to communicate with the service, by returning an *IBinder* object. You must always implement this method, but if you don't want to allow binding, then you should return *null*. |
| 3 | **onUnbind()**  The system calls this method when all clients have disconnected from a particular interface published by the service. |
| 4 | **onRebind()**  The system calls this method when new clients have connected to the service, after it had previously been notified that all had disconnected in its *onUnbind(Intent)*. |
| 5 | **onCreate()**  The system calls this method when the service is first created using *onStartCommand()* or *onBind()*. This call is required to perform one-time set-up. |
| 6 | **onDestroy()**  The system calls this method when the service is no longer used and is being destroyed. Your service should implement this to clean up any resources such as threads, registered listeners, receivers, etc. |

The following skeleton service demonstrates each of the life cycle methods −

package com.tutorialspoint;

import android.app.Service;

import android.os.IBinder;

import android.content.Intent;

import android.os.Bundle;

public class HelloService extends Service {

/\*\* indicates how to behave if the service is killed \*/

int mStartMode;

/\*\* interface for clients that bind \*/

IBinder mBinder;

/\*\* indicates whether onRebind should be used \*/

boolean mAllowRebind;

/\*\* Called when the service is being created. \*/

@Override

public void onCreate() {

}

/\*\* The service is starting, due to a call to startService() \*/

@Override

public int onStartCommand(Intent intent, int flags, int startId) {

return mStartMode;

}

/\*\* A client is binding to the service with bindService() \*/

@Override

public IBinder onBind(Intent intent) {

return mBinder;

}

/\*\* Called when all clients have unbound with unbindService() \*/

@Override

public boolean onUnbind(Intent intent) {

return mAllowRebind;

}

/\*\* Called when a client is binding to the service with bindService()\*/

@Override

public void onRebind(Intent intent) {

}

/\*\* Called when The service is no longer used and is being destroyed \*/

@Override

public void onDestroy() {

}

}

## Example

This example will take you through simple steps to show how to create your own Android Service. Follow the following steps to modify the Android application we created in *Hello World Example* chapter −

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | You will use Android StudioIDE to create an Android application and name it as *My Application* under a package *com.example.tutorialspoint7.myapplication* as explained in the *Hello World Example* chapter. |
| 2 | Modify main activity file *MainActivity.java* to add *startService()* and *stopService()* methods. |
| 3 | Create a new java file *MyService.java* under the package *com.example.My Application*. This file will have implementation of Android service related methods. |
| 4 | Define your service in *AndroidManifest.xml* file using <service.../> tag. An application can have one or more services without any restrictions. |
| 5 | Modify the default content of *res/layout/activity\_main.xml* file to include two buttons in linear layout. |
| 6 | No need to change any constants in *res/values/strings.xml* file. Android studio take care of string values |
| 7 | Run the application to launch Android emulator and verify the result of the changes done in the application. |

Following is the content of the modified main activity file **MainActivity.java**. This file can include each of the fundamental life cycle methods. We have added *startService()* and *stopService()* methods to start and stop the service.

package com.example.tutorialspoint7.myapplication;

import android.content.Intent;  
  
import android.os.Bundle;  
import android.app.Activity;  
import android.util.Log;  
import android.view.View;  
  
public class MainActivity extends Activity {  
 String msg = "Android : ";  
  
 */\*\* Called when the activity is first created. \*/* @Override  
 public void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.*activity\_main*);  
 Log.*d*(msg, "The Activity onCreate() event");  
 }  
  
 public void startService(View view) {  
 startService(new Intent(getBaseContext(), MyService.class));  
 Log.*d*(msg, "The startService() event");  
 }  
  
 *// Method to stop the service* public void stopService(View view) {  
 stopService(new Intent(getBaseContext(), MyService.class));  
 Log.*d*(msg, "The stopService() event");  
 }  
}

Following is the content of **MyService.java**. This file can have implementation of one or more methods associated with Service based on requirements. For now we are going to implement only two methods *onStartCommand()* and *onDestroy()* −

**package** com.example.david.myapplication;  
  
import android.app.Service;  
import android.content.Intent;  
import android.os.IBinder;import android.widget.Toast;  
import android.util.Log;  
  
*/\*\*  
 \* Created by TutorialsPoint7 on 8/23/2016.  
 \*/*public class MyService extends Service {  
 String msg = "Android : ";@Override  
 public IBinder onBind(Intent intent) {  
 return null;  
 }  
  
