Services overview

Archived: 2026-04-06 01:53:07 UTC

A `Service` is an application component that can perform long-running operations in the background. It does not provide a user interface. Once started, a service might continue running for some time, even after 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 can handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

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. You should run any blocking operations on a separate thread within the service to avoid Application Not Responding (ANR) errors.

Types of Services

These are the three different types of services:

Foreground

> A foreground service performs some operation that is noticeable to the user. For example, an audio app would use a foreground service to play an audio track. Foreground services must display a Notification. Foreground services continue running even when the user isn't interacting with the app.
>
> When you use a foreground service, you must display a notification so that users are actively aware that the service is running. This notification cannot be dismissed unless the service is either stopped or removed from the foreground.
>
> Learn more about how to configure foreground services in your app.
>
> **Note:** The WorkManager API offers a flexible way of scheduling tasks, and is able to run these jobs as foreground services if needed. In many cases, using WorkManager is preferable to using foreground services directly.

Background

> A background service performs an operation that isn't directly noticed by the user. For example, if an app used a service to compact its storage, that would usually be a background service.
>
> **Note:** If your app targets API level 26 or higher, the system imposes restrictions on running background services when the app itself isn't in the foreground. In most situations, for example, you shouldn't access location information from the background. Instead, schedule tasks using WorkManager.

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, receive

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 started and bound 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 of callback methods: `onStartCommand()` to allow components to start it and `onBind()` to allow binding.

Regardless of whether your service 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.

Choosing between a service and a thread

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

If you must perform work outside of your main thread, but only while the user is interacting with your application, you should instead create a new thread in the context of another application component. 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()` , and stop it in `onStop()` . Also consider using thread pools and executors from the `java.util.concurrent` package or Kotlin coroutines instead of the traditional `Thread` class. See the Threading on Android document for more information about moving execution to background threads.

Remember that if you do use a service, it still runs in your application's main thread by default, so you should still create a new thread within the service if it performs intensive or blocking operations.

The basics

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

`onStartCommand()`
: The system invokes this method by calling `startService()` when another component (such as an activity) requests that the service be started. When 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 complete by calling `stopSelf()` or `stopService()` . If you only want to provide binding, you don't need to implement this method.

`onBind()`
: The system invokes this method by calling `bindService()` when another component wants to bind with the service (such as to perform RPC). 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; however, if you don't want to allow binding, you should return null.

`onCreate()`

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

`onDestroy()`

The system invokes 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, or receivers. This is the last call that the service receives.

If a component starts the service by calling `startService()` (which results in a call to `onStartCommand()`), the service continues to run 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, the service runs only as long as the component is bound to it. After the service is unbound from all of its clients, the system destroys it.

The Android system stops 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, it's less likely to be killed; if the service is declared to run in the foreground, it's rarely killed. If the service is started and is long-running, the system lowers its position in the list of background tasks over time, and the service becomes highly susceptible to killing—if your service is started, 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, but this also depends on the value that you return from `onStartCommand()`. For more information about when the system might destroy a service, see the Processes and Threading document.

In the following sections, you'll see how you can create the `startService()` and `bindService()` service methods, as well as how to use them from other application components.

Declaring a service in the manifest

You must declare all services in your application's manifest file, just as you do for activities and other components.

To declare your service, add a `<service>` element as a child of the `<application>` element. Here is an example:

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

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

See the `<service>` element reference for more information about declaring your service in the manifest.

There are other attributes that you can include in the `<service>` element to define properties such as the permissions that are required to start the service and the process in which the service should run. The `android:name` attribute is the only required attribute—it specifies the class name of the service. After you publish your application, leave this name unchanged to avoid the risk of breaking code due to dependence on explicit intents to start or bind the service (read the blog post, Things That Cannot Change).

Caution: To ensure that your app is secure, always use an explicit intent when starting a `Service` and don't declare intent filters for your services. Using an implicit intent to start a service is a security hazard because you cannot be certain of the service that responds to the intent, and the user cannot see which service starts. Beginning with Android 5.0 (API level 21), the system throws an exception if you call `bindService()` with an implicit intent.

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.

Note: Users can see what services are running on their device. If they see a service that they don't recognize or trust, they can stop the service. In order to avoid having your service stopped accidentally by users, you need to add the `android:description` attribute to the `<service>` element in your app manifest. In the description, provide a short sentence explaining what the service does and what benefits it provides.

Creating a started service

A started service is one that another component starts by calling `startService()`, which results 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. 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 complete 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 complete, the service stops itself and is destroyed.

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. If your service performs intensive or blocking operations while the user interacts with an activity from the same application, the service slows down activity performance. To avoid impacting application performance, start a new thread inside the service.

