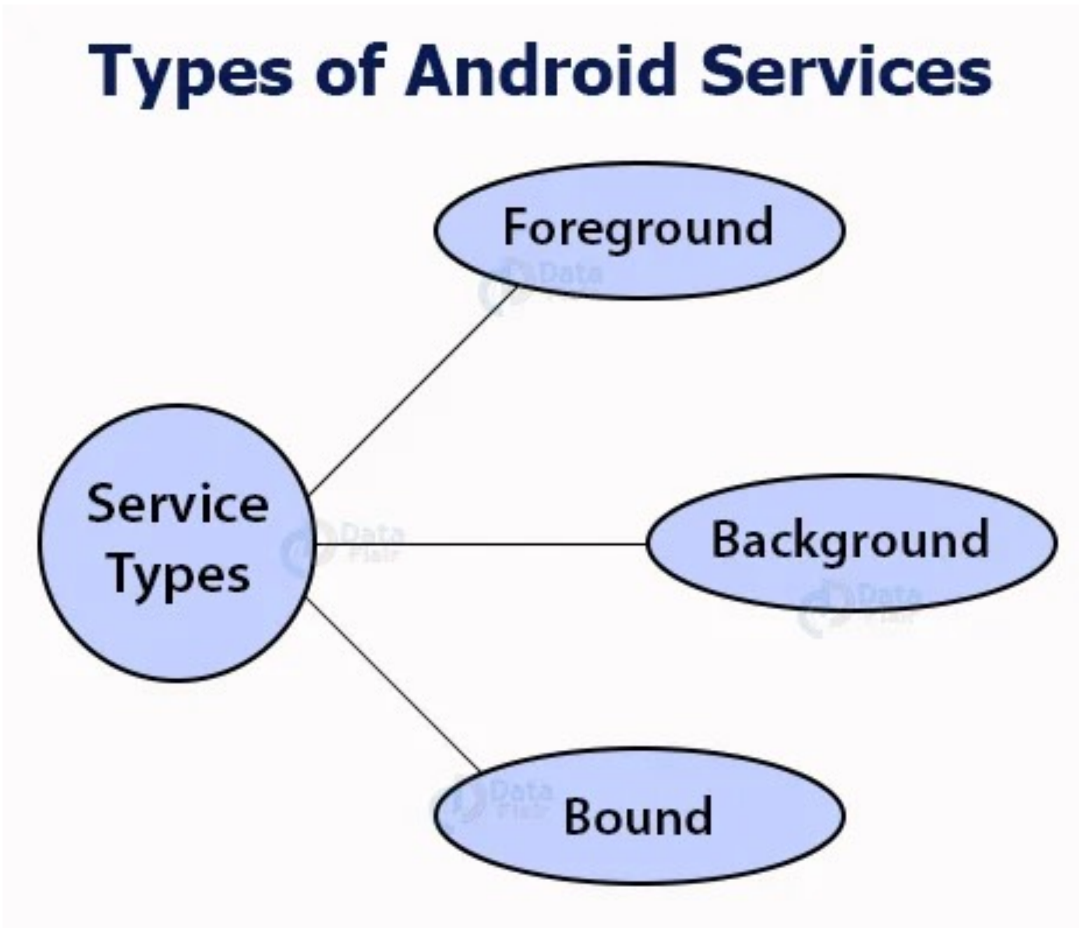
What are Android Services?

Android Services are the application components that run in the background. We can understand it as a process that doesn't need any direct user interaction. As they perform long-running processes without user intervention, they have no User Interface. They can be connected to other components and do **inter-process communication (IPC)**.

Types of Android Services

When we talk about services, they can be of three types as shown in the figure below:



The working of these three services is below:

1. Foreground Services

Foreground services are those services that are visible to the users. The users can interact with them at ease and track what's happening. These services continue to run even when users are using other applications.

The perfect example of this is Music Player and Downloading.

2. Background Services

These services run in the background, such that the user can't see or access them. These are the tasks that don't need the user to know them.

Syncing and Storing data can be the best example.

3. Bound Services

Bound service runs as long as some other ***application component*** is bound to it. Many components can bind to one service at a time, but once they all unbind, the service will destroy.

To bind an application component to the service, **bindService()** is used.

Lifecycle of Android Services

Android services life-cycle can have two forms of services and they follow two paths, that are:

- Started Service

- Bounded Service

Let us see these services and their approach.

