SERVICES AND BROADCAST RECEIVERS

COMP 41690

DAVID COYLE > D.COYLE@UCD.IE

BASIC BUILDING BLOCKS

Activities: UI building blocks

Content Providers: handling and storing data

Services: background processes

Broadcase receivers: handle system wide messages

Three of the four component types—activities, services, and broadcast receivers—are activated by an asynchronous message called an **intent**. Intents bind individual components to each other at runtime (you can think of them as the messengers that request an action from other components), whether the component belongs to your app or another.

SERVICES

A Service is an application component that can perform long-running operations in the background and does not provide a user interface.

Other application components can **start** or **bind** to a service to interact with it.

For example, a service might handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

https://developer.android.com/guide/components/services.html

CREATING A SERVICE

As with Activities, Services are declared in the manifest.

```
<manifest ... >
  ...
  <application ... >
      <service android:name=".ExampleService" />
      ...
  </application>
</manifest>
```

To create a service, you must create a subclass of **Service**.

You then override callback methods that handle key aspects of the service lifecycle and provide a mechanism for components to bind to the service.

```java
public class ExampleService extends Service {
    int mStartMode;       // indicates how to behave if the service is killed
    IBinder mBinder;      // interface for clients that bind
    boolean mAllowRebind; // indicates whether onRebind should be used

    @Override
    public void  onCreate () {
        // The service is being created
    }
    @Override
    public int  onStartCommand (Intent intent, int flags, int startId) {
        // The service is starting, due to a call to  startService()
        return mStartMode;
    }
    @Override
    public IBinder  onBind (Intent intent) {
        // A client is binding to the service with  bindService()
        return mBinder;
    }
    @Override
    public boolean  onUnbind (Intent intent) {
        // All clients have unbound with  unbindService()
        return mAllowRebind;
    }
    @Override
    public void  onRebind (Intent intent) {
        // A client is binding to the service with  bindService() ,
        // after onUnbind() has already been called
    }
    @Override
    public void  onDestroy () {
        // The service is no longer used and is being destroyed
    }
}
```

onStartCommand()

The system calls this method when another component, such as an activity, requests that the service be started.

If you implement this, it is your responsibility to implement the code that will stop the service when its work is done, by calling stopSelf() or stopService().

onBind()

The system calls this method 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 onBind(), but if you don't want to allow binding, then you should return null.

SERVICES

A service can essentially take two forms:

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.

