# What is a service?

A service is a component in Android that performs long-running operations in the background without a user interface

# How many types of services are there?

There are three types of services in Android:

* + - Background Services (Started Service)
    - Bound Services
    - Foreground Services

# Does Service have a UI interface?

No, services do not have a UI interface. They run in the background and are typically used for operations that don't require user interaction.

# What happens when you run a UI-blocking long-running task on a service?

Running a UI-blocking long-running task on a service can cause the UI to become unresponsive and lead to an ANR (Application Not Responding) error. It's important to offload such tasks to background threads or use async mechanisms to avoid blocking the UI thread.

# Define Foreground service in one line?

A foreground service is a type of service that has a notification associated with it, indicating to the user that the service is running in the foreground, even when the app is not in the foreground.

# What is the most important thing for a foreground service to run?

The most important thing for a foreground service to run is to show a persistent notification to the user, informing them that the service is active.

# Which other API offers a flexible way of scheduling tasks other than a foreground service?

*WorkManager* API offers a flexible way of scheduling tasks in the background, providing features like constraints, retries, and guaranteed execution.

# Define a background service in one line?

A background service is a type of service that runs in the background without a UI and is used for tasks that do not require user interaction.

# What are the limitations when a service runs as a background service?

When a service runs as a background service, it may be subject to limitations imposed by the system, such as restrictions on background execution in newer Android versions (e.g., Android 8+).

# Define Bound service in one line?

A bound service is a type of service that allows other components (like activities) to bind to it and interact with it through an interface (usually an *IBinder*).

# When is a service considered as background, foreground, and bound?

A service is considered:

* + - Background when it runs in the background without a foreground notification.
    - Foreground when it runs in the foreground with a persistent notification.
    - Bound when other components bind to it to interact with it.

# Can a service behave as a background, foreground, or bound at the same time?

No, a service can only behave as one type at a time. It can be either a background service, a foreground service, or a bound service based on how it is configured and used.

# Can any other app component use the service of any other app?

Yes, app components from different apps can interact with each other's services using explicit intents if the service allows it through proper configuration and permissions.

# Explain a case in which we should use a background thread instead of a service?

If the task is short-lived and doesn't need to continue running if the app is closed or the component is destroyed, using a background thread (like *AsyncTask* or Kotlin coroutines) within the activity or fragment might be more appropriate than starting a service.

# What are subclasses of the Service name and their uses?

The subclasses of the Service class in Android include:

* *IntentService*: Used for handling asynchronous requests on a single worker thread.
* *JobIntentService*: Similar to *IntentService* but provides compatibility with Android 8.0+ *JobScheduler*.
* *ForegroundService*: Used for tasks that require to be in the foreground with a notification.
* *JobService* (deprecated): Previously used for background tasks scheduled with *JobScheduler*, now replaced by *JobIntentService* and *WorkManager*.

# What are the most important callbacks of Service that you should override?

The most important callbacks of the Service class are:

* *onCreate*(): Called when the service is created.
* *onStartCommand*(Intent intent, int flags, int startId): Called when the service is started.
* *onBind*(*Intent* *intent*): Called when a client binds to the service (for bound services).
* *onDestroy*(): Called when the service is destroyed.

# When is the onStartCommand() method invoked?

The onStartCommand() method is invoked when another component (such as an activity) calls startService() to start the service.

# When is it the developer's responsibility to stop the service, and how many ways can it be stopped?

It is the developer's responsibility to stop the service when it is no longer needed. The service can be stopped using *stopSelf*() from within the service or by another component calling *stopService*().

# When is the *onBind*() callback invoked?

The *onBind*() callback is invoked when a component binds to the service using *bindService*(). It returns an IBinder interface that the client can use to interact with the service.

# How to start a service as bound as well as foreground at the same time and invoke both *onBind*() and onStartCommand() methods?

To start a service as bound and foreground simultaneously, you can use a combination of *startForegroundService*() (for foreground) and *bindService*() (for bound). This way, both *onBind*() and onStartCommand() methods will be invoked.

# What is the role of IBinder, and when is *onBind*() necessary, and when should it return null?

The IBinder interface is used for communication between the service and the client (e.g., activity). *onBind*() is necessary for bound services to provide an IBinder implementation, and it should return null only if the service does not support binding.

# When is the *onCreate*() callback invoked, and what is its role?

The *onCreate*() callback is invoked when the service is initially created. It is used for initialization tasks such as setting up variables, initializing components, or acquiring resources needed by the service.