 @Override  
 public int onStartCommand(Intent intent, int flags, int startId) {  
 *// Let it continue running until it is stopped.* Log.*d*(msg, "The Service onStartCommand() event");  
 Toast.*makeText*(this, "Service Started", Toast.*LENGTH\_LONG*).show();  
 return *START\_STICKY*;  
  
 }  
  
 @Override  
 public void onDestroy() {  
 super.onDestroy();  
 Log.*d*(msg, "The service onDestroy() event");  
 Toast.*makeText*(this, "Service Destroyed", Toast.*LENGTH\_LONG*).show();  
 }  
}

Following will the modified content of *AndroidManifest.xml* file. Here we have added <service.../> tag to include our service −

<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.tutorialspoint7.myapplication">

<application

android:allowBackup="true"

android:icon="@mipmap/ic\_launcher"

android:label="@string/app\_name"

android:supportsRtl="true"

android:theme="@style/AppTheme">

<activity android:name=".MainActivity">

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

</intent-filter>

</activity>

<service android:name=".MyService" />

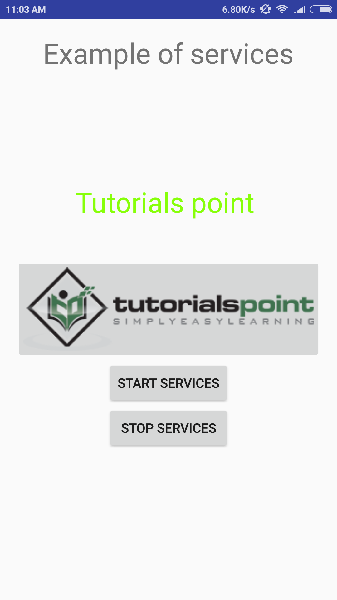
</application>

</manifest>

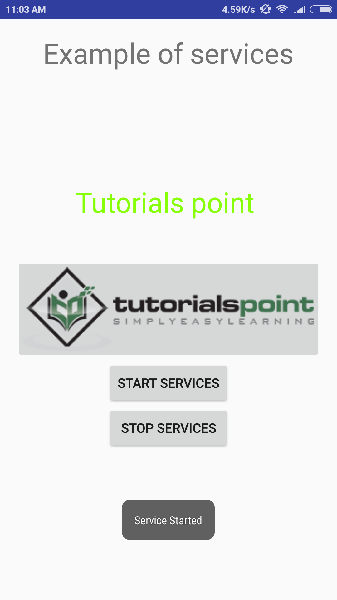
Following will be the content of **res/layout/activity\_main.xml** file to include two buttons −

*<?*xml version="1.0" encoding="utf-8"*?>*<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:tools="http://schemas.android.com/tools" android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 tools:context=".MainActivity">  
  
 <TextView  
 android:id="@+id/textView1"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="Example of services"  
 android:layout\_alignParentTop="true"  
 android:layout\_centerHorizontal="true"  
 android:textSize="30dp" />  
  
 <Button  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:id="@+id/button2"  
 android:text="Start Services"  
 android:onClick="startService"  
 android:layout\_below="@+id/textView1"  
 android:layout\_centerHorizontal="true" />  
  
 <Button  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="Stop Services"  
 android:id="@+id/button"  
 android:onClick="stopService"  
 android:layout\_below="@+id/button2"  
 android:layout\_alignLeft="@+id/button2"  
 android:layout\_alignStart="@+id/button2"  
 android:layout\_alignRight="@+id/button2"  
 android:layout\_alignEnd="@+id/button2" />  
  
</RelativeLayout>

Let's try to run our modified **Hello World!** application we just modified. I assume you had created your **AVD** while doing environment setup. To run the app from Android studio, open one of your project's activity files and click Run Android StudioRun Icon icon from the tool bar. Android Studio installs the app on your AVD and starts it and if everything is fine with your set-up and application, it will display following Emulator window −



Now to start your service, let's click on **Start Service** button, this will start the service and as per our programming in *onStartCommand()* method, a message *Service Started* will appear on the bottom of the the simulator as follows −



To stop the service, you can click the Stop Service button.

# Android - Broadcast Receivers

**Broadcast Receivers**simply respond to broadcast messages from other applications or from the system itself. These messages are sometime called events or intents. For example, applications can also initiate broadcasts to let other applications know that some data has been downloaded to the device and is available for them to use, so this is broadcast receiver who will intercept this communication and will initiate appropriate action.

