47. An Overview of Android Started and Bound Services

The Android Service class is designed specifically to allow applications to initiate and perform background tasks. Unlike broadcast receivers, which are intended to perform a task quickly and then exit, services are designed to perform tasks that take a long time to complete (such as downloading a file over an internet connection or streaming music to the user) but do not require a user interface.

In this chapter, an overview of the different types of services available will be covered, including *started services*, *bound services* and *intent services*. Once these basics have been covered, subsequent chapters will work through a number of examples of services in action.

47.1 Started Services

Started services are launched by other application components (such as an activity or even a broadcast receiver) and potentially run indefinitely in the background until the service is stopped, or is destroyed by the Android runtime system in order to free up resources. A service will continue to run if the application that started it is no longer in the foreground, and even in the event that the component that originally started the service is destroyed.

By default, a service will run within the same main thread as the application process from which it was launched (referred to as a *local service*). It is important, therefore, that any CPU intensive tasks be performed in a new thread within the service. Instructing a service to run within a separate process (and therefore known as a *remote service*) requires a configuration change within the manifest file.

Unless a service is specifically configured to be private (once again via a setting in the manifest file), that service can be started by other components on the same Android device. This is achieved using the Intent mechanism in the same way that one activity can launch another, as outlined in preceding chapters.

Started services are launched via a call to the *startService()* method, passing through as an argument an Intent object identifying the service to be started. When a started service has completed its tasks, it should stop itself via a call to *stopSelf()*. Alternatively, a running service may be stopped by another component via a call to the *stopService()* method, passing through as an argument the matching Intent for the service to be stopped.

Services are given a high priority by the Android system and are typically among the last to be terminated in order to free up resources.

47.2 Intent Service

As previously outlined, services run by default within the same main thread as the component from which they are launched. As such, any CPU intensive tasks that need to be performed by the service should take place within a new thread, thereby avoiding impacting the performance of the calling application.

The *IntentService* class is a convenience class (subclassed from the Service class) that sets up a worker thread for handling background tasks and handles each request in an asynchronous manner. Once the service has handled all queued requests, it simply exits. All that is required when using the

IntentService class is that the *onHandleIntent()* method be implemented containing the code to be executed for each request.

For services that do not require synchronous processing of requests, IntentService is the recommended option. Services requiring synchronous handling of requests will, however, need to subclass from the Service class and manually implement and manage threading to handle any CPU intensive tasks efficiently.

47.3 Bound Service

A *bound service* is similar to a started service with the exception that a started service does not generally return results or permit interaction with the component that launched it. A bound service, on the other hand, allows the launching component to interact with, and receive results from, the service. Through the implementation of interprocess communication (IPC), this interaction can also take place across process boundaries. An activity might, for example, start a service to handle audio playback. The activity will, in all probability, include a user interface providing controls to the user for the purpose of pausing playback or skipping to the next track. Similarly, the service will quite likely need to communicate information to the calling activity to indicate that the current audio track has completed and to provide details of the next track that is about to start playing.

A component (also referred to in this context as a *client*) starts and *binds* to a bound service via a call to the *bindService()* method. Also, multiple components may bind to a service simultaneously. When the service binding is no longer required by a client, a call should be made to the *unbindService()* method. When the last bound client unbinds from a service, the service will be terminated by the Android runtime system. It is important to keep in mind that a bound service may also be started via a call to *startService()*. Once started, components may then bind to it via *bindService()* calls. When a bound service is launched via a call to *startService()* it will continue to run even after the last client unbinds from it.

A bound service must include an implementation of the *onBind()* method which is called both when the service is initially created and when other clients subsequently bind to the running service. The purpose of this method is to return to binding clients an object of type *IBinder* containing the information needed by the client to communicate with the service.

In terms of implementing the communication between a client and a bound service, the recommended technique depends on whether the client and service reside in the same or different processes and whether or not the service is private to the client. Local communication can be achieved by extending the Binder class and returning an instance from the *onBind()* method. Interprocess communication, on the other hand, requires Messenger and Handler implementation. Details of both of these approaches will be covered in later chapters.