# If a service is already running and *startService*() is called, does *onCreate*() callback invoked?

No, if a service is already running and *startService*() is called again, the *onCreate*() callback will not be invoked. It will only call *onStartCommand*() with the new intent.

# What will be the stop behaviour of a service started with *startService*()?

A service started with *startService*() will stop itself when *stopSelf*() is called or when another component calls *stopService*().

# What will be the stop behaviour of a service started with *bindService*()?

A service started with *bindService*() will stop itself when all clients unbind from it using *unbindService*().

# In which case will *onStartCommand*() not be called?

*onStartCommand*() will not be called if the service is already running and startService() is called again with the same intent without calling *stopService*() in between.

# What will be the stop *behavior* of a service started with *bindService*() if onStartCommand() is not called?

If *onStartCommand*() is not called (e.g., due to the service already running), the service will continue to run until all clients unbind from it, at which point *onDestroy*() will be called.

# When is a service least likely to be killed, rarely killed, and highly susceptible to killing?

A service is:

* Least likely to be killed when it's a foreground service with a persistent notification.
* Rarely killed when it's a bound service bound to an activity or other component.
* Highly susceptible to killing when it's a background service running without being bound or in the foreground.

# How can you set the restart behaviour of the service?

You can set the restart behaviour of a service using the START\_STICKY, *START*\_NOT\_STICKY, or START\_*REDELIVER*\_INTENT constants returned by the *onStartCommand*() method.

# How to declare a Service?

To declare a service in Android, you need to create a subclass of the Service class and declare it in the AndroidManifest.xml file within the <service> tag.

# Which attribute is must require to declare a service in the manifest?

The android:name attribute is a required attribute to declare a service in the manifest, specifying the class name of the service.

# Why should we take care that the service name remains unchanged while making a build?

It's important to keep the service name unchanged while making a build because changing the service name can break existing components that depend on the service, such as activities binding to the service.

# By which intent is it good practice to start a service, an implicit or explicit, and give an example of starting a service with both intents?

It's generally good practice to start a service with an explicit intent, as it specifies the exact component to start. Here are examples of starting a service with both intents:

Explicit Intent



Implicit Intent



# What happens if you *bindService()* with an implicit intent?

Binding a service with an implicit intent may lead to unpredictable behaviour or errors because the system may not be able to determine which service to bind to.

# What is the role of the *android:exported* attribute in the service tag while declaring in the manifest?

The *android:exported* attribute in the service tag specifies whether the service can be accessed by components outside the app (true) or not (false).

# What is the role of the android:description attribute in the service tag while declaring in the manifest?

The android:description attribute in the service tag is used to provide a description of the service for users or developers.

# What is a started service?

A started service is a service that is explicitly started using the startService() method and continues to run until it is explicitly stopped or stops itself.

# What is the behavior of the lifecycle of a started service?

The lifecycle of a started service includes the onCreate(), onStartCommand(), and onDestroy() methods. It starts when startService() is called, processes commands in onStartCommand(), and stops when stopSelf() or stopService() is called.

# How can you pass data to a service, and where can you get the data?

You can pass data to a service using intents (putExtra methods) when starting the service. In the service, you can retrieve the data from the intent received in the onStartCommand() method.

# Which subclass of a service uses a worker thread to do tasks?

The IntentService subclass of a service uses a worker thread to handle asynchronous tasks sequentially in the background.

# Which subclass of service is not recommended to use starting from Android 8, and what are the limitations due to which it is not recommended?

The BackgroundService subclass of service is not recommended to use starting from Android 8 (API level 26) onwards. This is due to limitations imposed on background execution, such as background service restrictions and battery optimization policies.

# Which subclass of service is deprecated in favor of Work Manager?

The JobService subclass of service is deprecated in favor of using the Work Manager API for scheduling background tasks in a more efficient and flexible way.

# On which version is the subclass of service that is not recommended deprecated, and what is the replacement for this deprecated class?

The BackgroundService subclass of service that is not recommended is deprecated starting from Android 8 (API level 26). The replacement for this deprecated class is using Work Manager for background task scheduling.

# What is the role of the integer value returned by onStartCommand(), and why is it important for the return value to be one of the specified constants?

The integer value returned by onStartCommand() specifies the behavior of the service when the system kills it due to resource constraints. It's important for the return value to be one of the specified constants (START\_STICKY, START\_NOT\_STICKY, START\_REDELIVER\_INTENT) to define how the service should behave after being killed and restarted by the system.