There are following two important steps to make BroadcastReceiver works for the system broadcasted intents −

* Creating the Broadcast Receiver.
* Registering Broadcast Receiver

There is one additional steps in case you are going to implement your custom intents then you will have to create and broadcast those intents.

## Creating the Broadcast Receiver

A broadcast receiver is implemented as a subclass of **BroadcastReceiver**class and overriding the onReceive() method where each message is received as a **Intent** object parameter.

public class MyReceiver extends BroadcastReceiver {

@Override

public void onReceive(Context context, Intent intent) {

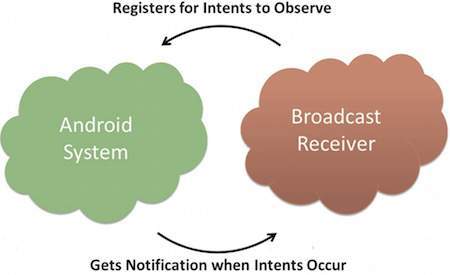
Toast.makeText(context, "Intent Detected.", Toast.LENGTH\_LONG).show();

}

}

## Registering Broadcast Receiver

An application listens for specific broadcast intents by registering a broadcast receiver in *AndroidManifest.xml* file. Consider we are going to register *MyReceiver* for system generated event ACTION\_BOOT\_COMPLETED which is fired by the system once the Android system has completed the boot process.



#### BROADCAST-RECEIVER

<application

android:icon="@drawable/ic\_launcher"

android:label="@string/app\_name"

android:theme="@style/AppTheme" >

<receiver android:name="MyReceiver">

<intent-filter>

<action android:name="android.intent.action.BOOT\_COMPLETED">

</action>

</intent-filter>

</receiver>

</application>

Now whenever your Android device gets booted, it will be intercepted by BroadcastReceiver *MyReceiver* and implemented logic inside *onReceive()* will be executed.

There are several system generated events defined as final static fields in the **Intent** class. The following table lists a few important system events.

|  |  |
| --- | --- |
| **Sr.No** | **Event Constant & Description** |
| 1 | **android.intent.action.BATTERY\_CHANGED**  Sticky broadcast containing the charging state, level, and other information about the battery. |
| 2 | **android.intent.action.BATTERY\_LOW**  Indicates low battery condition on the device. |
| 3 | **android.intent.action.BATTERY\_OKAY**  Indicates the battery is now okay after being low. |
| 4 | **android.intent.action.BOOT\_COMPLETED**  This is broadcast once, after the system has finished booting. |
| 5 | **android.intent.action.BUG\_REPORT**  Show activity for reporting a bug. |
| 6 | **android.intent.action.CALL**  Perform a call to someone specified by the data. |
| 7 | **android.intent.action.CALL\_BUTTON**  The user pressed the "call" button to go to the dialer or other appropriate UI for placing a call. |
| 8 | **android.intent.action.DATE\_CHANGED**  The date has changed. |
| 9 | **android.intent.action.REBOOT**  Have the device reboot. |

## Broadcasting Custom Intents

If you want your application itself should generate and send custom intents then you will have to create and send those intents by using the *sendBroadcast()* method inside your activity class. If you use the *sendStickyBroadcast(Intent)* method, the Intent is **sticky**, meaning the *Intent*you are sending stays around after the broadcast is complete.

public void broadcastIntent(View view) {

Intent intent = new Intent();

intent.setAction("com.tutorialspoint.CUSTOM\_INTENT");

sendBroadcast(intent);

}

This intent *com.tutorialspoint.CUSTOM\_INTENT* can also be registered in similar way as we have regsitered system generated intent.

<application

android:icon="@drawable/ic\_launcher"

android:label="@string/app\_name"

android:theme="@style/AppTheme" >

<receiver android:name=".MyReceiver">

<intent-filter>

<action android:name="com.tutorialspoint.CUSTOM\_INTENT">

</action>

</intent-filter>

</receiver>

</application>

## Example

This example will explain you how to create *BroadcastReceiver* to intercept custom intent. Once you are familiar with custom intent, then you can program your application to intercept system generated intents. So let's follow the following steps to modify the Android application we created in *Hello World Example* chapter −

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | You will use Android studio to create an Android application and name it as *My Application* under a package *com.example.tutorialspoint7.myapplication* as explained in the *Hello World Example* chapter. |
| 2 | Modify main activity file *MainActivity.java* to add *broadcastIntent()*method. |
| 3 | Create a new java file called *MyReceiver.java* under the package *com.example.tutorialspoint7.myapplication* to define a BroadcastReceiver. |
| 4 | An application can handle one or more custom and system intents without any restrictions. Every intent you want to intercept must be registered in your *AndroidManifest.xml* file using <receiver.../> tag |
| 5 | Modify the default content of *res/layout/activity\_main.xml* file to include a button to broadcast intent. |
| 6 | No need to modify the string file, Android studio take care of string.xml file. |
| 7 | Run the application to launch Android emulator and verify the result of the changes done in the application. |