The `Service` class is the base class for all services. When you extend this class, it's important to create a new thread in which the service can complete all of its work; the service uses your application's main thread by default, which can slow the performance of any activity that your application is running.

The Android framework also provides the `IntentService` subclass of `Service` that uses a worker thread to handle all of the start requests, one at a time. Using this class is **not recommended** for new apps as it will not work well starting with Android 8 Oreo, due to the introduction of Background execution limits. Moreover, it's deprecated starting with Android 11. You can use JobIntentService as a replacement for `IntentService` that is compatible with newer versions of Android.

The following sections describe how you can implement your own custom service, however you should strongly consider using WorkManager instead for most use cases. Consult the guide to background processing on Android to see if there is a solution that fits your needs.

Extending the Service class

You can extend the `Service` class to handle each incoming intent. Here's how a basic implementation might look:

```
class HelloService : Service() {

    private var serviceLooper: Looper? = null
    private var serviceHandler: ServiceHandler? = null

    // Handler that receives messages from the thread
    private inner class ServiceHandler(looper: Looper) : Handler(looper) {

        override fun handleMessage(msg: Message) {
            // 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 (e: InterruptedException) {
                // 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)
        }
```

```
    }

    override fun 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("ServiceStartArguments", Process.THREAD_PRIORITY_BACKGROUND).apply {
            start()

            // Get the HandlerThread's Looper and use it for our Handler
            serviceLooper = looper
            serviceHandler = ServiceHandler(looper)
        }
    }

    override fun onStartCommand(intent: Intent, flags: Int, startId: Int): Int {
        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
        serviceHandler?.obtainMessage()?.also { msg ->
            msg.arg1 = startId
            serviceHandler?.sendMessage(msg)
        }

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

    override fun onBind(intent: Intent): IBinder? {
        // We don't provide binding, so return null
        return null
    }

    override fun onDestroy() {
        Toast.makeText(this, "service done", Toast.LENGTH_SHORT).show()
    }
}
```

```
public class HelloService extends Service {
  private Looper serviceLooper;
  private ServiceHandler serviceHandler;

  // 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);
    }
}

@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 doesn't 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
  serviceLooper = thread.getLooper();
  serviceHandler = new ServiceHandler(serviceLooper);
}

@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 = serviceHandler.obtainMessage();
    msg.arg1 = startId;
    serviceHandler.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();
  }
}
```

The example code handles all incoming calls in `onStartCommand()` and posts the work to a `Handler` running on a background thread. It works just like an `IntentService` and processes all requests serially, one after another. You could change the code to run the work on a thread pool, for example, if you'd like to run multiple requests simultaneously.

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. 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 are pending intents to start the service. In that case, those intents are delivered. This is suitable for media players (or similar services) that are not executing commands but are 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` to `startService()` or `startForegroundService()`. The Android system calls the service's `onStartCommand()` method and passes it the `Intent`, which specifies which service to start.

Note: If your app targets API level 26 or higher, the system imposes restrictions on using or creating background services unless the app itself is in the foreground. If an app needs to create a foreground service, the app should call `startForegroundService()`. That method creates a background service, but the method signals to the system that the service will promote itself to the foreground. Once the service has been created, the service must call its `startForeground()` method within five seconds.

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

```
startService(Intent(this, HelloService::class.java))
```

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

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

If the service doesn't also provide binding, the intent that is 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, 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 doesn't stop or destroy the service unless it must recover system memory and the service continues to run after `onStartCommand()` returns. 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.

If your service handles multiple requests to `onStartCommand()` concurrently, you shouldn't stop the service when you're done processing a start request, as you might have 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 receives a new start request before you are able to call `stopSelf(int)`, the ID doesn't match and the service doesn't stop.

Caution: To avoid wasting system resources and consuming battery power, ensure that your application stops its services when it's done working. 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 receives 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()` to create a long-standing connection. It generally doesn't allow components to *start* it by calling `startService()`.

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, 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 same way that you must when the service is started through `onStartCommand()`.

To create a bound service, you must 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. After the client receives the `IBinder`, it can begin interacting with the service through that interface.

Multiple clients can bind to the service simultaneously. When a client is done interacting with the service, it calls `unbindService()` to unbind. When 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. For these reasons, the bound service discussion appears in a separate document about Bound Services.

Sending notifications to the user

When a service is running, it can notify the user of events using snackbar notifications or status bar notifications.

A snackbar notification is a message that appears on the surface of the current window for only a moment before disappearing. 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 to use when background work such as a file download has completed, 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 display the downloaded file).

