Apuntes

Descargar estos apuntes

Tema 10. Corrutinas y Servicios

Índice

Corrutinas Kotlin

Una corrutina de Koytlin es un conjunto de sentencias que realizan una tarea específica, con la capacidad de suspender o resumir su ejecución sin bloquear un hilo. Esto permite que tengas diferentes corrutinas cooperando entre ellas y no significa que exista un hilo por cada corrutina, al contrario, puedes ejecutar varias en un solo.

Las corrutinas son parte del paquete kotlinx.coroutines, por lo que necesitas especificar la dependencia correspondiente en en `build.gradle`. A día de hoy:

```
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.5.1"
```

Las corrutinas en el desarrollo de Android, aunque son un elemento que va a permitirnos crear códigos más sencillos, al principio pueden parecer complejas debido a su nivel de abstracción.

Alcance de las corrutinas

Cuando creamos coroutines asumimos de manera implícita dos aspectos muy importantes: El ámbito (CoroutineScope y GlobalScope) y el contexto (CoroutineContext).

El uso de `GlobalScope` permite crea corrutinas de nivel superior, esto quiere decir que tienen la capacidad de vivir hasta que termine la aplicación y es trabajo del desarrollador el control para su cancelación, por lo que no se aconseja usar este ámbito.

```
private fun ejemploGlobalScope() {
          GlobalScope.launch(Dispatchers.Main) {
            launch(Dispatchers.IO) {
               delay(3000)
            }
            Toast.makeText(requireActivity(), "GlobalScope",
            Toast.LENGTH_SHORT).show()
        }
    }
```

> 📌 con `launch` lanzamos la corrutina en el contexto de la Main para poder mostrar el Toast y dentro de esta lanzamos otra para procesos largos (los tres segundos de retardo).

Para reducir este alcance, Kotlin nos permite crear los espacios donde queremos que se dé la concurrencia. Esto lo haremos mediante `CoroutineScope`.

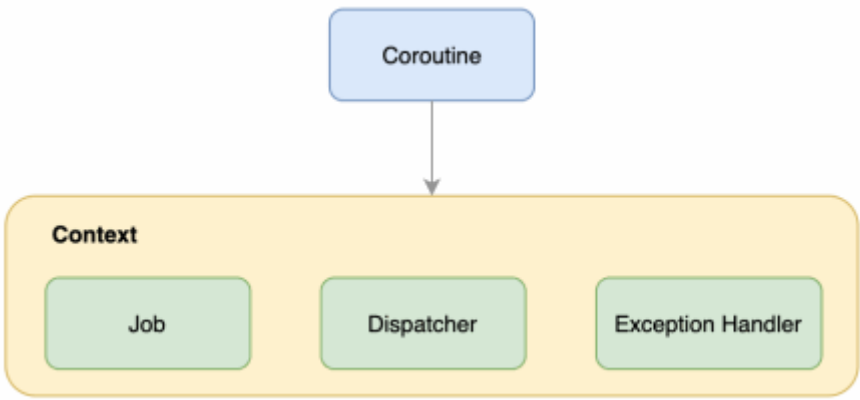
```
var coroutineScope= CoroutineScope(Dispatchers.Main)
private fun ejemploCoroutineScopeConMain()
{
   coroutineScope?.launch {
    while (isActive) {
        delay(3000)
        Toast.makeText(requireActivity(), "[CoroutineScope-Main] I'm alive o
        "${Thread.currentThread().name}!", Toast.LENGTH_SHORT).show()
    }
  }
}
//Al salir del fragment se cancela la corutina
override fun onDetach() {
    super.onDetach()
    coroutineScope?.cancel()
}
```

> 📌esta corrutina permanece activa hasta que sea cancelada de alguna forma. Está lanzada con `launch` y en el alcance de la Main, por lo que podremos usar el Toast para mostrar la información. El alcance personalizado lo hemos conseguido con el constructor de `CoroutineScope()`

Cada vez que usamos un constructor de coroutines en realidad estamos haciendo una llamada a una función que recibe como primer parámetro un objeto de tipo `CoroutineContext`.

CoroutineContext

Las coroutines siempre se ejecutan en algún contexto que está representado por un valor del tipo CoroutineContext.



El contexto de Coroutine es un conjunto de varios elementos. Los elementos principales son el

Job de la Coroutine, su dispatcher y también su Exception handler.

- **Job** De acuerdo con la documentación oficial "Un Job es una cosa cancelable con un ciclo de vida que culmina con su finalización. Los trabajos de coroutine se crean con `launch coroutine builder`. Ejecuta un bloque de código especificado y se completa a la finalización de este bloque".
  La ejecución de un trabajo no produce un valor de resultado. Deberíamos utilizar una interfaz `Deferred` para un trabajo que produzca un resultado. Un trabajo con resultado (Deferred), se crea con el `async coroutine builder` y el resultado se puede recuperar con el método `await()`, que lanza una excepción si el Deferred ha fallado.
  Los jobs tienen un par de funciones interesantes que pueden ser muy útiles.
  Pero es importante entender que un Job puede tener a su vez otro Job padre.
  Ese job padre tiene cierto control sobre los hijos, y ahí es donde entran en juego estas funciones:
  - job.join -> Con está función, se puede bloquear la corrutina asociada con el job hasta que todos los jobs hijos hayan finalizado. Todas las funciones de suspension que se llaman dentro de una corrutina están vinculadas a job, así que el job puede detectar cuándo finalizan todos los jobs hijos y después continuar la ejecución. job.join() es una función de suspensión en sí misma, por lo que debe llamarse dentro de otra corrutina.
  - job.cancel -> Esta función cancelará todos sus jobs hijos asociados. job.cancel() esta es una función normal, por lo que no requiere una corrutina para ser llamada.

