## Broadcast Component in Android

## An Overview of Broadcast Intents

Broadcast intents are Intent objects that are broadcast via a call to the sendBroadcast(), sendStickyBroadcast() or sendOrderedBroadcast() methods of the Activity class (the latter being used when results are required from the broadcast). In addition to providing a messaging and event system between application components, broadcast intents are also used by the Android system to notify interested applications about key system events (such as the external power supply or headphones being connected or disconnected).

When a broadcast intent is created, it must include an action string in addition to optional data and a category string. As with standard intents, data is added to a broadcast intent using key-value pairs in conjunction with the putExtra() method of the intent object.

## An Overview of Broadcast Receivers

An application listens for specific broadcast intents by registering a broadcast receiver. Broadcast receivers are implemented by extending the Android BroadcastReceiver class and overriding the onReceive() method. The broadcast receiver may then be registered, either within code (for example within an activity), or within a manifest file. Part of the registration implementation involves the creation of intent filters to indicate the specific broadcast intents the receiver is required to listen for. This is achieved by referencing the action string of the broadcast intent. When a matching broadcast is detected, the onReceive() method of the broadcast receiver is called, at which point the method has 5 seconds within which to perform any necessary tasks before returning. It is important to note that a broadcast receiver does not need to be running all the time. In the event that a matching intent is detected, the Android runtime system will automatically start up the broadcast receiver before calling the onReceive() method.

## Example

MainActivity

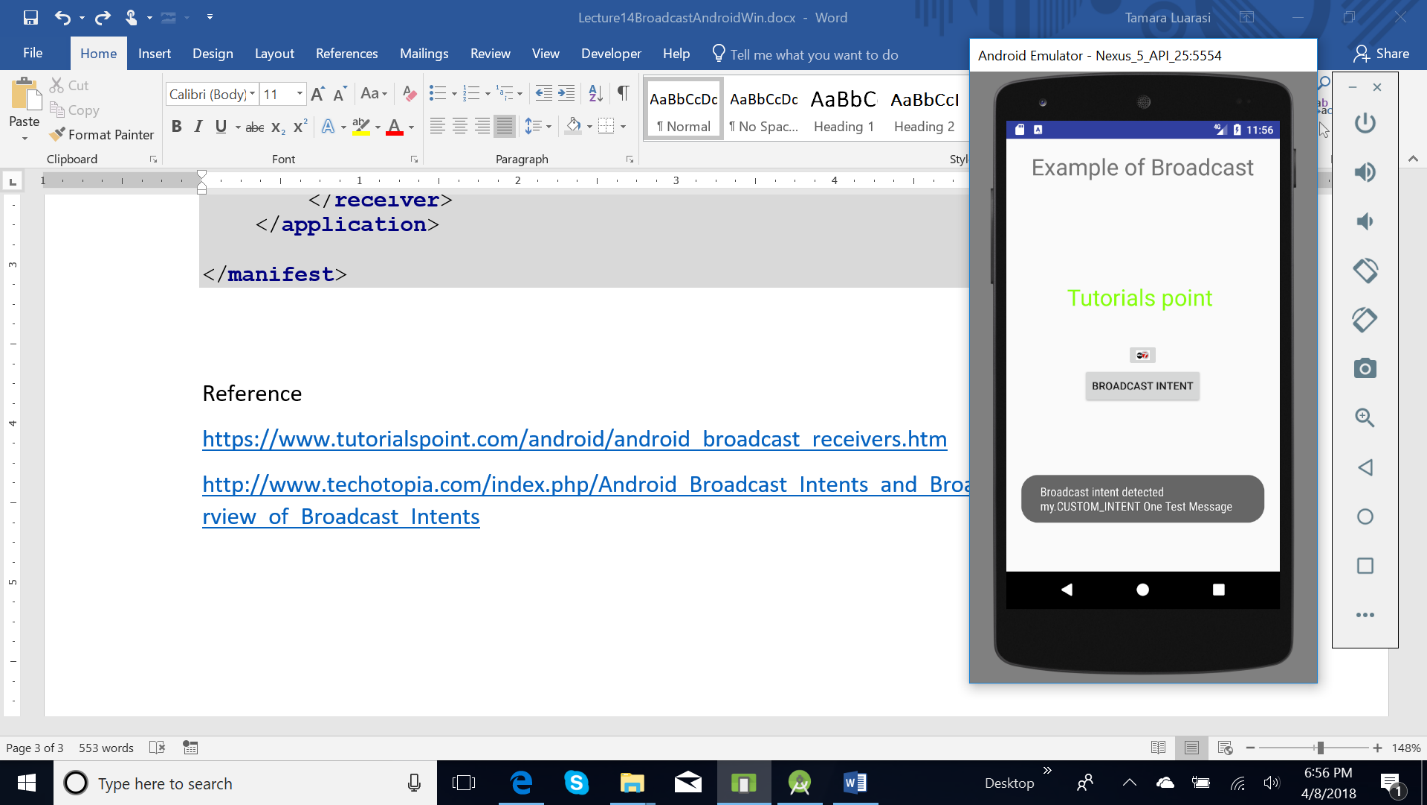
**import** android.app.Activity;  
**import** android.content.Intent;  
**import** android.os.Bundle;  
**import** android.view.View;  
**public class** MainActivity **extends** Activity {  
 */\*\* Called when the activity is first created. \*/* @Override  
 **public void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.***activity\_main***);  
 } *// broadcast a custom intent.* **public void** broadcastIntent(View view){  
 Intent intent = **new** Intent();  
 intent.setAction(**"my.CUSTOM\_INTENT"**);  
 intent.addFlags(Intent.***FLAG\_INCLUDE\_STOPPED\_PACKAGES***);  
 intent.putExtra(**"Message"**,**"One Test Message"**);  
 sendBroadcast(intent);  
 }  
}