47.4 The Anatomy of a Service

A service must, as has already been mentioned, be created as a subclass of the Android Service class (more specifically *android.app.Service*) or a sub-class thereof (such as *android.app.IntentService*). As part of the subclassing procedure, one or more of the following superclass callback methods must be overridden, depending on the exact nature of the service being created:

- **onStartCommand()** – This is the method that is called when the service is started by another component via a call to the *startService()* method. This method does not need to be implemented

for bound services.

- **onBind()** – Called when a component binds to the service via a call to the *bindService()* method. When implementing a bound service, this method must return an *IBinder* object facilitating communication with the client. In the case of *started services*, this method must be implemented to return a NULL value.
- **onCreate()** – Intended as a location to perform initialization tasks, this method is called immediately before the call to either *onStartCommand()* or the *first* call to the *onBind()* method.
- **onDestroy()** – Called when the service is being destroyed.
- **onHandleIntent()** – Applies only to IntentService subclasses. This method is called to handle the processing for the service. It is executed in a separate thread from the main application.

Note that the IntentService class includes its own implementations of the *onStartCommand()* and *onBind()* callback methods so these do not need to be implemented in subclasses.

47.5 Controlling Destroyed Service Restart Options

The *onStartCommand()* callback method is required to return an integer value to define what should happen with regard to the service in the event that it is destroyed by the Android runtime system. Possible return values for these methods are as follows:

- **START_NOT_STICKY** – Indicates to the system that the service should not be restarted in the event that it is destroyed unless there are pending intents awaiting delivery**.**
- **START_STICKY** – Indicates that the service should be restarted as soon as possible after it has been destroyed if the destruction occurred after the *onStartCommand()* method returned. In the event that no pending intents are waiting to be delivered, the *onStartCommand()* callback method is called with a NULL intent value. The intent being processed at the time that the service was destroyed is discarded.
- **START_REDELIVER_INTENT** – Indicates that, if the service was destroyed after returning from the *onStartCommand()* callback method, the service should be restarted with the current intent redelivered to the *onStartCommand()* method followed by any pending intents**.**

47.6 Declaring a Service in the Manifest File

In order for a service to be useable, it must first be declared within a manifest file. This involves embedding an appropriately configured <*service*> element into an existing <*application*> entry. At a minimum, the <service> element must contain a property declaring the class name of the service as illustrated in the following XML fragment:

```
<application
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name" >
    <activity
        android:label="@string/app_name"
        android:name=".TestActivity" >
        <intent-filter>
          <action android:name="android.intent.action.MAIN" />
          <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <service android:name="MyService>
     </service>
```

```
        </application>
    </manifest>
```

By default, services are declared as public, in that they can be accessed by components outside of the application package in which they reside. In order to make a service private, the *android:exported* property must be declared as *false* within the <service> element of the manifest file. For example:

```
<service android:name="MyService"
android:exported="false">
</service>
```

As previously discussed, services run within the same process as the calling component by default. In order to force a service to run within its own process, add an *android:process* property to the <service> element, declaring a name for the process prefixed with a colon (:):

```
<service android:name="MyService"
android:exported="false"
android:process=":myprocess">
</service>
```

The colon prefix indicates that the new process is private to the local application. If the process name begins with a lower case letter instead of a colon, however, the process will be global and available for use by other components.

Finally, using the same intent filter mechanisms outlined for activities, a service may also advertise capabilities to other applications running on the device. For more details on intent filters, refer to the chapter entitled *An Overview of Android Intents*.

47.7 Starting a Service Running on System Startup

Given the background nature of services, it is not uncommon for a service to need to be started when an Android based system first boots up. This can be achieved by creating a broadcast receiver with an intent filter configured to listen for the system *android.intent.action.BOOT_COMPLETED* intent. When such an intent is detected, the broadcast receiver would simply invoke the necessary service and then return. Note that, in order to function, such a broadcast receiver will need to request the *android.permission.RECEIVE_BOOT_COMPLETED* permission.

