Intent Service

Now that we understand how system services work, we can use another service concept to substantially simplify our Updater service. If you recall, our Updater service is an always-on, always-running service that periodically goes to the cloud and pulls down the latest timeline updates. Since by default a service runs in the same thread as the user interface (i.e., it runs on the UI thread), we had to create a separate thread called `Updater` within the Updater service that is responsible for the actual network connection. We then started this thread in the service's `onCreate()` and `onStartCommand()` methods. We ran it forever until `onDestroy()` got called. However, our Updater thread would sleep between the updates for some amount of time. All this worked well in Chapter 8, but there's a simpler way to accomplish this task, shown in Example 13-9.

An `IntentService` is a subclass of `Service` and is also activated by a `startService()` intent. Unlike a regular service, it runs on its own *worker* thread, so it doesn't block our precious UI thread. Also, once it's done, it's done. This means it runs only once, but we will use an Alarm later to run it periodically. Any call to the intent's `startService()` will recreate it.

Unlike a regular service, we don't override `onCreate()`, `onStartCommand()`, `onDestroy()`, and `onBind()`, but rather a new `onHandleIntent()` method. This method is where we want to put our code that goes online and handles the network updates. Also, unlike a regular service, an `IntentService` has a default constructor that must be provided.

In short, instead of creating a separate thread and delaying network updates as in a regular service, we can simplify our code by using an `IntentService` to run status updates on its worker thread. Now we just need something to periodically wake up our `IntentService` so it knows it needs to handle the updating job. For that, we'll use the Alarm manager, another system service.

The key to Intent services is the `onHandleIntent()` method, a block of code that will run on a separate thread.

Example 13-9. UpdaterService.java based on IntentService

```
package com.marakana.yamba8;

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

public class UpdaterService1 extends IntentService { // ❶
  private static final String TAG = "UpdaterService";

  public static final String NEW_STATUS_INTENT = "com.marakana.yamba.NEW_STATUS";
  public static final String NEW_STATUS_EXTRA_COUNT = "NEW_STATUS_EXTRA_COUNT";
  public static final String RECEIVE_TIMELINE_NOTIFICATIONS
    = "com.marakana.yamba.RECEIVE_TIMELINE_NOTIFICATIONS";
```

```
  public UpdaterService1() { // ❷
    super(TAG);

    Log.d(TAG, "UpdaterService constructed");
  }

  @Override
  protected void onHandleIntent(Intent inIntent) { // ❸
    Intent intent;
    Log.d(TAG, "onHandleIntent'ing");
    YambaApplication yamba = (YambaApplication) getApplication();
    int newUpdates = yamba.fetchStatusUpdates();
    if (newUpdates > 0) { // ❹
      Log.d(TAG, "We have a new status");
      intent = new Intent(NEW_STATUS_INTENT);
      intent.putExtra(NEW_STATUS_EXTRA_COUNT, newUpdates);
      sendBroadcast(intent, RECEIVE_TIMELINE_NOTIFICATIONS);
    }
  }
}
```

❶ We now subclass `IntentService` instead of its parent, `Service`.

❷ A default constructor is needed. This is a good place to give your service a name, which can be useful in TraceView, for example, to help identify various threads.

❸ This is the key method. The work inside of it takes place on a separate worker thread and doesn't interfere with the main UI thread.

❹ The rest of the code in this section broadcasts the change, as described in "Broadcasting Intents" on page 165.

At this point, our service is updated. An easy way to test it would be to change the Start/Stop Service menu item to a Refresh button. To do that, update your *menu.xml* file to include the new item shown in Example 13-10, and change its handling in our `BaseActivity` class.

