Charlie Collins
Michael Galpin
Matthias Käppler

Android IN PRACTICE

Includes 91 Techniques

MANNING

SAMPLE CHAPTER

Android in Practice

by Charlie Collins,
Michael D. Galpin,
and Matthias Käppler

Chapter 5

brief contents

5 Managing background tasks with Services

In this chapter

- Multitasking with `Services`
- Creating background tasks
- Reviving tasks that have been killed

> *I am the greatest. I said that even before I knew I was. Don't tell me I can't do something. Don't tell me it's impossible. Don't tell me I'm not the greatest. I'm the double greatest.*
>
> —Muhammad Ali

`Services` are a killer feature of Android. That's a bold statement, but it's accurate. It might be more accurate to say that *multitasking* is a killer feature of Android, and the way to fully implement multitasking on Android is by using `Services`. Don't take our word on this: watch TV instead. One of the most successful commercials for the popular Motorola Droid touted its ability to multitask and ridiculed "other" phones that couldn't "walk and chew gum at the same time."

Unfortunately, multitasking is one of the most often misunderstood features, even from a technical standpoint. For years, we've used desktop and laptop computers. These kinds of computers have defined how we expect multitasking to work. If

I start to load a web page in my browser and then change windows to type in a word processor, I expect that the web page will continue to load even without my attention. As programmers, we often begin a build of our code and then switch to another program while the build goes on. What would we do if the build stopped when we switched to another window? This is the multitasking world that mobile applications live in. In this chapter, we'll learn how Android's `Services` allow for multitasking when the traditional desktop multitasking doesn't work. First, let's understand how multitasking works on Android devices.

5.1 It's all about the multitasking

An easy way to realize how valuable multitasking is on a computing device is to live without it. For some applications, this is no big deal—everything the application does is confined to the device anyways. An example of this might be a note-taking application. If all the app does is stores notes on your device, then you probably don't care if it can multitask. But if your app stores your notes on a remote server so that they can be accessed (both read and write) from any computer/device, then multitasking starts to become nice. Why? If your app can run in the background, then it can keep the notes on the device and on the server in sync. Without this, you'll need to resync with the server every time you launch the app. That may not sound like a big deal, but this is a network operation that could be slow. The user is going to experience this slowness every time they want to use the application. In fact, it may be the first thing they experience when they launch the application. Can you say bad user experience?

The obvious solution is to let applications run in the background indefinitely, as desktop applications do. But what works fine on a desktop computer doesn't work well on a mobile device. The main problem is memory. A desktop computer has a lot of it. When it runs low, it uses virtual memory or paging to expand the available memory by using hard disk or similar storage. When an application isn't in the foreground, its real memory is often swapped out for virtual memory. When it comes back into the foreground, it'll need to get its data swapped back into real memory. This can be a slow process and make your computer seem sluggish.

On a mobile device, the amount of real memory available is low. One could imagine many apps going in and out of virtual memory. Suddenly, any time you changed apps, your device would seem to be bogged down and unresponsive. Nobody wants a device like that.

On an Android device, when you move an application into the background, it'll continue to run much like an application on a desktop computer. It's possible that it could run like this for a long time, but this is far from guaranteed. Instead, if or when memory becomes low, the Android OS will terminate your application. This may seem harsh, but it's not really. This removes the need for virtual memory and swapping, as we learned about in chapter 3. Plus, the OS will also send events to your application to let it know that this is about to happen. That gives you a chance to save the state of your application.

If this were the end of the multitasking story on Android, then you'd have to agree that Android wouldn't qualify as a multitasking OS. You might get lucky and be able to

multitask for a while, but it'd be difficult to design an application around this. Even worse, there'd be nothing to talk about in this chapter! Fortunately, for all of us, Android gives you more multitasking options, and these are all built around `Services`. We first saw `Services` in chapter 2, but now it's time to take a much more detailed look at them. In this chapter, we'll look at all of the many aspects of `Services`.

We'll start with the basics—how to create `Services` and how to start them automatically when the device boots up. We'll learn about two of the most common design patterns for using `Services`, using them to centralize access to and cache data, and periodically executing `Services` to check for remote events and potentially publishing `Notifications` about remote events. This will naturally lead us into a discussion about scheduling `Services` and how to make sure these schedules are executed even when a device is asleep or low on memory. Finally, we'll learn about a new feature in Android 2.2, Cloud to Device Messaging, and see how we can use our remote servers to schedule and interact with `Services`. Let's get started by discussing why we'd want to use a `Service`.

5.2 Why services and how to use them

We stated this earlier, but it bears repeating: `Services` are the way to fully implement multitasking on Android. You'll need other technologies as well, and we'll examine those, but `Services` are the building blocks for any kind of multitasking on Android. Now `Services` aren't for running indefinitely in the background. If you need to start a task separate from your main application, consider using a `Service`. For example, let's say that you need to upload some large file to a remote server. This could take a long time. It's possible that the user will leave your app before this upload finishes. If the upload is tied to the app, it might still finish as long as the app runs in the background. But if the app gets terminated to free up memory, then that upload could potentially be disrupted in midstream. Another example might be building some kind of complex data structure. A common example of this would be creating a `ContentProvider` for Android's systemwide search. This may involve downloading some data, processing it, and then storing it on the device, probably in a SQLite database. This is a one-time task that could take a long time to execute. You don't want it tied to your application's lifecycle, or this task may never finish correctly. `Services` are perfect for these kinds of one-time tasks, as well as any kind of recurring task.

GRAB THE PROJECT: STOCKPORTFOLIO You can get the source code for this project, and/or the packaged APK to run it, at the *Android in Practice* code website. Because some code listings here are shortened to focus on specific concepts, we recommend that you download the complete source code and follow along within Eclipse (or your favorite IDE or text editor).

Source: http://mng.bz/APOO, APK file: http://mng.bz/4iDX

This all sounds well and good, but let's consider a more concrete example. In this chapter, we'll develop an application called StockPortfolio. It'll allow the user to track their stock portfolio—what stocks they own, how many shares of each stock, and how

much they paid for the stocks. Further , it'll allow the user to set alerts, so that if a stock's price falls too low or rises too high, they'll be notified so they can sell or buy . This is a simple application, but it benefits from multitasking via `Services` in two ways. First, it'll fetch the latest stock data in th e background and cache it locally . That way, when the user launches the app, it'll imme diately display accurate stock data, with no wait time for the user. Second, by running in the background, it can also compare the current stock prices to see if they're at a le vel where the user wants to receive a notification. This way the user can receive the notification without having the application open. All of this sounds simple enough, bu t such an application wouldn't be possible on some mobile devices. Even on Android, you need to be aware of some "gotchas." By the end of the chapter , you'll understand not only how to create such a `Service`, but how to get it run periodically in th e background, even under low-memor y situations where the OS may have to kill the `Service`.

TECHNIQUE 13 Creating a Service

This chapter is all about `Services`, and we'll cover them in great detail. But we're going to start off small and discuss the basics. `Services` have some unique characteristics, as they're designed to fill the ni che of background processing given the style of multitasking supported by the Android OS. It comes as no surprise that creating and starting a `Service` isn't as simple as implementing an inter face and invoking a method.

PROBLEM

You need to monitor the prices of stocks at all times, not only when the user has the application open in the foreground.

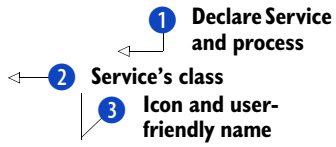
SOLUTION

The Android way of per forming background processing is to use a `Service`. If all you cared about was retrieving data in the ba ckground while the user had the application open, then you could spawn a thread from your `Activity`. If you wanted it to run continuously, then you could use a `java.util.Timer`. You might also want to consider Android's `AsyncTask` as a convenient way to orches trate the spawned thread and its interaction with the UI. (Chapter 6 has a lot more information about thread s and `AsyncTasks`.) The problem with this approach is that once your application leaves the foreground, the OS could terminate it at any time.

It might seem like this isn't the case in practice. It's easy to create an app that starts a `Timer` that continues to run when the applic ation leaves the foreground. You could let it run on a test devise for a long time, but it will only a ppear as if it's never killed. This is misleading though, since it's atypical usage. Typically, users are using lots of different apps, making calls, sending emails, and so on. All of these require memory and make it more likely that the OS will terminate your applicat ion. So don't be fooled: if you need to keep running in the background, you need a `Service`. To create a `Service`, you'll need to declare it in your mani fest file. The following listing shows how the `Service` for our stock portfolio, called `PortfolioManagerService`, is declared.