Following is the content of the modified main activity file **MainActivity.java**. This file can include each of the fundamental life cycle methods. We have added *broadcastIntent()* method to broadcast a custom intent.

package com.example.tutorialspoint7.myapplication;

import android.app.Activity;

import android.content.Intent;

import android.os.Bundle;

import android.view.View;

public class MainActivity extends Activity {

/\*\* Called when the activity is first created. \*/

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

}

// broadcast a custom intent.

public void broadcastIntent(View view){

Intent intent = new Intent();

intent.setAction("com.tutorialspoint.CUSTOM\_INTENT"); sendBroadcast(intent);

}

}

Following is the content of **MyReceiver.java**:

package com.example.tutorialspoint7.myapplication;

import android.content.BroadcastReceiver;

import android.content.Context;

import android.content.Intent;

import android.widget.Toast;

/\*\*

\* Created by TutorialsPoint7 on 8/23/2016.

\*/

public class MyReceiver extends BroadcastReceiver{

@Override

public void onReceive(Context context, Intent intent) {

Toast.makeText(context, "Intent Detected.", Toast.LENGTH\_LONG).show();

}

}

Following will the modified content of *AndroidManifest.xml* file. Here we have added <receiver.../> tag to include our service:

<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.tutorialspoint7.myapplication">

<application

android:allowBackup="true"

android:icon="@mipmap/ic\_launcher"

android:label="@string/app\_name"

android:supportsRtl="true"

android:theme="@style/AppTheme">

<activity android:name=".MainActivity">

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

</intent-filter>

</activity>

<receiver android:name="MyReceiver">

<intent-filter>

<action android:name="com.tutorialspoint.CUSTOM\_INTENT">

</action>

</intent-filter>

</receiver>

</application>

</manifest>

Following will be the content of **res/layout/activity\_main.xml** file to include a button to broadcast our custom intent −

<RelativeLayout

xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:tools="http://schemas.android.com/tools"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:paddingLeft="@dimen/activity\_horizontal\_margin"

android:paddingRight="@dimen/activity\_horizontal\_margin"

android:paddingTop="@dimen/activity\_vertical\_margin"

android:paddingBottom="@dimen/activity\_vertical\_margin"

tools:context=".MainActivity">

<TextView

android:id="@+id/textView1"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Example of Broadcast"

android:layout\_alignParentTop="true"

android:layout\_centerHorizontal="true"

android:textSize="30dp" />

<TextView

android:id="@+id/textView2"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Tutorials point "

android:textColor="#ff87ff09"

android:textSize="30dp"

android:layout\_above="@+id/imageButton"

android:layout\_centerHorizontal="true"

android:layout\_marginBottom="40dp" />

<ImageButton

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/imageButton"

android:src="@drawable/abc"

android:layout\_centerVertical="true"

android:layout\_centerHorizontal="true" />

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/button2"

android:text="Broadcast Intent"

android:onClick="broadcastIntent"

android:layout\_below="@+id/imageButton"

android:layout\_centerHorizontal="true" />

</RelativeLayout>

Let's try to run our modified **Hello World!** application we just modified. I assume you had created your **AVD** while doing environment set-up. To run the app from Android studio, open one of your project's activity files and click Run Eclipse Run Icon icon from the tool bar. Android Studio installs the app on your AVD and starts it and if everything is fine with your set-up and application, it will display following Emulator window −



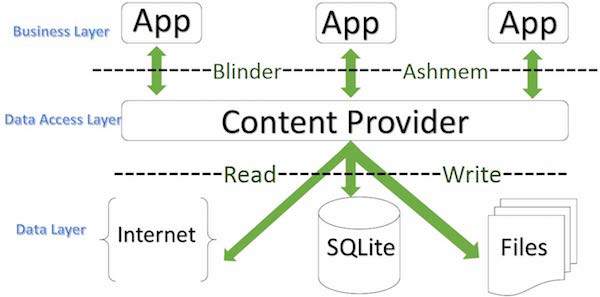
Now to broadcast our custom intent, let's click on **Broadcast Intent** button, this will broadcast our custom intent *"com.tutorialspoint.CUSTOM\_INTENT"*which will be intercepted by our registered BroadcastReceiver i.e. MyReceiver and as per our implemented logic a toast will appear on the bottom of the the simulator as follows −



You can try implementing other BroadcastReceiver to intercept system generated intents like system boot up, date changed, low battery etc.