Example 13-10. res/xml/menu.xml

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  ...
  <item android:title="@string/titleRefresh" android:id="@+id/itemRefresh"
    android:icon="@android:drawable/ic_menu_rotate"></item>
</menu>
```

I've replaced `itemToggle` with `itemRefresh` so that the names make more sense. We must also add the appropriate string to the *strings.xml* file.

Now we need to update our *BaseActivity.java* file to handle this new Refresh button (see Example 13-11). To do that, we change the appropriate case statement in `onOptionsItemSelected()`. Additionally, we can now remove `onMenuOpened()` altogether

because we no longer need to change the state of that toggle button—it doesn't exist any more.

Example 13-11. BaseActivity.java with support for the Refresh button

```
public class BaseActivity extends Activity {
  ...
  @Override
  public boolean onOptionsItemSelected(MenuItem item) {

    switch (item.getItemId()) {
    ...
    case R.id.itemRefresh:
      startService(new Intent(this, UpdaterService.class)); // ❶
      break;
    ...
    }
    return true;
  }
  ...
}
```

❶ We simply fire off an intent to start our Updater service.

So our options menu now has a Refresh button that will start a service and have it update the status data in the background. We can use this button to test whether this new feature works well.

Another way to add the same functionality would have been to use an `AsyncTask`. In fact, `AsyncTask` would probably be slightly more appropriate in this case from a design point of view, to keep all the functionality at the UI level, but we've already discussed it in "Threading in Android" on page 65. Here we wanted to demonstrate quickly how an `IntentService` is started, and as you can see, it works just like any other service.

Next, we want to have our Updater service triggered periodically. To do that, we'll use the Alarm manager.

Alarms

The previous incarnation of our Updater service had a regular service that was always running in a loop, pulling network updates, then sleeping for some amount of time, and then looping again. With `IntentService`, we turned the process around. Our Updater service now runs only once when fired up by the `startService()` intent. Now we need a way to have something fire these intents every so often.

Android comes with yet another system service just for that. The Alarm service, represented by the `AlarmManager` class, lets you schedule certain things to happen at certain times. The time can be recurring, which makes it easy to start our service every so often. And the event that happens is an intent, or more precisely, a `PendingIntent`.

Pending intents

A `PendingIntent` is a combination of an intent and an action to be executed on it. Typically this is used for future intents that you are passing to someone else. Create a pending intent via one of the static methods in the `PendingIntent` class. Since there are only a handful of ways to send an intent, there are only a handful of static methods to create pending intents along with their actions. If you recall, you typically use an intent to start an activity via `startActivity()`, start a service via `startService()`, or send a broadcast via `sendBroadcast()`. So, to create a pending intent that will execute `startService()` with our intent in the future, we call the `getService()` static method.

Adding an Interval to Preferences

Now that we know how to leave an intent for someone to execute later and how to tell an Alarm service to repeat that periodically, we need to choose where to implement this feature. One good place is our existing `BootReceiver`, but before we do that, we'll add another option to our preferences, shown in Example 13-12.

Example 13-12. strings.xml with arrays for interval options

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  ...

  <!-- ❶ -->
  <string-array name="interval">
    <item>Never</item>
    <item>Fifteen minutes</item>
    <item>Half hour</item>
    <item>An hour</item>
    <item>Half day</item>
    <item>Day</item>
  </string-array>

  <!-- ❷ -->
  <string-array name="intervalValues">
    <item>0</item>
    <item>900000</item>
    <item>1800000</item>
    <item>3600000</item>
    <item>43200000</item>
    <item>86400000</item>
  </string-array>
</resources>
```

❶ These will be the names of options that show up in the list.

❷ These will be their corresponding values.

Now that we have these arrays, we can update *prefs.xml* as shown in Example 13-13 to add to our list of intervals.

Example 13-13. prefs.xml with support for interval preference setting

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
  ...
  <!-- ❶ -->
  <ListPreference android:entryValues="@array/intervalValues"
    android:summary="@string/summaryUpdaterInterval"
    android:title="@string/titleUpdaterInterval"
    android:entries="@array/interval" android:key="interval" />
</PreferenceScreen>
```

❶ This is the list preference. It shows a list of entities, as represented by `android:entities`. The value associated with it comes from `android:entityValues`.

Now we are ready to update `BootReceiver` and add the Alarm service alarms.

Updating BootReceiver

If you recall from "BootReceiver" on page 162, a `BootReceiver` wakes up every time the device is booted up. So far, our `BootReceiver` just starts our Updater service. That was fine when the Updater service was always on and running, but now it would cause only a one-time execution of the Updater.

We can use the Alarm service instead to periodically fire intents that start our Updater service, as shown in Example 13-14. To do that, we'll get the reference to the Alarm manager, create a pending intent to be started each time, and set up the interval at which to start the updates. Because our pending intent is meant to start a service, we'll use the `PendingIntent.getService()` call, as described in "Pending intents" on page 209.

Example 13-14. BootReceiver.java updated with Alarm service calls to periodically start the Updater service