Listing 5.1 Declaring the `PortfolioManagerService`

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.manning.aip.portfolio"
      android:versionCode="1"
      android:versionName="1.0">
   <application android:icon="@drawable/icon"
         android:label="@string/app_name">
      <activity android:name=".ViewStocks"
            android:label="@string/app_name">
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
               <category android:name="android.intent.category.LAUNCHER"/>
         </intent-filter>
      </activity>
   <service android:process=":stocks_background"
      android:name="PortfolioManagerService"
      android:icon="@drawable/icon"
      android:label="@string/service_name"/>
   </application>
   <uses-sdk android:minSdkVersion="8" />
   <uses-permission android:name="android.permission.INTERNET"/>
</manifest>
```



This is a straightforward manifest, so we'll focus on the `Service` declaration. `Services` are important enough in Android that they get their own tag! The first part of this declaration is significant. The first attribute that we declare (1) is the `Service`'s process, specifying the OS-level process that the `Service` will run in. This is an optional attribute—if you don't specify it, then the `Service` will run in the same process as your application.

Having a `Service` in the same process as your main application will change the way the OS classifies your application process. This is generally good (it'll be less likely that your application process will be killed to free up memory). But it also means that your application and `Service` share the same memory allocated to the process that they run in. This can cause your application to run low on memory more often and cause more garbage collections. That can lead to a laggy/jerky user experience, as sometimes the UI will be frozen while garbage collection occurs. By putting the `Service` in its own named process, you avoid this potential problem.

All you have to do is supply a `process` attribute. Now you might notice that the value of this attribute is `:stocks_background`. The colon prefix is significant—it indicates that this separate process is private to the application. The only application that can start or interact (bind) with the `Service` is going to be your application. If we removed the colon, then the `Service` would still be in its own process, but it would be a global process. If your `Service` provides some feature that you want other applications to have access to, then you might want to do this. We'll look at global `Services` later in this chapter.

Getting back to listing 5.1, the next thing we declare is the `Service`'s `name` attribute (2). This is the only attribute that's required in a `Service` declaration. It specifies

the class of the `Service` (relative to the package of your application, like for activities). Next, we declare two more optional attributes for our `Service`. These are the `icon` and `label` ❸. The Android OS allows users to see all running `Services` on their device and potentially stop them. The OS uses the `icon` and `label` when the user views this list of running `Services`, as shown in figure 5.1

Now that we've declared our `Service`, we still need to implement it. This is as easy as extending `android.app.Service`. You aren't required to do much in this extension, but you'll often want to override the `Service`'s lifecycle methods. Here's the basic structure of the `PortfolioManagerService`.

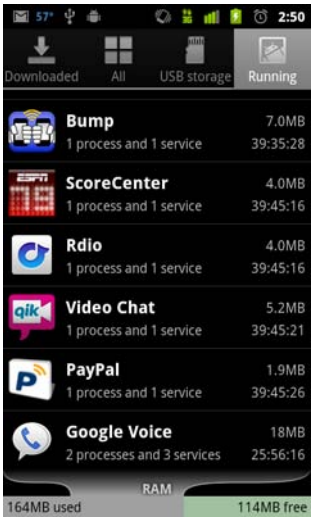


Figure 5.1 Viewing running Services

Listing 5.2 Declaring the `PortfolioManagerService`

```
public class PortfolioManagerService extends Service {
   @Override
   public void onCreate() {
      // ...
   }
   @Override
   public IBinder onBind(Intent intent) {
      // ...
   }
  @Override
  public void onDestroy() {
      // ...
   }
}
```

Start Service

❶ Establish communication channel

Release resources when Service is killed

The code in listing 5.2 shows the outline of our `Service` (we'll look at the details of its methods later). You only need to implement one method: `onBind` ❶. This method allows other components—typically activities or perhaps other `Services`—to communicate with the `Service`. Remember, a `Service` will usually be running in its own process, so communicating with it isn't as simple as invoking its methods. Interprocess communication (IPC) is necessary. The `onBind` method is where the IPC channel is established.

The other methods that we chose to override in listing 5.2 are `onCreate` and `onDestroy`. These are optional. If your `Service` does all of its work within the `onBind` (an example might be uploading data to a remote server), then you may not need to override `onCreate`. If you need to do some processing outside the context of an `onBind` call, then you'll probably set that up in the `onCreate` method. Finally, as the

name suggests, `onDestroy` is called when a `Service` is being killed. You should release any resources being used by your `Service` here.

DISCUSSION

You've seen all the basics of declaring and creating a `Service`. There are some key things to take away from this. First, the `Service` will run in its own process. This decouples it from the application's process, so that it won't be terminated when the application is terminated. Second, because it's in its own process, you can only communicate with it through IPC. We'll get into the mechanics of how to do this on Android later in this chapter. Before we do, one more lifecycle method is worth mentioning. Many applications will want to implement the `onStartCommand` (or the deprecated `onStart`, if you're developing for pre-Android 2.0 devices). This allows additional parameters to be passed to the `Service` when it's first started. If you want to expose some configuration parameters of your `Service`, this is a common way to do it. An example might be to let the user decide on how often to check for new stock data. This assumes that you want to manually start the `Service` from your application. Often you'll want to automatically start the `Service` with no interaction from the user. Our next technique shows how to do this.

TECHNIQUE 14 Starting a Service automatically

One common use for a `Service` is periodically downloading information and potentially raising a `Notification` if a given condition is met. `Services` are well suited for this, but the question of when to start the `Service` now becomes significant.

PROBLEM

We want to show the user notifications if the price of a stock goes above or below certain levels. But we don't want to require the user to launch the application just to enable `Notifications`. Instead, we'd like our `Service` to begin running automatically, right after the device has booted up.

SOLUTION

The solution is to use a `BroadcastReceiver` to listen for Android's `BOOT_COMPLETED` event. This event is fired by the OS right after the device finishes booting up, which gives us an easy way to do something when the device is booted. To make this happen we need to declare it in our manifest as shown in the following listing.

Listing 5.3 Declaring a `BroadcastReceiver` for the boot complete event

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.flexware.stocks"
      android:versionCode="1"
      android:versionName="1.0">
...
      <receiver android:name="PortfolioStartupReceiver"
         android:process=":stocks_background">
            <intent-filter>
               <action android:name=
```

1 Declare BroadcastReceiver

2 Put in same process as Service

```
                "android.intent.action.BOOT_COMPLETED"/>
            </intent-filter>
        </receiver>
    </application>
...
</manifest>
```

Declare event to listen for (3)

In listing 5.3, we start off by declaring the `BroadcastReceiver`. This is similar to declaring a `Service` (it has many of the same attributes). We once again declare the class for the `BroadcastReceiver` by using the `name` attribute (1). Next, we declare that we want the `BroadcastReceiver` to be in a different process from our main application (2). If you compare this to listing 4.1, you'll see that we want it to be in the same process as our `Service`.

Going back to listing 5.3, the last important thing for us to declare about our `BroadcastReceiver` is what kind of events that it should listen to (3). We do this using the (hopefully) now familiar intent-filter paradigm. The `BOOT_COMPLETED` event (or action) is a predefined event in Android. In fact, there may be many other `BroadcastReceivers` listening for this event as well, and they'll all get a chance to do their thing when the device boots. Now that we've declared our `BroadcastReceiver`, we need to implement it. The next listing shows its implementation.

Listing 5.4 Starting our `Service` with a `BroadcastReceiver`

```
public class PortfolioStartupReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        Intent stockService =
            new Intent(context, PortfolioManagerService.class);
        context.startService(stockService);
    }
}
```

Start Service (1)

Our `BroadcastReceiver` couldn't be simpler. It creates a new `Intent` and uses that `Intent` to start the `Service` (1). This will cause the `onCreate` and then the `onStartCommand` methods to be invoked on our `Service`, and then return back to the `BroadcastReceiver`. Since a `BroadcastReceiver` should return quickly, those two methods on our `Service` should execute quickly as well. So if you need to do anything time-consuming in those methods, it's better to do such things in their own thread.

About installing on the SD card

One of the most-requested features for Android 2.2 was the ability to install apps on the SD card instead of on the internal memory. This seems like a great option for users, since much more space is available on the SD card than on the internal memory. If you choose to enable this though, be careful about relying on the device boot event as we've described in this section.

(continued)

The `BOOT_COMPLETED` event will be fired before the SD card is mounted, before your application is available. But there's another, similar event that you can listen for: the `ACTION_EXTERNAL_APPLICATIONS_AVAILABLE` event. This event will be fired after the SD card is mounted. If your app is on the SD card, it can listen for this event and start services at that point.

At the time this book was written, there was an open bug in Android (8485) that could prevent an app on the SD card from receiving this broadcast.

DISCUSSSION

You may be asking why we need to run the `BroadcastReceiver` in a different process. The answer is that it's often desirable to share objects between a `Service` and the `BroadcastReceiver` that started it or invoked it. We want the `BroadcastReceiver` and `Service` to be in the same process, so we don't have to use IPC. We'll see this technique later in this chapter when we discuss best practices for keeping your `Service` running continuously. In this case, it's not absolutely necessary. We'll see other cases where a `BroadcastReceiver` is invoked by the system's `AlarmManager` or by a push notification coming from Google's Cloud to Device Messaging service and then used to start our `Service` using this technique.

Finally, note that starting a `Service` at device boot isn't useful only for `Services` that can trigger `Notifications` to be sent. It's also useful if you're prefetching and caching data in the `Service`. When the user first opens your app, all of their data will already be loaded and ready to use—which is a positive experience for the user.

TECHNIQUE 15 Communicating with a Service

A `Service` can be used to perform useful tasks in the background. We saw a simple example in chapter 2 where the `Service` published `Notifications` for the user. But you'll usually want to send data back and forth to a `Service`. This is the case for our StockPortfolio service.

PROBLEM

We need to tell our `Service` what stocks to watch. For each stock, the `Service` needs to know two things: the ticker symbol, and the price levels at which the user should be notified. Since our `Service` is going to run in a different process, passing data to it isn't as simple as invoking a method on an object. We need some type of interprocess communication (IPC). Fortunately, the Android OS provides this.