# Android - Content Providers

A content provider component supplies data from one application to others on request. Such requests are handled by the methods of the ContentResolver class. A content provider can use different ways to store its data and the data can be stored in a database, in files, or even over a network.



#### CONTENTPROVIDER

**sometimes it is required to share data across applications. This is where content providers become very useful.**

Content providers let you centralize content in one place and have many different applications access it as needed. A content provider behaves very much like a database where you can query it, edit its content, as well as add or delete content using insert(), update(), delete(), and query() methods. In most cases this data is stored in an **SQlite** database.

A content provider is implemented as a subclass of **ContentProvider** class and must implement a standard set of APIs that enable other applications to perform transactions.

public class My Application extends ContentProvider {

}

## Content URIs

To query a content provider, you specify the query string in the form of a URI which has following format −

<prefix>://<authority>/<data\_type>/<id>

Here is the detail of various parts of the URI −

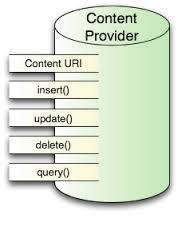
|  |  |
| --- | --- |
| **Sr.No** | **Part & Description** |
| 1 | **prefix**  This is always set to content:// |
| 2 | **authority**  This specifies the name of the content provider, for example *contacts*, *browser* etc. For third-party content providers, this could be the fully qualified name, such as *com.tutorialspoint.statusprovider* |
| 3 | **data\_type**  This indicates the type of data that this particular provider provides. For example, if you are getting all the contacts from the *Contacts*content provider, then the data path would be *people* and URI would look like this *content://contacts/people* |
| 4 | **id**  This specifies the specific record requested. For example, if you are looking for contact number 5 in the Contacts content provider then URI would look like this *content://contacts/people/5*. |

## Create Content Provider

This involves number of simple steps to create your own content provider.

* First of all you need to create a Content Provider class that extends the *ContentProviderbaseclass.*
* Second, you need to define your content provider URI address which will be used to access the content.
* Next you will need to create your own database to keep the content. Usually, Android uses SQLite database and framework needs to override *onCreate()* method which will use SQLite Open Helper method to create or open the provider's database. When your application is launched, the *onCreate()* handler of each of its Content Providers is called on the main application thread.
* Next you will have to implement Content Provider queries to perform different database specific operations.
* Finally register your Content Provider in your activity file using <provider> tag.

Here is the list of methods which you need to override in Content Provider class to have your Content Provider working −



#### CONTENTPROVIDER

* **onCreate()** This method is called when the provider is started.
* **query()** This method receives a request from a client. The result is returned as a Cursor object.
* **insert()**This method inserts a new record into the content provider.
* **delete()** This method deletes an existing record from the content provider.
* **update()** This method updates an existing record from the content provider.
* **getType()** This method returns the MIME type of the data at the given URI.

## Example

This example will explain you how to create your own *ContentProvider*. So let's follow the following steps to similar to what we followed while creating *Hello World Example*−

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | You will use Android StudioIDE to create an Android application and name it as *My Application* under a package *com.example.MyApplication*, with blank Activity. |
| 2 | Modify main activity file *MainActivity.java* to add two new methods *onClickAddName()* and *onClickRetrieveStudents()*. |
| 3 | Create a new java file called *StudentsProvider.java* under the package *com.example.MyApplication* to define your actual provider and associated methods. |
| 4 | Register your content provider in your *AndroidManifest.xml* file using <provider.../> tag |
| 5 | Modify the default content of *res/layout/activity\_main.xml* file to include a small GUI to add students records. |
| 6 | No need to change string.xml.Android studio take care of string.xml file. |
| 7 | Run the application to launch Android emulator and verify the result of the changes done in the application. |

Following is the content of the modified main activity file **src/com.example.MyApplication/MainActivity.java**. This file can include each of the fundamental life cycle methods. We have added two new methods *onClickAddName()* and *onClickRetrieveStudents()* to handle user interaction with the application.

package com.example.MyApplication;

import android.net.Uri;

import android.os.Bundle;

import android.app.Activity;

import android.content.ContentValues;

import android.content.CursorLoader;

import android.database.Cursor;

import android.view.Menu;

import android.view.View;

import android.widget.EditText;

import android.widget.Toast;

public class MainActivity extends Activity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