  > ✋ Con GlobalScope el padre no va a esperar la finalización de sus hijos y una vez que el padre es cancelado, los otros trabajos van a seguir corriendo aparte. Es decir, en este caso es responsabilidad del desarrollador llevar el control del tiempo de vida de las coroutines porque no hay sincronización con los trabajos hijos.

- **Dispacher** En Kotlin, todas las Coroutines deben ejecutarse en un dispatcher incluso cuando se ejecutan en el hilo principal. Los Dispatchers son un tipo de contextos de corrutina que especifican el hilo o hilos que pueden ser utilizados por la corrutina para ejecutar su código. Hay dispatchers que solo usan un hilo (como Main) y otros que definen un grupo de hilos que se optimizarán para ejecutar todas las corrutinas que reciben. Para especificar dónde deben ejecutarse las coroutines, Kotlin proporciona tres Dispatchers que puedes utilizar:

- `Dispatchers.Main`. Hilo principal en Android, interactúa con la UI y realiza trabajos ligeros como: llamar a funciones de suspensión, llamar a funciones de la interfaz de usuario y actualizar LiveData
- `Dispatchers.IO`. En general, todas las tareas que bloquearán el hilo mientras esperan la respuesta. Optimizado para la E/S de disco y red fuera del hilo principal: solicitudes de bases de datos, lectura/escritura de archivos,sensores, conexión en red, ...
- `Dispatchers.Default`. Optimizado para el trabajo intensivo de la CPU fuera del hilo principal: ordenar una lista, análisis de JSON, utilidades

Dispatcher	Description	Uses
Dispatchers.Main	Main thread on Android	- Calling suspend functions - Call UI functions - Update LiveData
Dispatchers.IO	Disk and network IO*	- Database - File IO - Networking
Dispatchers.Default	CPU intensive work	- Sorting a list / other algorithms - Parsing JSON - DiffUtils

- **Exception Handler**. Este elemento es opcional del CoroutineContext, y nos permite manejar excepciones no capturadas. Y de esta forma mejorar la expericencia de usuario

Funciones de Suspensión

Las corrutinas se basan en las **funciones de suspension** que son funciones normales a las que se les agrega el modificador `suspend`, las funciones de suspensión tienen la capacidad de bloquear la ejecución de la corrutina mientras están haciendo su trabajo. Una vez que termina, el resultado de la operación se devuelve y se puede utilizar en la siguiente línea.
De esta manera se agregan dos nuevas operaciones (además de invocar / llamar y regresar):

- suspender - pausa la ejecución de la corrutina actual, guardando todas las variables locales. El hilo actual puede continuar con su trabajo, mientras que el código de suspensión se ejecuta en otro hilo.
- reanudar : continúa una corrutina suspendida desde el lugar donde se pausó cuando el resultado está listo.

Las funciones de suspensión solo pueden llamarse o ejecutarse dentro de una corrutina o dentro de otra función de suspensión.

Algunas de las funciones usadas con frecuencias para lanzar corrutinas son funciones de suspensión en si mismas, tales como `delay, join, await, withContext, supervisorScope, etc.`.

withContext

Esta es una función de suspensión que permite cambiar fácilmente el contexto que se utilizará para ejecutar una parte del código dentro de una corrutina. `WithContext` es una función de suspensión en sí misma.

Builders de corrutinas

Los constructores de coroutinas sirven para iniciar las corrutinas. Tenemos diferentes builders dependiendo de lo que queramos hacer, e incluso técnicamente se pueden crear personalizados. Pero para la mayoría de los casos son suficientes con los que proporciona la librería:

- `runBlocking`, este builder bloquea el hilo actual hasta que se terminen todas las tareas dentro de esa corrutina. No suele ser una funcionalidad muy usada de las corrutinas, ya que precisamente lo que se pretende con estas es la sincronización de más de una tarea.

```
    fun ejemploRunBlocking()
    {
            runBlocking (Dispatchers.IO){
                //Lanzamos tres corrutinas con launch
                for (i in 1..3) {
                    launch(Dispatchers.Default) {
                        //delay es una funcion de suspensión por lo que al
                        //lanzar las tres corrutinas el tiempo no duran 5seg
                        //ya que se hacen las tres intercaladas
                        //a diferencia de sleep
                        Log.d("CORRUTINA","${Thread.currentThread().name}")
                        delay(5000)
                    }
                }
            }
        Toast.makeText(requireActivity(),"Ya han terminado las tareas lanzad
        "con el constructor runBlocking." +
        "\nSe desbloquea el hilo principal y se muestra este texto",
        Toast.LENGTH_LONG).show()
    }
```

> 📌en este ejemplo lanzamos una corrutina que a su vez lanza otras tres corrutinas, la primera con `runBlocking` bloqueará el hilo principal, por lo que el `Toast` no ocurrirá hasta que no terminen todas las subrutinas hijas de

> `runBlocking`. `launch` se lanza, en este ejemplo, en un contexto que no es el Main, por lo que nunca podremos introducir ninguna interacción con las vistas (por eso la salida la realizamos mediante log). Cada una de las tres corrutinas lanzadas por launch permanecerá 5 segundos de espera, pero al realizarse al mismo tiempo, el retardo es de poco más de 5 segundo. Si el bucle no lanzara las tres subrutinas con launch y solo dejaramos el `delay` el retardo sería de unos 15 segundos aproximadamente.