1. Started Service

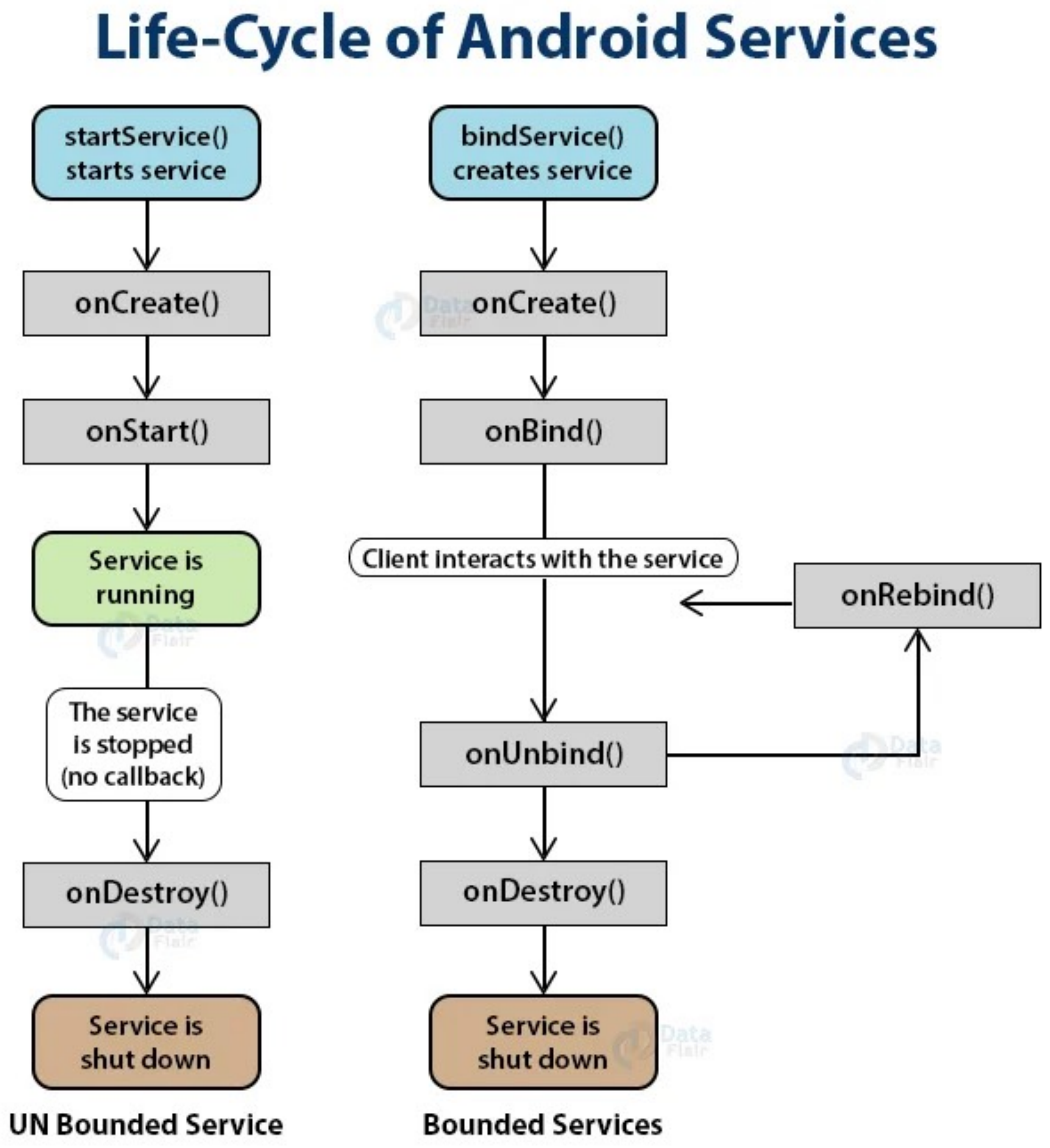
A service becomes started only when an application component calls **startService()**. It performs a single operation and doesn't return any result to the caller. Once this service starts, it runs in the background even if the component that created it destroys. This service can be stopped only in one of the two cases:

- By using the stopService() method.
- By stopping itself using the stopSelf() method.

2. Bound Service

A service is bound only if an application component binds to it using **bindService()**. It gives a client-server relation that lets the components interact with the service. The components can send requests to services and get results.

This service runs in the background as long as another application is bound to it. Or it can be unbound according to our requirement by using the **unbindService()** method.



IntentService()

There's an additional service class, that extends Service class, **IntentService** Class. It is a base class for services to handle asynchronous requests. It enables running an operation on a single background. It executes long-running programs without affecting any user's interface interaction. Intent services run and execute in the background and terminate themself as soon as they are executed completely.