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 either of these two 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 aren't entirely separate. You can bind to a service that is already started with `startService()`. For example, you can start a background music service 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 such as this, `stopService()` or `stopSelf()` doesn't actually stop the service until all of the 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:

```
class ExampleService : Service() {
    private var startMode: Int = 0             // indicates how to behave if the service is killed
    private var binder: IBinder? = null        // interface for clients that bind
    private var allowRebind: Boolean = false   // indicates whether onRebind should be used

    override fun  onCreate () {
        // The service is being created
    }

    override fun  onStartCommand (intent: Intent?, flags: Int, startId: Int): Int {
        // The service is starting, due to a call to startService()
        return startMode
    }

    override fun  onBind (intent: Intent): IBinder? {
```

```
        // A client is binding to the service with bindService()
        return binder
    }

    override fun onUnbind(intent: Intent): Boolean {
        // All clients have unbound with unbindService()
        return allowRebind
    }

    override fun onRebind(intent: Intent) {
        // A client is binding to the service with bindService(),
        // after onUnbind() has already been called
    }

    override fun onDestroy() {
        // The service is no longer used and is being destroyed
    }
}
```

```
public class ExampleService extends Service {
    int startMode;       // indicates how to behave if the service is killed
    IBinder binder;      // interface for clients that bind
    boolean allowRebind; // 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 startMode;
    }
    @Override
    public IBinder onBind(Intent intent) {
        // A client is binding to the service with bindService()
        return binder;
    }
    @Override
    public boolean onUnbind(Intent intent) {
        // All clients have unbound with unbindService()
        return allowRebind;
    }
    @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.

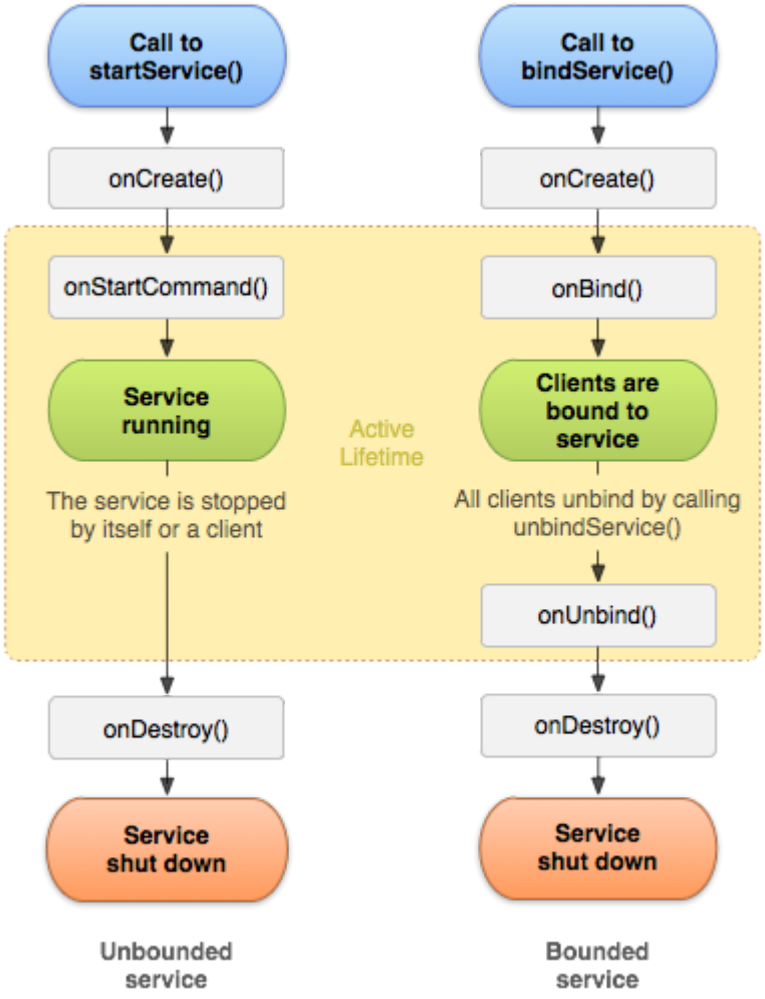

Figure 2. The service lifecycle. The diagram on the left shows the lifecycle when the service is created with `startService()` and the diagram on the right shows the lifecycle when the service is created with `bindService()`.

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. 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()`).

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

- The **entire lifetime** of a service occurs between the time that `onCreate()` is called and the time that `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 can create the thread where the music is played in `onCreate()`, and then it can stop the thread in `onDestroy()`.

  Note: 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()`.

  If the service is started, the active lifetime ends at 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 isn't a respective callback for the service (there's no `onStop()` callback). Unless the service is bound to a client, the system destroys it when the service is stopped— `onDestroy()` is the only callback received.

For more information about creating a service that provides binding, see the Bound Services document, which includes more information about the `onRebind()` callback method in the section about Managing the lifecycle of a bound service.