SOLUTION

To send data to our `Service`, we need to use Android's IPC mechanism. This mechanism allows `Services` to be exposed to other processes and for serialized data to be sent between the processes. This is similar to enterprise IPC mechanisms such as CORBA and Windows COM. Those systems consist of an *interface definition language*

(IDL) to describe the interface of what's being exposed and a proxy class to be used by clients of the interface. Android uses a similar pattern. It even has its own IDL, known as *Android IDL* or *AIDL*. Here's an AIDL description of the interface that we want to expose to our `Service`.

Listing 5.5 IStockService.aidl: The external interface into the stock portfolio service

```
package com.flexware.stocks.service;

import com.flexware.stocks.Stock;

interface IStockService{
   void addToPortfolio(in Stock stock);
   List<Stock> getPortfolio();
}
```

1 Import another AIDL

2 Operations that will be exposed

As you can see from listing 5.5, AIDL looks a lot like Java. It uses packages and imports, like Java. The main difference is that you can only import other AIDL definitions. You'll notice in this case that we're importing a `Stock` object 1. This is the same `Stock` class that we'll use in the UI of our application (we'll see how this is done shortly). Our interface is simple. It only exposes two methods to the outside world 2. Note how this method uses the `Stock` type and how we mark this input parameter as `in`. This indicates that the parameter will be passed in, but its value won't be returned to the caller. It's needed here because `Stock` is a complex type. If it were a Java primitive type, it wouldn't be needed.

AIDL types and parameters

Marking an input parameter as in is similar to marking it as `final` in Java. You can modify the value of any input parameter, but if it's marked `in`, then its new value won't be passed back to the caller. The `in` modifier is known as a *directional tag*. There are two other possible values: `out` and `inout`. The `out` modifier indicates that whatever data you pass in will be ignored. A blank/default value will be created by the `Service`, and its final value will be passed back. An `inout` value indicates that a value should be passed in, and that it can be modified with its new value passed back. It's important to figure out what you need. Data sent through IPC must be marshalled and unmarshalled, which can be an expensive process. A parameter marked as `inout` will be marshalled/unmarshalled twice. As mentioned, you don't need to specify a directional tag for primitive values. These are `in` only—they're always immutable values.

This small definition can be used to generate a lot of code. If you're using the command line then you'll want to use the `aidl` tool. If you're using Eclipse, it'll automatically generate code from any .aidl files it finds in your project. It'll put the generated Java classes in the `/gen` directory (the same place it puts the generated R.java file.) For this to work, it needs to resolve that import reference. You'll need another .aidl file for this:

```
package com.flexware.stocks;

parcelable Stock;
```

This file (Stock.aidl) declares the `Stock` class reference in listing 5.5. It declares the package of the class, as AIDL does in listing 5.5, but all it does is reference a `Parcelable`. This Java class can be used in your application, but it can also be turned into an `android.os.Parcel`—serialized so that instances of this class can be sent between processes. The following listing shows this `Stock` class.

Listing 5.6 The `Stock` class, a `Parcelable` class that can be sent over IPC

```
public class Stock implements Parcelable{
   // user defined
   private String symbol;
   private double maxPrice;
   private double minPrice;
   private double pricePaid;
   private int quantity;
   // dynamic retrieved
   private String name;
   private double currentPrice;
   // db assigned
   private int id;
   private Stock(Parcel parcel){
      this.readFromParcel(parcel);
   }
   public static final Parcelable.Creator<Stock> CREATOR =
      new Parcelable.Creator<Stock>() {

      public Stock createFromParcel(Parcel source) {
         return new Stock(source);
      }

      public Stock[] newArray(int size) {
         return new Stock[size];
      }
   };
   public int describeContents() {
      return 0;
   }
   @Override
   public void writeToParcel(Parcel parcel, int flags) {
      parcel.writeString(symbol);
      parcel.writeDouble(maxPrice);
      parcel.writeDouble(minPrice);
      parcel.writeDouble(pricePaid);
      parcel.writeInt(quantity);
   }

   public void readFromParcel(Parcel parcel){
      symbol = parcel.readString();
      maxPrice = parcel.readDouble();
      inPrice = parcel.readDouble();
      pricePaid = parcel.readDouble();
```

1 Implement Parcelable interface

2 Private constructor for Parcel

3 Static factory called CREATOR

4 Serialize to Parcel

5 Deserialize from Parcel

```
        quantity = parcel.readInt();
    }
}
```

This listing shows all of the basics of making of a class that' s a `Parcelable`. The interface (1) only states that you need to implement the `writeToParcel` method (4). As the name of this method implies, this is the method where you serialize an instance of your class into a `Parcel` (2). As you can see from the listing, the `Parcel` class has useful methods for serializing primitives an d strings. This is all you have to implement so that an instance of the class can be sent to another process. But you need to deserialize that `Parcel` back into a `Stock`. To do this, the Android runtime will look for a static field called `CREATOR` (3) that will be of type `Parcelable.Creator`. This interface defines a factor y method called `createFromParcel`. In listing 5.6, we've given our `Parcelable` class its own `readFromParcel` method (5) that the `Creator` delegates to. Once again, the `Parcel` class has several methods to assist you in retrieving the serialized data from the `Parcel`. One key thing to notice here is that you must read values from the `Parcel` in the same order as you wrote them to the `Parcel`. For example, the `symbol` field is the first value written to the `Parcel` in the `writeToParcel` method, so it' s also the first field read from the `Parcel` in the `readFromParcel` method.

Now we have a data structure that can be sent back and forth between the process where our main application runs and the process where our background service runs. In listing 5.5, we defined the operations that the background ser vice exposes to the main application. A Java interface can be generated from the interface defined in the .aidl file. You can generate this manually using the `aidl` tool, or it'll be generated for you automatically if you're using Eclipse and the Android Developer Tools. In the following listing, you can see what this generated code looks like.

Listing 5.7 Java interface generated from AIDL interface

```
package com.flexware.stocks.service;
public interface IStockService extends android.os.IInterface
{
/** Local-side IPC implementation stub class. */
public static abstract class Stub extends android.os.Binder
➥     implements com.flexware.stocks.service.IStockService
{
// generated code
}
public void addToPortfolio(com.flexware.stocks.Stock stock)
➥     throws android.os.RemoteException;
public java.util.List<com.flexware.stocks.Stock> getPortfolio()
➥     throws android.os.RemoteException;
}
```

(1) Stub class

This is what you'd expect from the AIDL in listing 5.5. The interface and its two operations are directly translated. The only thing interesting is the `Stub` abstract class (1). As the name implies, this is a classic stub class that implements the interface (but not the

operations, which are still abstract), adding lots of generated boilerplate code. You'll want to extend this abstract class, implementing the `IStockService` methods, to leverage the generated boilerplate code. You'll also want to return your implementation class from the `onBind` method of your `Service`'s class. Take a look at the following to see how this works.

Listing 5.8 The `PortfolioManagerService` class

```
public class PortfolioManagerService extends Service {
   private final StocksDb db = new StocksDb(this);
   // Other methods omitted
   @Override
   public IBinder onBind(Intent intent) {
      return new IStockService.Stub() {
         public void addToPortfolio(Stock stock)
            throws RemoteException {
               db.addStock(stock);
            }

         public List<Stock> getPortfolio()
            throws RemoteException {
               return db.getStocks();
            }
      };
   }
}
```

Extend Service

Helper class for persisted data

(1) Return class that extends stub class

The `PortfolioManagerService` class shows you a typical `Service` that supports remote communication. You might recall that in chapter 2, we saw a `Service` that didn't support remote communication, so its `onBind` method returned null. Here, (1) we're supporting IPC, so we need to return a class that extends the generated `Stub` class from listing 5.7. In our example, we used an anonymous inner class that extended `Stub`, as our implementation is simple: we're delegating to a helper class `StocksDb`. This class uses Android's embedded SQLite database to save the stocks that the user wants retrieved on demand. A call to `addToPortfolio` will execute an insert statement and a `getPortfolio` call will execute a simple query. The last thing we want to do is show how this is used by the main application. The following listing shows the application's main `Activity` and how it binds and calls the `Service`.

Listing 5.9 The main `Activity` binding to the `Service`

```
public class ViewStocks extends ListActivity {

   private ArrayList<Stock> stocks;
   private IStockService stockService;
   private ServiceConnection connection = new ServiceConnection(){

      public void onServiceConnected(ComponentName className,
            IBinder service) {
         stockService = IStockService.Stub.asInterface(service);
```

Generated service interface

```
            try {
               stocks = (ArrayList<Stock>)
                     stockService.getPortfolio();
               if (stocks == null){
                  stocks = new ArrayList<Stock>(0);
               }
               refresh();
            } catch (RemoteException e) {
               Log.e(LOGGING_TAG, "Exception retrieving
                                   portfolio from service",e);
            }
         }
         public void onServiceDisconnected(ComponentName className) {
            stockService = null;
         }
      };
      @Override
      public void onStart(Bundle savedInstanceState) {
         super.onStart();
         bindService(new Intent(IStockService.class.getName()), connection,
           Context.BIND_AUTO_CREATE);
          ... // UI code omitted
       }
}
```

(1) Refresh UI when data is retrieved

(2) Bind to remote service

The code in listing 5.9 is a sampling of code from a `ListActivity`. The first thing we do in listing 5.9 is define a `ServiceConnection`, a delegate that will reflect the lifecycle of our connection to our remote service. We use the generated stub to take the remote service interface (represented as an `android.os.IBinder`) and get an implementation of the local interface. Next, in our Activity's `onStart` method, we use the `bindService` method (2), available on any `Context` object (such as an `Activity` or `Service`) to bind to the remote service. We pass in the name of the class of the service that we want to bind to, our connection delegate, and a flag indicating to automatically create the service if necessary. Invoking a service running in another process is much faster than making a call over the network, but it's still a slow operation that shouldn't be done on the main UI thread (`bindService` will cause this binding to happen asynchronously). The `onServiceConnected` method in the `ServiceConnection` acts as a callback to this asynchronous binding of the service. When it's called, we know that our service is bound and we can retrieve data from it and refresh the UI (1).

