101. Servicios

Un Service es un componente de Android que realiza tareas de larga duración en segundo plano, sin proporcionar una interfaz de usuario. Se utiliza para operaciones como reproducir música, manejar descargas de red o interactuar con un servidor.

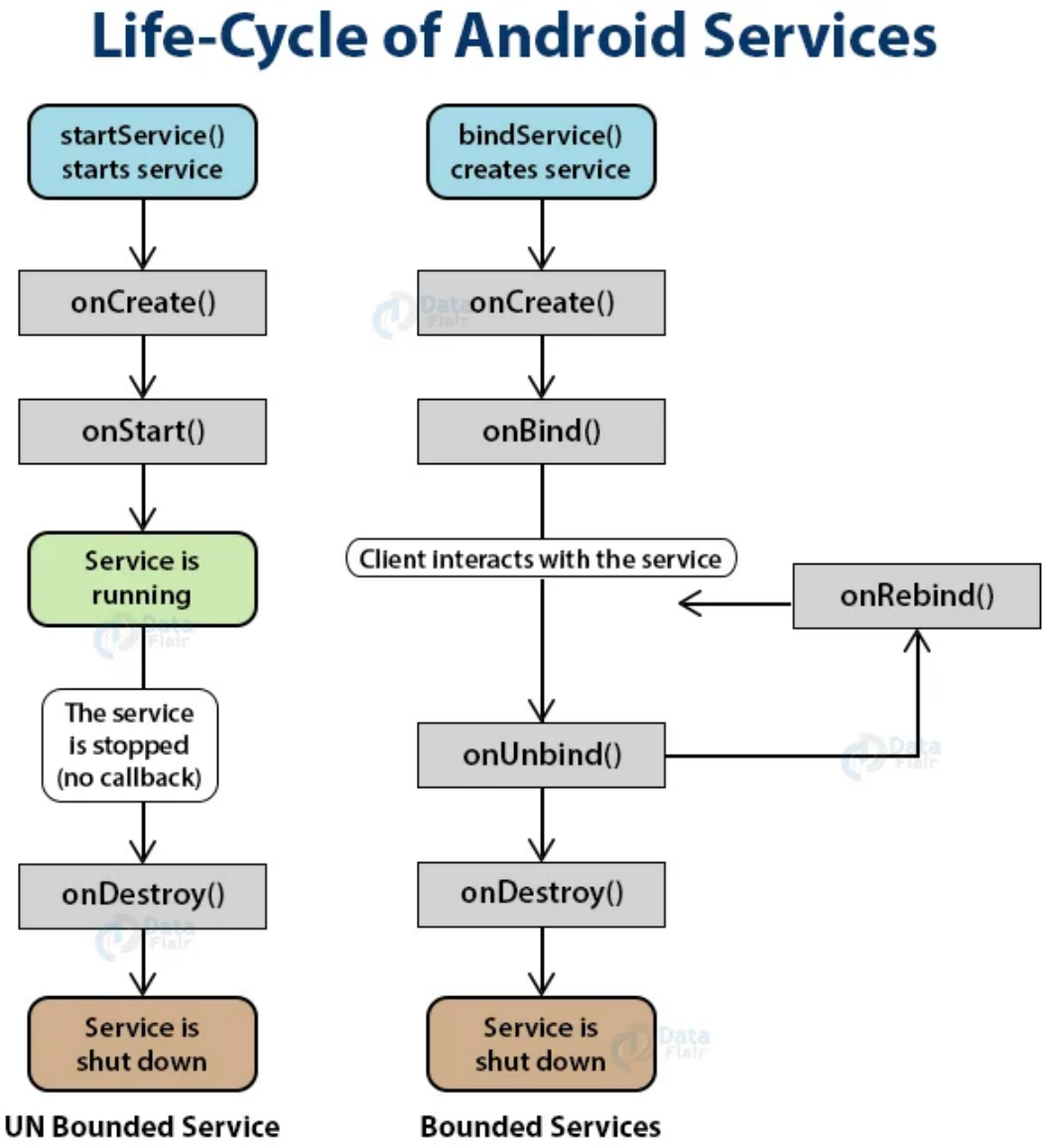
En Android suele hablarse de: * Servicios en Primer Plano, ejemplo aplicación de audio * Servicios en segundo Plano, ejemplo para comprimir archivos

Y se clasifican en dos tipos: * Seervicios Iniciados (Started Service) y * Servicios enlazados (bindingService)

Tanto los Started como los Bound Service se pueden ejecutar en primer o segundo plano.

