Services

A `Service` is an application component that can perform long-running operations in the background and does not provide a user interface. Another application component can start a service and it will continue to run in the background even if the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform interprocess communication (IPC). For example, a service might handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

A service can essentially take two forms:

Started

> A service is "started" when an application component (such as an activity) starts it by calling `startService()`. Once started, a service can run in the background indefinitely, even if the component that started it is destroyed. Usually, a started service performs a single operation and does not return a result to the caller. For example, it might download or upload a file over the network. When the operation is done, the service should stop itself.

Bound

> A service is "bound" when an application component binds to it by calling `bindService()`. A bound service offers a client-server interface that allows components to interact with the service, send requests, get results, and even do so across processes with interprocess communication (IPC). A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.

Although this documentation generally discusses these two types of services separately, your service can work both ways—it can be started (to run indefinitely) and also allow binding. It's simply a matter of whether you implement a couple callback methods: `onStartCommand()` to allow components to start it and `onBind()` to allow binding.

Regardless of whether your application is started, bound, or both, any application component can use the service (even from a separate application), in the same way that any component can use an activity—by starting it with an `Intent`. However, you can declare the service as private, in the manifest file, and block access from other applications. This is discussed more in the section about Declaring the service in the manifest.

> **Caution:** A service runs in the main thread of its hosting process—the service does **not** create its own thread and does**not** run in a separate process (unless you specify otherwise). This means that, if your service is going to do any CPU intensive work or blocking operations (such as MP3 playback or networking), you should create a new thread within the service to do that work. By using a separate thread, you will reduce the risk of Application Not Responding (ANR) errors and the application's main thread can remain dedicated to user interaction with your activities.

The Basics

To create a service, you must create a subclass of `Service` (or one of its existing subclasses). In your implementation, you need to override some callback methods that handle key aspects of the service lifecycle and provide a mechanism for components to bind to the service, if appropriate. The most important callback methods you should override are:

`onStartCommand()`

The system calls this method when another component, such as an activity, requests that the service be started, by calling `startService()`. Once this method executes, the service is started and can run in the background indefinitely. If you implement this, it is your responsibility to stop the service when its work is done, by calling `stopSelf()` or `stopService()`. (If you only want to provide binding, you don't need to implement this method.)

`onBind()`

The system calls this method when another component wants to bind with the service (such as to perform RPC), by calling `bindService()`. In your implementation of this method, you must provide an interface that clients use to communicate with the service, by returning an `IBinder`. You must always implement this method, but if you don't want to allow binding, then you should return null.

`onCreate()`

The system calls this method when the service is first created, to perform one-time setup procedures (before it calls either `onStartCommand()` or `onBind()`). If the service is already running, this method is not called.

`onDestroy()`

The system calls this method when the service is no longer used and is being destroyed. Your service should implement this to clean up any resources such as threads, registered listeners, receivers, etc. This is the last call the service receives.

If a component starts the service by calling `startService()` (which results in a call to `onStartCommand()`), then the service remains running until it stops itself with `stopSelf()` or another component stops it by calling `stopService()`.

If a component calls `bindService()` to create the service (and `onStartCommand()` is *not* called), then the service runs only as long as the component is bound to it. Once the service is unbound from all clients, the system destroys it.

The Android system will force-stop a service only when memory is low and it must recover system resources for the activity that has user focus. If the service is bound to an activity that has user focus, then it's less likely to be killed, and if the service is declared to run in the foreground (discussed later), then it will almost never be killed. Otherwise, if the service was started and is long-running, then the system will lower its position in the list of background tasks over time and the service will become highly susceptible to killing—if your service is started, then you must design it to gracefully handle restarts by the system. If the system kills

your service, it restarts it as soon as resources become available again (though this also depends on the value you return from `onStartCommand()`, as discussed later). For more information about when the system might destroy a service, see the Processes and Threadingdocument.

Creating a Started Service

A started service is one that another component starts by calling `startService()`, resulting in a call to the service's`onStartCommand()` method.

When a service is started, it has a lifecycle that's independent of the component that started it and the service can run in the background indefinitely, even if the component that started it is destroyed. As such, the service should stop itself when its job is done by calling `stopSelf()`, or another component can stop it by calling `stopService()`.

An application component such as an activity can start the service by calling `startService()` and passing an `Intent`that specifies the service and includes any data for the service to use. The service receives this `Intent` in the`onStartCommand()` method.