47.8 Summary

Android services are a powerful mechanism that allows applications to perform tasks in the background. A service, once launched, will continue to run regardless of whether the calling application is the foreground task or not, and even in the event that the component that initiated the service is destroyed.

Services are subclassed from the Android Service class and fall into the category of either *started services* or *bound services*. Started services run until they are stopped or destroyed and do not inherently provide a mechanism for interaction or data exchange with other components. Bound services, on the other hand, provide a communication interface to other client components and generally run until the last client unbinds from the service.

By default, services run locally within the same process and main thread as the calling application. A new thread should, therefore, be created within the service for the purpose of handling CPU intensive tasks. Remote services may be started within a separate process by making a minor configuration change to the corresponding <service> entry in the application manifest file.

The IntentService class (itself a subclass of the Android Service class) provides a convenient mechanism for handling asynchronous service requests within a separate worker thread.

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.LAUNCHER" />
            </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 Android Monitor tool window using either the tools menu button located in the far left corner of the status bar or the *Alt-6* keyboard shortcut. Within the tool window, make sure that the *logcat* tab is selected before accessing 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 started
```

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 three times performing a 10-second wait on each loop. 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");

        for (int i = 0; i < 3; i++) {
            long endTime = System.currentTimeMillis() +
                    10 * 1000;
            while (System.currentTimeMillis() < endTime) {
                synchronized (this) {
                    try {
                        wait(endTime - System.currentTimeMillis());
                    } 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*. Remaining in the Properties tool window, change the layout_width property of the Button widget to *wrap_content* so that it is sized to accommodate the text.

With the new Button still selected, locate the *onClick* property in the Properties 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 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 a New Thread for Service Tasks

As outlined in *A Basic Overview of Android Threads and Thread Handlers*, 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 a new thread. The following *onStartCommand()* method from the *MyService.java* file, for example, has been modified to launch the task within a new thread using the most basic of thread handling examples:

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

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

        final int currentId = startId;

        Runnable r = new Runnable() {
               public void run() {

                for (int i = 0; i < 3; i++)
                {
                       long endTime = System.currentTimeMillis() +
                                                              10*1000;

                       while (System.currentTimeMillis() < endTime) {
                              synchronized (this) {
                                     try {
                                            wait(endTime -
```

```
                                        System.currentTimeMillis());
                                        } catch (Exception e) {
                                    }
                                }
                            }
                            Log.i(TAG, "Service running " + currentId);
                        }
                        stopSelf();
                    }
                };

                Thread t = new Thread(r);
                t.start();
                return Service.START_STICKY;
            }
```

When the application is now run, it should be possible to touch the *Start Service* button multiple times. Each time a new thread will be created by the service to process the task. The LogCat output will now also include a number referencing the startId of each service request.

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 threads within a service to avoid making the main thread of the application unresponsive.

49. Android Local Bound Services – A Worked Example

As outlined in some detail in the previous chapters, bound services, unlike started services, provide a mechanism for implementing communication between an Android service and one or more client components. The objective of this chapter is to build on the overview of bound services provided in *An Overview of Android Started and Bound Services* before embarking on an example implementation of a *local* bound service in action.

49.1 Understanding Bound Services

In common with started services, bound services are provided to allow applications to perform tasks in the background. Unlike started services, however, multiple client components may *bind* to a bound service and, once bound, interact with that service using a variety of different mechanisms.

Bound services are created as sub-classes of the Android Service class and must, at a minimum, implement the *onBind()* method. Client components bind to a service via a call to the *bindService()* method. The first bind request to a bound service will result in a call to that service's *onBind()* method (subsequent bind requests do not trigger an *onBind()* call). Clients wishing to bind to a service must also implement a ServiceConnection subclass containing *onServiceConnected()* and *onServiceDisconnected()* methods which will be called once the client-server connection has been established or disconnected, respectively. In the case of the *onServiceConnected()* method, this will be passed an IBinder object containing the information needed by the client to interact with the service.