}

public void onClickAddName(View view) {

// Add a new student record

ContentValues values = new ContentValues();

values.put(StudentsProvider.NAME,

((EditText)findViewById(R.id.editText2)).getText().toString());

values.put(StudentsProvider.GRADE,

((EditText)findViewById(R.id.editText3)).getText().toString());

Uri uri = getContentResolver().insert(

StudentsProvider.CONTENT\_URI, values);

Toast.makeText(getBaseContext(),

uri.toString(), Toast.LENGTH\_LONG).show();

}

public void onClickRetrieveStudents(View view) {

// Retrieve student records

String URL = "content://com.example.MyApplication.StudentsProvider";

Uri students = Uri.parse(URL);

Cursor c = managedQuery(students, null, null, null, "name");

if (c.moveToFirst()) {

do{

Toast.makeText(this,

c.getString(c.getColumnIndex(StudentsProvider.\_ID)) +

", " + c.getString(c.getColumnIndex( StudentsProvider.NAME)) +

", " + c.getString(c.getColumnIndex( StudentsProvider.GRADE)),

Toast.LENGTH\_SHORT).show();

} while (c.moveToNext());

}

}

}

Create new file StudentsProvider.java under *com.example.MyApplication*package and following is the content of **src/com.example.MyApplication/StudentsProvider.java** −

package com.example.MyApplication;

import java.util.HashMap;

import android.content.ContentProvider;

import android.content.ContentUris;

import android.content.ContentValues;

import android.content.Context;

import android.content.UriMatcher;

import android.database.Cursor;

import android.database.SQLException;

import android.database.sqlite.SQLiteDatabase;

import android.database.sqlite.SQLiteOpenHelper;

import android.database.sqlite.SQLiteQueryBuilder;

import android.net.Uri;

import android.text.TextUtils;

public class StudentsProvider extends ContentProvider {

static final String PROVIDER\_NAME = "com.example.MyApplication.StudentsProvider";

static final String URL = "content://" + PROVIDER\_NAME + "/students";

static final Uri CONTENT\_URI = Uri.parse(URL);

static final String \_ID = "\_id";

static final String NAME = "name";

static final String GRADE = "grade";

private static HashMap<String, String> STUDENTS\_PROJECTION\_MAP;

static final int STUDENTS = 1;

static final int STUDENT\_ID = 2;

static final UriMatcher uriMatcher;

static{

uriMatcher = new UriMatcher(UriMatcher.NO\_MATCH);

uriMatcher.addURI(PROVIDER\_NAME, "students", STUDENTS);

uriMatcher.addURI(PROVIDER\_NAME, "students/#", STUDENT\_ID);

}

/\*\*

\* Database specific constant declarations

\*/

private SQLiteDatabase db;

static final String DATABASE\_NAME = "College";

static final String STUDENTS\_TABLE\_NAME = "students";

static final int DATABASE\_VERSION = 1;

static final String CREATE\_DB\_TABLE =

" CREATE TABLE " + STUDENTS\_TABLE\_NAME +

" (\_id INTEGER PRIMARY KEY AUTOINCREMENT, " +

" name TEXT NOT NULL, " +

" grade TEXT NOT NULL);";

/\*\*

\* Helper class that actually creates and manages

\* the provider's underlying data repository.

\*/

private static class DatabaseHelper extends SQLiteOpenHelper {

DatabaseHelper(Context context){

super(context, DATABASE\_NAME, null, DATABASE\_VERSION);

}

@Override

public void onCreate(SQLiteDatabase db) {

db.execSQL(CREATE\_DB\_TABLE);

}

@Override

public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {

db.execSQL("DROP TABLE IF EXISTS " + STUDENTS\_TABLE\_NAME);

onCreate(db);

}

}

@Override

public boolean onCreate() {

Context context = getContext();

DatabaseHelper dbHelper = new DatabaseHelper(context);

/\*\*

\* Create a write able database which will trigger its

\* creation if it doesn't already exist.

