Mobile programming

Omid Jafarinezhad

Android 07

Services

- A Service is an application component that can perform long-running operations in the background and does not provide a user interface

- 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

Service 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**.

Service forms(2)

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**.

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.**

Should you use a service or a thread?

A service is simply a component that can run in the background even when the user is not interacting with your application. Thus, you should create a service only if that is what you need.

If you need to perform work outside your main thread, but only while the user is interacting with your application, then you should probably instead create a new thread and not a service. For example, if you want to play some music, but only while your activity is running, you might create a thread in onCreate(), start running it in onStart(), then stop it in onStop(). Also consider using AsyncTask or HandlerThread, instead of the traditional Thread class.

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()
- onBind()
- onCreate()
- onDestroy()

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.

Caution

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**

if the service is declared to run in the foreground (discussed later), then it will almost never be killed.

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)

Declaring a service in the manifest

Like activities (and other components), **you must declare all services in your application's manifest file**. There are other attributes you can include in the <service> element to define properties such as permissions required to start the service and the process in which the service should run

Additionally, you can ensure that your service is available to only your app by including the **android:exported** attribute and setting it to "false". This effectively stops other apps from starting your service, even when using an explicit intent.

```
<manifest ... >
  ...
  <application ... >
      <service android:name=".ExampleService" />
      ...
  </application>
</manifest>
```

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()

Caution

A service 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.

What if scenario?

- app resume and the main activity start a service in main thread
- activity destroy
- **What is the thread id of the service?**

Creating a Started 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**.

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**.

IntentService

- 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.

```java
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.
      try {
          Thread.sleep(5000);
      } catch (InterruptedException e) {
          // Restore interrupt status.
          Thread.currentThread().interrupt();
      }
  }
}
```

Caution

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);
}
```

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.

```
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.
          try {
              Thread.sleep(5000);
          } catch (InterruptedException e) {
              // Restore interrupt status.
              Thread.currentThread().interrupt();
          }
          // Stop the service using the startId, so that we don't stop
          // the service in the middle of handling another job
          stopSelf(msg.arg1);
      }
  }
```

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



```
@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);
}
```

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