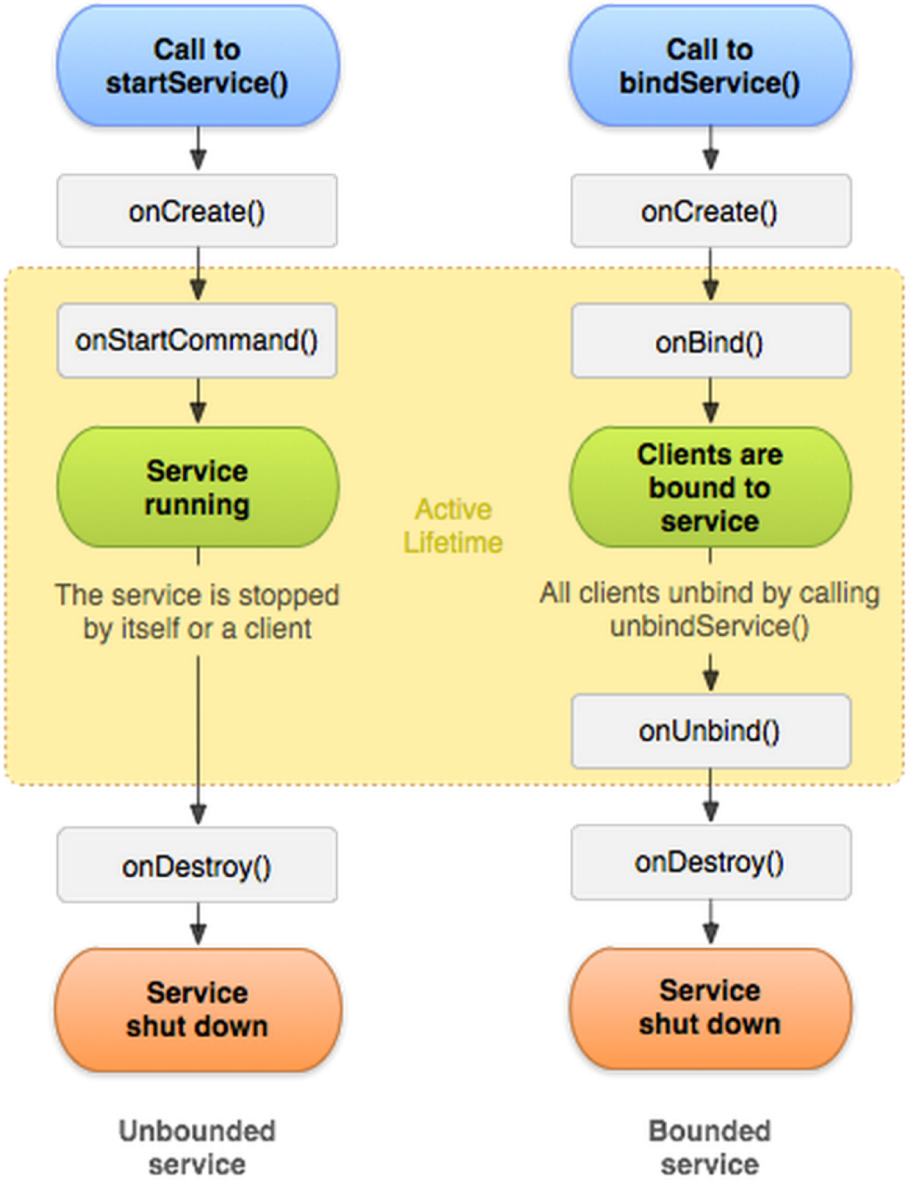
Bound

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 etc.

A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.

https://developer.android.com/guide/components/services.html

The **service lifecycle**.



On the left: the lifecycle when the service is created with startService().

On the right: the lifecycle when the service is created with bindService().

SERVICE CLASSES

There are two classes you will typically extend:

Service

This is the base class for all services. When you extend this class, it's important that you create a new thread in which to do all the service's work, because the service uses your application's main thread, by default, which could slow the performance of any activity your application is running.

IntentService

This is a subclass of Service that uses a worker thread to handle all start requests, one at a time. This is the best option if you don't require that your service handle multiple requests simultaneously. All you need to do is implement onHandleIntent(), which receives the intent for each start request.

INTENTSERVICE

The **IntentService** does the following:

- Creates a default worker thread that executes all intents delivered to onStartCommand() separate from your application's main thread.
- Creates a work queue that passes one intent at a time to your onHandleIntent() implementation, so you never have to worry about multi-threading.
- Stops the service after all start requests have been handled, so you never have to call stopSelf().
- Provides default implementation of onBind() that returns null.
- Provides a default implementation of onStartCommand() that sends the intent to the work queue and then to your onHandleIntent() implementation.

```java
public class HelloIntentService extends IntentService {

  /**
   * A constructor is required, and must call the super  IntentService(String)
   * constructor with a name for the worker thread.
   */
  public HelloIntentService() {
      super("HelloIntentService");
  }

  /**
   * The IntentService calls this method from the default worker thread with
   * the intent that started the service. When this method returns, IntentService
   * stops the service, as appropriate.
   */
  @Override
  protected void onHandleIntent(Intent intent) {
      // Normally we would do some work here, like download a file.
      // For our sample, we just sleep for 5 seconds.
      long endTime = System.currentTimeMillis() + 5*1000;
      while (System.currentTimeMillis() < endTime) {
          synchronized (this) {
              try {
                  wait(endTime - System.currentTimeMillis());
              } catch (Exception e) {
              }
          }
      }
  }
}
```

CREATING A BOUND SERVICE

A bound service is one that allows other application components to bind to it in order to create a long-standing connection

- To create a bound service, you must implement the onBind() callback method to return an IBinder that defines the interface for communication with the service.
- Other application components call bindService() to retrieve the IBinder.
- They can then interact with the service through the IBinder interface, e.g. begin calling methods on the service.