\*/

db = dbHelper.getWritableDatabase();

return (db == null)? false:true;

}

@Override

public Uri insert(Uri uri, ContentValues values) {

/\*\*

\* Add a new student record

\*/

long rowID = db.insert( STUDENTS\_TABLE\_NAME, "", values);

/\*\*

\* If record is added successfully

\*/

if (rowID > 0) {

Uri \_uri = ContentUris.withAppendedId(CONTENT\_URI, rowID);

getContext().getContentResolver().notifyChange(\_uri, null);

return \_uri;

}

throw new SQLException("Failed to add a record into " + uri);

}

@Override

public Cursor query(Uri uri, String[] projection,

String selection,String[] selectionArgs, String sortOrder) {

SQLiteQueryBuilder qb = new SQLiteQueryBuilder();

qb.setTables(STUDENTS\_TABLE\_NAME);

switch (uriMatcher.match(uri)) {

case STUDENTS:

qb.setProjectionMap(STUDENTS\_PROJECTION\_MAP);

break;

case STUDENT\_ID:

qb.appendWhere( \_ID + "=" + uri.getPathSegments().get(1));

break;

default:

}

if (sortOrder == null || sortOrder == ""){

/\*\*

\* By default sort on student names

\*/

sortOrder = NAME;

}

Cursor c = qb.query(db, projection, selection,

selectionArgs,null, null, sortOrder);

/\*\*

\* register to watch a content URI for changes

\*/

c.setNotificationUri(getContext().getContentResolver(), uri);

return c;

}

@Override

public int delete(Uri uri, String selection, String[] selectionArgs) {

int count = 0;

switch (uriMatcher.match(uri)){

case STUDENTS:

count = db.delete(STUDENTS\_TABLE\_NAME, selection, selectionArgs);

break;

case STUDENT\_ID:

String id = uri.getPathSegments().get(1);

count = db.delete( STUDENTS\_TABLE\_NAME, \_ID + " = " + id +

(!TextUtils.isEmpty(selection) ? "

AND (" + selection + ')' : ""), selectionArgs);

break;

default:

throw new IllegalArgumentException("Unknown URI " + uri);

}

getContext().getContentResolver().notifyChange(uri, null);

return count;

}

@Override

public int update(Uri uri, ContentValues values,

String selection, String[] selectionArgs) {

int count = 0;

switch (uriMatcher.match(uri)) {

case STUDENTS:

count = db.update(STUDENTS\_TABLE\_NAME, values, selection, selectionArgs);

break;

case STUDENT\_ID:

count = db.update(STUDENTS\_TABLE\_NAME, values,

\_ID + " = " + uri.getPathSegments().get(1) +

(!TextUtils.isEmpty(selection) ? "

AND (" +selection + ')' : ""), selectionArgs);

break;

default:

throw new IllegalArgumentException("Unknown URI " + uri );

}

getContext().getContentResolver().notifyChange(uri, null);

return count;

}

@Override

public String getType(Uri uri) {

switch (uriMatcher.match(uri)){

/\*\*

\* Get all student records

\*/

case STUDENTS:

return "vnd.android.cursor.dir/vnd.example.students";

/\*\*

\* Get a particular student

\*/

case STUDENT\_ID:

return "vnd.android.cursor.item/vnd.example.students";

default:

throw new IllegalArgumentException("Unsupported URI: " + uri);

}

}

}

Following will the modified content of *AndroidManifest.xml* file. Here we have added <provider.../> tag to include our content provider:

<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.MyApplication">

<application

android:allowBackup="true"

android:icon="@mipmap/ic\_launcher"

android:label="@string/app\_name"

android:supportsRtl="true"

android:theme="@style/AppTheme">

<activity android:name=".MainActivity">

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

</intent-filter>

</activity>

<provider android:name="StudentsProvider"

android:authorities="com.example.MyApplication.StudentsProvider"/>

</application>

</manifest>

Following will be the content of **res/layout/activity\_main.xml** file−

<?xml version="1.0" encoding="utf-8"?>

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:tools="http://schemas.android.com/tools"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:paddingBottom="@dimen/activity\_vertical\_margin"

android:paddingLeft="@dimen/activity\_horizontal\_margin"

android:paddingRight="@dimen/activity\_horizontal\_margin"

android:paddingTop="@dimen/activity\_vertical\_margin"

tools:context="com.example.MyApplication.MainActivity">

<TextView

android:id="@+id/textView1"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Content provider"

android:layout\_alignParentTop="true"

android:layout\_centerHorizontal="true"

android:textSize="30dp" />

<TextView

android:id="@+id/textView2"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Tutorials point "

android:textColor="#ff87ff09"

android:textSize="30dp"

android:layout\_below="@+id/textView1"

android:layout\_centerHorizontal="true" />

<ImageButton

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/imageButton"

android:src="@drawable/abc"

android:layout\_below="@+id/textView2"