101.0.1 Ciclo de vida

Un servicio se crea derivando de la clase Service o algunas de sus subclases.



Algunos de los estados/callback de estos dos servicios.

- **onStartCommand()**: Método donde se inicia la tarea en un servicio iniciado en otro componenete (como podria ser una Activity) llamando a `startService()` y el servicio se ejecuta en segundo plano hasta que se detiene con stopSelf() o desde el componente que lo inicio con stopService()
- **onBind()**: Método para manejar las conexiones en un servicio enlazado. El sistema lo inicia llamando a bindService() cuando otro componente crea un IPC para obtener el servicio. En tu implementación de este método, debes proporcionar una interfaz que los clientes utilicen para comunicarse con el servicio devolviendo una IBinder. Siempre debes implementar este método, pero debes devolver NULL si no deseas permitir los enlaces.
- **onCreate()** : El sistema invoca a este método para realizar procedimientos únicos de configuración cuando el servicio se crea por primera vez (antes de llamar a onStartCommand() o onBind()). Si el servicio ya se está ejecutando, no se llama a este método.
- **onDestroy()**: Método donde se limpian los recursos cuando el servicio ya no es necesario.

Los servicios deben detenerse explícitamente llamando a **stopSelf()** o **stopService()**.

101.0.2 Declaración del servicio en el Manifest

Utilizando la etiqueta `<service />`

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

101.0.3 Started Service.

Se inician llamando al método `startService()`. Una vez iniciados, pueden continuar ejecutándose indefinidamente, incluso si la actividad que los inició se destruye.

Son ideales para tareas que no están directamente relacionadas con la interacción del usuario, como descargar un archivo.

Un componente de una aplicación, como una actividad, puede iniciar el servicio llamando a `startService()` y pasando una Intent que especifique el servicio y que adem´sa incluya los datos. El servicio recibe este Intent en el método `onStartCommand()`.

Por ejemplo, imagina que una actividad necesita guardar datos en una base de datos en línea. La actividad puede iniciar un servicio complementario y enviar los datos que se guardarán pasando un intent a `startService()`. El servicio recibe el intent en `onStartCommand()`, se conecta a Internet y realiza la transacción de la base de datos. Cuando se completa la transacción, el servicio se detiene a sí mismo y se destruye.

La clase Service es la clase base para todos los servicios. Cuando extiendes esta clase, es importante que **crees un subproceso nuevo**, en el que el servicio pueda completar todo su trabajo. El servicio usa el subproceso principal de tu aplicación de forma predeterminada, lo que puede ralentizar el rendimiento de cualquier actividad que tu aplicación esté ejecutando.

El framework de Android también proporciona la subclase **IntentService de Service**, que usa un subproceso de trabajo para controlar todas las solicitudes de inicio, una a la vez. **No se recomienda** usar esta clase para apps nuevas, ya que no funcionará bien a partir de Android 8 Oreo debido a la introducción de los límites de ejecución en segundo plano. Además, dejará de estar disponible a partir de Android 11. Puedes usar JobIntentService para reemplazar IntentService, que es compatible con las versiones más recientes de Android.

En las siguientes secciones, se describe cómo implementar tu propio servicio personalizado. Sin embargo, debes considerar usar `WorkManager` . Consulta la guía para el procesamiento en segundo plano en Android para ver si existe una solución que se adapte a tus necesidades.