49.2 Bound Service Interaction Options

There are two recommended mechanisms for implementing interaction between client components and a bound service. In the event that the bound service is local and private to the same application as the client component (in other words it runs within the same process and is not available to components in other applications), the recommended method is to create a subclass of the Binder class and extend it to provide an interface to the service. An instance of this Binder object is then returned by the *onBind()* method and subsequently used by the client component to directly access methods and data held within the service.

In situations where the bound service is not local to the application (in other words, it is running in a different process from the client component), interaction is best achieved using a Messenger/Handler implementation.

In the remainder of this chapter, an example will be created with the aim of demonstrating the steps involved in creating, starting and interacting with a local, private bound service.

49.3 An Android Studio Local Bound Service Example

The example application created in the remainder of this chapter will consist of a single activity and a bound service. The purpose of the bound service is to obtain the current time from the system and return that information to the activity where it will be displayed to the user. The bound service will be local and private to the same application as the activity.

Launch Android Studio and follow the usual steps to create a new project, entering *LocalBound* 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 *LocalBoundActivity* with the remaining fields set to the default values.

Once the project has been created, the next step is to add a new class to act as the bound service.

49.4 Adding a Bound Service to the Project

To add a new class to the project, right-click on the package name (located under *app -> java -> com.ebookfrenzy.localbound*) within the Project tool window and select the *New -> Service -> Service* menu option. Specify *BoundService* as the class name and make sure that both the *Exported* and *Enabled* options are selected before clicking on *Finish* to create the class. By default, Android Studio will load the *BoundService.java* file into the editor where it will read as follows:

```
package com.ebookfrenzy.localbound;

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

public class BoundService extends Service {
    public BoundService() {
    }

    @Override
    public IBinder onBind(Intent intent) {
        // TODO: Return the communication channel to the service.
        throw new UnsupportedOperationException("Not yet implemented");
    }
}
```

49.5 Implementing the Binder

As previously outlined, local bound services can communicate with bound clients by passing an appropriately configured Binder object to the client. This is achieved by creating a Binder subclass within the bound service class and extending it by adding one or more new methods that can be called by the client. In most cases, this simply involves implementing a method that returns a reference to the bound service instance. With a reference to this instance, the client can then access data and call methods within the bound service directly.

For the purposes of this example, therefore, some changes are needed to the template *BoundService* class created in the preceding section. In the first instance, a Binder subclass needs to be declared. This class will contain a single method named *getService()* which will simply return a reference to the current service object instance (represented by the *this* keyword). With these requirements in mind, edit the *BoundService.java* file and modify it as follows:

```
package com.ebookfrenzy.localbound;
```

```
import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.os.Binder;

public class BoundService extends Service {

    private final IBinder myBinder = new MyLocalBinder();

    public BoundService() {
    }

    @Override
    public IBinder onBind(Intent intent) {
        // TODO: Return the communication channel to the service.
        throw new UnsupportedOperationException("Not yet implemented");
    }

    public class MyLocalBinder extends Binder {
        BoundService getService() {
            return BoundService.this;
        }
    }
}
```

Having made the changes to the class, it is worth taking a moment to recap the steps performed here. First, a new subclass of Binder (named *MyLocalBinder*) is declared. This class contains a single method for the sole purpose of returning a reference to the current instance of the *BoundService* class. A new instance of the *MyLocalBinder* class is created and assigned to the *myBinder* IBinder reference (since Binder is a subclass of IBinder there is no type mismatch in this assignment).

Next, the *onBind()* method needs to be modified to return a reference to the *myBinder* object and a new public method implemented to return the current time when called by any clients that bind to the service:

```
package com.ebookfrenzy.localbound;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Locale;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.os.Binder;

public class BoundService extends Service {

    private final IBinder myBinder = new MyLocalBinder();

    public BoundService() {
    }

    @Override
```

```
    public IBinder onBind(Intent intent) {
        return myBinder;
    }

    public String getCurrentTime() {
        SimpleDateFormat dateformat =
                new SimpleDateFormat("HH:mm:ss MM/dd/yyyy",
                        Locale.US);
        return (dateformat.format(new Date()));
    }

    public class MyLocalBinder extends Binder {
        BoundService getService() {
            return BoundService.this;
        }
    }
}
```