For instance, suppose an activity needs to save some data to an online database. The activity can start a companion service and deliver it the data to save by passing an intent to `startService()`. The service receives the intent in `onStartCommand()`, connects to the Internet and performs the database transaction. When the transaction is done, the service stops itself and it is destroyed.

> **Caution:** A services runs in the same process as the application in which it is declared and in the main thread of that application, by default. So, if your service performs intensive or blocking operations while the user interacts with an activity from the same application, the service will slow down activity performance. To avoid impacting application performance, you should start a new thread inside the service.

Traditionally, there are two classes you can extend to create a started service:

`Service`

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

`IntentService`

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

The following sections describe how you can implement your service using either one for these classes.

Extending the IntentService class

Because most started services don't need to handle multiple requests simultaneously (which can actually be a dangerous multi-threading scenario), it's probably best if you implement your service using the `IntentService` class.

The `IntentService` does the following:

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

All this adds up to the fact that all you need to do is implement `onHandleIntent()` to do the work provided by the client. (Though, you also need to provide a small constructor for the service.)

Here's an example implementation of `IntentService`:

```
public class HelloIntentService extends IntentService {

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

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

That's all you need: a constructor and an implementation of `onHandleIntent()`.

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

For example, `onStartCommand()` must return the default implementation (which is how the intent gets delivered to `onHandleIntent()`):

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    Toast.makeText(this, "service starting", Toast.LENGTH_SHORT).show();
    return super.onStartCommand(intent,flags,startId);
}
```

Besides `onHandleIntent()`, the only method from which you don't need to call the super class is `onBind()` (but you only need to implement that if your service allows binding).

In the next section, you'll see how the same kind of service is implemented when extending the base `Service` class, which is a lot more code, but which might be appropriate if you need to handle simultaneous start requests.

Extending the Service class

As you saw in the previous section, using `IntentService` makes your implementation of a started service very simple. If, however, you require your service to perform multi-threading (instead of processing start requests through a work queue), then you can extend the `Service` class to handle each intent.

For comparison, the following example code is an implementation of the `Service` class that performs the exact same work as the example above using `IntentService`. That is, for each start request, it uses a worker thread to perform the job and processes only one request at a time.

```
public class HelloService extends Service {
  private Looper mServiceLooper;
  private ServiceHandler mServiceHandler;

  // Handler that receives messages from the thread
  private final class ServiceHandler extends Handler {
      public ServiceHandler(Looper looper) {
          super(looper);
      }
      @Override
      public void handleMessage(Message msg) {
          // Normally we would do some work here, like download a file.
          // For our sample, we just sleep for 5 seconds.
          long endTime = System.currentTimeMillis() + 5*1000;
          while (System.currentTimeMillis() < endTime) {
              synchronized (this) {
                  try {
                      wait(endTime - System.currentTimeMillis());
                  } catch (Exception e) {
                  }
              }
          }
          // Stop the service using the startId, so that we don't stop
          // the service in the middle of handling another job
          stopSelf(msg.arg1);
```

```
        }
    }

    @Override
    public void onCreate() {
      // Start up the thread running the service.  Note that we create a
      // separate thread because the service normally runs in the process's
      // main thread, which we don't want to block.  We also make it
      // background priority so CPU-intensive work will not disrupt our UI.
      HandlerThread thread = new HandlerThread("ServiceStartArguments",
              Process.THREAD_PRIORITY_BACKGROUND);
      thread.start();

      // Get the HandlerThread's Looper and use it for our Handler
      mServiceLooper = thread.getLooper();
      mServiceHandler = new ServiceHandler(mServiceLooper);
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Toast.makeText(this, "service starting", Toast.LENGTH_SHORT).show();

        // For each start request, send a message to start a job and deliver the
        // start ID so we know which request we're stopping when we finish the job
        Message msg = mServiceHandler.obtainMessage();
        msg.arg1 = startId;
        mServiceHandler.sendMessage(msg);

        // If we get killed, after returning from here, restart
        return START_STICKY;
    }

    @Override
    public IBinder onBind(Intent intent) {
        // We don't provide binding, so return null
        return null;
    }

    @Override
    public void onDestroy() {
      Toast.makeText(this, "service done", Toast.LENGTH_SHORT).show();
    }
}
```

As you can see, it's a lot more work than using `IntentService`.

However, because you handle each call to `onStartCommand()` yourself, you can perform multiple requests simultaneously. That's not what this example does, but if that's what you want, then you can create a new thread for each request and run them right away (instead of waiting for the previous request to finish).