```
@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();
}
```

onStartCommand()

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**
- **START_STICKY**
- **START_REDELIVER_INTENT**

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.

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.

If the service is not already running, the system first calls onCreate(), then calls onStartCommand().

You should never call onStartCommand() directly.

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

Caution

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.

stopSelf(int)

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**

Bound Services

A bound service is the server in a client-server interface. A bound service allows components (such as activities) to bind to the service, send requests, receive responses, and even perform interprocess communication (IPC). **A bound service typically lives only while it serves another application component and does not run in the background indefinitely.**

Bound Service - server

A bound service is an **implementation of the Service** class that allows other applications to bind to it and interact with it. **To provide binding for a service**, you must **implement the onBind() callback method**. This method **returns an IBinder object that defines the programming interface that clients can use to interact with the service**.

Bound Service - client

A **client** can bind to the service by **calling bindService()**. When it does, it must **provide an implementation of ServiceConnection**, which **monitors the connection with the service**. The **bindService() method returns immediately without a value**, but when the **Android system creates the connection between the client and service, it calls onServiceConnected() on the ServiceConnection, to deliver the IBinder** that the client can use to communicate with the service.

onBind()

- **Multiple clients can connect to the service at once**. However, the system **calls your service's onBind() method** to retrieve the **IBinder only when the first client binds**. The system then **delivers the same IBinder to any additional clients that bind, without calling onBind() again**.

- **When the last client unbinds from the service, the system destroys the service**

provides binding

When creating a service that provides binding, **you must provide an IBinder that provides the programming interface that clients can use to interact with the service**. There are three ways you can define the interface:

- Extending the Binder class
- Using a Messenger
- Using AIDL

Extending the Binder class

If your **service is private to your own application and runs in the same process as the client (which is common)**, you should create your interface by extending the Binder class and returning an instance of it from onBind(). The client receives the Binder and can use it to directly **access public methods available in either the Binder implementation or even the Service**.

This is the preferred technique when your service is merely a background worker for your own application. **The only reason you would not create your interface this way is because your service is used by other applications or across separate processes**

Using a Messenger

If you need your interface **to work across different processes**, you can **create an interface for the service with a Messenger**. In this manner, the **service defines a Handler that responds to different types of Message objects.** This Handler is the basis for a Messenger that can then share an IBinder with the client, **allowing the client to send commands to the service using Message objects**. Additionally, **the client can define a Messenger of its own so the service can send messages back**.

This is the simplest way to perform interprocess communication (IPC), because the **Messenger queues all requests into a single thread** so that you don't have to design your service to be **thread-safe**.

Using AIDL - Android Interface Definition Language

AIDL performs all the work to decompose objects into primitives that the operating system can understand and marshall them across processes to perform IPC. The previous technique, using a **Messenger**, **is actually based on AIDL as its underlying structure**. **If you want your service to handle multiple requests simultaneously, then you can use AIDL directly**. In this case, your service must be capable of multi-threading and be built thread-safe.

To use AIDL directly, you must create an **.aidl file that defines the programming interface**. The Android SDK tools use this file to generate an abstract class that implements the interface and handles IPC, which you can then extend within your service.

Extending the Binder class

1. In your service, create an instance of Binder that either:
    - contains **public methods** that the client can call
    - returns the **current Service instance**, which has public methods the client can call
    - or, returns an **instance of another class hosted by the service** with public methods the client can call
1. Return this instance of Binder from the onBind() callback method.
2. In the client, receive the Binder from the onServiceConnected() callback method and make calls to the bound service using the methods provided.

```java
public class LocalService extends Service {
    // Binder given to clients
    private final IBinder mBinder = new LocalBinder();
    // Random number generator
    private final Random mGenerator = new Random();

    /**
     * Class used for the client Binder.  Because we know this service always
     * runs in the same process as its clients, we don't need to deal with IPC.
     */
    public class LocalBinder extends Binder {
        LocalService getService() {
            // Return this instance of LocalService so clients can call public methods
            return LocalService.this;
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        return mBinder;
    }

    /** method for clients */
    public int getRandomNumber() {
      return mGenerator.nextInt(100);
    }
}
```

```
public class BindingActivity extends Activity {
    LocalService mService;
    boolean mBound = false;

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

    @Override
    protected void onStart() {
        super.onStart();
        // Bind to LocalService
        Intent intent = new Intent(this, LocalService.class);
        bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
    }

    @Override
    protected void onStop() {
        super.onStop();
        // Unbind from the service
        if (mBound) {
            unbindService(mConnection);
            mBound = false;
        }
    }
```

```
public class BindingActivity extends Activity {
    LocalService mService;
    boolean mBound = false;

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

    @Override
    protected void onStart() {
        super.onStart();
        // Bind to LocalService
        Intent intent = new Intent(this, LocalServi
        bindService(intent, mConnection, Context.BI
    }

    @Override
    protected void onStop() {
        super.onStop();
        // Unbind from the service
        if (mBound) {
            unbindService(mConnection);
            mBound = false;
        }
    }
```

```
/** Called when a button is clicked (the button in the layout file attaches to
  * this method with the android:onClick attribute) */
public void onButtonClick(View v) {
    if (mBound) {
        // Call a method from the LocalService.
        // However, if this call were something that might hang, then this request s
        // occur in a separate thread to avoid slowing down the activity performance
        int num = mService.getRandomNumber();
        Toast.makeText(this, "number: " + num, Toast.LENGTH_SHORT).show();
    }
}

/** Defines callbacks for service binding, passed to bindService() */
private ServiceConnection mConnection = new ServiceConnection() {