Certain important features of Intent are :

- It queues up the upcoming request and executes them one by one.
- Once the queue is empty it stops itself, without the user’s intervention in its lifecycle.
- It does proper thread management by handling the requests on a separate thread.

Methods of Android Services

The service base class defines certain callback methods to perform operations on applications. When we talk about Android services it becomes quite obvious that these services will do some operations and they’ll be used. The following are a few important methods of Android services :

- onStartCommand()
- onBind()
- onCreate()
- onUnbind()
- onDestroy()
- onRebind()

Let us see these methods in detail:

1. onStartCommand()

The system calls this method whenever a component, say an **activity** requests ‘start’ to a service, using **startService()**.

Once we use this method it’s our duty to stop the service using **stopService()** or **stopSelf()**.

2. onBind()

This is invoked when a component wants to bind with the service by calling **bindService()**. In this, we must provide an interface for clients to communicate with the service. For interprocess communication, we use the **IBinder object**.

It is a must to implement this method. If in case binding is not required, we should return **null** as implementation is mandatory.

3. onUnbind()

The system invokes this when all the clients disconnect from the interface published by the service.

4. onRebind()

The system calls this method when new clients connect to the service. The system calls it after the **onBind()** method.

5. onCreate()

This is the first callback method that the system calls when a new component starts the service. We need this method for a one-time set-up.

6. onDestroy()

This method is the final clean up call for the system. The system invokes it just before the service destroys. It cleans up resources like threads, receivers, registered listeners, etc.

Broadcast Receiver in Android With Example

Broadcast in android is the system-wide events that can occur when the device starts, when a message is received on the device or when incoming calls are received, or when a device goes to airplane mode, etc. Broadcast Receivers are used to respond to these system-wide events. Broadcast Receivers allow us to register for the system and application events, and when that event happens, then the register receivers get notified. There are mainly two types of Broadcast Receivers:

- **Static Broadcast Receivers:** These types of Receivers are declared in the manifest file and works even if the app is closed.
- **Dynamic Broadcast Receivers:** These types of receivers work only if the app is active or minimized.

Since from API Level 26, most of the broadcast can only be caught by the dynamic receiver, so we have implemented dynamic receivers in our sample project given below. There are some static fields defined in the Intent class which can be used to broadcast different events. We have taken a change of airplane mode as a broadcast event, but there are many events for which broadcast register can be used. Following are some of the important system-wide generated intents:-

android.intent.action.BATTERY_LOW :	Indicates low battery condition on the device.
android.intent.action.BOOT_COMPLETED	This is broadcast once after the system has finished booting
android.intent.action.CALL	To perform a call to someone specified by the data
android.intent.action.DATE_CHANGED	Indicates that the date has changed
android.intent.action.REBOOT	Indicates that the device has been a reboot
android.net.conn.CONNECTIVITY_CHANGE	The mobile network or wifi connection is changed(or reset)
android.intent.ACTION_AIRPLANE_MODE_CHANGED	This indicates that airplane mode has been switched on or off.

The two main things that we have to do in order to use the broadcast receiver in our application are:

Creating the Broadcast Receiver:

```
class AirplaneModeChangeReceiver:BroadcastReceiver() {
    override fun onReceive(context: Context?, intent: Intent?) {
        // logic of the code needs to be written here
    }
}
```

Registering a BroadcastReceiver:

```
IntentFilter(Intent.ACTION_AIRPLANE_MODE_CHANGED).also {
                    // receiver is the broadcast receiver that we have registered
                // and it is the intent filter that we have created
                registerReceiver(receiver,it)
    }
```

An Android Room Database and Repository

Data from the app can be saved on users' devices in different ways. We can store data in the user's device in SQLite tables, shared preferences, and many more ways. In this article, we will take a look at **saving data, reading, updating, and deleting data in Room Database** on Android. We will perform CRUD operations using Room Database on Android. In this article, we will take a look at performing CRUD operations in Room Database in Android.

What is Room?

Room is a persistence library that provides an abstraction layer over the SQLite database to allow a more robust database.

With the help of room, we can easily create the database and perform CRUD operations very easily.

Components of Room

The three main components of the room are **Entity, Database, and DAO**.

Entity: Entity is a modal class that is annotated with @Entity. This class is having variables that will be our columns and the class is our table.
Database: It is an abstract class where we will be storing all our database entries which we can call Entities.
DAO: The full form of DAO is a **Database access object** which is an interface class with the help of it we can perform different operations in our database.