Notice that the `onStartCommand()` method must return an integer. The integer is a value that describes how the system should continue the service in the event that the system kills it (as discussed above, the default implementation for `IntentService` handles this for you, though you are able to modify it). The return value from `onStartCommand()` must be one of the following constants:

`START_NOT_STICKY`

> If the system kills the service after `onStartCommand()` returns, *do not* recreate the service, unless there are pending intents to deliver. This is the safest option to avoid running your service when not necessary and when your application can simply restart any unfinished jobs.

`START_STICKY`

If the system kills the service after `onStartCommand()` returns, recreate the service and call `onStartCommand()`, but *do not* redeliver the last intent. Instead, the system calls `onStartCommand()` with a null intent, unless there were pending intents to start the service, in which case, those intents are delivered. This is suitable for media players (or similar services) that are not executing commands, but running indefinitely and waiting for a job.

`START_REDELIVER_INTENT`

If the system kills the service after `onStartCommand()` returns, recreate the service and call `onStartCommand()` with the last intent that was delivered to the service. Any pending intents are delivered in turn. This is suitable for services that are actively performing a job that should be immediately resumed, such as downloading a file.

For more details about these return values, see the linked reference documentation for each constant.

Starting a Service

You can start a service from an activity or other application component by passing an `Intent` (specifying the service to start) to `startService()`. The Android system calls the service's `onStartCommand()` method and passes it the `Intent`. (You should never call `onStartCommand()` directly.)

For example, an activity can start the example service in the previous section (`HelloSevice`) using an explicit intent with `startService()`:

```
Intent intent = new Intent(this, HelloService.class);
startService(intent);
```

The `startService()` method returns immediately and the Android system calls the service's `onStartCommand()` method. If the service is not already running, the system first calls `onCreate()`, then calls `onStartCommand()`.

If the service does not also provide binding, the intent delivered with `startService()` is the only mode of communication between the application component and the service. However, if you want the service to send a result back, then the client that starts the service can create a `PendingIntent` for a broadcast (with `getBroadcast()`) and deliver it to the service in the `Intent` that starts the service. The service can then use the broadcast to deliver a result.

Multiple requests to start the service result in multiple corresponding calls to the service's `onStartCommand()`. However, only one request to stop the service (with `stopSelf()` or `stopService()`) is required to stop it.

Stopping a service

A started service must manage its own lifecycle. That is, the system does not stop or destroy the service unless it must recover system memory and the service continues to run after `onStartCommand()` returns. So, the service must stop itself by calling `stopSelf()` or another component can stop it by calling `stopService()`.

Once requested to stop with `stopSelf()` or `stopService()`, the system destroys the service as soon as possible.

However, if your service handles multiple requests to `onStartCommand()` concurrently, then you shouldn't stop the service when you're done processing a start request, because you might have since received a new start request (stopping at the end of the first request would terminate the second one). To avoid this problem, you can use `stopSelf(int)` to ensure that your request to stop the service is always based on the most recent start request. That is, when you call `stopSelf(int)`, you pass the ID of the start request (the `startId` delivered to `onStartCommand()`) to which your stop request corresponds. Then if the service received a new start request before you were able to call `stopSelf(int)`, then the ID will not match and the service will not stop.

> **Caution:** It's important that your application stops its services when it's done working, to avoid wasting system resources and consuming battery power. If necessary, other components can stop the service by calling `stopService()`. Even if you enable binding for the service, you must always stop the service yourself if it ever received a call to `onStartCommand()`.

For more information about the lifecycle of a service, see the section below about Managing the Lifecycle of a Service.

Creating a Bound Service

A bound service is one that allows application components to bind to it by calling `bindService()` in order to create a long-standing connection (and generally does not allow components to *start* it by calling `startService()`).

You should create a bound service when you want to interact with the service from activities and other components in your application or to expose some of your application's functionality to other applications, through interprocess communication (IPC).

To create a bound service, you must implement the `onBind()` callback method to return an `IBinder` that defines the interface for communication with the service. Other application components can then call `bindService()` to retrieve the interface and begin calling methods on the service. The service lives only to serve the application component that is bound to it, so when there are no components bound to the service, the system destroys it (you do *not* need to stop a bound service in the way you must when the service is started through `onStartCommand()`).