- `launch`, este es el builder más usado. A diferencia de runBlocking, no bloqueará el subproceso actual (si se usan los dispatchers adecuados).Launch devuelve un Job, que permitirá realizar las funciones explicadas anteriormente.
  Cuando **launch** se usa sin parámetros, hereda el contexto (y por lo tanto el dispatcher) del CoroutineScope desde el que se inicia. Puede iniciarse dentro de otra corrutina omitiendose o no el Dispachers, pero si no está dentro de una corrutina necesitará obligatoriamente especificar el alcance. Launch nos devuelve un Job, que permitirá realizar las funciones explicadas anteriormente.

```
...
//launch especificando el alcance mediante CoroutineScope
CoroutineScope(Dispatchers.Main).launch()
{
//Al estar dentro de una corrutina padre, se puede omitir
//el alcance e incluso el Dispatchers si no queremos cambiar
//el contexto del padre, en este caso si lo cambiamos
launch(Dispatchers.IO){  ...    }
}
...
```

- `async`, este otro builder permite ejecutar varias tareas en segundo plano en paralelo. No es una función de suspensión en sí misma, por lo que cuando ejecutamos async, el proceso en segundo plano se inicia, pero la siguiente línea se ejecuta de inmediato. `async` siempre debe llamarse dentro de otra corrutina, y devuelve un job especializado que se llama `Deferred`.
  `Deferred` tiene una nueva función de suspensión llamada `await()` que es la que bloquea. Llamaremos a await() solo cuando necesitemos el resultado. Si el resultado aún no esta listo, la corrutina se suspende en ese punto. Si ya tenemos el resultado, simplemente lo devolverá y continuará.

```
private fun ejemploAsyncAway()
{
    var cadena:String?=null
    CoroutineScope(Dispatchers.Main).launch {
        val job=async {
            for (i in 1..5) {
                tiempo.text =i.toString()
                withContext(Dispatchers.IO) { delay(1000)}
            }
            "Ha terminado la corrutina async"
        }
        cadena=job.await()
        Toast.makeText(requireActivity(),cadena,Toast.LENGTH_SHORT).show()
    }
}
```

> 📌 lanzamos el `async` dentro de una corrutina launch con contesto Main, para poder pintar cada segundo el texto que nos interese (en este caso un contador de 1 a 5). Cambiamos el Dispacher a IO para realizar el retardo y devolvemos una cadena como respuesta del `async` . Cuando termine el trabajo la corrutina, recuperaremos la cadena con `await` y se lanzará el Toast que la utiliza.

🖊 Copia los códigos de los ejemplos y completalos hasta hacerlos funcionar. Para ayudarte en la comprensión puedes analizar el ejercicio resuelto de la Barra de Progreso

ViewModel y Corrutinas

ViewModel es extendido con una propiedad `viewModelScope` que se encarga de cancelar las corrutinas cuando ya no son necesarias.

```
//propiedad viewModelScope extendida de CoroutineScope y
//que a su vez extiende el ViewModel
val ViewModel.viewModelScope: CoroutineScope
   get()
   set()
```

Gracias al comportamiento de `CoroutineScope` a través de la propiedad `viewModelScope` se realiza un seguimiento de todas las corrutinas que se crean en este ambito. Por lo tanto, si se cancela un alcance, se cancelan todas las corrutinas creadas en él.

Por ejemplo, en el siguiente código creamos un ViewModel que nos gestionará un `MutableLiveData` de tipo ArrayList de Strings y que tiene un método que simula una descarga de datos que se guardarán en el objeto:

```
class ItemViewModel : ViewModel() {
    private val valores =
        arrayOf("item1", "item2", "item3", "item4", "item5", "item6", "item7", "
    private var liveData =
        MutableLiveData<ArrayList<String>>() //:MutableLiveData<String> by lazy
    val datos: LiveData<ArrayList<String>> get() = liveData

    fun descargarDatos() {
        val random = Random()
        var aux = ArrayList<String>()
        //Simulación de una descarga lenta de datos
        val numeroElementos = random.nextInt(10)
        viewModelScope.launch {
            for (i in 0..numeroElementos) {
                aux.add(valores[random.nextInt(valores.size - 1)])
                delay(1000)
            }
            liveData.value = aux
        }
    }
}
```

> 📌 en el método **descargarDatos** se usa la propiedad `viewModelScope` para lanzar la corrutina que producirá el retardo y la lista generada con elementos aleatorios.

El uso del ViewModel desde las distintas partes de la aplicación, es igual al que hemos visto en temas anteriores.