Visible processes and bound Services

In our example, the application and `Service` each run in their own process, but there's only so much memory to be spread out among these processes. For first-generation Android devices, this is generally 16 MB per process, and 24 MB per process on second-generation devices. So when all of those 16 or 24 MB pieces of the pie have been handed out, the OS must kill some processes. Different processes are viewed as being more or less important, as we discussed in chapter 3.

DISCUSSION

Communicating with a remote service is one of the more complicated techniques that you'll see. There are several steps in the process, but they're quite straightfor ward. Still, you can' t be blamed for wondering whether it' s worth all the trouble. What makes it more complex is that you're co mmunicating across processes. That means that a channel for communication must be created and data must be marshalled and unmarshalled as it goes between the processes. This is definitely worth it if you want to decouple the execution of your application fr om the user interacting with it. It' s one of the features of the Android platform that give it an advantage over its competitors. One common use case for this is to use a `Service` to manage and cache data from remote servers.

TECHNIQUE 16 Using a Service for caching data

A `Service` often needs to work with the same data as your main application. Both components can retrieve and manage this data. But as we saw in the previous section, it's possible for your main app to communicate with a `Service`. This makes it possible to have the `Service` manage all of the data, and if the data comes from over the Web, the `Service` can cache the data from the server.

PROBLEM

You have an application that also has a background `Service`. Both the main application and the `Service` need to use data from a remote ser ver. You want to centralize the access to this data in one place and cache it, since retrieving it over the network is slow and expensive. You want to do this from the background `Service`, so that it can retrieve the data even when the main application isn't being used and so that it can be exposed to the main application via IPC with the background `Service`.

SOLUTION

This is a common application pattern for Android apps. Part of why it's so common is because it's fairly straightfor ward. It buil ds on the other techniques that we've discussed so far . Your background `Service` can be started at device boot. Then it can retrieve data over the network. This can be done periodically, as needed. Finally, once the user launches your application, one of your app's activities can bind to the `Service` and invoke one of its methods to return the data that the `Service` downloaded from the network.

This simple pattern is followed by many popular Android apps. So how would we apply it to our stock po rtfolio application? For that app lication, the list of stocks that the user wants to track is mana ged locally, stored in a local SQLite database. To track the current price of the stock, we'll download this data over the network. To make all of this happen, we only need to modify our `Service`. Here's the new version.

Listing 5.10 Stock `Service` now with caching

```
public class PortfolioManagerService extends Service {
   private final StocksDb db = new StocksDb(this);
   private long timestamp = 0L;
```

1 Keep timestamp of last update

```
    private static final int MAX_CACHE_AGE = 15*60*1000;
       // 15 minutes
    @Override
    public IBinder onBind(Intent intent) {
       return new IStockService.Stub() {
          public Stock addToPortfolio(Stock stock)
                        throws RemoteException {
             Stock s = db.addStock(stock);
             updateStockData();
             return s;
          }

          public List<Stock> getPortfolio() throws RemoteException {
          ArrayList<Stock> stocks = db.getStocks();
          long currTime = System.currentTimeMillis();
          if (currTime - timestamp <= MAX_CACHE_AGE){
             return stocks;
          }
          Stock[] currStocks = new Stock[stocks.size()];
          stocks.toArray(currStocks);
          try {
             ArrayList<Stock> newStocks =
                               fetchStockData(currStocks);
             updateStockData(newStocks);
             return newStocks;
          } catch (Exception e) {
             Log.e("PortfolioManagerService",
                "Exception getting stock data",e);
             throw new RemoteException();
          }
       }
    };
}
... // code for retrieving stock data omitted
```

2 Cache data up to 15 minutes

3 Refresh cache whenever stock added

4 Use cached if fresh enough

5 Get data from server

6 Persist fresh data

The code in listing 5.10 expands on the `Service` first shown in listing 5.2. To allow for caching, we need a couple of things. We want to set a time limit (2) on how stale our cache can be before we bypass it and go back to the server. To determine the freshness of our cache, we need to keep track of the last time (1) we downloaded data from the server. Next, we need to add some cache management code to our two operations that we expose, `addToPortfolio` and `getPortfolio`. For `addToPortfolio`, we add the `Stock` to the local database, and then we call `updateStockData` (3). This method will retrieve data from the network, and then update the stocks stored in our local database. We'll look at its code shortly. Because we added a new stock, we need to get information about it from the network, so we might as well get information about all of our stocks and update our cache.

For the `getPortfolio` method, we start by retrieving the cached data from our local database and see if this data is fresh enough. In the previous listing, we set a simple policy of allowing cached data to be used if it's less than 15 minutes old. You could imagine a much more sophisticated caching policy, where you'd be more aggressive if the current time was during stock market trading hours, but otherwise passive. This

policy is good enough for our application, so we check if the current time minus the last timestamp is less than 15 minutes (4). If so, then we return the cached data. Otherwise, we retrieve data from the network (5) and then update our cache (6) with the fresh data. We do this by calling another variant of `updateStockData`.

Listing 5.11 Updating cached stock data

```
private void updateStockData() throws IOException{
   ArrayList<Stock> stocks = db.getStocks();
   Stock[] currStocks = new Stock[stocks.size()];
   currStocks = stocks.toArray(currStocks);
   stocks = fetchStockData(currStocks);
   updateStockData(stocks);
}

private void updateStockData(ArrayList<Stock> stocks){
   timestamp = System.currentTimeMillis();
   Stock[] currStocks = new Stock[stocks.size()];
   currStocks = stocks.toArray(currStocks);
   for (Stock stock : currStocks){
      db.updateStockPrice(stock);
   }
   checkForAlerts(stocks);
}
```

(1) Get stocks
(2) Get fresh data
(3) Update cached data
(4) Update latest price of the stock

These two methods are what the `Service` uses to refresh its cached data. The first method takes no arguments and is used when the user adds a new stock. It retrieves the full list of stocks (1) that the user is monitoring by retrieving this data from the local database. Then it uses the `fetchStockData` method (2) to get the latest information on the `Stock` from the network. Finally, it delegates to the second method (3), which takes in a list of `Stock` objects and updates their prices in the database. This method then iterates over the list of `Stocks`, and updates the price of each `Stock` (4).

DISCUSSION

Caching of data can make a huge difference in the performance of any application. The more expensive that data is to retrieve, the bigger the benefit of caching it will be. This is true for mobile applications, which often rely heavily on data from remote servers. The network connection speeds on mobile networks are generally never great, and are often quite slow. Storing data in a local database is a great way to cache that data. Putting all of the management of that data into a background `Service` allows its retrieval/updates to be done in the background, and not be tied to the user using the application. Having this data in the background `Service` allows that `Service` to do other things with that data. A common example of this is to create notifications based on the data that's retrieved from the server.

TECHNIQUE 17 Creating notifications

`Notifications` are one of the most significant features of mobile applications. They allow your application to interact with users in an asynchronous manner—the users don't have to be directly interacting with your application (have it open) in order for

your application to communicate important, time-critical information. It should come as no surprise that background `Services` are integral to such notifications, as they're the key feature of the Android platform that enables your application to operate in an asynchronous manner.

PROBLEM

You want to alert your user when some significant events happen, even if your users aren't using your application at the time of that event. You want to provide them with detailed information about this event, and ma ke it actionable so that they can immediately use your application to respond ap propriately to the event. The event may come from a remote system, or it might be local to the device. Either way, you want to incorporate all of the various capabilities of Android to alert users, so that they can act on the event in a meaningful way.

SOLUTION

The Android platform offers a flexible an d extensible notification system. The simplest type of notification offered by Android is known as a *toast notification*, or a *toast*. Toasts are often used by an `Activity` to alert the user to an event, but they can also be launched from a `Service`. Toasts are designed to display information to the user—they're not interactive. T o get the kind of interactivity we desire, we need to use an `android.app.Notification`. A `Notification` allows the user to interact with your application by wrapping an `Intent`. It can be displayed on the status bar , create a sound, vibrate the phone, and even trigger custom colored flashing LEDs.

For our stock portfolio application, users can enter a minimum and maximum price level for each of the stocks in their portfolio. Each time we download the latest price information from the network, we want to check whether any of the stock prices have gone below the minimum price or e xceeded the maximum price. The following listing shows how we can add this logic to our `Service`.

