48. Implementing an Android Started Service – A Worked Example

The previous chapter covered a considerable amount of information relating to Android services and, at this point, the concept of services may seem somewhat overwhelming. In order to reinforce the information in the previous chapter, this chapter will work through an Android Studio tutorial intended to gradually introduce the concepts of started service implementation.

Within this chapter, a sample application will be created and used as the basis for implementing an Android service. In the first instance, the service will be created using the *IntentService* class. This example will subsequently be extended to demonstrate the use of the *Service* class. Finally, the steps involved in performing tasks within a separate thread when using the Service class will be implemented. Having covered started services in this chapter, the next chapter, entitled *"Android Local Bound Services – A Worked Example"*, will focus on the implementation of bound services and client-service communication.

48.1 Creating the Example Project

Launch Android Studio and follow the usual steps to create a new project, entering *ServiceExample* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). Continue to proceed through the screens, requesting the creation of an Empty Activity named *ServiceExampleActivity* using the default values for the remaining options.

48.2 Creating the Service Class

Before writing any code, the first step is to add a new class to the project to contain the service. The first type of service to be demonstrated in this tutorial is to be based on the IntentService class. As outlined in the preceding chapter (*"An Overview of Android Started and Bound Services"*), the purpose of the IntentService class is to provide the developer with a convenient

mechanism for creating services that perform tasks asynchronously within a separate thread from the calling application.

Add a new class to the project by right-clicking on the *com.ebookfrenzy.serviceexample* package name located under *app -> java* in the Project tool window and selecting the *New -> Java Class* menu option. Within the resulting *Create New Class* dialog, name the new class *MyIntentService*. Finally, click on the *OK* button to create the new class.

Review the new *MyIntentService.java* file in the Android Studio editor where it should read as follows:

```
package com.ebookfrenzy.serviceexample;

/**
 * Created by <name> on <date>.
 */
public class MyIntentService {
}
```

The class needs to be modified so that it subclasses the IntentService class. When subclassing the IntentService class, there are two rules that must be followed. First, a constructor for the class must be implemented which calls the superclass constructor, passing through the class name of the service. Second, the class must override the *onHandleIntent()* method. Modify the code in the *MyIntentService.java* file, therefore, so that it reads as follows:

```
package com.ebookfrenzy.serviceexample;

import android.app.IntentService;
import android.content.Intent;

public class MyIntentService extends IntentService {

    @Override
    protected void onHandleIntent(Intent arg0) {

    }

    public MyIntentService() {
        super("MyIntentService");
    }
}
```

All that remains at this point is to implement some code within the

onHandleIntent() method so that the service actually does something when invoked. Ordinarily this would involve performing a task that takes some time to complete such as downloading a large file or playing audio. For the purposes of this example, however, the handler will simply output a message to the Android Studio Logcat panel:

```
package com.ebookfrenzy.serviceexample;

import android.app.IntentService;
import android.content.Intent;
import android.util.Log;

public class MyIntentService extends IntentService {

    private static final String TAG =
              "ServiceExample";

    @Override
    protected void onHandleIntent(Intent arg0) {
        Log.i(TAG, "Intent Service started");
    }

    public MyIntentService() {
        super("MyIntentService");
    }
}
```

48.3 Adding the Service to the Manifest File

Before a service can be invoked, it must first be added to the manifest file of the application to which it belongs. At a minimum, this involves adding a <service> element together with the class name of the service.

Double-click on the *AndroidManifest.xml* file (*app -> manifests*) for the current project to load it into the editor and modify the XML to add the service element as shown in the following listing:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.ebookfrenzy.serviceexample">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
```

```
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".ServiceExampleActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCH
            </intent-filter>
        </activity>
        <service android:name=".MyIntentService" />
    </application>

</manifest>
```

48.4 Starting the Service

Now that the service has been implemented and declared in the manifest file, the next step is to add code to start the service when the application launches. As is typically the case, the ideal location for such code is the *onCreate()* callback method of the activity class (which, in this case, can be found in the *ServiceExampleActivity.java* file). Locate and load this file into the editor and modify the *onCreate()* method to add the code to start the service:

```
package com.ebookfrenzy.serviceexample;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.content.Intent;

public class ServiceExampleActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_service_example);
        Intent intent = new Intent(this, MyIntentService.class);
        startService(intent);
    }
}
```

All that the added code needs to do is to create a new Intent object primed with the class name of the service to start and then use it as an argument to the *startService()* method.

48.5 Testing the IntentService Example

The example IntentService based service is now complete and ready to be tested. Since the message displayed by the service will appear in the Logcat panel, it is important that this is configured in the Android Studio environment.

Begin by displaying the Logcat tool window before clicking on the menu in the upper right-hand corner of the panel (which will probably currently read *Show only selected application*). From this menu, select the *Edit Filter Configuration* menu option.

In the *Create New Logcat Filter* dialog name the filter *ServiceExample* and, in the *by Log Tag* field, enter the TAG value declared in *ServiceExampleActivity.java* (in the above code example this was *ServiceExample*).

When the changes are complete, click on the *OK* button to create the filter and dismiss the dialog. The newly created filter should now be selected in the Android tool window.

With the filter configured, run the application on a physical device or AVD emulator session and note that the "Intent Service Started" message appears in the Logcat panel. Note that it may be necessary to change the filter menu setting back to ServiceExample after the application has launched:

```
06-29 09:05:16.887 3389-
3948/com.ebookfrenzy.serviceexample I/ServiceExample: Intent Service s
```

Had the service been tasked with a long-term activity, the service would have continued to run in the background in a separate thread until the task was completed, allowing the application to continue functioning and responding to the user. Since all our service did was log a message, it will have simply stopped upon completion.

48.6 Using the Service Class

While the IntentService class allows a service to be implemented with minimal coding, there are situations where the flexibility and synchronous nature of the Service class will be required. As will become evident in this chapter, this involves some additional programming work to implement.

In order to avoid introducing too many concepts at once, and as a demonstration of the risks inherent in performing time-consuming service

tasks in the same thread as the calling application, the example service created here will not run the service task within a new thread, instead relying on the main thread of the application. Creation and management of a new thread within a service will be covered in the next phase of the tutorial.

48.7 Creating the New Service

For the purposes of this example, a new class will be added to the project that will subclass from the Service class. Right-click, therefore, on the package name listed under *app -> java* in the Project tool window and select the *New -> Service -> Service* menu option. Create a new class named *MyService* with both the *Exported* and *Enabled* options selected.

The minimal requirement in order to create an operational service is to implement the *onStartCommand()* callback method which will be called when the service is starting up. In addition, the *onBind()* method must return a null value to indicate to the Android system that this is not a bound service. For the purposes of this example, the *onStartCommand()* method will loop 3 times sleeping for 10 seconds on each loop iteration. For the sake of completeness, stub versions of the *onCreate()* and *onDestroy()* methods will also be implemented in the new *MyService.java* file as follows:

```
package com.ebookfrenzy.serviceexample;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;

public class MyService extends Service {

    public MyService() {
    }

    private static final String TAG =
            "ServiceExample";

    @Override
    public void onCreate() {
        Log.i(TAG, "Service onCreate");
    }
```

```
    @Override
    public int onStartCommand(Intent intent, int flags, int startId)
{

        Log.i(TAG, "Service onStartCommand " + startId);

        int i = 0;
        while (i <= 3) {

            try {
                Thread.sleep(10000);
                i++;
            } catch (Exception e) {
            }
            Log.i(TAG, "Service running");
        }
        return Service.START_STICKY;
    }

    @Override
    public IBinder onBind(Intent arg0) {
        Log.i(TAG, "Service onBind");
        return null;
    }

    @Override
    public void onDestroy() {
        Log.i(TAG, "Service onDestroy");
    }
}
```

With the service implemented, load the *AndroidManifest.xml* file into the editor and verify that Android Studio has added an appropriate entry for the new service which should read as follows:

```
<service
 android:name=".MyService"
            android:enabled="true"
            android:exported="true" >
</service>
```

48.8 Modifying the User Interface

As will become evident when the application runs, failing to create a new thread for the service to perform tasks creates a serious usability problem. In

order to be able to appreciate fully the magnitude of this issue, it is going to be necessary to add a Button view to the user interface of the *ServiceExampleActivity* activity and configure it to call a method when "clicked" by the user.

Locate and load the *activity_service_example.xml* file in the Project tool window (*app -> res -> layout -> activity_service_example.xml*). Delete the TextView and add a Button view to the layout. Select the new button, change the text to read "Start Service" and extract the string to a resource named *start_service*.

With the new Button still selected, locate the *onClick* property in the Attributes panel and assign to it a method named *buttonClick*.

Next, edit the *ServiceExampleActivity.java* file to add the *buttonClick()* method and remove the code from the *onCreate()* method that was previously added to launch the MyIntentService service:

```
package com.ebookfrenzy.serviceexample;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.content.Intent;
import android.view.View;

public class ServiceExampleActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_service_example);
        Intent intent = new Intent(this, MyIntentService.class);
        startService(intent);
    }

    public void buttonClick(View view)
    {
        Intent intent = new Intent(this, MyService.class);
        startService(intent);
    }
}
```

All that the *buttonClick()* method does is create an intent object for the new service and then start it running.

48.9 Running the Application

Run the application and, once loaded, touch the *Start Service* button. Within the Logcat tool window (using the *ServiceExample* filter created previously) the log messages will appear indicating that the *onCreate()* method was called and that the loop in the *onStartCommand()* method is executing.

Before the final loop message appears, attempt to touch the *Start Service* button a second time. Note that the button is unresponsive. After approximately 20 seconds, the system may display a warning dialog containing the message "ServiceExample isn't responding". The reason for this is that the main thread of the application is currently being held up by the service while it performs the looping task. Not only does this prevent the application from responding to the user, but also to the system, which eventually assumes that the application has locked up in some way.

Clearly, the code for the service needs to be modified to perform tasks in a separate thread from the main thread.

48.10 Creating an AsyncTask for Service Tasks

As outlined in *"A Basic Overview of Threads and AsyncTasks"*, when an Android application is first started, the runtime system creates a single thread in which all application components will run by default. This thread is generally referred to as the *main thread*. The primary role of the main thread is to handle the user interface in terms of event handling and interaction with views in the user interface. Any additional components that are started within the application will, by default, also run on the main thread.

As demonstrated in the previous section, any component that undertakes a time consuming operation on the main thread will cause the application to become unresponsive until that task is complete. It is not surprising, therefore, that Android provides an API that allows applications to create and use additional threads. Any tasks performed in a separate thread from the main thread are essentially performed in the background. Such threads are typically referred to as *background* or *worker* threads.

A very simple solution to this problem involves performing the service task within an AsyncTask instance. To add this support to the app, modify the *MyService.java* file create an AsyncTask subclass containing the timer code from the *onStartCommand()* method:

```
.
.
import android.os.AsyncTask;
.
.
private class SrvTask extends AsyncTask<Integer, Integer, String> {

    @Override
    protected String doInBackground(Integer... params) {

        int startId = params[0];
        int i = 0;
        while (i <= 3) {

            publishProgress(params[0]);
            try {
                Thread.sleep(10000);
                i++;
            } catch (Exception e) {
            }
        }
        return("Service complete " + startId);
    }

    @Override
    protected void onPostExecute(String result) {
        Log.i(TAG, result);
    }

    @Override
    protected void onPreExecute() {

    }

    @Override
    protected void onProgressUpdate(Integer... values) {
        Log.i(TAG, "Service Running " + values[0]);
    }
}
```

Next, modify the *onStartCommand()* method to execute the task in the background, this time using the thread pool executor to allow multiple instances of the task to run in parallel:

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {

    AsyncTask task = new SrvTask().executeOnExecutor(
                     AsyncTask.THREAD_POOL_EXECUTOR, startId);

    return Service.START_STICKY;
}
```

When the application is now run, it should be possible to touch the *Start Service* button multiple times. When doing so, the Logcat output should indicate more than one task running simultaneously (subject to CPU core limitations):

```
I/ServiceExample: Service Running 1
I/ServiceExample: Service Running 2
I/ServiceExample: Service Running 1
I/ServiceExample: Service Running 2
I/ServiceExample: Service Running 1
I/ServiceExample: Service Running 2
I/ServiceExample: Service Running 1
I/ServiceExample: Service Running 2
I/ServiceExample: Service complete 1
I/ServiceExample: Service complete 2
```

With the service now handling requests outside of the main thread, the application remains responsive to both the user and the Android system.

48.11 Summary

This chapter has worked through an example implementation of an Android started service using the *IntentService* and *Service* classes. The example also demonstrated the use of asynchronous tasks within a service to avoid making the main thread of the application unresponsive.