Realiza el ejercicio propuesto del cronómetro

Servicios y tareas de larga duración

Las corrutinas están diseñadas para trabajar tareas que deben finalizar cuando el usuario sale de un determinado alcance o termina una interacción, pero a la hora de realizar lo que llamamos **long-time running tasks**, es decir tareas que tienen que estar durante un gran tiempo en segundo plano y no tienen porqué estar ligadas a la actividad que las ha lanzado o a la interacción con el usuario, es necesario pensar en otras respuestas:

1. Tareas que no necesitan ejecutarse de inmediato y procesarse de forma continua, incluso cuando el usuario coloca la aplicación en segundo plano o se reinicia el dispositivo. En este caso lo recomendable es el uso de `WorkManager`.
2. En casos concretos en los que se realizan tareas largas como la reproducción de contenido multimedia o la navegación activa, se recomienda el uso de `Servicios` pero en primer plano.

Servicios

Los Servicios son componentes que se ejecutan en background (o segundo plano) y que que no proporciona una interfaz de usuario, por lo que no interactúan con el usuario.
La plataforma Android ofrece una gran cantidad de servicios predefinidos, disponibles regularmente a través de la clase Manager. De esta manera, en nuestras actividades podremos acceder a estos servicios a través del método `getSystemService()`.
Si el servicio creado va a realizar tareas que tengan un coste computacional elevado, se pueden provocar problemas en la aplicación (Application Not Responding o ANR), para evitar situaciones como estas, el servicio puede ejecutarse en un hilo secundario. Si la aplicación está orientada al nivel de API 26 o un nivel superior, el sistema impone restricciones en la ejecución de servicios en segundo plano cuando la aplicación misma no se encuentra en primer plano. En estos casos es mejor usar una tarea programada, `WorkManager`.

> ⚠ Por otro lado, si necesitamos utilizar servicios propios, estos deben ir declarados en el archivo `AndroidManifest.xml`.

Tenemos varias opciones para que nuestra activity se comunique con un servicio.

- Usar **intent**: es un escenario simple donde no se requiere comunicación directa ni notificaciones de procesos. El servicio recibe los datos vía intent y realiza la tarea que sea necesaria sin devolver ningún tipo de dato. Un escenario donde podemos utilizar esto es cuando necesitamos actualizar un content provider. El servicio recibe los datos vía intent, actualiza el content provider y este notifica a nuestra app que el contenido está actualizado, sin que se requiera alguna acción extra por nuestro servicio.
- Usar **receivers**: otro método es emitir eventos y registrar receptores (receivers, clase `BroadcastReceiver`) para establecer la comunicación. Por ejemplo, una activity registra dinámicamente un receiver para un evento, y el servicio es el encargado de emitir ese evento. Este escenario se usa cuando es necesario que el escenario notifique que terminó de hacer alguna tarea.

Construyendo un Servicio

Para crear un Servicio tendremos que extender de `Service` por lo que tendremos que redefinir y conocer 4 métodos principales:

- `onCreate`: De manera similar a un Activity, se ejecuta la primera vez que se crea el servicio y sirve para inicializar variables.
- `onStartCommand`: Se ejecuta cuando el servicio recibe un intent lanzado mediante el comando startService. Si el servicio ya está iniciado, los intents sucesivos no volverán a generar un `onCreate` si no que pasaran directamente aquí. Es importante tener en cuenta que si se crean hilos en este método, debe comprobarse que el hilo no esté ya funcionando o podrían accidentalmente crearse hilos nuevos cada vez que se pasa por aquí.
- `onBind`: Este comando es llamado cuando se ejecuta un bindService() desde un Activity. Sirve para devolver una referencia a modo de IBinder al Activity de la instancia del servicio.
- `onDestroy`: Cuando se llama al comando `stopService()` o dentro del mismo service al método `stopSelft()` se pasa por este método, dónde deben terminarse hilos o tareas que puedan estar ejecutándose.

Ejemplo de un servicio sencillo, podemos analizar su funcionamiento visualizando las lineas de LogCat:

```
class MyService: Service() {
    override fun onCreate() {
        super.onCreate()
        Log.d("DATO","Creando servicio....")
    }
    override fun onStartCommand
                (intent: Intent?, flags: Int, startId: Int): Int {
        super.onStartCommand(intent, flags, startId)
        Log.d("DATO","Recibiendo Intent ....")
        while(true)  Log.d("DATO","Mostrando intent ....+" +
                            "${intent?.getStringExtra("CADENA")}")
        return START_STICKY
    }
    override fun onBind(p0: Intent?): IBinder? {
        Log.d("DATO","Enlazando Intent ....")
        return null
    }
    override fun onDestroy() {
        Log.d("DATO","Destruyendo Intent ....")
        super.onDestroy()
    }
}
```

> 📌 Hay que destacar que este tipo de servicios hay que pararlos explícitamente, aunque el sistema también podría pararlos si se encontrase escaso de memoria. Si en el método onStartCommand se ha devuelto **`START_STICKY`** , el sistema podrá revivir el servicio cuando vuelva a tener memoria disponible, aunque hay que tener en cuenta que entonces el intent que recibirá onStartCommand será nulo (null).