Listing 5.12 Checking maximum and minimum levels

```
private void updateStockData(List<Stock> stocks){
   // existing code omitted
   checkForAlerts(stocks);
}

private void checkForAlerts(Iterable<Stock> stocks){
   for (Stock stock : stocks){
      double current = stock.getCurrentPrice();
      if (current > stock.getMaxPrice()){
         createHighPriceNotification(stock);
         continue;
      }
      if (current < stock.getMinPrice()){
         createLowPriceNotification(stock);
      }
   }
}
```

1 Check for alerts after update

2 High price notification

3 Low price notification

The easiest way to add the price alert checking logic is to call it (1) after we update our locally cached data with new data from the network. This involves iterating over each stock and creating a specific `Notification` depending on whether the current price is higher (2) than the user's maximum or lower (3) than the user's minimum price. Note that we've created a specific method for creating each of these different `Notifications`. Here's how we create high-price `Notifications`.

Listing 5.13 Creating a high price `Notification`

```
private static final int HIGH_PRICE_NOTIFICATION = 1;
private void createHighPriceNotification(Stock stock) {
   NotificationManager mgr = (NotificationManager)
      getSystemService(Context.NOTIFICATION_SERVICE);
   int dollarBill = R.drawable.dollar_icon;
   String shortMsg = "High price alert: " + stock.getSymbol();
   long time = System.currentTimeMillis();
   Notification n = new Notification(dollarBill, shortMsg, time);
   String title = stock.getName();
   String msg = "Current price $" + stock.getCurrentPrice() +
      " is high";
   Intent i = new Intent(this, NotificationDetails.class);
   i.putExtra("stock", stock);
   PendingIntent pi = PendingIntent.getActivity(this, 0, i, 0);

   n.setLatestEventInfo(this, title, msg, pi);
   n.defaults |= Notification.DEFAULT_SOUND;
   long[] steps = {0, 500, 100, 200, 100, 200};
   n.vibrate = steps;
   n.ledARGB = 0x80009500;
   n.ledOnMS = 250;
   n.ledOffMS = 500;
   n.flags |= Notification.FLAG_SHOW_LIGHTS;
   mgr.notify(HIGH_PRICE_NOTIFICATION, n);
}
```

Get Notification service

Notification with ticker info (1)

(2) Intent for launch

Expanded Notification info

(3) Add sound

(4) Vibrate phone

(5) Flash lights

The method in listing 5.12 shows many of the options available for creating `Notifications`. At its most basic, you need to create the information that will be shown on the status bar (ticker). This includes an icon (image) (1), a short message, and when the `Notification` should be shown. We could stop here, but we want the `Notification` to be actionable. To do this, we want to start an `Activity` when the user selects the `Notification`. To do that, we need an `Intent` (2). Note that the `Stock` object that the `Notification` pertains to is added to the `Intent` as an extra. We can do this because the `Stock` class is a `Parcelable`, the OS can easily serialize/deserialize a `Stock` object. The `Intent` then gets wrapped in a `PendingIntent`—an `Intent` that will be activated sometime in the future.

The rest of the code shows some of the other options available to you for making the user notice your `Notifications`. You can have the device play a sound (3). In this case, we used the default sound that the user has set for `Notifications`. You could also include a sound file with your application and use it here instead. Next, we have

the device vibrate (4) when the `Notification` is sent. We pass in an array of longs for this. The first value in the array is how long to wait until the vibration start. After that, it's a pattern of values, alternating how long the vibration should be on and then how long it should be off. Once the end of the array is reached, the phone will stop vibrating. Finally, we can also make the LEDs on the phone flash (5). The presence and type of these lights varies from device to device, but if you specify something that the device can't do, the OS will degrade this appropriately . In this case, we specified an ARGB hexadecimal color (green) for the LED, and then an on/off pattern. In this case, the pattern will be repeated indefinitely.

If/when the user expands the status bar to see more information about the `Notification`, they'll be shown the `contentTitle` and `contentText`. In listing 5.12, we specified these values using the `setLatestEventInfo` method. This method also takes the `PendingIntent` that we created, so that if the user taps on the `Notification` then the `Intent` that was wrapped by the `PendingIntent` will be used to start the `Activity` associated with it. This is a convenience method that allows you to specify these values and combines them with a predefined view You can also specify your own custom view The next listing shows a custom view being used to create the `Notification` for low prices.

Listing 5.14 Creating a low price `Notification`

```
private static final int LOW_PRICE_NOTIFICATION = 0;
private void createLowPriceNotification(Stock stock){
   NotificationManager mgr = (NotificationManager)
      getSystemService(Context.NOTIFICATION_SERVICE);
   int dollarBill = R.drawable.dollar_icon;
   String shortMsg = "Low price alert: " + stock.getSymbol();
   long time = System.currentTimeMillis();
   Notification n = new Notification(dollarBill, shortMsg, time);
   String pkg = getPackageName();
   RemoteViews view =
        new RemoteViews(pkg, R.layout.notification_layout);
   String msg = "Current price $" + stock.getCurrentPrice() +
        " is low";
   view.setTextViewText(R.id.notification_message, msg);
   n.contentView = view;
   Intent i = new Intent(this, NotificationDetails.class);
   i.putExtra("stock", stock);
   PendingIntent pi = PendingIntent.getActivity(this, 0, i, 0);
   n.contentIntent = pi;
   n.defaults |= Notification.DEFAULT_SOUND;
   long[] steps = {0, 500, 100, 500, 100, 500, 100, 500};
   n.vibrate = steps;
   n.ledARGB = 0x80A80000;
   n.ledOnMS = 1;
   n.ledOffMS = 0;
   n.flags |= Notification.FLAG_SHOW_LIGHTS;
   mgr.notify(LOW_PRICE_NOTIFICATION, n);
}
```

(1) Get RemoteViews

Set text on View

Set View to be used

Set PendingIntent

The createLowPriceNotification in listing 5.13 is similar to createHighPriceNotification. The messaging, icons, vibration pattern, and lights are a little different, but these are the same APIs that we saw in listing 5.12. The significant difference is that we no longer use the setLastEventInfo method on the Notification object. Instead, we use a custom View. The tricky part about creating a View in this situation is that we're creating it from our background Service, which is running in a separate process from whatever application that the user is currently viewing. In fact, since this is executing from within a Service, we can't even use the layout inflater system service, since it needs an Activity to inflate a View. Fortunately, Android has the RemoteViews class to deal with this situation. It only needs the package name of our application and an XML view (1) to inflate the View. Here's the View that we're going to inflate.

Listing 5.15 Custom XML layout used for a Notification

```
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/notification_layout_root"
  android:orientation="horizontal"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:padding="5dp">
    <ImageView android:id="@+id/notification_icon_left"
               android:layout_width="wrap_content"
               android:layout_height="fill_parent"
               android:layout_marginRight="5dp"
               android:src="@drawable/radioactive_icon"
               />
    <TextView android:id="@+id/notification_message"
              android:layout_width="wrap_content"
              android:layout_height="fill_parent"
              android:textColor="#000"
              />
    <ImageView android:id="@+id/notification_icon_right"
               android:layout_width="wrap_content"
               android:layout_height="fill_parent"
               android:layout_marginLeft="5dp"
               android:src="@drawable/radioactive_icon"
               />
</LinearLayout>
```

(1) TextView to display message

The View for the Notification is a simple LinearLayout that flows horizontally. It has a text message (1) flanked by icons to its left and right. For the text message, we use a TextView with an ID so that we can retrieve it and set its text. This needs to be done from the setLowPriceNotification method, but that's part of our background Service. The familiar findViewById method is only available from an Activity, not from a Service. Fortunately, the RemoteViews class has a variety of methods to work around this. Back in listing 5.13, you can see that we used the setTextViewText method to set

the text value of the message that will be shown in our `Notification`. The `RemoteViews` class has several other similar methods to handle variations on this situation.

Once the `View` is created and ready, it's set as the `contentView` of the `Notification`. Also note that we needed to set the `contentIntent` of the `Notification` as well. We didn't have to do this in the `setHighPriceNotification` method because we used the `setLastEventInfo` method that took care of this for us.

DISCUSSION

Android provides application developers with a rich set of APIs for creating and managing and `Notifications`. We've gotten a good look at many of them in this technique. Now do you really want to play a sound, vibrate the phone for several seconds, and flash the LEDs every time you need to send a `Notification`? This is a rhetorical question on the way to the bigger question: what's the point of all these literal bells and whistles for `Notifications`? After all, if you compare it to other popular mobile platforms, you get many more capabilities, but is that necessarily a good thing? Like any other feature, it's possible to go overboard. But these rich capabilities give you many opportunities to create distinctive `Notifications` for your application, and that's valuable.

Remember that `Notifications` are usually raised while the user is using a different application, or perhaps even more commonly, while the user is not using the phone at all. Maybe it's sitting in their pocket or lying on the desk in front of them. If your `Notification` is distinctive, they'll recognize that a `Notification` is from your application without even viewing it on their phone. This makes them much more likely to react to your `Notification`, and in turn your application—which is a good thing.

The combination of background `Services` and `Notifications` is powerful and compelling. But to make it work effectively we need to understand scheduling and how this interacts with your `Service`'s lifecycle.