The above code would successfully launch the corresponding broadcast receiver on a device running an Android version earlier than 3.0. On more recent versions of Android, however, the intent would not be received by the broadcast receiver. This is because Android 3.0 introduced a launch control security measure that prevents components of stopped applications from being launched via an intent. An application is considered to be in a stopped state if the application has either just been installed and not previously launched, or been manually stopped by the user using the application manager on the device. To get around this, however, a flag can be added to the intent before it is sent to indicate that the intent is to be allowed to start a component of a stopped application. This flag is FLAG\_INCLUDE\_STOPPED\_PACKAGES.

Broadcast receivers  
**import** android.content.BroadcastReceiver;  
**import** android.content.Context;  
**import** android.content.Intent;  
**import** android.widget.Toast;  
  
**public class** MyReceiver **extends** BroadcastReceiver{  
 @Override  
 **public void** onReceive(Context context, Intent intent) {  
 String message = **"Broadcast intent detected "** + intent.getAction();  
  
 Toast.*makeText*(context, message+**" "**+intent.getStringExtra(**"Message"**),  
 Toast.***LENGTH\_LONG***).show();  
  
 }  
}

Broadcast Registration

*<?***xml version="1.0" encoding="utf-8"***?>*<**manifest xmlns:android="http://schemas.android.com/apk/res/android"  
 package="com.example.user.broadcastexample"**>  
  
 <**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="my.CUSTOM\_INTENT"**>  
 </**action**>  
 </**intent-filter**>  
  
 </**receiver**>  
 </**application**>  
  
</**manifest**>



## System Broadcasted Intents

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 **android.intent.action.ACTION\_SHUTDOWN**

which is fired by the system once the Android system has completed the boot process.

<**intent-filter**>  
 <**action android:name="android.intent.action.BATTERY\_LOW"**>  
 </**action**>  
</**intent-filter**>

Or by code

IntentFilter filter = new IntentFilter(); intentFilter.addAction(

getPackageName() + " **android.intent.action.BATTERY\_LOW** ");

MyReceiver myReceiver = new MyReceiver(); registerReceiver(myReceiver, filter);

# Services Component in Android

A Service is an application component that can perform long-running operations in the background, and it does not provide a user interface. Another application component can start a service, and it continues to run in the background even if the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform interprocess communication (IPC). For example, a service can handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

We already know the lifecycle of an Android Activity; it can be started, stopped, destroyed if the system resources become too low and maybe can be recreated. On the contrary, a service in Android is designed to have a longer life.

An **Android Service can be started from an Activity, from a Broadcast receiver** and other services too.

It is important to notice that using Android Service we do not create automatically new threads, so if we implement a simple logic inside our Service, that doesn’t require long time processing, we don’t need to run it a separate thread. If we have to implement complex logic, with longtime processing, we have to take care of creating a new thread, otherwise, the Android service runs on the main thread and it could cause ANR problem.  
A service in Android is used for two main reasons:

* Implement multi-task
* Enable Inter-Process-Communication (IPC)

A typical example of the first case is an app required to download data from a remote server. In this case, we can have Activity that interacts with a user and starts a service that accomplishes the work in the background while the user can use the app. When the Android service finishes, it sends a message to the user.  
In the second case, we want to “share” some common functions so that different app can re-use them. For example, we can suppose we have an Android Service that sends an email and we want to share this service with several apps so that we don’t have to rewrite the same code. In this case, we can use IPC so that the service exposes a “remote” interface that can be called by other apps.  
In this post, we cover the first case: we have a local service, local means that the service can be seen just inside our APK.