To create a bound service, the first thing you must do is define the interface that specifies how a client can communicate with the service. This interface between the service and a client must be an implementation of `IBinder` and is what your service must return from the `onBind()` callback method. Once the client receives the `IBinder`, it can begin interacting with the service through that interface.

Multiple clients can bind to the service at once. When a client is done interacting with the service, it calls `unbindService()` to unbind. Once there are no clients bound to the service, the system destroys the service.

There are multiple ways to implement a bound service and the implementation is more complicated than a started service, so the bound service discussion appears in a separate document about Bound Services.

Sending Notifications to the User

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

A toast notification is a message that appears on the surface of the current window for a moment then disappears, while a status bar notification provides an icon in the status bar with a message, which the user can select in order to take an action (such as start an activity).

Usually, a status bar notification is the best technique when some background work has completed (such as a file completed downloading) and the user can now act on it. When the user selects the notification from the expanded view, the notification can start an activity (such as to view the downloaded file).

See the Toast Notifications or Status Bar Notifications developer guides for more information.

Running a Service in the Foreground

A foreground service is a service that's considered to be something the user is actively aware of and thus not a candidate for the system to kill when low on memory. A foreground service must provide a notification for the status bar, which is placed under the "Ongoing" heading, which means that the notification cannot be dismissed unless the service is either stopped or removed from the foreground.

For example, a music player that plays music from a service should be set to run in the foreground, because the user is explicitly aware of its operation. The notification in the status bar might indicate the current song and allow the user to launch an activity to interact with the music player.

To request that your service run in the foreground, call `startForeground()`. This method takes two parameters: an integer that uniquely identifies the notification and the`Notification` for the status bar. For example:

```
Notification notification = new Notification(R.drawable.icon, getText(R.string.ticker text),
        System.currentTimeMillis());
Intent notificationIntent = new Intent(this, ExampleActivity.class);
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, notificationIntent, 0);
notification.setLatestEventInfo(this, getText(R.string.notification title),
        getText(R.string.notification_message), pendingIntent);
startForeground(ONGOING_NOTIFICATION, notification);
```

To remove the service from the foreground, call `stopForeground()`. This method takes a boolean, indicating whether to remove the status bar notification as well. This method does*not* stop the service. However, if you stop the service while it's still running in the foreground, then the notification is also removed.

> **Note:** The methods `startForeground()` and `stopForeground()` were introduced in Android 2.0 (API Level 5). In order to run your service in the foreground on older versions of the platform, you must use the previous `setForeground()` method—see the `startForeground()` documentation for information about how to provide backward compatibility.

For more information about notifications, see Creating Status Bar Notifications.

Managing the Lifecycle of a Service

The lifecycle of a service is much simpler than that of an activity. However, it's even more important that you pay close attention to how your service is created and destroyed, because a service can run in the background without the user being aware.

The service lifecycle—from when it's created to when it's destroyed—can follow two different paths:

- A started service

The service is created when another component calls `startService()`. The service then runs indefinitely and must stop itself by calling `stopSelf()`. Another component can also stop the service by calling `stopService()`. When the service is stopped, the system destroys it..

- A bound service

The service is created when another component (a client) calls `bindService()`. The client then communicates with the service through an `IBinder` interface. The client can close the connection by calling `unbindService()`. Multiple clients can bind to the same service and when all of them unbind, the system destroys the service. (The service does *not*need to stop itself.)

These two paths are not entirely separate. That is, you can bind to a service that was already started with `startService()`. For example, a background music service could be started by calling `startService()` with an `Intent` that identifies the music to play. Later, possibly when the user wants to exercise some control over the player or get information about the current song, an activity can bind to the service by calling `bindService()`. In cases like this, `stopService()` or `stopSelf()` does not actually stop the service until all clients unbind.

Implementing the lifecycle callbacks

Like an activity, a service has lifecycle callback methods that you can implement to monitor changes in the service's state and perform work at the appropriate times. The following skeleton service demonstrates each of the lifecycle methods:

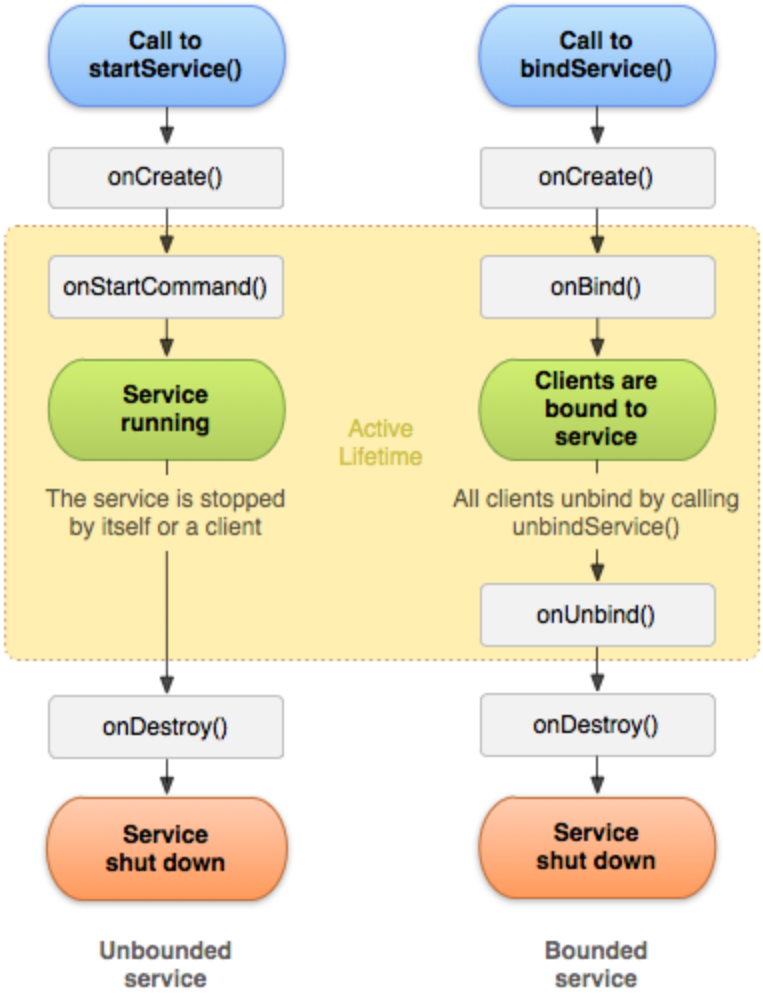

Figure 2. The service lifecycle. The diagram on the left shows the lifecycle when the service is created with `startService()` and the diagram on the right shows the lifecycle when the service is created with `bindService()`.

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

    @Override
    public void onCreate() {
        // The service is being created
    }
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        // The service is starting, due to a call to startService()
        return mStartMode;
    }
    @Override
    public IBinder onBind(Intent intent) {
        // A client is binding to the service with bindService()
        return mBinder;
    }
```

```
    @Override
    public boolean onUnbind(Intent intent) {
        // All clients have unbound with unbindService()
        return mAllowRebind;
    }
    @Override
    public void onRebind(Intent intent) {
        // A client is binding to the service with bindService(),
        // after onUnbind() has already been called
    }
    @Override
    public void onDestroy() {
        // The service is no longer used and is being destroyed
    }
}
```

> **Note:** Unlike the activity lifecycle callback methods, you are *not* required to call the superclass implementation of these callback methods.

By implementing these methods, you can monitor two nested loops of the service's lifecycle:

- The **entire lifetime** of a service happens between the time `onCreate()` is called and the time `onDestroy()` returns. Like an activity, a service does its initial setup in `onCreate()` and releases all remaining resources in `onDestroy()`. For example, a music playback service could create the thread where the music will be played in `onCreate()`, then stop the thread in `onDestroy()`.

  The `onCreate()` and `onDestroy()` methods are called for all services, whether they're created by `startService()` or `bindService()`.
- The **active lifetime** of a service begins with a call to either `onStartCommand()` or `onBind()`. Each method is handed the `Intent` that was passed to either `startService()` or `bindService()`, respectively.

  If the service is started, the active lifetime ends the same time that the entire lifetime ends (the service is still active even after `onStartCommand()` returns). If the service is bound, the active lifetime ends when `onUnbind()` returns.

> **Note:** Although a started service is stopped by a call to either `stopSelf()` or `stopService()`, there is not a respective callback for the service (there's no `onStop()` callback). So, unless the service is bound to a client, the system destroys it when the service is stopped—`onDestroy()` is the only callback received.

Figure 2 illustrates the typical callback methods for a service. Although the figure separates services that are created by `startService()` from those created by `bindService()`, keep in mind that any service, no matter how it's started, can potentially allow clients to bind to it. So, a service that was initially started with `onStartCommand()` (by a client calling `startService()`) can still receive a call to `onBind()` (when a client calls `bindService()`).