5.3 Scheduling and Services

Running in the background on a traditional desktop computer or server is fairly straightforward. It's much more complicated on a mobile operating system like Android, where memory is more scarce. Anything that's running in the background could be killed by the OS to free up memory to be used by an application that the user is interacting with. This feature of the OS is great for the user, as it ensures that their applications are always responsive, but it doesn't make life easy on application developers. If you want to run in the background indefinitely, then you can't assume that you can start a `Service` and let it go. You must assume that the OS will kill it and that you'll need to resurrect it. You need some hooks into the OS to do this, and fortunately, Android provides them. Traditionally, this has been accessing the system alarm services via Android's `android.app.AlarmManager` class. With the introduction of Android's Cloud to Device Messaging service in Android 2.2, developers have another way of doing this by sending wake-up calls from their servers to their `Service` on a

specific device. In this section, we'll learn about various techniques for using these parts of the Android platform to make your background Services more robust.

TECHNIQUE 18 Using the AlarmManager

The Linux gurus out there will surely be familiar with Linux's system-level alarms and timers. These utilities are available to Android processes as well. But you don't need to read the manual. Instead, Android provides a simple Java API for setting system-level alarms, including both one-time and repeating alarms. It's the key API in Android for executing your program at some point in the future and making sure it happens even if your application or `Service` isn't running at that time.

PROBLEM

Your `Service` needs to execute code at some point in the future. But even though your `Service` may be currently running, you can't guarantee that it'll still be running at that point. If that was the case—or if it was okay for your code to not execute if your `Service` isn't running in the future—then you could use a combination of Java's `Timer` and `TimerTask` along with Android's `Handler`. The following listing shows such a naïve implementation.

Listing 5.16 Using a `Timer` and a `Handler` to schedule `Services` (DON'T DO THIS!)

```
Calendar when = Calendar.getInstance();
when.add(Calendar.MINUTE, 2);
final Handler handler = new Handler();
TimerTask task = new TimerTask(){
   @Override
   public void run() {
      handler.post(new Runnable(){
         public void run() {
            updateStockData();
         }
      });
   }
};
Timer timer = new Timer();
timer.scheduleAtFixedRate(task, when.getTime(), 15*60*1000);
```

If you can live with your `Service` and scheduled operations being killed by the OS, then use code like listing 5.15. This code will call the `updateStockData` method that we saw in listing 5.11. The first call will be two minutes from the current time. After that, it'll be called every 15 minutes, for as long as the `Service` is running. This is the desired behavior, except for the "for as long as the `Service` is running" part. Instead we'd like to change this "for as long as the device is turned on."

SOLUTION

To ensure that our code is executed at the desired time, we can't rely on the `Service` because the OS could kill the `Service` to free up memory. We must use the OS to schedule the execution, and to do this we must use the `android.app.AlarmManager` class. This system service is like the layout inflat or or notification manager services. In our

stock portfolio application, we've already created a `BroadcastReceiver` that's invoked when the device finishes booting up. Currently it starts the `Service` at that time, but here you see a new version that instead schedules the `Service` to be executed.

Listing 5.17 Using a device boot receiver to schedule `Service` execution

```
public class PortfolioStartupReceiver extends BroadcastReceiver {
   private static final int FIFTEEN_MINUTES = 15*60*1000;
   @Override
   public void onReceive(Context context, Intent intent) {
      AlarmManager mgr = (AlarmManager)
         context.getSystemService(Context.ALARM_SERVICE);

      Intent i = new Intent(context, AlarmReceiver.class);
      PendingIntent sender = PendingIntent.getBroadcast(context, 0,
         i, PendingIntent.FLAG_CANCEL_CURRENT);
      Calendar now = Calendar.getInstance();
      now.add(Calendar.MINUTE, 2);
      mgr.setRepeating(AlarmManager.RTC_WAKEUP,
         now.getTimeInMillis(),FIFTEEN_MINUTES, sender);
   }
}
```

Get AlarmManager

1 Create Intent to be scheduled

2 Schedule Intent

If you compare listings 5.16 and 5.4, you'll see that we've changed the implementation of the `onReceive` method. Now instead of starting the `Service`, we'll schedule it. We create an `Intent` (1) for the `BroadcastReceiver` that will receive the alarm from the `AlarmManager`. Note that we once again wrap the `Intent` in a `PendingIntent`, similar to what we did for a `Notification`. This is because the `Intent` won't be executed now, but in the future. Then we use the `AlarmManager` (2) to schedule the `PendingIntent` for execution. By specifying the type of alarm as `RTC_WAKEUP`, we're instructing the OS to execute this alarm even if the device has been put to sleep (that's what the *wakeup* suffix represents; the RTC part says we're measuring start time in absolute system time). We've set the alarm to first go off in two minutes from the current time, and then to go off every 15 minutes subsequently. Note that our `Intent` wasn't for the `Service` directly, but instead for a class called `AlarmReceiver`. The following listing shows this class.

Listing 5.18 `AlarmReceiver`, a `BroadcastReceiver` for handling system alarms

```
public class AlarmReceiver extends BroadcastReceiver {

   @Override
   public void onReceive(Context context, Intent intent) {
      Intent stockService =
         new Intent(context, PortfolioManagerService.class);
      context.startService(stockService);
   }

}
```

This class should look familiar. It's equivalent to the original `PortfolioStartupReceiver` class shown in listing 5.4. All it does is create an `Intent` for the `PortfolioManagerService` and then immediately start that `Service`. But now we want that

Service to update the stock data and check whether it needs to send Notifications to the user. The next listing shows how we need to modify the Service.

WHAT'S IN THE INTENT? You might notice that the AlarmReceiver's onReceive method has an Intent passed in to it, per the onReceive method's specification from BroadcastReceiver. This is the same Intent you created in the PortfolioStartupReceiver, wrapped in a PendingIntent. It's not exactly the same, because it could be serialized and then deserialized. But any extended data added (using the Intent's putExtra methods) to the Intent created in listing 5.16 will be present in the Intent received in listing 5.18, and can be retrieved using the getExtra methods.

Listing 5.19 Modified `Service` to work with system alarms

```
public class PortfolioManagerService extends Service {
   // other code omitted

   @Override
   public int onStartCommand(Intent intent, int flags, int startId) {
      updateStockData();
      return Service.START_NOT_STICKY;
   }
}
```

To get our Service to work properly with the system alarms, we need to override another of android.app.Service's lifecycle methods: the onStartCommand method. This method will be invoked each time a client context calls startService, such as in listing 5.18, even if the Service is already running. All we want to do is call our updateStockData method, since it'll take care of retrieving fresh data from the network, updating the locally cached data in our database, checking whether we need to send out Notifications, and send them out if so.

Note that this method must return an integer. The value of that integer tells the OS what to do with the Service if it's killed by the OS. The START_NOT_STICKY flag indicates that the OS can forget about this Service if it has to kill it. That makes sense in this example, since we know that we have an alarm scheduled to restart the Service later. Alternatively, we could've returned START_STICKY. This would instruct the OS to restart the Service itself.

SERVICE ONSTART VERSUS ONSTARTCOMMAND If you dig around the Internet looking for examples of starting a Service periodically, you might see code that overrides onStart instead of overriding onStartCommand as we did in listing 5.18. This older lifecycle method was deprecated in Android 2.0. It has no return value, unlike onStartCommand, so it can't provide the OS any information on what to do if the Service is killed. You should always use onStartCommand, unless you need to write code specifically for devices running pre-2.0 versions of Android.

DISCUSSION

Using the `AlarmManager` sounds harmless enough. After all, it's another set of APIs that are part of the Android platform. But it's powerful. It allows us to decouple the execution of background code from the process executing that background code. Take a look at the `Service` that we've developed up to this point. It'll start up two minutes after a device boots, and will then poll data from the Internet every 15 minutes until the device shuts down. The device could even be asleep, and our alarm will still execute. To get this behavior, all we had to do was specify an alarm type (`RTC_WAKEUP`) when we scheduled the alarm.

Behind the scenes, the `AlarmManager` must obtain a *wake lock* to prevent the device from going to sleep. This wake lock is held while the `onReceive` method of the `BroadcastReceiver` that receives the alarm is executing. In this case, that `BroadcastReceiver` is our `AlarmReceiver` class shown in listing 5.17. But once its `onReceive` method returns, it again becomes possible for the device to go to sleep, and for your `Service` to stop executing. Our next technique discusses how you can prevent this from happening.

TECHNIQUE 19 Keeping Services awake

In the previous technique, we learned about the `AlarmManager`, and in particular how it can help us to resurrect our killed `Service`. But that resurrection could be short-lived. Having the alarm go off isn't good enough. We also want to make sure that we finish the work that the `Service` needs to do—retrieve fresh stock data from the Internet and send out `Notifications` if needed. To do this, we'll need to use some of Android's power management APIs, and we'll need to think carefully about Android processes.

PROBLEM

If a device is asleep, we still want our `Service` to execute. We want it to keep the device awake long enough to create `Notifications` for the user. We don't want our users to not receive `Notifications` because their device was asleep in their pocket.