101.0.3.1 Crear un servicio iniciado (Started Service)

Se debe extender la clase `Service`:

```
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()
    }
}
```

101.0.3.2 Iniciar el servicio

Desde una Activity (no exclusivamente) se puede iniciar enviado un Intent que será procesado por `startService()

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

101.0.3.3 Detener el Servicio

[TODO]

101.0.4 Bound Service.

Un Service que permite el enlace a componentes (IPC) (como actividades)y se conecta llamando a `bindService()`. Este tipo de servicio ofrece una interfaz cliente-servidor que permite la comunicación entre el `Service` y los componentes.

Es útil para tareas que deben interactuar con la aplicación, como reproducir música.

Un servicio vinculado es una implementación de la clase `Service` que permite que otras aplicaciones se vinculen e interactúen con él. Para vincular un servicio, implementa el método de devolución de llamada (callback) `onBind()`. Este método muestra un objeto `IBinder` que define la interfaz de programación que los clientes pueden usar para interactuar con el servicio.

Si permites que tu servicio se inicie y se vincule, cuando se inicie, el sistema no lo destruirá cuando se hayan desvinculado todos los clientes. En su lugar, debes detener explícitamente el servicio llamando a `stopSelf()` o `stopService()`.

Un cliente se vincula a un servicio llamando a `bindService()`. Cuando lo hace, debe implementar `ServiceConnection`, que supervisa la conexión con el servicio. El valor que se muestra de bindService() indica si el servicio solicitado existe y si se le permite al cliente acceder a él.

Puedes garantizar que tu servicio esté disponible solo para tu aplicación incluyendo el atributo **android:exported **y configurándolo en false. Esto impide que otras aplicaciones inicien tu servicio, incluso cuando se utiliza una intent explícita.

101.0.4.1 Vincualar un servicio iniciado

Se puede iniciar un servicio llamando a startService(), lo que permite que el servicio se ejecute de forma indefinida. También puedes llamar a bindService() para permitir que un cliente se vincule con el servicio.

Un cliente se vincula a un servicio llamando a bindService(). Cuando lo hace, debe implementar ServiceConnection, que supervisa la conexión con el servicio. El valor que se muestra de bindService() indica si el servicio solicitado existe y si se le permite al cliente acceder a él.

Cuando el sistema Android crea la conexión entre el cliente y el servicio, llama a onServiceConnected() en ServiceConnection. El método onServiceConnected() incluye un argumento IBinder que el cliente usa para comunicarse con el servicio vinculado.

Puedes conectar varios clientes a un servicio simultáneamente. Sin embargo, el sistema almacena en caché el canal de comunicación del servicio IBinder. En otras palabras, el sistema llama al método onBind() del servicio para generar el IBinder solo cuando se vincula el primer cliente. El sistema entonces entrega el mismo IBinder a todos los clientes adicionales que se vinculan a ese mismo servicio, sin volver a llamar a onBind().

Cuando se desvincula el último cliente del servicio, el sistema destruye el servicio, a menos que este se haya iniciado con startService().

La parte más importante de la implementación de tu servicio vinculado es definir la interfaz que mostrará tu método de devolución de llamada onBind(). En la siguiente sección, se analizan varias formas en las que puedes definir la interfaz IBinder de tu servicio.

101.0.4.2 Cómo crear un servicio vinculado (Bound Service)

Al crear un servicio que proporciona la capacidad de crear una vinculación, debes proporcionar un `IBinder` que contenga la interfaz de programación que los clientes pueden usar para interactuar con el servicio. Puedes definir la interfaz de las siguientes tres maneras:

1. Extender la clase IBinder.

Si tu servicio es privado para tu propia aplicación y se ejecuta en el mismo proceso que el cliente, que es lo común, crea tu interfaz extendiendo la clase Binder y mostrando una instancia de ella desde onBind(). El cliente recibe el Binder y puede usarlo para acceder directamente a métodos públicos disponibles en la implementación de Binder o Service. Esta es la técnica preferida cuando tu servicio es solo un trabajador en segundo plano de tu propia aplicación. El único caso de uso en el que esta no es la forma preferida de crear tu interfaz es si otras aplicaciones usan tu servicio o se usa entre procesos separados.