```
package com.marakana.yamba8;

import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;

public class BootReceiver extends BroadcastReceiver {

  @Override
  public void onReceive(Context context, Intent callingIntent) {

    // Check if we should do anything at boot at all
    long interval = ((YambaApplication) context.getApplicationContext())
        .getInterval(); // ❶
    if (interval == YambaApplication.INTERVAL_NEVER)  // ❷
```

```
      return;

    // Create the pending intent
    Intent intent = new Intent(context, UpdaterService.class);  // ❸
    PendingIntent pendingIntent = PendingIntent.getService(context, -1, intent,
        PendingIntent.FLAG_UPDATE_CURRENT); // ❹

    // Setup alarm service to wake up and start service periodically
    AlarmManager alarmManager = (AlarmManager) context
        .getSystemService(Context.ALARM_SERVICE); // ❺
    alarmManager.setInexactRepeating(AlarmManager.ELAPSED_REALTIME, System
        .currentTimeMillis(), interval, pendingIntent); // ❻

    Log.d("BootReceiver", "onReceived");
  }

}
```

The previous code assumes that the phone is awake and will not work when the device is asleep. When your device is asleep, a different approach is needed (not discussed in this book).

❶ Our Yamba application has a simple getter method to return the value of the interval preference.

❷ We check the user's preference to set the frequency of checks for network updates. A value of `INTERVAL_NEVER` (zero) means not to check for updates at all.

❸ This is the intent that will run to start our Updater service.

❹ Here we wrap that intent with the action to start a service and get a new pending intent. The value `-1` is for a request code that is currently not being used. The flag in the final argument indicates whether this intent already exists. We need just to update it and not recreate it.

❺ We get the reference to `AlarmManager` via the usual `getSystemService()` call.

❻ `setInexactRepeating()` specifies that we'd like this pending intent to be sent repeatedly, but we're not concerned with being exactly on time. The `ELAPSED_REALTIME` flag will keep the alarm from waking up the phone just to run the updates. The other parameters are the current time as the start time for this alarm, our desired interval, and the actual pending intent to execute when the alarm runs.

You can now install this application on a device (and thus install the updated `BootReceiver`), and then reboot the device. Once the device starts, the LogCat should indicate that the `BootReceiver` ran and started the Updater service by posting a pending intent to the Alarm service.

Sending Notifications

Here's an opportunity to introduce yet another system service—this time the Notification service. We worked hard to have our Updater service run in the background and get the latest status updates, but what's the point of all this work if the user is not made aware that there's something new to look at? A standard Android UI approach to this would be to post a notification to the notification bar up at the top of the screen. To do that, we use the Notification system service.

We're going to make the Updater service responsible for posting the notifications, since it is the part of the app that knows of new statuses in the first place. To do that, we'll get the reference to the system Notification service, create a new `Notification` object, and update it with the latest information. The notification itself will contain a pending intent so that when the user clicks on it, it takes the user to Timeline activity to view the latest status updates. Example 13-15 shows the new code.

Example 13-15. UpdaterService.java with Notifications

```
package com.marakana.yamba8;

import android.app.IntentService;
import android.app.Notification;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.content.Intent;
import android.util.Log;

public class UpdaterService extends IntentService {
  private static final String TAG = "UpdaterService";

  public static final String NEW_STATUS_INTENT = "com.marakana.yamba.NEW_STATUS";
  public static final String NEW_STATUS_EXTRA_COUNT = "NEW_STATUS_EXTRA_COUNT";
  public static final String RECEIVE_TIMELINE_NOTIFICATIONS = "com.marakana.yamba.
    RECEIVE_TIMELINE_NOTIFICATIONS";

  private NotificationManager notificationManager; // ❶
  private Notification notification; // ❷

  public UpdaterService() {
    super(TAG);