    @Override
    public void onServiceConnected(ComponentName className,
            IBinder service) {
        // We've bound to LocalService, cast the IBinder and get LocalService instan
        LocalBinder binder = (LocalBinder) service;
        mService = binder.getService();
        mBound = true;
    }

    @Override
    public void onServiceDisconnected(ComponentName arg0) {
        mBound = false;
    }
};
```

Note

- The reason the **service and client must be in the same application is so the client can cast the returned object and properly call its APIs**. The service and client must also be in the same process, because this technique does not perform any marshalling across processes

- In the example above, **the onStop() method unbinds the client from the service. Clients should unbind from services at appropriate times**

Using a Messenger

- The **service implements a Handler that receives a callback** for each call from a client.
- The **Handler is used to create a Messenger object** (which is a reference to the Handler).
- The **Messenger creates an IBinder that the service returns to clients** from onBind().
- **Clients use the IBinder to instantiate the Messenger** (that references the service's Handler), which the client uses to send Message objects to the service.
- The **service receives each Message in its Handler**—specifically, in the handleMessage() method.

```
public class MessengerService extends Service {
    static final int MSG_SAY_HELLO = 1;
    /* Handler of incoming messages from clients.   */
    class IncomingHandler extends Handler {
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case MSG_SAY_HELLO:
                    Toast.makeText(getApplicationContext(), "hello!",
Toast.LENGTH_SHORT).show();
                    break;
                default:
                    super.handleMessage(msg);
            }
        }
    }
    /*  Target we publish for clients to send messages to IncomingHandler.     */
    final Messenger mMessenger = new Messenger(new IncomingHandler());
    /* When binding to the service, we return an interface to our messenger for sending
messages to the service. */
    @Override
    public IBinder onBind(Intent intent) {
        Toast.makeText(getApplicationContext(), "binding", Toast.LENGTH_SHORT).show();
        return mMessenger.getBinder();
    }
```

```
public class ActivityMessenger extends Activity {
    /** Messenger for communicating with the service. */
    Messenger mService = null;

    /** Flag indicating whether we have called bind on the service. */
    boolean mBound;

    /**
     * Class for interacting with the main interface of the service.
     */
    private ServiceConnection mConnection = new ServiceConnection() {
        public void onServiceConnected(ComponentName className, IBinder service) {
            // This is called when the connection with the service has been
            // established, giving us the object we can use to
            // interact with the service.  We are communicating with the
            // service using a Messenger, so here we get a client-side
            // representation of that from the raw IBinder object.
            mService = new Messenger(service);
            mBound = true;
        }

        public void onServiceDisconnected(ComponentName className) {
            // This is called when the connection with the service has been
            // unexpectedly disconnected -- that is, its process crashed.
            mService = null;
            mBound = false;
        }
    };
```

```
public class ActivityMessenger extends Activity {
    /** Messenger for communicating with the service
    Messenger mService = null;

    /** Flag indicating whether we have called bind
    boolean mBound;

    /**
     * Class for interacting with the main interface
     */
    private ServiceConnection mConnection = new Serv
        public void onServiceConnected(ComponentName
            // This is called when the connection wi
            // established, giving us the object we
            // interact with the service.  We are co
            // service using a Messenger, so here we
            // representation of that from the raw I
            mService = new Messenger(service);
            mBound = true;
        }

        public void onServiceDisconnected(ComponentN
            // This is called when the connection wi
            // unexpectedly disconnected -- that is,
            mService = null;
            mBound = false;
        }
    };
```

```
public void sayHello(View v) {
    if (!mBound) return;
    // Create and send a message to the service, using a supported 'what' value
    Message msg = Message.obtain(null, MessengerService.MSG_SAY_HELLO, 0, 0);
    try {
        mService.send(msg);
    } catch (RemoteException e) {
        e.printStackTrace();
    }
}

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

@Override
protected void onStart() {
    super.onStart();
    // Bind to the service
    bindService(new Intent(this, MessengerService.class), mConnection,
        Context.BIND_AUTO_CREATE);
}

