

# Android - Services

A Service is an application component designed to perform a longer-running operation without user interface. like every component each service class must be declared in the manifest file. Note that services run in the main thread of the application. There is a confusion about this, because services don't have UI it is a common mistake to believe that they are running in a background thread, however this is not true, **regular services run on the main thread of the application**. If your service is going to do any CPU intensive or networking operations, it should create its own thread in which to do that work.

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 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 (this depends on the value you return from `onStartCommand()`, as discussed later).

A Service can either be started (common) or bounded (or both at the same time):

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

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

# Creating a Started Service

For creating a **started** service there are two classes you can inherit from:

**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.

## Extending Service class:

As we have seen the `IntentService` is quite simple it will perform all the tasks passed one after the other in a queue, however if you wish to perform all the task simultaneously and create multi-threading tasks you must **extends the Service class**. By doing so you must override the **`onStartCommand`** method that receives the intents and creates the background threads yourself (Note: you will also need to override the **`onBind`** method since services can also be binded).

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 `IntentService` handles this for you). The return value must be one of the following constants:

`START_NOT_STICKY` - If the system kills the service, *do not* recreate the service. This is the safest option to avoid running your service when not necessary.

`START_STICKY` - If the system kills the service it will recreate the service and call `onStartCommand()`, with a null intent. This is suitable for media players (or similar services) that are running indefinitely and waiting for a job (for a new intent).

`START_REDELIVER_INTENT` - If the system kills the service, it recreate the service and call `onStartCommand()` with the last intent that was delivered to the service. This is suitable for services that are actively performing a job that should be immediately resumed, such as downloading a file.

## finishing the service

Another thing which you will have to take care of when subclassing Service is **manually finishing the service and freeing all of it's resources when the task is done**. When you want to stop this service you have two options:

A. from within the service you can call **stopSelf()**, this will destroy the service (you can also stop a single task using the **stopSelf(int)** and pass the ID you got in the **onStartCommand()**).

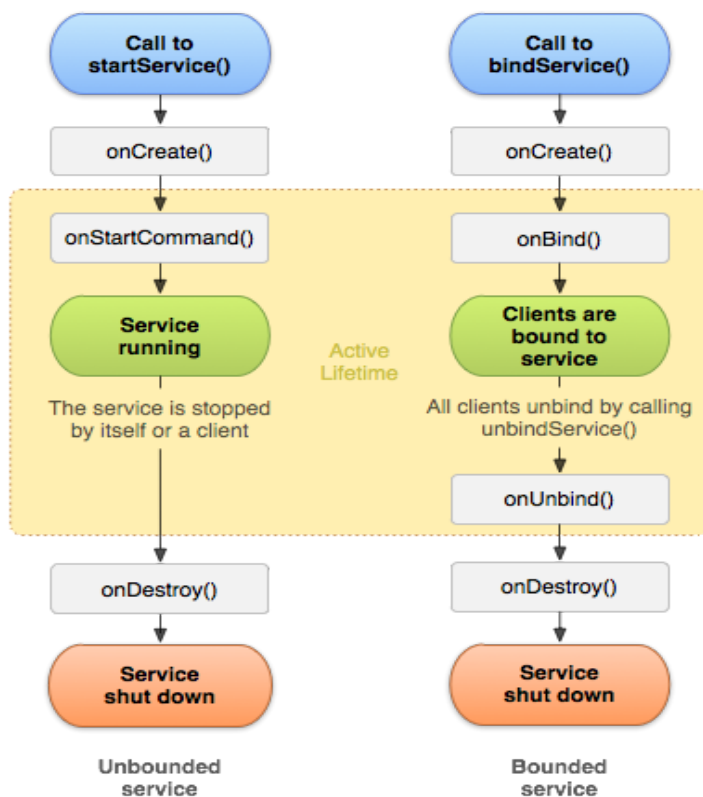
B. From the activity or anywhere else you can use the context's **stopService(intent)** method and pass the original intent used to create the service.

## Service Lifecycle

Services run without UI so it is very important to understand when it is created and destroyed.

A started service - The service is created when another component calls **startService()**. The service then runs indefinitely and must be stopped by himself or by another component. 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.)



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

The **active lifetime** of a service begins with a call to either `onStartCommand()` or `onBind()`.

If the service is started, the active lifetime ends the same time that the entire lifetime

ends (the service is still active even after `onStartCommand()` returns thus you must finish it yourself). If the service is bound, the active lifetime ends when `onUnbind()` returns.

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

## Foreground Services

A foreground service is a service that's considered to be something the user is actively aware of and thus not a candidate for the system to kill when low on memory. It can live forever. A foreground service must provide a notification for the status bar, this notification cannot be dismissed unless the service is either stopped or removed from the foreground.

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

To request that your service run in the foreground, call the context `startForeground()`. This method takes two parameters: an integer that uniquely identifies the notification (must not be 0) and the `Notification` for the status bar. It is important to understand that this function does not create the service, after creating the service with the `startService` you should call it from within the service (from service `onCreate` function for example).

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

**Reference:** [Services](#), Android Developer guide