At this point, the bound service is complete and is ready to be added to the project manifest file. Locate and double-click on the *AndroidManifest.xml* file for the *LocalBound* project in the Project tool window and, once loaded into the Manifest Editor, verify that Android Studio has already added a <service> entry for the service as follows:

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=" .LocalBoundActivity" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

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

        <service
            android:name=".BoundService"
            android:enabled="true"
            android:exported="true" >
        </service>
    </application>

</manifest>
```

The next phase is to implement the necessary code within the activity to bind to the service and call the *getCurrentTime()* method.

49.6 Binding the Client to the Service

For the purposes of this tutorial, the client is the *LocalBoundActivity* instance of the running

application. As previously noted, in order to successfully bind to a service and receive the IBinder object returned by the service's *onBind()* method, it is necessary to create a ServiceConnection subclass and implement *onServiceConnected()* and *onServiceDisconnected()* callback methods. Edit the *LocalBoundActivity.java* file and modify it as follows:

```
package com.ebookfrenzy.localbound;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.os.IBinder;
import android.content.Context;
import android.content.Intent;
import android.content.ComponentName;
import android.content.ServiceConnection;
import com.ebookfrenzy.localbound.BoundService.MyLocalBinder;

public class LocalBoundActivity extends AppCompatActivity {

    BoundService myService;
    boolean isBound = false;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_local_bound);
    }

    private ServiceConnection myConnection = new ServiceConnection()
    {
        @Override
        public void onServiceConnected(ComponentName className,
                                       IBinder service) {
            MyLocalBinder binder = (MyLocalBinder) service;
            myService = binder.getService();
            isBound = true;
        }

        @Override
        public void onServiceDisconnected(ComponentName name) {
            isBound = false;
        }
    };
}
```

The *onServiceConnected()* method will be called when the client binds successfully to the service. The method is passed as an argument the IBinder object returned by the *onBind()* method of the service. This argument is cast to an object of type MyLocalBinder and then the *getService()* method of the binder object is called to obtain a reference to the service instance, which, in turn, is assigned to *myService*. A Boolean flag is used to indicate that the connection has been successfully established.

The *onServiceDisconnected()* method is called when the connection ends and simply sets the Boolean flag to false.

Having established the connection, the next step is to modify the activity to bind to the service. This

involves the creation of an intent and a call to the *bindService()* method, which can be performed in the *onCreate()* method of the activity:

```
@Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_local_bound);
        Intent intent = new Intent(this, BoundService.class);
        bindService(intent, myConnection, Context.BIND_AUTO_CREATE);
}
```

49.7 Completing the Example

All that remains is to implement a mechanism for calling the *getCurrentTime()* method and displaying the result to the user. As is now customary, Android Studio will have created a template *activity_local_bound.xml* file for the activity containing only a TextView. Load this file into the Layout Editor tool and, using Design mode, select the TextView component and change the ID to *myTextView*. Add a Button view beneath the TextView and change the text on the button to read "Show Time", extracting the text to a string resource named *show_time*. On completion of these changes, the layout should resemble that illustrated in Figure 49-1. If any constraints are missing, click on the Infer Constraints button in the Layout Editor toolbar.

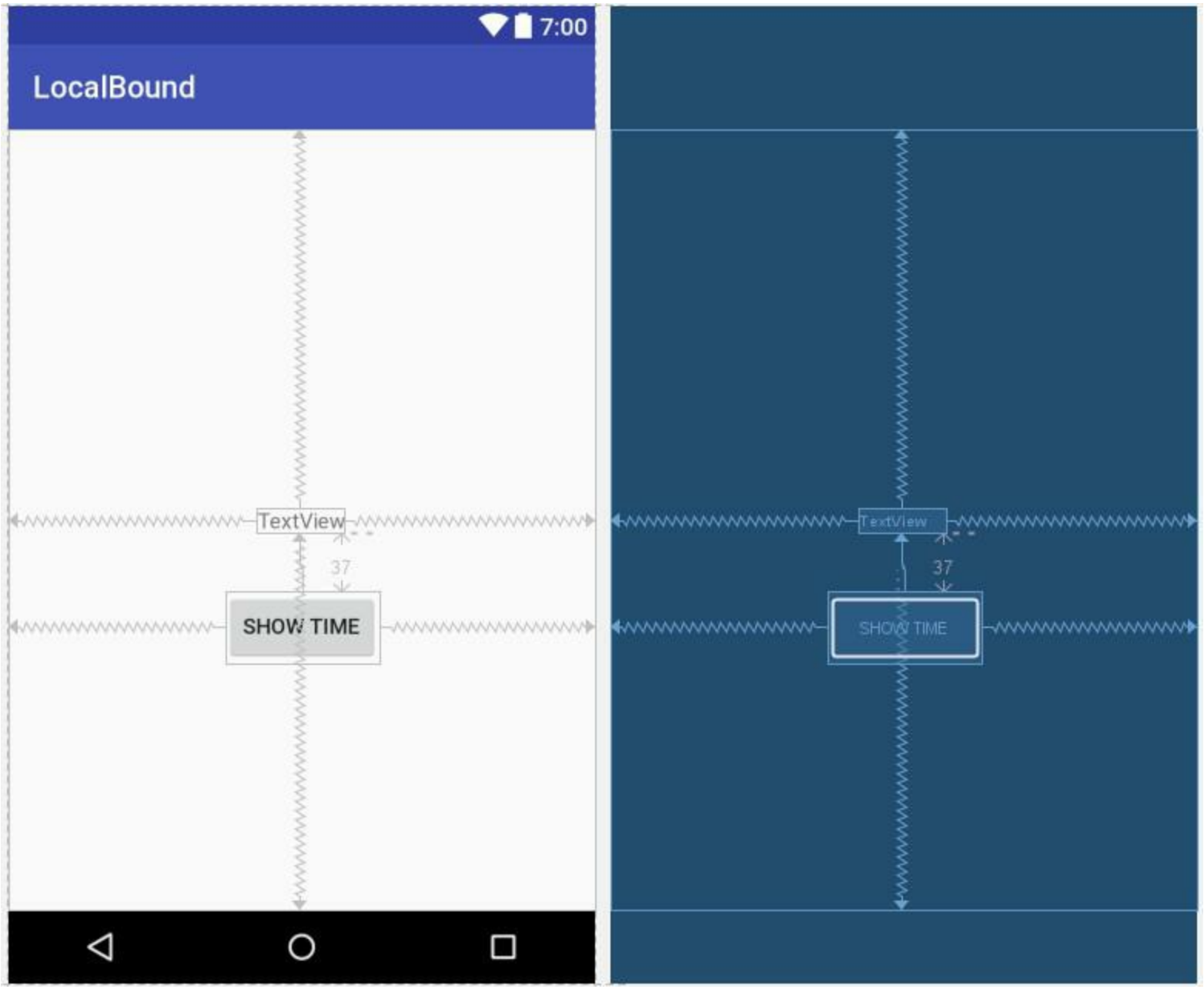

Figure 49-1

Complete the user interface design by selecting the Button and configuring the *onClick* property to call a method named *showTime*.

Finally, edit the code in the *LocalBoundActivity.java* file to implement the *showTime()* method. This method simply calls the *getCurrentTime()* method of the service (which, thanks to the *onServiceConnected()* method, is now available from within the activity via the *myService* reference) and assigns the resulting string to the TextView:

```
package com.ebookfrenzy.localbound;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.os.IBinder;
import android.content.Context;
import android.content.Intent;
import android.content.ComponentName;
import android.content.ServiceConnection;
import com.ebookfrenzy.localbound.BoundService.MyLocalBinder;
import android.view.View;
```