@Override
protected void onStop() {
    super.onStop();
    // Unbind from the service
    if (mBound) {
        unbindService(mConnection);
        mBound = false;
```

Using AIDL

Using AIDL is necessary only if you allow clients from different applications to access your service for IPC and want to handle multithreading in your service. **If you do not need to perform concurrent IPC across different applications**, you should create your interface by **implementing a Binder** or, **if you want to perform IPC, but do not need to handle multithreading,** implement your interface using **a Messenger**. Regardless, **be sure that you understand Bound Services before implementing an AIDL**

Defining an AIDL Interface

You must define your AIDL interface in **an .aidl file** using the Java programming language syntax, then save it in the source code (**in the src/ directory) of both the application hosting the service and any other application that binds to the service**

When you build each application that contains the .aidl file, the Android SDK tools generate an IBinder interface based on the .aidl file and save it in the project's gen/ directory

To create a bounded service using AIDL

Create the .aidl file: This file defines the **programming interface with method signatures**

Implement the interface: The **Android SDK tools generate an interface** in the Java programming language, based on your .aidl file. **This interface has an inner abstract class named Stub that extends Binder and implements methods from your AIDL interface**. You must extend the Stub class and implement the methods.

Expose the interface to clients: Implement a Service and override onBind() to return your implementation of the Stub class.

Create the .aidl file

By default, AIDL supports the following data types:

- All primitive types in the Java programming language (such as int, long, char, boolean, and so on)
- String
- CharSequence
- List
- Map

Create the .aidl file (2)

- When defining your service interface, be aware that:
  - Methods can take zero or more parameters, and return a value or void
  - All non-primitive parameters require a directional tag indicating which way the data goes. Either in, out, or inout (see the example below)

- Primitives are in by default, and cannot be otherwise
- **Caution**: You should limit the direction to what is truly needed, because marshalling parameters is expensive.

Create the .aidl file (3)

Simply save your .aidl file in your project's src/ directory and when you build your application, the SDK tools generate the IBinder interface file in your project's gen/ directory. (**for example, IRemoteService.aidl results in IRemoteService.java**)

```
// IRemoteService.aidl
package com.example.android;

// Declare any non-default types here with import statements

/** Example service interface */
interface IRemoteService {
    /** Request the process ID of this service, to do evil things with it. */
    int getPid();

    /** Demonstrates some basic types that you can use as parameters
     * and return values in AIDL.
     */
    void basicTypes(int anInt, long aLong, boolean aBoolean, float aFloat,
            double aDouble, String aString);
}
```

Implement the interface

```
private final IRemoteService.Stub mBinder = new IRemoteService.Stub() {
    public int getPid(){
        return Process.myPid();
    }
    public void basicTypes(int anInt, long aLong, boolean aBoolean,
        float aFloat, double aDouble, String aString) {
        // Does nothing
    }
};
```

Expose the interface to clients

```
public class RemoteService extends Service {
    @Override
    public void onCreate() {
        super.onCreate();
    }

    @Override
    public IBinder onBind(Intent intent) {
        // Return the interface
        return mBinder;
    }

    private final IRemoteService.Stub mBinder = new IRemoteService.Stub() {
        public int getPid(){
            return Process.myPid();
        }
        public void basicTypes(int anInt, long aLong, boolean aBoolean,
            float aFloat, double aDouble, String aString) {
            // Does nothing
        }
    };
}
```

Expose the interface to clients

```
public class RemoteService extends Service {
    @Override
    public void onCreate() {
        super.onCreate();
    }

    @Override
    public IBinder onBind(Intent intent) {
        // Return the interface
        return mBinder;
    }

    private final IRemoteService.Stub mBinder = new IRemoteService.Stub() {
        public int getPid(){
            return Process.myPid();
        }
        public void basicTypes(int anInt, long aLong, boolean aBoolean,
            float aFloat, double aDouble, String aString) {
            // Does nothing
        }
    };
}
```

Client receives the IBinder