Una manera de inicializar este servicio, es desde una actividad y de forma muy parecida a iniciar una activity:

```
class MainActivity : AppCompatActivity() {
    lateinit var intento:Intent
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        findViewById<MaterialButton>(R.id.boton).setOnClickListener{
            intento=Intent(applicationContext,MyService::class.java)
            intento.putExtra("CADENA","Dato Pasado desde Main")
            startService(intento)
        }
    }
    override fun onDestroy() {
        super.onDestroy()
        stopService(intento)
    }
}
```

> ⚠ No olvidar que se debe declarar los servicios usados en el manifest.
>
> **`<service android:name=".MyService"/>`**

Como ya se ha indicado, todo este código funcionaría en primer plano, si deseamos crear un hilo para realizar tareas en segundo plano, habría que hacerlo en onStartCommand:

```
class MyServiceSegundoPlano: Service() {
     var hilo=Thread()
    override fun onStartCommand
                (intent: Intent?, flags: Int, startId: Int): Int {
        super.onStartCommand(intent, flags, startId)
        val cadena=intent?.getStringExtra("CADENA")
        Log.d("DATO","Recibiendo Intent ....")
        if(hilo==null || !hilo?.isAlive) {
            hilo = Thread {
                while (true) Log.d(
                    "DATO", "Mostrando intent ...." +
                            "${cadena}"
                )
            }
            hilo.start()
        }

        return START_STICKY
    }
    override fun onBind(p0: Intent?): IBinder? {
        return null
    }
    override fun onDestroy() {
        Log.d("DATO","Destruyendo Intent ....")
        super.onDestroy()
    }
}
```

Prueba los dos ejemplos anteriores y mira el resultado en el Logcat

Existen dos tipos de servicios. Aquel que se inicia explícitamente a través del método `startService()` llamado **Started Service** y el que se liga a un componente con `bindService()` llamado **Bound Service**.

- Started: Una vez iniciado el servicio, se ejecuta de forma indefinida en segundo plano, incluso si se destruye el componente que lo inició.
- Bound: En este caso, el servicio ofrece una interfaz de tipo cliente-servidor a aquellos componentes que se asocian o enlazan a dicho servicio. Los componentes asociados o enlazados (bound) al servicio pueden enviar peticiones, conseguir resultados y respuestas o, incluso, llevar a cabo tareas de comunicación con otros procesos. En este caso, el servicio sólo se ejecuta cuando hay algún componente asociado o enlazado a él.

Aplicación ejemplo para crear un servicio para poder consumirlo desde una actividad

Vamos a crear un servicio para que desde una actividad podamos consumirlo. El objetivo del servicio será recuperar datos de forma periódica para que de esta forma la actividad pueda desplegarlos a través de una ListView, siempre pudiendo pedir los datos más actualizados de este servicio.
Vamos a enlazar el servicio enlazándolo con la actividad (Bound) Teniendo el concepto claro, pasemos a codificarlo:

1. Creamos un nuevo proyecto llamado MiServicio.
2. Como dato adicional, a la actividad principal del proyecto la he llamado Consumer:
3. Creamos una nueva clase llamada MiServicio que es en dónde vamos a escribir todo el código correspondiente a lo que hace el servicio. A continuación te proporciono el código que deberá tener esta clase:

```
    class MiService : Service() {
      var timer: Timer?=null
      var miBinder: IBinder = MiBinder()
      lateinit var lista: ArrayList<String>
      var array = arrayOf("blanco", "azul", "verde")
      var pos = 0
      override  fun onCreate() {
          pos = 0
          super.onCreate()
          timer=Timer()
          lista = ArrayList()
          pollForUpdates()
      }
      fun pollForUpdates() {
15        timer?.scheduleAtFixedRate(object : TimerTask() {
              override  fun run() {
                  if (lista.size >= array.size) {
                      lista.removeAt(0)
                  }
                  lista.add(array[pos])
                  pos++
                  if (pos >= array.size) pos = 0
              }
          }, 0, UPDATE_INTERVAL)
      }
      override  fun onDestroy() {
          super.onDestroy()
          timer?.cancel()
      }
      override fun onBind(intent: Intent?): IBinder {
          return miBinder
      }
      fun getLista(): List<String> {
          return lista
      }
36    internal inner class MiBinder : Binder() {
          val service: MiService
              get() = this@MiService
39    }
      companion object {
          private const val UPDATE_INTERVAL: Long = 5000
      }
  }
```

📌En primer lugar podemos observar que esta clase hereda de **`Service`** . Dentro de la clase MiServicio declaramos un **`Timer`** que nos ayudará a retardar las actualizaciones de la información que proveerá el servicio junto con la constante estática UPDATE_INTERVAL que fijamos con un valor de 5 segundos. Tenemos un ArrayList y un arreglo de cadenas que nos servirán para jugar con

la información del servicio; y un IBinder que permite interactuar con un objeto de forma remota.
Dentro de los métodos que estamos creando en nuestro servicio se encuentra `pollForUpdates()` en el que definimos la tarea cuya ejecución se repetirá según el tiempo que hayamos establecido, mediante el método de la clase Timer `scheduleAtFixedRat()` , **Línea 15**
Con respecto al uso de IBinder, la documentación de Android nos dice que no debemos implementarla de manera directa, y que para efectos prácticos debemos utilizar una clase que extienda de Binder. Es por ello que creamos una clase interna llamada MyBinder **Líneas 36-39**. El resto es incluir los métodos anulados `onDestroy()` y `onBind()` , para cancelar el servicio o para devolverlo respectivamente.