# Write the role of each specific constant returned by onStartCommand() and how they change the behavior of the service, explaining each with a real-life example?

START\_STICKY: If the service is killed, the system will recreate it and call onStartCommand() with a null intent, but the service's previous state (like running or paused) is not restored. Example: A music player service that should continue playing music even if the app is in the background.

START\_NOT\_STICKY: If the service is killed, it will not be restarted until the application explicitly calls startService() again. Example: A service for uploading photos that doesn't need to be restarted if it's killed.

START\_REDELIVER\_INTENT: If the service is killed, the system will recreate it and redeliver the last intent sent to it. Example: A download service that needs to continue downloading files even if the service is temporarily killed.

# What are the changes in the way of starting a service from Android 28?

Starting from Android 28 (API level 28), background execution restrictions have become more stringent. Services that run in the background may be subject to limitations and may not run indefinitely.

# What if a developer wants the service to send back a result to the activity that started it? What would be the best approach?

The best approach for a service to send back a result to the activity is to use a callback mechanism or broadcast intents. The service can send the result back to the activity through a callback interface or by broadcasting an intent with the result data.

# What happens if you call startService() and stopService() multiple times using the same intent?

If you call startService() multiple times with the same intent without calling stopService() in between, the service will receive multiple start requests but will only have one instance running. The service will be stopped only when stopService() is called the same number of times as startService().

# What is the role of startID passed in onStartCommand() as an argument?

The startId passed in onStartCommand() is an identifier that represents each start request for the service. It is used to track and manage multiple start requests, especially when stopping the service with stopSelf(int startId).

# What will be the stop behavior of the app if it is stopped by passing startID in stopSelf() as a parameter?

If the app is stopped by passing the startId in stopSelf() as a parameter, the service will be stopped only when all start requests identified by that startId have been stopped. This ensures that the service is stopped correctly based on its start requests.

# Does stopService() stop the service even if it's running as foreground as well as bound?

Yes, stopService() will stop the service regardless of whether it's running as a foreground service, bound service, or both. It will stop the service completely and release any resources associated with it.

# By which way can a bound service communicate with a client or bind component?

A bound service can communicate with a client or bind component by implementing an interface that extends IBinder. The client can then use this interface to interact with the service and access its methods.

# Can multiple components or clients bind to a bound service simultaneously?

Yes, multiple components or clients can bind to a bound service simultaneously. Each client will receive its own instance of the service connection, allowing them to interact with the service independently.

# Does unbindService() get called every time a service is unbound?

Yes, unbindService() is called every time a client unbinds from a bound service. It's important to properly handle the unbinding process to release resources and clean up connections.

# What is the destroy behavior of a service, and how is it destroyed when started with startService() and when started with bindService()?

The destroy behavior of a service involves the onDestroy() method, which is called when the service is being destroyed. When started with startService(), the service will be destroyed when stopSelf() or stopService() is called. When started with bindService(), the service will be destroyed when all clients unbind from it using unbindService().

# How to run a service as both foreground and bound at the same time?

To run a service as both foreground and bound at the same time, you can first start the service using startForegroundService() to make it a foreground service with a notification. Then, you can bind to the service using bindService() to establish a connection between the service and the client.

# Does stopService() or stopSelf() stop the service if it runs as a started service and then binds any components and remains bound?

Yes, both stopService() and stopSelf() will stop the service even if it's running as a started service and remains bound to components. The service will be stopped based on the stop request received, regardless of its binding status.

# When is onRebind() called?

The onRebind() method is called when a client that was previously bound to the service using bindService() calls bindService() again after being unbound and reconnected.

# Explain the service lifecycle callback methods flow for both started as foreground and bound services?

For a started service, the flow of lifecycle callback methods is onCreate() (if not already created) -> onStartCommand() -> onDestroy().

For a foreground service, it's the same as a started service but with startForeground() called in onStartCommand() to show a notification.

For a bound service, the flow is onCreate() (if not already created) -> onBind() -> onUnbind() (if all clients unbind) -> onRebind() (if rebind occurs) -> onDestroy() (if no clients are bound).

# What is active lifetime vs. entire lifetime of a service?

The active lifetime of a service refers to the time between its onCreate() and onDestroy() callbacks, during which it's actively running and performing tasks.

The entire lifetime of a service includes the active lifetime plus any time it spends in a started state without being destroyed, including periods when it's not actively performing tasks (e.g., paused or waiting for callbacks).