1. Usar Messenger

Si necesitas que tu interfaz funcione en diferentes procesos, puedes crear una para el servicio con un Messenger. De esta manera, el servicio define un Handler que responde a diferentes tipos de objetos Message.
Este Handler es la base para un Messenger que luego pueda compartir un IBinder con el cliente y permitir que el cliente envíe comandos al servicio mediante objetos Message. Además, el cliente puede definir un Messenger propio para que el servicio pueda devolver mensajes.

Esta es la manera más sencilla de establecer comunicación entre procesos (IPC), ya que Messenger pone en cola todas las solicitudes en un solo subproceso a fin de que no debas diseñar tu servicio de modo que sea seguro para subprocesos.

101.0.4.3 Extender la clase IBinder

Pasos a seguir:

1. En tu servicio, crea una instancia de Binder que realice una de las siguientes acciones:
2. Que contenga métodos públicos que el cliente pueda llamar
3. Que muestre la instancia Service actual, que tiene métodos públicos que el cliente puede llamar
4. Que muestre una instancia de otra clase alojada por el servicio con métodos públicos que el cliente pueda llamar
5. Exponer esta instancia Binder desde el método de devolución de llamada onBind()
6. En el cliente, recibir `Binder` del método de devolución de llamada `onServiceConnected()` y realice llamadas al servicio vinculado usando los métodos proporcionados

```
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
    }
}
```

LocalBinder proporciona el método getService() para que los clientes obtengan la instancia actual de LocalService. Esto permite a los clientes llamar a métodos públicos en el servicio. Por ejemplo, los clientes pueden llamar a getRandomNumber() desde el servicio.

A continuación, se proporciona una actividad que se vincula a LocalService y llama a getRandomNumber() cuando se hace clic en un botón:

```
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()
        }
    }
}
```

101.0.5 Intent Service.

Cabe destacar que, a partir de Android 11 (API nivel 30), IntentService está marcado como **obsoleto** y se recomienda utilizar `JobIntentService` o `WorkManager` para tareas en segundo plano.

101.0.6 Servicios en Primer Plano.

Para tareas importantes o visibles para el usuario (como la reproducción de música), es recomendable utilizar un **Foreground Service**, que muestra una notificación persistente.

Los servicios en primer plano (Foreground Service) en Android son una forma especial de servicios que notifican al usuario sobre su operación activa, generalmente a través de una notificación persistente. Son ideales para tareas que el usuario percibe como en ejecución, como la reproducción de música, la grabación de audio o la obtención de la ubicación en tiempo real

Ejemplo:
* Paso 1 Creación del servicio

```
class MyForegroundService : Service() {

    override fun onStartCommand(intent: Intent, flags: Int, startId: Int): Int {
        // Código para iniciar el servicio en primer plano
        startForegroundService()
        return START_STICKY
    }

    private fun startForegroundService() {
        val notification = createNotification()
        startForeground(NOTIFICATION_ID, notification)
    }

    override fun onBind(intent: Intent): IBinder? {
        return null
    }

    // Método para crear la notificación
    private fun createNotification(): Notification {
        // Código para crear la notificación
```

```
        // ...
    }
}
```

- Paso 2 Crear la notificación

```
private fun createNotification(): Notification {
    val notificationChannelId = "MY_CHANNEL_ID"

    // Crear el canal de notificación en Android Oreo y superior
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        val name = "My Channel"
        val importance = NotificationManager.IMPORTANCE_DEFAULT
        val channel = NotificationChannel(notificationChannelId, name, importance)
        val notificationManager: NotificationManager =
getSystemService(NOTIFICATION_SERVICE) as NotificationManager
        notificationManager.createNotificationChannel(channel)
    }

    val builder = NotificationCompat.Builder(this, notificationChannelId)
    return builder
        .setContentTitle("Servicio en Primer Plano")
        .setContentText("Este servicio está corriendo en primer plano.")
        // Icono y otras configuraciones de la notificación
        .setSmallIcon(R.drawable.ic_notification)
        .build()
}
```

- Paso 3 Iniciar el servicio

```
val serviceIntent = Intent(this, MyForegroundService::class.java)
startService(serviceIntent)
```

101.1 Sevicios incluidos en el Sistema Android

1. **LocationManager**: Para acceder a la funcionalidad de localización.

