# Bound services overview

developer.android.com/develop/background-work/services/foreground-services

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

This document describes how to create a bound service, including how to bind to the service from other application components. For additional information about services in general, such as how to deliver notifications from a service and set the service to run in the foreground, refer to the Services overview.

## The basics

A bound service is an implementation of the `Service` class that lets other applications bind to it and interact with it. To provide binding for a service, you 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.

### Bind to a started service

As discussed in the Services overview, you can create a service that is both started and bound. That is, you can start a service by calling `startService()`, which lets the service run indefinitely. You can also let a client bind to the service by calling `bindService()`.

If you let your service be started and bound, then when the service starts, the system *doesn't* destroy the service when all clients unbind. Instead, you must explicitly stop the service by calling `stopSelf()` or `stopService()`.

Although you usually implement either `onBind()` *or* `onStartCommand()`, it's sometimes necessary to implement both. For example, a music player might find it useful to let its service run indefinitely and also provide binding. This way, an activity can start the service to play some music, and the music continues to play even if the user leaves the application. Then, when the user returns to the application, the activity can bind to the service to regain control of playback.

For more information about the service lifecycle when adding binding to a started service, see the Manage the lifecycle of a bound service section.

A client binds to a service by calling `bindService()`. When it does, it must provide an implementation of `ServiceConnection`, which monitors the connection with the service. The return value of `bindService()` indicates whether the requested service exists and whether the client is permitted access to it.

When the Android system creates the connection between the client and service, it calls `onServiceConnected()` on the `ServiceConnection`. The `onServiceConnected()` method includes an `IBinder` argument, which the client then uses to communicate with the bound service.

You can connect multiple clients to a service simultaneously. However, the system caches the `IBinder` service communication channel. In other words, the system calls the service's `onBind()` method to generate the `IBinder` only when the first client binds. The system then delivers that same `IBinder` to all additional clients that bind to that same service, without calling `onBind()` again.

When the last client unbinds from the service, the system destroys the service, unless the service was started using `startService()`.

The most important part of your bound service implementation is defining the interface that your `onBind()` callback method returns. The following section discusses several ways that you can define your service's `IBinder` interface.

## Create a bound service

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:

**Extend the Binder class**

If your service is private to your own application and runs in the same process as the client, which is common, 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 the `Service`.

This is the preferred technique when your service is merely a background worker for your own application. The only use case when this is not the preferred way to create your interface is if your service is used by other applications or across separate processes.

**Use 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, letting the client 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.

**Use AIDL**

Android Interface Definition Language (AIDL) decomposes objects into primitives that the operating system can understand and marshalls them across processes to perform IPC. The previous technique, using a `Messenger`, is actually based on AIDL as its underlying structure.
As mentioned in the preceding section, the `Messenger` creates a queue of all the client requests in a single thread, so the service receives requests one at a time. If, however, you want your service to handle multiple requests simultaneously, then you can use AIDL directly. In this case, your service must be thread-safe and capable of multithreading.

To use AIDL directly, 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.

**Note:** For most applications, AIDL isn't the best choice to create a bound service, because it might require multithreading capabilities and can result in a more complicated implementation. Therefore, this document does not discuss how to use it for your service. If you're certain that you need to use AIDL directly, see the AIDL document.

## Extend the Binder class

If only the local application uses your service and it doesn't need to work across processes, then you can implement your own `Binder` class that provides your client direct access to public methods in the service.

**Note:** This works only if the client and service are in the same application and process, which is most common. For example, this works well for a music application that needs to bind an activity to its own service that's playing music in the background.

Here's how to set it up:

1. In your service, create an instance of `Binder` that does one of the following:
    - Contains public methods that the client can call.
    - Returns the current `Service` instance, which has public methods the client can call.
    - Returns an instance of another class hosted by the service with public methods the client can call.
2. Return this instance of `Binder` from the `onBind()` callback method.
3. In the client, receive the `Binder` from the `onServiceConnected()` callback method and make calls to the bound service using the methods provided.

**Note:** The service and client must be in the same application so that 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 doesn't perform any marshalling across processes.

For example, here's a service that provides clients with access to methods in the service through a `Binder` implementation:

KotlinJava

```
class LocalService : Service() {
    // Binder given to clients.
    private val binder = LocalBinder()
    // Random number generator.
    private val mGenerator = Random()
    /** Method for clients.  */
    val randomNumber: Int
        get() = mGenerator.nextInt(100)
    /**
     * 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.
     */
    inner class LocalBinder : Binder() {
        // Return this instance of LocalService so clients can call public
methods.
        fun getService(): LocalService = this@LocalService
    }
    override fun onBind(intent: Intent): IBinder {
        return binder
    }
}
```

The `LocalBinder` provides the `getService()` method for clients to retrieve the current instance of `LocalService`. This lets clients call public methods in the service. For example, clients can call `getRandomNumber()` from the service.