## 2. Bound and Unbound Service

*Bound Service*

Service which call indefinitely in between activity. An Android component may bind itself to a Service using bindservice (). A bound service would run as long as the other application components are bound to it. As soon as they unbind, the service destroys itself.

#### Unbound Service

Service which call at the life span of calling activity. In this case, an application component starts the service, and it would continue to run in the background, even if the original component that initiated it is destroyed. For instance, when started, a service would continue to play music in the background indefinitely.

## 3. Android Service Lifecycle

A service can be run by the system, If someone calls Context.startService() or bindService() method.

#### onStartCommand()

This method is called when the service be started, by calling startService(). Once this method executes, the service is started and can run in the background indefinitely. If you implement this, it is your responsibility to stop the service when its work is done, by calling stopSelf() or stopService().  If you are defining your service as, bounded service then you don’t need to implement this method.

#### onBind()

You need to override this method, only if you are defining your service as bounded service. This method is called, when another component wants to bind with the service by calling bindService(). In your implementation of this method, you must provide an interface that clients use to communicate with the service, by returning an IBinder. You must always implement this method, but if you don’t want to allow binding, then you should return null.

#### onCreate()

This method is called while the service is first created. Here all the service initialization is done. This method is never called again.

#### onDestroy()

The system calls this method when the service is no longer used and is being destroyed. This method is used to, clean up any resources such as threads, registered listeners, receivers, etc. This is the last call the service receives.

## Creating a Android Service

Create a new class and extend it from android.app.Service. You need to override onStartCommand(), onBind(), onCreate() and onDestroy() method to handle the service lifecycle.

public class TestService extends Service {

@Override

public void onCreate() {

super.onCreate();

}

@Override

public void onDestroy() {

super.onDestroy();

}

@Override

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

return super.onStartCommand(intent, flags, startId);

}

@Override

public IBinder onBind(Intent arg0) {

return null;

}

}

## Start and Stop Service in Android

As we know a Service has to be started and eventually stopped so that it can accomplish its task. We can suppose to start it from an activity and we could pass to the service some information using Intent. We can suppose that our Activity has two buttons one to start and one to stop the Service:

**public void** startService(View view) {  
 startService(**new** Intent(getBaseContext(), MyService.**class**));  
}  
  
*// Method to stop the service***public void** stopService(View view) {  
 stopService(**new** Intent(getBaseContext(), MyService.**class**));  
}

and the code of the MyService

**import** android.app.Service;  
**import** android.content.Intent;  
**import** android.os.IBinder;  
**import** android.support.annotation.Nullable;  
**import** android.util.Log;  
**import** android.widget.Toast;  
  
  
**public class** MyService **extends** Service {  
 **private boolean isRunning** = **false**;  
 **private static final** String ***TAG*** = **"HelloService"**;  
 @Override  
 **public void** onCreate() {  
 Log.*i*(***TAG***, **"Service onCreate"**);  
 **isRunning** = **true**;  
 }  
  
 @Nullable  
 @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.* Toast.*makeText*(**this**, **"Service Started"**, Toast.***LENGTH\_LONG***).show();  
  
 *//Creating new thread for my service  
 //Always write your long running tasks*

*//in a separate thread, to avoid ANR* **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
  
  
 *//Your logic that service will perform will be placed here  
 //In this example we are just looping and waits for*

*// 1000 milliseconds in each loop.* **for** (**int** i = 0; i < 5; i++) {  
 **try** {  
 Thread.*sleep*(1000);  
 } **catch** (Exception e) {  
 }  
  
 **if**(**isRunning**){  
 Log.*i*(***TAG***, **"Service running"**);  
 }  
 }  
  
 *//Stop service once it finishes its task* stopSelf();  
 }  
 }).start();  
  
 **return** Service.***START\_STICKY***;  
  
 }  
  
 @Override  
 **public void** onDestroy() {  
 **super**.onDestroy();  
 Toast.*makeText*(**this**, **"Service Destroyed"**, Toast.***LENGTH\_LONG***).show();  
 }  
}

Note:

Both codes are only relevant when the phone runs out of memory and kills the service before it finishes executing. START\_STICKY tells the OS to recreate the service after it has enough memory and call onStartCommand() again with a null intent. START\_NOT\_STICKY tells the OS to not bother recreating the service again. There is also a third code START\_REDELIVER\_INTENT that tells the OS to recreate the service and redeliver the same intent to onStartCommand().

**Reference**

<https://www.tutorialspoint.com/android/android_broadcast_receivers.htm>

<http://www.techotopia.com/index.php/Android_Broadcast_Intents_and_Broadcast_Receivers#An_Overview_of_Broadcast_Intents>

http://stacktips.com/tutorials/android/android-service-example