```
IRemoteService mIRemoteService;
private ServiceConnection mConnection = new ServiceConnection() {
    // Called when the connection with the service is established
    public void onServiceConnected(ComponentName className, IBinder service) {
        // Following the example above for an AIDL interface,
        // this gets an instance of the IRemoteInterface, which we can use to call on the service
        mIRemoteService = IRemoteService.Stub.asInterface(service);
    }

    // Called when the connection with the service disconnects unexpectedly
    public void onServiceDisconnected(ComponentName className) {
        Log.e(TAG, "Service has unexpectedly disconnected");
        mIRemoteService = null;
    }
};
```

Reporting Work Status

How to report the status of a work request run in a background service to the component that sent the request

The recommended way to send and receive status is to use a **LocalBroadcastManager**, which limits broadcast Intent objects to components in your own app

Report Status From an IntentService

```
public final class Constants {
    ...
    // Defines a custom Intent action
    public static final String BROADCAST_ACTION =
        "com.example.android.threadsample.BROADCAST";
    ...
    // Defines the key for the status "extra" in an Intent
    public static final String EXTENDED_DATA_STATUS =
        "com.example.android.threadsample.STATUS";
    ...
}
public class RSSPullService extends IntentService {
...
    /*
     * Creates a new Intent containing a Uri object
     * BROADCAST_ACTION is a custom Intent action
     */
    Intent localIntent =
            new Intent(Constants.BROADCAST_ACTION)
            // Puts the status into the Intent
            .putExtra(Constants.EXTENDED_DATA_STATUS, status);
    // Broadcasts the Intent to receivers in this app.
    LocalBroadcastManager.getInstance(this).sendBroadcast(localIntent);
...
}
```

Receive Status Broadcasts from an IntentService

```
// Broadcast receiver for receiving status updates from the IntentService
private class DownloadStateReceiver extends BroadcastReceiver {
    // Prevents instantiation
    private DownloadStateReceiver() {
    }
    // Called when the BroadcastReceiver gets an Intent it's registered to
receive
    @
    public void onReceive(Context context, Intent intent) {
        /*
         * Handle Intents here.
         */
    }
}
```

Receive Status Broadcasts from an IntentService

```
// Class that displays photos
public class DisplayActivity extends FragmentActivity {
    ...
    public void onCreate(Bundle stateBundle) {
        ...
        super.onCreate(stateBundle);
        ...
        // The filter's action is BROADCAST_ACTION
        IntentFilter mStatusIntentFilter = new IntentFilter(
                Constants.BROADCAST_ACTION);

        // Adds a data filter for the HTTP scheme
        mStatusIntentFilter.addDataScheme("http");
        ...
```

Receive Status Broadcasts from an IntentService

```
// Instantiates a new DownloadStateReceiver
DownloadStateReceiver mDownloadStateReceiver =
        new DownloadStateReceiver();
// Registers the DownloadStateReceiver and its intent filters
LocalBroadcastManager.getInstance(this).registerReceiver(
        mDownloadStateReceiver,
        mStatusIntentFilter);
...
```

Note

A single BroadcastReceiver can handle more than one type of broadcast Intent object, each with its own action. This feature allows you to run different code for each action, without having to define a separate BroadcastReceiver for each action. To define another IntentFilter for the same BroadcastReceiver, create the IntentFilter and repeat the call to registerReceiver().
For example:

```
/*
 * Instantiates a new action filter.
 * No data filter is needed.
 */
statusIntentFilter = new IntentFilter(Constants.ACTION_ZOOM_IMAGE);
...
// Registers the receiver with the new filter
LocalBroadcastManager.getInstance(getActivity()).registerReceiver(
        mDownloadStateReceiver,
        mIntentFilter);
```

Managing the Lifecycle of a Service