Here's an activity that binds to `LocalService` and calls `getRandomNumber()` when a button is clicked:

KotlinJava

```
class BindingActivity : Activity() {
    private lateinit var mService: LocalService
    private var mBound: Boolean = false
    /** Defines callbacks for service binding, passed to bindService().  */
    private val connection = object : ServiceConnection {
        override fun onServiceConnected(className: ComponentName, service:
IBinder) {
            // We've bound to LocalService, cast the IBinder and get LocalService
instance.
            val binder = service as LocalService.LocalBinder
            mService = binder.getService()
            mBound = true
        }
        override fun onServiceDisconnected(arg0: ComponentName) {
            mBound = false
        }
    }
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.main)
    }
    override fun onStart() {
        super.onStart()
        // Bind to LocalService.
        Intent(this, LocalService::class.java).also { intent ->
            bindService(intent, connection, Context.BIND_AUTO_CREATE)
        }
    }
    override fun onStop() {
        super.onStop()
        unbindService(connection)
        mBound = false
    }
    /** Called when a button is clicked (the button in the layout file attaches to
     * this method with the android:onClick attribute).  */
    fun onButtonClick(v: View) {
        if (mBound) {
            // Call a method from the LocalService.
            // However, if this call is something that might hang, then put this
request
            // in a separate thread to avoid slowing down the activity
performance.
            val num: Int = mService.randomNumber
            Toast.makeText(this, "number: $num", Toast.LENGTH_SHORT).show()
        }
    }
}
```

The preceding sample shows how the client binds to the service using an implementation of `ServiceConnection` and the `onServiceConnected()` callback. The next section provides more information about this process of binding to the service.

**Note:** In the preceding example, the `onStop()` method unbinds the client from the service. Unbind clients from services at appropriate times, as discussed in the Additional notes section.

For more sample code, see the `LocalService.java` class and the `LocalServiceActivities.java` class in ApiDemos.

## Use a Messenger

If you need your service to communicate with remote processes, then you can use a `Messenger` to provide the interface for your service. This technique lets you perform interprocess communication (IPC) without the need to use AIDL.

Using a `Messenger` for your interface is simpler than using AIDL because `Messenger` queues all calls to the service. A pure AIDL interface sends simultaneous requests to the service, which must then handle multithreading.

For most applications, the service doesn't need to perform multithreading, so using a `Messenger` lets the service handle one call at a time. If it's important that your service be multithreaded, use AIDL to define your interface.

Here's a summary of how to use a `Messenger`:

1. The service implements a `Handler` that receives a callback for each call from a client.
2. The service uses the `Handler` to create a `Messenger` object (which is a reference to the `Handler`).
3. The `Messenger` creates an `IBinder` that the service returns to clients from `onBind()`.
4. 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.
5. The service receives each `Message` in its `Handler`—specifically, in the `handleMessage()` method.

In this way, there are no *methods* for the client to call on the service. Instead, the client delivers *messages* (`Message` objects) that the service receives in its `Handler`.

Here's a simple example service that uses a `Messenger` interface:

KotlinJava

```
/** Command to the service to display a message.  */
private const val MSG_SAY_HELLO = 1
class MessengerService : Service() {
    /**
     * Target we publish for clients to send messages to IncomingHandler.
     */
    private lateinit var mMessenger: Messenger
    /**
     * Handler of incoming messages from clients.
     */
    internal class IncomingHandler(
            context: Context,
            private val applicationContext: Context = context.applicationContext
    ) : Handler() {
        override fun handleMessage(msg: Message) {
            when (msg.what) {
                MSG_SAY_HELLO ->
                    Toast.makeText(applicationContext, "hello!",
Toast.LENGTH_SHORT).show()
                else -> super.handleMessage(msg)
            }
        }
    }
    /**
     * When binding to the service, we return an interface to our messenger
     * for sending messages to the service.
     */
    override fun onBind(intent: Intent): IBinder? {
        Toast.makeText(applicationContext, "binding", Toast.LENGTH_SHORT).show()
        mMessenger = Messenger(IncomingHandler(this))
        return mMessenger.binder
    }
}
```

The `handleMessage()` method in the `Handler` is where the service receives the incoming `Message` and decides what to do, based on the `what` member.