    Log.d(TAG, "UpdaterService constructed");
  }

  @Override
  protected void onHandleIntent(Intent inIntent) {
    Intent intent;
    this.notificationManager = (NotificationManager) getSystemService(NOTIFICATION_
        SERVICE); // ❸
    this.notification = new Notification(android.R.drawable.stat_notify_chat,
        "", 0); // ❹
```

```
    Log.d(TAG, "onHandleIntent'ing");
    YambaApplication yamba = (YambaApplication) getApplication();
    int newUpdates = yamba.fetchStatusUpdates();
    if (newUpdates > 0) {
      Log.d(TAG, "We have a new status");
      intent = new Intent(NEW_STATUS_INTENT);
      intent.putExtra(NEW_STATUS_EXTRA_COUNT, newUpdates);
      sendBroadcast(intent, RECEIVE_TIMELINE_NOTIFICATIONS);
      sendTimelineNotification(newUpdates); // ❺
    }
  }

  /**
   * Creates a notification in the notification bar telling user there are new
   * messages
   *
   * @param timelineUpdateCount
   *          Number of new statuses
   */
  private void sendTimelineNotification(int timelineUpdateCount) {
    Log.d(TAG, "sendTimelineNotification'ing");
    PendingIntent pendingIntent = PendingIntent.getActivity(this, -1,
        new Intent(this, TimelineActivity.class),
        PendingIntent.FLAG_UPDATE_CURRENT); // ❻
    this.notification.when = System.currentTimeMillis(); // ❼
    this.notification.flags |= Notification.FLAG_AUTO_CANCEL; // ❽
    CharSequence notificationTitle = this
        .getText(R.string.msgNotificationTitle); // ❾
    CharSequence notificationSummary = this.getString(
        R.string.msgNotificationMessage, timelineUpdateCount);
    this.notification.setLatestEventInfo(this, notificationTitle,
        notificationSummary, pendingIntent); // ❿
    this.notificationManager.notify(0, this.notification);
    Log.d(TAG, "sendTimelineNotificationed");
  }

}
```

❶ This is just our local reference to the `NotificationManager` class, which is our access to the Notification system service.

❷ We create a class-global `Notification` object and update it each time there's a new notification for our listeners.

❸ We obtain the reference to the Notification service by using the usual `getSystemService()` call.

❹ We create the notification object that we'll reuse later. For now, we just specify the standard icon to use with our notification, and leave the text and timestamp to be updated later when we are about to post this notification.

❺ We call our private `sendTimelineNotification()` method once we know there are new statuses for the user.

❻ This pending intent will be kicked off when the user checks the notification in the notification bar and clicks on the actual item. In this case, we want to take the user to the Timeline activity, so we create an intent for that.

❼ We're now updating the data for the most recent notification. This is the timestamp that indicates when it happened.

❽ This flag tells the Notification manager to cancel this notification as soon as the user clicks on it. The notification will be removed from the notification bar at that point.

❾ Here we get the notification's title and summary from our *strings.xml* file. Notice that the summary has parameters, so we can use `String.format()` to update the actual number of new statuses.

❿ Finally, we tell the Notification manager to post this notification. In this case, we do not need the ID, so we specify zero. An ID can be used to refer to a notification later, usually in order to cancel it.

At this point our application is yet again complete. We now have a way to notify the user of any new status updates so he can stay on top of what is going on in the world.

Summary

At this point you have seen a few system services—Sensor, Location, Alarm, and Notification—and Android provides a few more services in addition to these. You might have noticed that most of them have a lot of similarities, and hopefully you have started extrapolating certain patterns. We have also used this chapter to somewhat simplify our Updater service and introduce Intent services and pending intents.