```
import android.widget.TextView;

public class LocalBoundActivity extends AppCompatActivity {

    BoundService myService;
    boolean isBound = false;

    public void showTime(View view)
    {
        String currentTime = myService.getCurrentTime();
        TextView myTextView =
                (TextView)findViewById(R.id.myTextView);
        myTextView.setText(currentTime);
    }
.
.
.
}
```

49.8 Testing the Application

With the code changes complete, perform a test run of the application. Once visible, touch the button and note that the text view changes to display the current date and time. The example has successfully started and bound to a service and then called a method of that service to cause a task to be performed and results returned to the activity.

49.9 Summary

When a bound service is local and private to an application, components within that application can interact with the service without the need to resort to inter-process communication (IPC). In general terms, the service's *onBind()* method returns an IBinder object containing a reference to the instance of the running service. The client component implements a ServiceConnection subclass containing callback methods that are called when the service is connected and disconnected. The former method is passed the IBinder object returned by the *onBind()* method allowing public methods within the service to be called.

Having covered the implementation of local bound services, the next chapter will focus on using IPC to interact with remote bound services.

50. Android Remote Bound Services – A Worked Example

In this, the final chapter dedicated to Android services, an example application will be developed to demonstrate the use of a messenger and handler configuration to facilitate interaction between a client and remote bound service.

50.1 Client to Remote Service Communication

As outlined in the previous chapter, interaction between a client and a local service can be implemented by returning to the client an IBinder object containing a reference to the service object. In the case of remote services, however, this approach does not work because the remote service is running in a different process and, as such, cannot be reached directly from the client.

In the case of remote services, a Messenger and Handler configuration must be created which allows messages to be passed across process boundaries between client and service.

Specifically, the service creates a Handler instance that will be called when a message is received from the client. In terms of initialization, it is the job of the Handler to create a Messenger object which, in turn, creates an IBinder object to be returned to the client in the *onBind()* method. This IBinder object is used by the client to create an instance of the Messenger object and, subsequently, to send messages to the service handler. Each time a message is sent by the client, the *handleMessage()* method of the handler is called, passing through the message object.

The simple example created in this chapter will consist of an activity and a bound service running in separate processes. The Messenger/Handler mechanism will be used to send a string to the service, which will then display that string in a Toast message.

50.2 Creating the Example Application

Launch Android Studio and follow the steps to create a new project, entering *RemoteBound* 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 *RemoteBoundActivity* with a corresponding layout resource file named *activity_remote_bound*.

50.3 Designing the User Interface

Locate the *activity_remote_bound.xml* file in the Project tool window and double-click on it to load it into the Layout Editor tool. With the Layout Editor tool in Design mode, delete the default TextView instance and drag and drop a Button widget from the palette so that it is positioned in the center of the layout. Change the text property of the button to read “Send Message” and extract the string to a new resource named *send_message*. Remaining within the Properties tool window, change the layout_width property to *wrap_content*.

Finally, configure the *onClick* property to call a method named *sendMessage*.

50.4 Implementing the Remote Bound Service

In order to implement the remote bound service for this example, add a new class to the project by right-clicking on the package name (located under *app -> java*) within the Project tool window and select the *New -> Service -> Service* menu option. Specify *RemoteService* as the class name and make sure that both the *Exported* and *Enabled* options are selected before clicking on *Finish* to create the class.

The next step is to implement the handler class for the new service. This is achieved by extending the Handler class and implementing the *handleMessage()* method. This method will be called when a message is received from the client. It will be passed a Message object as an argument containing any data that the client needs to pass to the service. In this instance, this will be a Bundle object containing a string to be displayed to the user. The modified class in the *RemoteService.java* file should read as follows once this has been implemented:

```
package com.ebookfrenzy.remotebound;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.widget.Toast;
import android.os.Messenger;

public class RemoteService extends Service {

    public RemoteService() {
    }

    class IncomingHandler extends Handler {
        @Override
        public void handleMessage(Message msg) {