All that a client needs to do is create a `Messenger` based on the `IBinder` returned by the service and send a message using `send()`. For example, here's an activity that binds to the service and delivers the `MSG_SAY_HELLO` message to the service:

This example doesn't show how the service can respond to the client. If you want the service to respond, you need to also create a `Messenger` in the client. When the client receives the `onServiceConnected()` callback, it sends a `Message` to the service that includes the client's `Messenger` in the `replyTo` parameter of the `send()` method.

You can see an example of how to provide two-way messaging in the `MessengerService.java` (service) and `MessengerServiceActivities.java` (client) samples.

## Bind to a service

Application components (clients) can bind to a service by calling `bindService()`. The Android system then calls the service's `onBind()` method, which returns an `IBinder` for interacting with the service.

The binding is asynchronous, and `bindService()` returns immediately *without* returning the `IBinder` to the client. To receive the `IBinder`, the client must create an instance of `ServiceConnection` and pass it to `bindService()`. The `ServiceConnection` includes a callback method that the system calls to deliver the `IBinder`.

**Note:** Only activities, services, and content providers can bind to a service—you **can't** bind to a service from a broadcast receiver.

To bind to a service from your client, follow these steps:

1. Implement `ServiceConnection`.
   Your implementation must override two callback methods:

   **`onServiceConnected()`**
   The system calls this to deliver the `IBinder` returned by the service's `onBind()` method.

   **`onServiceDisconnected()`**
   The Android system calls this when the connection to the service is unexpectedly lost, such as when the service crashes or is killed. This is *not* called when the client unbinds.

2. Call `bindService()`, passing the `ServiceConnection` implementation.
   **Note:** If the method returns false, your client does not have a valid connection to the service. However, do call `unbindService()` in your client. Otherwise, your client keeps the service from shutting down when it is idle.

3. When the system calls your `onServiceConnected()` callback method, you can begin making calls to the service, using the methods defined by the interface.
4. To disconnect from the service, call `unbindService()`.
   If your client is still bound to a service when your app destroys the client, destruction causes the client to unbind. It is a better practice to unbind the client as soon as it is done interacting with the service. Doing so lets the idle service shut down. For more information about appropriate times to bind and unbind, see the Additional notes section.

The following example connects the client to the service created previously by extending the Binder class, so all it needs to do is cast the returned `IBinder` to the `LocalBinder` class and request the `LocalService` instance:

With this `ServiceConnection`, the client can bind to a service by passing it to `bindService()`, as shown in the following example:

- The first parameter of `bindService()` is an `Intent` that explicitly names the service to bind.
  **Caution:** If you use an intent to bind to a `Service`, make sure your app is secure by using an explicit intent. Using an implicit intent to start a service is a security hazard because you can't be certain what service responds to the intent, and the user can't 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.

- The second parameter is the `ServiceConnection` object.

## Additional notes

Here are some important notes about binding to a service:

- Always trap `DeadObjectException` exceptions, which are thrown when the connection has broken. This is the only exception thrown by remote methods.
- Objects are reference counted across processes.
- You usually pair the binding and unbinding during the matching bring-up and tear-down moments of the client's lifecycle, as described in the following examples:

  If you want your activity to receive responses even while it is stopped in the background, bind during `onCreate()` and unbind during `onDestroy()`. Beware that this implies that your activity needs to use the service the entire time it's running, even in the background, so when the service is in another process, then you increase the weight of the process and it is more likely to be killed by the system.

  **Note:** You *don't* usually bind and unbind during your activity's `onResume()` and `onPause()` callbacks, because these callbacks occur at every lifecycle transition. Keep the processing that occurs at these transitions to a minimum.

  Also, if multiple activities in your application bind to the same service and there is a transition between two of those activities, the service might be destroyed and recreated as the current activity unbinds (during pause) before the next one binds (during resume). This activity transition for how activities coordinate their lifecycles is described in The activity lifecycle.

For more sample code showing how to bind to a service, see the `RemoteService.java` class in ApiDemos.

## Manage the lifecycle of a bound service

When a service is unbound from all clients, the Android system destroys it (unless it was started using `startService()`). So, you don't have to manage the lifecycle of your service if it's purely a bound service. The Android system manages it for you based on whether it is bound to any clients.

However, if you choose to implement the `onStartCommand()` callback method, then you must explicitly stop the service, because the service is now considered *started*. In this case, the service runs until the service stops itself with `stopSelf()` or another component calls `stopService()`, regardless of whether it is bound to any clients.