4. Ahora pasemos a codificar la actividad que consumirá este servicio. Para la cuál definimos la interfaz gráfica con un botón para refrescar la información del servicio y una ListView para desplegar los datos del mismo.

```
class MainActivity : Activity() {
    var miservicio: MiService?=null
    lateinit  var values: ArrayList<String>
    lateinit  var adapter: ArrayAdapter<String>
    lateinit  var lista: ListView
    lateinit var boton: Button
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        lista = findViewById(R.id.listView)
        boton = findViewById(R.id.button)
        doBindService()
        values = ArrayList()
        adapter = ArrayAdapter<String>(this, android.R.layout.
                                        simple_list_item_1, values)
        lista.setAdapter(adapter)
        boton.setOnClickListener {  showServiceData()}
    }
19  private val mConnection: ServiceConnection = object : ServiceConnection
    {
        override fun onServiceConnected(name: ComponentName,
                                         service: IBinder) {
            miservicio = (service as MiBinder).service
            Toast.makeText(this@MainActivity, "Connectado",
                        Toast.LENGTH_LONG).show()
        }
        override fun onServiceDisconnected(name: ComponentName) {
            miservicio = null
        }
30  }
    fun doBindService() {
32      bindService(Intent(this, MiService::class.java), mConnection,
                            BIND_AUTO_CREATE)
    }
    fun showServiceData() {
        if (miservicio != null) {
37          val listaTrabajo: List<String> = miservicio!!.getLista()
            values.clear()
            values.addAll(listaTrabajo)
            adapter.notifyDataSetChanged()
        }
    }
}
```

📌 Declaramos un ArrayList y un objeto de tipo MiServicio. Para poder trabajar con el servicio será indispensable crear un objeto de tipo **`ServiceConnection`** **Línea 19 - 30** , por medio del cual realizaremos las funciones necesarias para que nuestra actividad pueda conectarse con el servicio creado y podamos así, acceder a la información que nos provee. Este objeto será inicializado a través

> del método `bindService()` **Línea 32**, que es llamado en el `onCreate()` de la activity. Como vamos a llenar una ListView con estos datos, es importante crear un adaptador. Luego, en el método propio `showServiceData()` actualizamos la lista con los datos recibidos del servicio **Línea 37**.

5. IMPORTANTE , una vez que hemos terminado la lógica de la aplicación nos faltará dar de alta el servicio creado, en el archivo AndroidManifest.xml.

Reconstruye el ejemplo para entender el funcionamiento del la comunicación mediante Bind

BroadCastReceiver

Un broadcast receiver es el componente que permite recibir y responder ante eventos del sistema (aviso de batería baja, un SMS recibido,…) o eventos producidos por otras aplicaciones o servicios.
Algunos eventos son de acceso privilegiado y hay que establecerlos en el archivo de manifiesto de la aplicación con la etiqueta `<uses_permission>`.
Para crear un broadcast receiver simplemente tenemos que crear una clase que herede de `BroadcastReceiver` y anular el método `onReceive()`, que es el método que se ejecutará cada vez que se produzca el evento al que se suscriba nuestro broadcast receiver (similar al onCreate() de una activity).
La clase BroadcastReceiver es capaz de recibir intents a través del método `sendBroadcast()`.
Otra de las clase utiles es `PendingIntent`, que es un token que le damos a otra aplicación a través de un Notification Manager, Alarm Manager u otras aplicaciones de terceros, que les permiten a estos componentes utilizar los permisos de nuestra aplicación para ejecutar un trozo de código predefinido.

Aplicación ejemplo para activar la vibración y la alarma del dispositivo

Vamos a definir un broadcast receiver que escuche los cambios de estado del teléfono. Queremos tener en la pantalla de nuestra aplicación un EditText para introducir los segundos que pasarán antes de que nos vibre el teléfono y un botón para iniciar el conteo. El layout de la actividad principal se omite por su sencillez.
Pasamos a explicar la clase que hereda de `BroadCastReceiver`.

```
class BroadCastAlarma : BroadcastReceiver() {
    @RequiresApi(Build.VERSION_CODES.O)
    override fun onReceive(context: Context, intent: Intent?) {
        Toast.makeText(
            context,
            "Vibración activa porque el tiempo se ha terminado",
            Toast.LENGTH_LONG
        ).show()
        val vibrator = context.getSystemService(Context.VIBRATOR_SERVICE)
                                                        as Vibrator
        vibrator.vibrate(VibrationEffect.createOneShot(8000,
                                        VibrationEffect.DEFAULT_AMPLITUDE))
    }
}
```

> 📌 Como vemos implementamos el método **`onRecive()`** que recibe el contexto desde donde es llamado y el intent que lo disparará. Visualizamos un toast para indicar que el tiempo de espera finalizó y que el modo vibración se activará. A continuación activamos la vibración durante ocho segundos. Para ello definimos un objeto de tipo **`Vibrator`** que lo inicializamos con el servicio propio del sistema y llamamos al método **`vibrate(`**) de dicho objeto.