            Bundle data = msg.getData();
            String dataString = data.getString("MyString");
            Toast.makeText(getApplicationContext(),
                    dataString, Toast.LENGTH_SHORT).show();
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        // TODO: Return the communication channel to the service.
        throw new UnsupportedOperationException("Not yet implemented");
    }
}
```

With the handler implemented, the only remaining task in terms of the service code is to modify the *onBind()* method such that it returns an IBinder object containing a Messenger object which, in turn, contains a reference to the handler:

```
final Messenger myMessenger = new Messenger(new IncomingHandler());
```

```
@Override
public IBinder onBind(Intent intent) {
      return myMessenger.getBinder();
}
```

The first line of the above code fragment creates a new instance of our handler class and passes it through to the constructor of a new Messenger object. Within the *onBind()* method, the *getBinder()* method of the messenger object is called to return the messenger's IBinder object.

50.5 Configuring a Remote Service in the Manifest File

In order to portray the communication between a client and remote service accurately, it will be necessary to configure the service to run in a separate process from the rest of the application. This is achieved by adding an *android:process* property within the <service> tag for the service in the manifest file. In order to launch a remote service it is also necessary to provide an intent filter for the service. To implement these changes, modify the *AndroidManifest.xml* file to add the required entries:

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".RemoteBoundActivity" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

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

        <service
            android:name=".RemoteService"
            android:enabled="true"
            android:exported="true"
            android:process=":my_process" >
        </service>
    </service>
    </application>

</manifest>
```

50.6 Launching and Binding to the Remote Service

As with a local bound service, the client component needs to implement an instance of the ServiceConnection class with *onServiceConnected()* and *onServiceDisconnected()* methods. Also, in common with local services, the *onServiceConnected()* method will be passed the IBinder object returned by the *onBind()* method of the remote service which will be used to send messages to the

server handler. In the case of this example, the client is *RemoteBoundActivity*, the code for which is located in *RemoteBoundActivity.java*. Load this file and modify it to add the ServiceConnection class and a variable to store a reference to the received Messenger object together with a Boolean flag to indicate whether or not the connection is established:

```
package com.ebookfrenzy.remotebound;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.os.IBinder;
import android.os.Message;
import android.os.Messenger;
import android.os.RemoteException;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.view.View;

public class RemoteBoundActivity extends AppCompatActivity {

    Messenger myService = null;
    boolean isBound;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_remote_bound);
    }

    private ServiceConnection myConnection =
            new ServiceConnection() {
                public void onServiceConnected(
                                ComponentName className,
                                        IBinder service) {
                    myService = new Messenger(service);
                    isBound = true;
                }

                public void onServiceDisconnected(
                                ComponentName className) {
                    myService = null;
                    isBound = false;
                }
            };
}
```

Next, some code needs to be added to bind to the remote service. This involves creating an intent that matches the intent filter for the service as declared in the manifest file and then making a call to the *bindService()* method, providing the intent and a reference to the ServiceConnection instance as arguments. For the purposes of this example, this code will be implemented in the activity's *onCreate()* method:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_remote_bound);

        Intent intent = new Intent(getApplicationContext(),
                                   RemoteService.class);

        bindService(intent, myConnection, Context.BIND_AUTO_CREATE);
    }
```

50.7 Sending a Message to the Remote Service

All that remains before testing the application is to implement the *sendMessage()* method in the *RemoteBoundActivity* class which is configured to be called when the button in the user interface is touched by the user. This method needs to check that the service is connected, create a bundle object containing the string to be displayed by the server, add it to a Message object and send it to the server:

```
    public void sendMessage(View view)
    {
         if (!isBound) return;

        Message msg = Message.obtain();

        Bundle bundle = new Bundle();
        bundle.putString("MyString", "Message Received");

        msg.setData(bundle);

        try {
            myService.send(msg);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
```

With the code changes complete, compile and run the application. Once loaded, touch the button in the user interface, at which point a Toast message should appear that reads “Message Received”.

50.8 Summary

In order to implement interaction between a client and remote bound service it is necessary to implement a handler/message communication framework. The basic concepts behind this technique have been covered in this chapter together with the implementation of an example application designed to demonstrate communication between a client and a bound service, each running in a separate process.