android:layout\_centerHorizontal="true" />

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/button2"

android:text="Add Name"

android:layout\_below="@+id/editText3"

android:layout\_alignRight="@+id/textView2"

android:layout\_alignEnd="@+id/textView2"

android:layout\_alignLeft="@+id/textView2"

android:layout\_alignStart="@+id/textView2"

android:onClick="onClickAddName"/>

<EditText

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/editText"

android:layout\_below="@+id/imageButton"

android:layout\_alignRight="@+id/imageButton"

android:layout\_alignEnd="@+id/imageButton" />

<EditText

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/editText2"

android:layout\_alignTop="@+id/editText"

android:layout\_alignLeft="@+id/textView1"

android:layout\_alignStart="@+id/textView1"

android:layout\_alignRight="@+id/textView1"

android:layout\_alignEnd="@+id/textView1"

android:hint="Name"

android:textColorHint="@android:color/holo\_blue\_light" />

<EditText

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/editText3"

android:layout\_below="@+id/editText"

android:layout\_alignLeft="@+id/editText2"

android:layout\_alignStart="@+id/editText2"

android:layout\_alignRight="@+id/editText2"

android:layout\_alignEnd="@+id/editText2"

android:hint="Grade"

android:textColorHint="@android:color/holo\_blue\_bright" />

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Retrive student"

android:id="@+id/button"

android:layout\_below="@+id/button2"

android:layout\_alignRight="@+id/editText3"

android:layout\_alignEnd="@+id/editText3"

android:layout\_alignLeft="@+id/button2"

android:layout\_alignStart="@+id/button2"

android:onClick="onClickRetrieveStudents"/>

</RelativeLayout>

Make sure you have following content of **res/values/strings.xml** file:

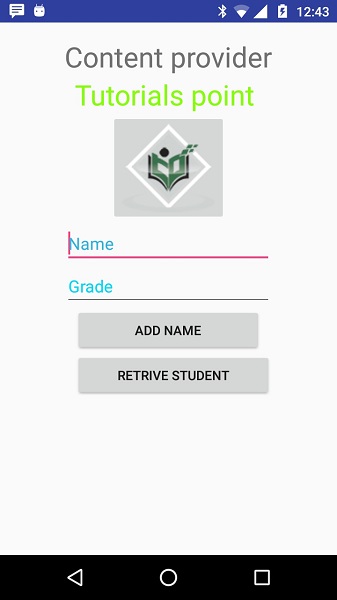
<?xml version="1.0" encoding="utf-8"?>

<resources>

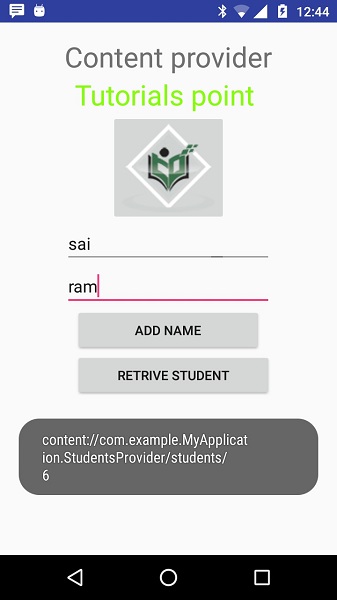
<string name="app\_name">My Application</string>

</resources>;

Let's try to run our modified **My Application** application we just created. I assume you had created your **AVD** while doing environment set-up. To run the app from Android Studio IDE, open one of your project's activity files and click Run Android StudioRun Icon icon from the tool bar. Android Studio installs the app on your AVD and starts it and if everything is fine with your set-up and application, it will display following Emulator window, be patience because it may take sometime based on your computer speed −



Now let's enter student **Name** and **Grade** and finally click on **Add Name**button, this will add student record in the database and will flash a message at the bottom showing ContentProvider URI along with record number added in the database. This operation makes use of our **insert()** method. Let's repeat this process to add few more students in the database of our content provider.



Once you are done with adding records in the database, now its time to ask ContentProvider to give us those records back, so let's click **Retrieve Students** button which will fetch and display all the records one by one which is as per our the implementation of our **query()** method.

You can write activities against update and delete operations by providing callback functions in **MainActivity.java** file and then modify user interface to have buttons for update and deleted operations in the same way as we have done for add and read operations.

This way you can use existing Content Provider like Address Book or you can use Content Provider concept in developing nice database oriented applications where you can perform all sort of database operations like read, write, update and delete as explained above in the example.