En la activity principal solamente tenemos que implementar la pulsación del botón, se recogerá la información introducida en el EditText, se programará una alarma y cuando esta termine se invocará al broadcast receiver definido anteriormente. Veamos el código:

```
class MainActivity :AppCompatActivity() {
    lateinit var texto:EditText
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        texto=findViewById<EditText>(R.id.editText)
        texto.setText("0")
        findViewById<Button>(R.id.button).setOnClickListener {  activar()}
    }

    private fun activar() {
        val tiempo = texto.text.toString().toInt()
        val intent = Intent(this,BroadCastAlarma::class.java)
        val pendingIntent = PendingIntent.getBroadcast(this,
                                                REQUESTCODE, intent, 0)
        val alarmManager = getSystemService(ALARM_SERVICE) as AlarmManager
        alarmManager[AlarmManager.RTC_WAKEUP, System.currentTimeMillis()
                                                    + tiempo * 1000] =
                                                         pendingIntent
        Toast.makeText(this, "Has fijado la alarma en "+tiempo+"segundos",
                                            Toast.LENGTH_LONG).show()
    }
    companion object{
        val REQUESTCODE=1111
    }
}
```

📌Lo primero que hacemos es recuperar la referencia al EditText y parseamos el contenido a Integer. Visualizamos un toast para informar que se va a establecer la cuenta atrás en los segundos especificados en el EditText.
¡¡Lo interesante!! Definimos un intent que apunta a nuestro broadcast receiver. Luego hacemos uso de un PendingIntent en el que se recoge el contexto en el que este objeto se ejecutará cuando sea atendido, un código de identificación, el intent donde se define la acción y un flag. Nosotros lo hacemos con `PendinIntent.getBroadcast()` , porque lo que hemos definido o quien va atender esta petición está en un BroadcastReceiver **Líneas 13 - 15**.
Usaríamos ***`nombrePendinIntent.getActivity()`*** *si iniciáramos una activity y* ***`nombrePendinIntent.getService()`*** *para un servicio.*
Finalmente creamos un objeto AlarmManager inicializándolo con el servicio propio del sistema (ALARM_SERVICE). Con el método set() definimos el tiempo en el que será lanzada la alarma (i segundos después del actual) y el PendingIntent que resuelve la lógica de nuestra aplicación y que se ha definido anteriormente **Líneas 16 - 19**.

> ⚠Para probar esta aplicación es necesario un terminal real y no el emulador, ya que éste último no puede reproducir la vibración.

Prueba el ejemplo del BroadcastReceiver

Sigue el ejercicio resuelto que muestra un Toast indicando si se ha conectado/desconectado el dispositivo a la alimentación

WorkManager

Android WorkManager es una biblioteca de procesamiento que se utiliza para ejecutar tareas en segundo plano que deberían ejecutarse de forma garantizada, pero no necesariamente de forma inmediata. Con WorkManager podemos poner en cola nuestro procesamiento en segundo plano incluso cuando la aplicación no se está ejecutando o el dispositivo se reinicia por alguna razón. WorkManager también nos permite definir las restricciones necesarias para ejecutar la tarea, por ejemplo, la disponibilidad de la red antes de iniciar la tarea en segundo plano.

Android WorkManager es parte de Android Jetpack y es perfecta para usar cuando la tarea:

1. No necesita ejecutarse en un momento específico
2. Puede aplazarse su ejecución
3. Se necesita que se ejecute incluso después de que se cierre la aplicación o se reinicie el dispositivo.
4. Tiene que cumplir con limitaciones como el suministro de batería o la disponibilidad de la red antes de la ejecución.

Para poder trabajar con `WorkManager` será necesario incluir la siguiente implementación:

```
    implementation 'androidx.work:work-runtime-ktx:2.6.0'
```

Crear un trabajo

Para crear un trabajo de este tipo, se deberá implementar una clase que herede de `Worker` y anular su método `doWork()` donde añadiremos el código que queramos que se ejecute en segundo plano (a diferencia de los servicios, no habrá que crear ningún hilo ya que se encarga el sistema).

```
class WorkManager(val context: Context, workerParams: WorkerParameters) :
    Worker(context, workerParams){
    override fun doWork(): Result {
       for(i in 0..5) {
           Thread.sleep(1000)
           Log.d("DATOS", i.toString())
       }
        var intento=Intent()
        intento.setComponent(
            ComponentName("com.ejemplos.b10.ejemploviewmodelcorrutinas",
            "com.ejemplos.b10.ejemploviewmodelcorrutinas.MainActivity")
        )
        //Se añade el Flag para que se pueda abrir una activity
        // desde un hilo en segundo plano
        intento.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK)
        if (intento.resolveActivity(context.packageManager) != null)
            startActivity(context,intento,null)
        return Result.success()
    }
}
```

> 📌 Este ejemplo crea una trabajo que realiza un retardo de 5 segundos y lanza un Intent para abrir otra aplicación. Como se puede ver todo el código está implementado en el método **`doWork()`** .