Additionally, if your service is started and accepts binding, then when the system calls your `onUnbind()` method, you can optionally return `true` if you want to receive a call to `onRebind()` the next time a client binds to the service. `onRebind()` returns void, but the client still receives the `IBinder` in its `onServiceConnected()` callback. The following figure illustrates the logic for this kind of lifecycle.

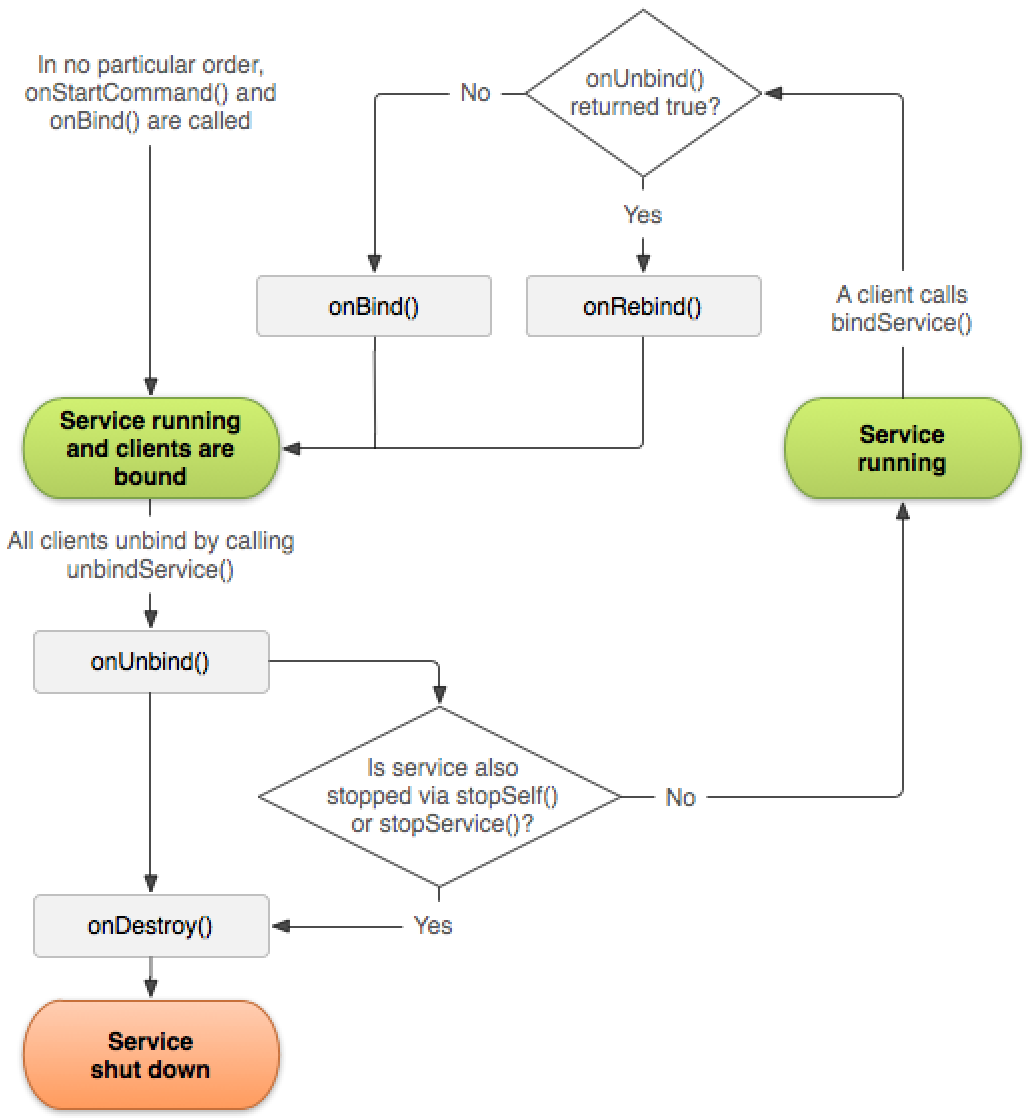
In no particular order, onStartCommand() and onBind() are called
No
onUnbind() returned true?
Yes
onBind()
onRebind()
A client calls bindService()
Service running and clients are bound
Service running
All clients unbind by calling unbindService()
onUnbind()
Is service also stopped via stopSelf() or stopService()?
No
onDestroy()
Yes
Service shut down

**Figure 1.** The lifecycle for a service that is started and also allows binding.

For more information about the lifecycle of a started service, see the Services overview.