Multiple clients can bind to the service at once. When a client is done interacting with the service, it calls unbindService() to unbind.

Once there are no clients bound to the service, the system destroys the service.

For more detail on Bound Services see: https://developer.android.com/guide/components/bound-services.html

NOTES ON SERVICES

Once running, a service can notify the user of events using Toast Notifications or Status Bar Notifications.

The Android system will force-stop a service only when memory is low and it must recover system resources for the activity that has user focus.

IntentService: If you override callback methods such as onCreate(), onStartCommand(), or onDestroy(), be sure to call the super implementation, so that the IntentService can properly handle the life of the worker thread.

BROADCAST RECEIVERS

A broadcast receiver is a component that responds to system-wide broadcast announcements.

Many broadcasts originate from the system - for example, a broadcast announcing that the screen has turned off, the battery is low, or a picture was captured.

Apps can also initiate broadcasts - for example, to let other apps know that some data has been downloaded to the device and is available for them to use.

Although broadcast receivers don't display a user interface, they may create a status bar notification to alert the user when a broadcast event occurs.

More commonly, though, a broadcast receiver is just a "gateway" to other components and is intended to do a very minimal amount of work.

BROADCAST RECEIVERS

Two steps to receive system broadcasted intents:

- Creating the **Broadcast Receiver**.
- Registering Broadcast Receiver.

If you want to broadcast your own custom intents there are additional steps:

- Implement your custom intents
- Create and broadcast those intents.

CREATE A RECEIVER

A broadcast receiver extends the BroadcastReceiver abstract class.

This means you have to implement the onReceive() method of this base class.

Whenever the event occurs Android calls the onReceive() method on the registered broadcast receiver.

```
package org.davidcoyle.myfirstapp;

import ...

public class MyReceiver extends BroadcastReceiver {
    public MyReceiver() {
    }

    @Override
    public void onReceive(Context context, Intent intent) {
        // TODO: This method is called when the BroadcastReceiver is receiving
        // an Intent broadcast.
        throw new UnsupportedOperationException("Not yet implemented");
    }
}
```

REGISTER A RECEIVER

There are two ways to register Android broadcastreceiver.

The first is static. Here the broadcast receiver is registered in an android application via AndroidManifest.xml file.

```
<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>
```

REGISTER A RECEIVER

There are two ways to register Android broadcastreceiver.

The first is static. Here the broadcast receiver is registered in an android application via AndroidManifest.xml file.

```
<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>
```

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

Example of system generated events defined as final static fields in the Intent class

Event Constant	Description
android.intent.action.BATTERY_CHANGED	Sticky broadcast containing the charging state, level, and other information about the battery.
android.intent.action.BATTERY_LOW	Indicates low battery condition on the device.
android.intent.action.BATTERY_OKAY	Indicates the battery is now okay after being low.
android.intent.action.BOOT_COMPLETED	This is broadcast once, after the system has finished booting.
android.intent.action.BUG_REPORT	Show activity for reporting a bug.
android.intent.action.CALL	Perform a call to someone specified by the data.
android.intent.action.CALL_BUTTON	The user pressed the "call" button to go to the dialer or other appropriate UI for placing a call.
android.intent.action.DATE_CHANGED	The date has changed.
android.intent.action.REBOOT	Have the device reboot.

PERMISSIONS

To register for some receivers you also need to request permission from the user.