Iniciar un trabajo

Este tipo de trabajos permite definir si queremos el hilo como tarea única o periodica cada `X` tiempo. Para ello se usan dos clases distintas **`OneTimeWorkRequestBuilder<T>()`** y **`PeriodicWorkRequestBuilder<T>()`**. Veamos ejemplos de la creación de trabajos de los dos tipos:

```
val work = OneTimeWorkRequestBuilder<MiWorkManager>()
val workPeriodico =PeriodicWorkRequestBuilder<MiWorkManager>(16, TimeUnit.MINUTE
                                                              .build()
```

En el caso de tarea única no hay nada que explicar, en cambio de la periodica podemos decir que le entra como primer argumento el intervalo que tiene que pasar hasta que se vuelva a realizar la siguiente iteración y el segundo argumento especifica las unidades del anterior (en este caso la tarea se repetirá cada 16 minutos).

> ✋ Algunos puntos importantes a tener en cuenta al trabajar con `WorkManager` son:
>
> - La primera ejecución ocurre inmediatamente después de que se cumplen las restricciones mencionadas por la solicitud de trabajo
> - El trabajo se puede encadenar con otras solicitudes de trabajo.
> - Si no se cancela se seguirá ejecutando hasta que acabe.
>
> Además de lo anterior en caso de trabajos de tipo periodicos, `PeriodicWorkRequest` , también se tendrá en cuenta que:
>
> - El trabajo se ejecuta varias veces.
> - Si no se cancela se seguirá ejecutando periódicamente.
> - La siguiente ejecución ocurre después de que transcurra el periodo de tiempo indicado en los argumentos.
> - El intervalo mínimo de repetición es de *15 minutos*.
> - El trabajo no se puede encadenar con otras solicitudes de trabajo.
> - El intervalo de tiempo entre 2 intervalos periódicos puede diferir según las restricciones de optimización del sistema operativo.

Una vez creado el objeto, para ejecutar la solicitud de trabajo necesitamos llamar al método `enqueue()` desde una instancia de `WorkManager` y pasar el `WorkRequest` :

```
val workPeriodico = PeriodicWorkRequestBuilder<WorkPeriodico>(16, TimeUnit.MINUT
                                                              .build()
WorkManager.getInstance(this).enqueue(workPeriodico)
```

El método `enqueue()` pone en cola una o más `WorkRequests` para que se ejecuten en segundo plano.

Funcionalidad de WorkManager

Las instancias de `WorkManager` proporcionan una serie de funcionalidades para manejarnos con las tareas. Por ejemplo, podemos controlar si las tareas están canceladas o cancelarlas si nos interesa, etc:

```
val workManager = WorkManager.getInstance(this)
val id:UUID
id = workPeriodico.id
if(!workManager.getWorkInfoById(id).isCancelled)
                workManager.cancelWorkById(id)
//En este caso se cancelan todas las tareas lanzadas por esta actividad
workManager.cancelAllWork()
```

Cuando cancelamos un trabajo, este no se cancela automáticamente sino que es el sistema el que se encarga de informar de la cancelación. Por lo que será tarea del programador atender a la información del sistema para parar el trabajo. Esto se hace desde la clase que hereda de `WorkManager`, por ejemplo de las siguiente manera:

```
class MiUnicoWork(val context: Context, workerParams: WorkerParameters) :
    Worker(context, workerParams){
    override fun doWork(): Result {
     ...
     //isStopped devolverá true cuando el sistema informe
     //que el trabajo ha sido cancelado desde código
        while(!isStopped)
        {
            Thread.sleep(1000)
            Log.d("DATOS", "Realizar tarea larga")
        }
     ...
        return Result.success()
    }
}
```

Restricciones de WorkManager

Una de las ventajas que proporciona esta clase, es la de permitir restringir la funcionalidad de la tarea hasta que se cumplan una serie de condiciones que se le hayan asociado. Logicamente, estas restricciones se impondrán al ser necesarios algunos requisitos para la funcionalidad de la app. Por ejemplo, si se necesita Internet para realizar una descarga de datos. En el siguiente código se han añadido todas las restricciones posibles:

```
//Se construyen las restricciones
val constraints = Constraints.Builder()
            .setRequiresBatteryNotLow(true)
            .setRequiredNetworkType(NetworkType.CONNECTED)
            .setRequiresCharging(true)
            .setRequiresStorageNotLow(true)
            .setRequiresDeviceIdle(true)
            .build()
val workUnico = OneTimeWorkRequestBuilder<MiUnicoWork>()
//Se añaden al trabajo
workUnico.setConstraints(constraints)
WorkManager.getInstance(this).enqueue(workUnico.build())
```

Comunicación con WorkManager

Si queremos comunicar información a la tarea, ya sea como datos de entrada antes de su ejecución o como datos de salida al terminar, se utiliza la clase `Data.Builder`. La información se añade al objeto mediante el ya conocido método del mapa clave-valor.
Para acceder a estos datos de entrada dentro del Worker, se usa la propiedad `inputData` dentro del método `doWork()`.

```
val inputData = Data.Builder()
                .putString("USER", user)
                .putString("PASS", pass)
                .build()
val workUnico = OneTimeWorkRequestBuilder<MiUnicoWork>()
workUnico.setInputData(inputData)
WorkManager.getInstance(this).enqueue(workUnico.build())
```

```
class MiUnicoWork(val context: Context, workerParams: WorkerParameters) :
    Worker(context, workerParams){
    override fun doWork(): Result {
     ...
    val user = inputData.getString("USER")
    val pass = inputData.getString("PASS")
     ...
        return Result.success()
    }
}
```

Ejercicio resuelto números primos

Ejercicio propuesto temporizador con vibración