   ```
   val locationManager = getSystemService(Context.LOCATION_SERVICE) as LocationManager
   ```

2. **NotificationManager**: Para mostrar notificaciones a los usuarios.

   ```
   val notificationManager = getSystemService(Context.NOTIFICATION_SERVICE) as
   NotificationManager
   ```

3. **ConnectivityManager**: Para manejar operaciones relacionadas con la conectividad de red.

```
val connectivityManager = getSystemService(Context.CONNECTIVITY_SERVICE) as
ConnectivityManager
```

1. **AlarmManager**: Para programar acciones para que se ejecuten en un momento futuro.

```
val alarmManager = getSystemService(Context.ALARM_SERVICE) as AlarmManager
```

1. **AudioManager**: Para controlar los volúmenes de audio y los modos de sonido del dispositivo.

```
val audioManager = getSystemService(Context.AUDIO_SERVICE) as AudioManager
```

1. **SensorManager**: Para acceder a los sensores del dispositivo.

```
val sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
```

1. **PowerManager**: Para interactuar con la administración de energía del dispositivo, incluyendo wakelocks.

```
val powerManager = getSystemService(Context.POWER_SERVICE) as PowerManager
```

1. **WindowManager**: Para acceder a elementos relacionados con la ventana, como el tamaño de la pantalla.

```
val windowManager = getSystemService(Context.WINDOW_SERVICE) as WindowManager
```

1. **TelephonyManager**: Para acceder a información relacionada con la telefonía.

```
val telephonyManager = getSystemService(Context.TELEPHONY_SERVICE) as TelephonyManager
```

101.2 Ejemplos

101.2.1 POWER_SERVICE

El servicio POWER_SERVICE en Android es utilizado para interactuar con diferentes aspectos de la administración de energía del dispositivo, como adquirir "wake locks". Un "wake lock" permite a tu aplicación mantener el dispositivo *despierto* (es decir, evitar que el dispositivo entre en modo de suspensión) durante una operación específica.

```
import android.content.Context
import android.os.PowerManager

class MyActivity : AppCompatActivity() {

    private lateinit var wakeLock: PowerManager.WakeLock

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Obtén el PowerManager del sistema
        val powerManager = getSystemService(Context.POWER_SERVICE) as PowerManager

        // Crea un WakeLock (asegúrate de liberarlo cuando ya no sea necesario)
        wakeLock = powerManager.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
"MyApp:MyWakelockTag")
    }

    fun someMethodRequiringWakelock() {
        // Adquiere el wake lock
        wakeLock.acquire()

        // Realiza la operación que requiere que el dispositivo permanezca despierto
```

```
        // ...

        // Libera el wake lock cuando hayas terminado
        if (wakeLock.isHeld) {
            wakeLock.release()
        }
    }
}
```

Importante dar los permisos adecuados:

```
<uses-permission android:name="android.permission.WAKE_LOCK" />
```

En el ejemplo, he utilizado `PARTIAL_WAKE_LOCK`, que mantiene el CPU funcionando pero permite que la pantalla y el teclado se apaguen. Existen otros tipos de wake locks para diferentes propósitos (como `SCREEN_DIM_WAKE_LOCK` para mantener la pantalla encendida pero atenuada). Escoge el tipo que mejor se ajuste a tus necesidades.

101.3 Apendice

Terminos: * AD : Android Developer * IPC = Interprocess communitacion * Started Service : Servicio Iniciado * Binding Service : Servicio Enlazado

Enlaces:
* Descripción general de Servicios * Ejemplo mediaplayerç * Ejemplo foreground service

Version 0.5 12-12-23
Versión 0.8 7-1-24