SOLUTION

To solve this problem, we'll need to use Android's PowerManager API. This is another system service on Android, and it allows us to control the power state on the device. Using this API, we can acquire what Android calls a *wake lock*. Acquiring a `WakeLock` allows your application to prevent the OS from putting the device to sleep (turning off the CPU). This is a significant capability that the OS provides to developers, and you must list it as a `<uses-permission>` in your AndroidManifest.xml file. Obviously if you misuse this, you'll severely affect the battery life of a device. With that in mind, there are several different types of wake locks. The most common type is the `PARTIAL_WAKE_LOCK`. This turns on the CPU, but keeps the screen (and if the device has a physical keyboard, the keyboard's backlight) turned off. Considering that the screen on a device is typically the single biggest drain on the battery, it's best to use a `PARTIAL_WAKE_LOCK` when possible. It also has the advantage that it won't be affected if the user

presses the power button on the device. The other types of wake locks—SCREEN_DIM_WAKE_LOCK, SCREEN_BRIGHT_WAKE_LOCK, and FULL_WAKE_LOCK—all turn the screen on, but because of that, the user pressing the power button can also dismiss them. It should come as no surprise that for a background Service, we definitely want to use a PARTIAL_WAKE_LOCK.

At this point, the solution to our problem may seem obvious. We can add code to our Service to acquire a WakeLock during its onStartCommand method, and then release it after we finish checking for Notifications. But there's a big problem with that approach. If the device is asleep, then the WakeLock acquired by the AlarmManager will be released once the onReceive method of our AlarmReceiver class finishes. This can (and will) happen before the onStartCommand of our Service is invoked. The device could go back to sleep before we even get a chance to acquire a WakeLock. Therefore, we must acquire a WakeLock in the onReceive method of AlarmReceiver, since that's the only place we're guaranteed that execution won't be suspended. Here's the new modified version of AlarmReceiver.

Listing 5.20 Modified AlarmReceiver, now with power management

```
public class AlarmReceiver extends BroadcastReceiver {
   private static PowerManager.WakeLock wakeLock = null;
   private static final String LOCK_TAG = "com.flexware.stocks";
   public static synchronized void acquireLock(Context ctx){
      if (wakeLock == null){
         PowerManager mgr = (PowerManager)
                        ctx.getSystemService(Context.POWER_SERVICE);
         wakeLock =
                        mgr.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
                            LOCK_TAG);
        wakeLock.setReferenceCounted(true);
      }
      wakeLock.acquire();
   }
   public static synchronized void releaseLock(){
      if (wakeLock != null){
         wakeLock.release();
      }
   }
      @Override
   public void onReceive(Context context, Intent intent) {
      acquireLock(context);
      Intent stockService =
         new Intent(context, PortfolioManagerService.class);
      context.startService(stockService);
   }
}
```

Shared WakeLock

Static method for acquiring

Static method for releasing

Acquire WakeLock before starting Service

The AlarmReceiver has received a major makeover. It has a WakeLock instance as a static variable. In addition, it also has two methods for acquiring and releasing the WakeLock. We used a static WakeLock with static acquire/release methods so that this can be shared between the AlarmReceiver instance and our background Service.

Normally, to share with a `Service` that you're starting, you'd pass it as part of the `Intent` (typically as an extra), but anything passed as part of the `Intent` must be a `Parcelable`. A `WakeLock` is a representation of a system setting, it's definitely not a `Parcelable`. So we use static variables and static methods to work around this.

Keep in mind that for this technique to work, `AlarmReceiver` and our `Service` must be running in the same process, or you'll face a tricky bug. If this is the case, then the same class loader will load them, and they'll share the static`WakeLock`. Otherwise they'll be in different class loaders and will have different copies of the `WakeLock`. Here's the declaration of `AlarmReceiver` from our AndroidManifest.xml file:

```
<receiver android:name="AlarmReceiver"
   android:process=":stocks_background" />
```

Now compare this to listing 5.1, and in particular the declaration of the `PortfolioManagerService`. Both components have `android:process=":stocks_background"`. Both will be run in a process outside of th e main application process, and will be in the same process. With this configuration, the technique will work. Now we need to add code to `PortfolioManagerService` to release the `WakeLock` so that the device can go back to sleep. The following listing shows the modified `checkForAlerts` method, now with power management code.

Listing 5.21 Releasing the `WakeLock` after checking for alerts

```
private void checkForAlerts(Iterable<Stock> stocks){
   try{
      for (Stock stock : stocks){
         double current = stock.getCurrentPrice();
         if (current > stock.getMaxPrice()){
            createHighPriceNotification(stock);
            continue;
         }
         if (current < stock.getMinPrice()){
            createLowPriceNotification(stock);
         }
      }
   } finally {
      AlarmReceiver.releaseLock();
      stopSelf();
   }
}
```

The main thing that we've done to this method is wrap its code in a `try-finally` sequence. Inside the `finally` block, we invoke the `releaseLock` static method from `AlarmReceiver`, and release the `WakeLock` that we a cquired during `AlarmReceiver`'s `onReceive` method.

DISCUSSION

It's important to think about the effect that the preceding code will have on batter y life. The CPU is going to be woken up to make a network call, update a local database, and possibly create `Notifications`. Without the power management code we added,

this wouldn't happen when the device is asleep. This whole process could take a few seconds, since it involves a network call. But we didn't turn on the screen, minimizing how much extra power is consumed.

Another thing to keep in mind is that a couple of other flags can be set on WakeLocks. These flags determine whether acquiring the WakeLock should cause the screen to turn on. Normally WakeLocks keep the screen from turning off, but with these extra flags they can also cause it to turn on if it's turned off. But those flags don't work with the PARTIAL_WAKE_LOCK type that we used. The PARTIAL_WAKE_LOCK is made for the "wake up, but stay in the background" kind of task like we're trying to accomplish with our Service. It's important that the Notifications that we create do more than create ticker text on the screen. The screen may be turned off, and we can't turn it on, so the user wouldn't see such Notifications. That's not a problem in our application, where our Notifications make a sound, vibrate the phone, and flash its LEDs. We didn't need to do all three of those things, but it's good that we did at least one of them.

TECHNIQUE 20 Using Cloud to Device Messaging

So far in this section, we've concentrated on how we can use the Android OS to schedule execution of our Service. The main driver for this was that we wanted our service to poll an Internet server to get fresh data about stocks. But polling is inherently inefficient. Most of your polls don't result in data that requires your Service to generate a Notification, so you poll too much. On the other hand, there will always be some window of time where an event has happened that you'd like to give your user a Notification about, but your Service hasn't polled yet, so you don't know about the event yet. You don't poll enough. In our application we're polling every 15 minutes. But you can imagine that with the volatility of the stock market, this interval may be unsatisfactory to the user. We can poll more often, but this will definitely have an effect on the battery life of the device. Android's Cloud to Device Messaging service provides an elegant alternative to this.

PROBLEM

We want to immediately notify our users of important events. The less time between when the event happens and when the user sees a Notification, the more valuable our application will be to the user. But extremely frequent polling will have a negative effect on battery life, and may also overly tax the servers that our background Service is polling. Further, as we've seen, the code to make background polling robust is complicated.

SOLUTION

If you took a poll of Android developers and asked them what the most important new feature in Android 2.2 (Froyo) was, many of them would instantly say Cloud to Device Messaging (C2DM). This is Android's answer to Apple Push Notification Service (APNS), only it has many advantages over APNS. With C2DM, remote web servers can send Intents to specific applications on specific Android devices. For our sample application, we can use C2DM to allow a server to tell our background Service to refresh its cache

and check for `Notifications`. To use C2DM requires a few steps of setup and several permissions. Here are some of the new additions to our AndroidManifest.xml.

Listing 5.22 Update manifest with C2DM permissions

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.flexware.stocks"
      android:versionCode="1"
      android:versionName="1.0">
   <application android:icon="@drawable/icon"
         android:label="@string/app_name">
      <!-- Code omitted -->
      <receiver android:name=".PushReceiver"
         android:permission=
            "com.google.android.c2dm.permission.SEND">
         <intent-filter>
            <action android:name=
                  "com.google.android.c2dm.intent.RECEIVE" />
            <category android:name="com.flexware.stocks" />
         </intent-filter>
         <intent-filter>
            <action android:name=
               "com.google.android.c2dm.intent.REGISTRATION"/>
            <category android:name="com.flexware.stocks" />
         </intent-filter>
      </receiver>   </application>
   <uses-sdk android:minSdkVersion="8" />
   <uses-permission android:name="android.permission.INTERNET"/>
   <permission android:name="com.example.myapp.permission.C2D_MESSAGE"
      android:protectionLevel="signature" />
   <uses-permission android:name=
      "com.example.myapp.permission.C2D_MESSAGE"/>
   <uses-permission android:name=
      "com.google.android.c2dm.permission.RECEIVE"/>
   <uses-permission android:name=
      "android.permission.MANAGE_ACCOUNTS"/>
   <uses-permission
      android:name="android.permission.WAKE_LOCK"/>
</manifest>
```

1 Declare receiver

What messages receiver should get 2

Handle C2DM registration messages 3

Check for Android 2.2

4 Permissions for C2DM

5 Access accounts

Need power management

Our manifest has a new `BroadcastReceiver` declared 1, called `PushReceiver`. We'll take a closer look at that class momentarily. It'll handle both registration messages 3 from the C2DM servers and app-specific messages 2 from our app servers, routed through the C2DM servers. We also need several new permissions for C2DM 4. Finally, we're going to access account information 5 as well. This isn't required for C2DM, but there are advantages to using this information, as we'll see shortly. Now that we see the permissions and declarations needed, let's take a look at initiating the C2DM registration process.

Listing 5.23 Requesting C2DM registration