For example:

```
<receiver android:name="BroadCastReceiver_Name" >
    <intent-filter>
        <action android:name="android.intent.action.PHONE_STATE" />
        <uses-permission android:name="android.permission.READ_PHONE_STATE" />
    </intent-filter>
</receiver>
```

DYNAMIC REGISTRATION

The second way of registering the broadcast receiver is dynamically or programmatically.

This is done using Context.registerReceiver() method.

```
IntentFilter filter = new IntentFilter(”org.davidcoyle.BroadcastReceiver");

MyReceiver myReceiver = new MyReceiver();

registerReceiver(myReceiver, filter);
```

DYNAMIC REGISTRATION

The second way of registering the broadcast receiver is dynamically or programmatically.

This is done using Context.registerReceiver() method.

```
IntentFilter filter = new IntentFilter(”org.davidcoyle.BroadcastReceiver");

MyReceiver myReceiver = new MyReceiver();

registerReceiver(myReceiver, filter);
```

Step 1:

Created an IntentFilter object that specifies which event/intent our receiver will listen to.

In this case it’s a custom action name. However could equally be an inbuilt action.

DYNAMIC REGISTRATION

The second way of registering the broadcast receiver is dynamically or programmatically.

This is done using Context.registerReceiver() method.

```
IntentFilter filter = new IntentFilter(”org.davidcoyle.BroadcastReceiver");

MyReceiver myReceiver = new MyReceiver();

registerReceiver(myReceiver, filter);
```

Step 2:

Instantiate our broadcast receiver

and Step 3:

Call Context.registerReceiver() to actually register our receiver so that is will be called when the relevant event occurs.

DYNAMIC REGISTRATION

Dynamically registered receivers live only as long as the component that does the registration (e.g. an Activity).

Dynamically registered broadcast receivers can be unregistered using Context.unregisterReceiver() method.

Note: make sure to unregister broadcast receivers. If you forget to unregister the broadcast receiver, the system will report a leaked broadcast receiver error.

```
1  @Override
2  protected void onPause() {
3      unregisterReceiver(mReceiver);
4      super.onPause();
5  }
```

CREATE AND BROADCAST AN INTENT

```
1  Intent intent = new Intent();
2  intent.addFlags(Intent.FLAG_INCLUDE_STOPPED_PACKAGES);
3  intent.setAction("com.pycitup.BroadcastReceiver");
4  intent.putExtra("Foo", "Bar");
5  sendBroadcast(intent);
```

- Created an Intent
- Set the action
- Put data (Foo="Bar") in it
- Send it using sendBroadcast().

This broadcasts the intent to all the interested broadcast receivers.

The receivers are called asynchronously, i.e., sendBroadcast() returns immediately and the execution continues while the receivers are run but doesn't wait until their completion. No results are propagated from the receivers.

NORMAL AND ORDERED BROADCASTS

Broadly there are two major classes of broadcasts:

Normal Broadcasts: sent with Context.sendBroadcast(). They're completely asynchronous, i.e. the broadcasts events/intents are received by all the receivers in an asynchronous fashion. The receivers are run in an undefined order, often at the same time. It's efficient but receivers cannot use results from other receivers or abort the entire chain of execution at a certain level.

Ordered Broadcasts: These are sent with Context.sendOrderedBroadcast(). They're delivered to one receiver at a time. The order can be controlled with android:priority attribute of the matching intent-filter.

Receivers with same priority will be executed in a random order. As each receiver executes, it can transmit the result to the next one or even abort the entire broadcast chain so that no other receivers receive the broadcast intent and are executed.

FURTHER DETAILS

Class overview:

https://developer.android.com/reference/android/content/BroadcastReceiver.html

A more detailed explanation:

http://codetheory.in/android-broadcast-receivers/

QUESTIONS?

Contact:

d.coyle@ucd.ie

Please ask in the Discussion Forum.

Next class:

Sensors