```
public class PortfolioStartupReceiver extends BroadcastReceiver {
   private static final String DEVELOPER_EMAIL_ADDRESS = "...";
```

1 Your email address

```
    @Override
    public void onReceive(Context context, Intent intent) {
        Intent registrationIntent =
            new Intent("com.google.android.c2dm.intent.REGISTER");
        registrationIntent.putExtra("app",
            PendingIntent.getBroadcast(context, 0,
                new Intent(), 0));
        registrationIntent.putExtra("sender", DEVELOPER_EMAIL_ADDRESS);
        context.startService(registrationIntent);
    }
}
```

As you can see in listing 5.22, we've once again modified the `PortfolioStartupReceiver` class that gets invoked when the device boots up. Now instead of using the `AlarmManager` here to schedule the execution of our `Service`, we're going to rely on C2DM. But we need to register for C2DM messages. This is a process where we tell the C2DM servers that our app wants to receive C2DM messages. The C2DM servers will respond by providing a registration ID. The code in listing 5.22 starts this process by requesting a registration ID. Most of this is generic code, and the only thing that you must supply is the email address (1) that you've used in conjunction with your Android apps. Once the device boots up, the receiver will send out this registration request. We need another `BroadcastReceiver` to handle the response from the C2DM servers (we saw this receiver declared in listing 5.21). In the next listing, you can see how it's implemented.

Listing 5.24 Registration and messaging receiver

```
public class PushReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        AlarmReceiver.acquireLock(context);
        if (intent.getAction().equals(
            "com.google.android.c2dm.intent.REGISTRATION")) {
                onRegistration(context, intent);
            } else if (intent.getAction().equals(
                "com.google.android.c2dm.intent.RECEIVE")) {
                    onMessage(context, intent);
        }
    }
    // code omitted
}
```

Our `PushReceiver` class is a `BroadcastReceiver`, so we must implement its `onReceive` method. Note that when we receive a message, we acquire the static `WakeLock` in a manner similar to the previous technique. There are two types of messages that it'll receive: one for registration events and one for events from your application server. To distinguish them, we look at the `Intent` that was sent from the C2DM server, and in particular at its `action` property. If we see it's a registration event, we invoke the `onRegistration` method as shown next.

Listing 5.25 Handling C2DM registration events (from `PushReceiver` class)

```
private void onRegistration(Context context, Intent intent) {
   String regId = intent.getStringExtra("registration_id");
   if (regId != null) {
      Intent i =
         new Intent(context, SendC2dmRegistrationService.class);
      i.putExtra("regId", regId);
      context.startService(i);
   }
}
```

Get registration ID (1)

Send registration ID to server and Service

To handle the registration event, we get the registration ID (1) from the C2DM servers and send it to our own application servers. We need this ID in order for our app servers to be able to send events to the C2DM servers. The C2DM servers will use the registration ID provided by our servers to route the message to the correct device, and then to the correct `BroadcastReceiver` on that device. We could send the registration ID to our servers from this `BroadcastReceiver`, but a `BroadcastReceiver` is designed to execute quickly, so we'll offload this to an `IntentService`.

Listing 5.26 `IntentService` for sending registration info to servers

```
public class SendC2dmRegistrationService extends IntentService {

   private static final String WORKER_NAME = "SendC2DMReg";
   public SendC2dmRegistrationService() {
      super(WORKER_NAME);
   }

   @Override
   protected void onHandleIntent(Intent intent) {
      try{
         String regId = intent.getStringExtra("regId");
         // TODO: Send the regId to the server
      } finally {
         AlarmReceiver.releaseLock();
      }
   }
}
```

(1) Get regId from Intent

(2) Make sure to release WakeLock

This `Service` gets the registration ID (1) that was passed in listing 5.24. Then, it sends this information to your server and releases the `WakeLock` (2) when it's done. Your server will use this information whenever it wants to send a message to your app. In addition to the registration ID from the device, it'll also need a `ClientLogin` auth token. This is a generic Google authentication and authorization mechanism. In general, a `ClientLogin` token allows a particular application to access a Google application/service in the name of a particular Google account. For C2DM, the service that you need authorization for is known as `ac2dm`, and the Google account in question is the account of the developer using C2DM. Your server will need to request this token using your email address and password. You might want to create a Google account specifically for your apps. If you use your personal Google account, then changing the password would affect your server's ability to send C2DM messages to Google's C2DM servers.

Once your server has the registration ID for a user and the `ClientLogin` auth token for your account, you can send messages to the app. As we saw in listing 5.23, messages from C2DM are processed by the `onMessage` method:

```
private void onMessage(Context context, Intent intent){
   Intent stockService =
      new Intent(context, PortfolioManagerService.class);
      stockService.putExtras(intent);
   context.startService(stockService);
}
```

This is the code to start the `PortfolioManagerService`. In this case, we've still acquired the static `WakeLock`. But as we saw in the previous technique, the `PortfolioManagerService` will release this `WakeLock` once it finishes its work.

DISCUSSION

In this example, we use a message pushed from the server to tell our background `Service` to update its cache and generate `Notifications` as needed. But the data that we push from the server can be much richer. When your application sends data to the C2DM servers, it can send arbitrary name-value pairs. Those name-value pairs can then be accessed from your receiver using the `Intent.getXXXExtra` methods. For our application, we could have our server track the high/low price events, and it could pass this information as part of the `Intent`. That could save our background `Service` from having to wait for data from the network, so that it can issue `Notifications` quicker.

Also, it should be noted that the preceding code doesn't deal with many of the error conditions that can arise when using C2DM. Google has developed a small, open source library for working with C2DM. It's not part of Android, but can be easily obtained from Google. This library encapsulates much of the code seen here, eliminating a lot of the boilerplate.

Is C2DM right for you?

C2DM was a huge new feature added in Android 2.2. Our discussion has been brief but hopefully you can see that C2DM creates many interesting opportunities. But does that mean you should use it? Keep in mind that C2DM requires that the user's device be running Android 2.2 or later. At the time that this book was written, more than 83% of devices were running 2.2+, and this number will grow over time. Still, you'll want to carefully examine the breakdown of Android versions "in the wild" and the potential impact on your app's success when you choose what API level to require. Remember that the Android Market won't show your app to a user if their device isn't capable of running it.

5.4 Summary

In this chapter, we've talked extensively about what multitasking is, along with the various tools that Android gives you to enable it in your applications. Providing true multitasking is one of the things that sets Android apart in the mobile space. But such a powerful capability has its side effects, and Android walks a fine line between

empowering applications and maintaining a quality user experience. The result is that we developers must deal with some complexity. We're hopeful that you'll agree that the result is worth this complexity. With multitasking, you can keep your application synchronized with data on your servers. This can make your app richer and more responsive.

For most of the history of Android to date, developers have walked a tightrope to get their background `Services` to be robust enough to judiciously retrieve data from the network. Some applications even go as far as to establish their own persistent connection with their servers, maintained from their background `Service`. This has its own set of pitfalls. But with the advent of Cloud to Device Messaging, the benefits of always being connected are more accessible to all applications. One of the often-overlooked features of C2DM is that it's not only for `Notifications`. You get a chance to execute code based on the message pushed to your application from your servers, and then decide if you want to show a `Notification`. You may want to synchronize data with your server, start another `Service`, and so forth. The fact that you process this message in the background gives you tremendous flexibility.

MOBILE DEVELOPMENT

Android IN PRACTICE

Collins • Galpin • Käppler

Free ebook
SEE INSERT

It's not hard to find the information you need to build your first Android app. Then what? If you want to build real apps, you will need some how-to advice, and that's what this book is about.

Android in Practice is a rich source of Android tips, tricks, and best practices, covering over 90 clever and useful techniques that will make you a more effective Android developer. Techniques are presented in an easy-to-read problem/solution/discussion format. The book dives into important topics like multitasking and services, testing and instrumentation, building and deploying applications, and using alternative languages.

What's Inside

- Techniques covering Android 1.x to 3.x
- Android for tablets
- Working with threads and concurrency
- Testing and building
- Using location awareness and GPS
- Styles and themes
- And much more!

This book requires a working knowledge of Java, but no prior experience with Android is assumed.

Charlie Collins is a mobile and web developer at MOVL, a contributor to several open source projects, and a coauthor of *GWT in Practice* and *Unlocking Android*. **Michael Galpin** is a developer at Bump Technologies and worked on two of the most downloaded apps on the Android Market, Bump, and eBay Mobile. **Matthias Käppler** is an Android and API engineer at Qype.

"In-depth coverage of the No. 1 smartphone platform."
—Gabor Paller, Ericsson

"Practical and immediately useful."
—Kevin McDonagh
Novoda

"Gets you thinking with an Android mindset."
—Norman Klein
POSMobility

"The hows and the whys. Highly Recommended!"
—Al Scherer, Follett Higher Education Group

"Go from regular old Java developer to cool Android app author!"
—Cheryl Jerozal, Atlassian

For access to the book's forum and a free ebook for owners of this book, go to www.manning.com/AndroidinPractice

ISBN 13: 978-1-935182-92-4
ISBN 10: 1-935182-92-7
54999
9 781935 182924

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]