

# Apuntes

[Descargar estos apuntes](#)

## Tema 10. Corrutinas y Servicios

### Índice

1. [Corrutinas Kotlin](#)
  1. [Alcance de las corrutinas](#)
  2. [CoroutineContext](#)
  3. [Funciones de Suspensión](#)
  4. [Builders de corrutinas](#)
  5. [ViewModel y Corrutinas](#)
2. [Servicios y tareas de larga duración](#)
  1. [Servicios](#)
  2. [WorkManager](#)

# Corrutinas Kotlin

Una [corrutina de Kotlin](#) 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** .

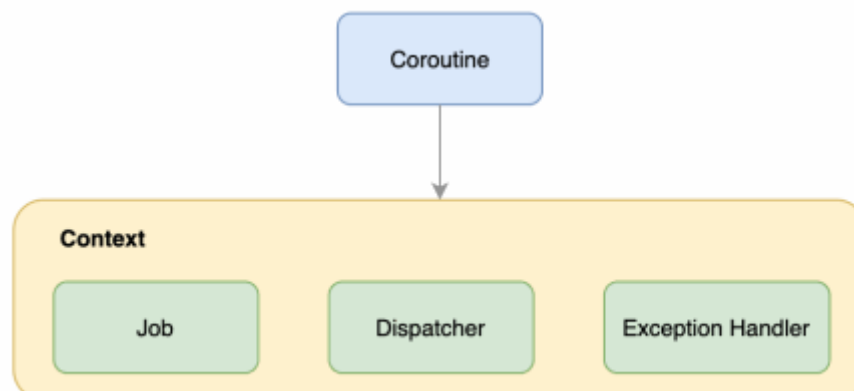
```
var coroutineScope= CoroutineScope(Dispatchers.Main)
private fun ejemploCoroutineScopeConMain()
{
    coroutineScope?.launch {
        while (isActive) {
            delay(3000)
            Toast.makeText(requireActivity(), "[CoroutineScope-Main] I'm alive o
            "${Thread.currentThread().name}!", 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 esta función, se puede bloquear la coroutine asociada con el job hasta que todos los jobs hijos hayan finalizado. Todas las funciones de suspensión que se llaman dentro de una coroutine 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 coroutine.
- `job.cancel` -> Esta función cancelará todos sus jobs hijos asociados. `job.cancel()` esta es una función normal, por lo que no requiere una coroutine 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.

- **Dispatcher** 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 coroutine que especifican el hilo o hilos que pueden ser utilizados por la coroutine 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 coroutines 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
<i>Dispatchers.Main</i>	Main thread on Android	- Calling suspend functions - Call UI functions - Update LiveData
<i>Dispatchers.IO</i>	Disk and network IO*	- Database - File IO - Networking
<i>Dispatchers.Default</i>	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 experiencia de usuario

## Funciones de Suspensión

Las corrutinas se basan en las **funciones de suspensión** 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 corrutinas 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 lanzadas
    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 segundos. Si el bucle no lanzara las tres subrutinas con `launch` y solo dejáramos 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 omitiéndose o no el `Dispatchers`, 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 está 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 Dispatcher 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.

## 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.

## 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 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 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 comunice 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 **stopSelf()** 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 líneas 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()
    }
}

```

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.

## WorkManager

Android `WorkManager` es una biblioteca de procesamiento en segundo plano 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 y 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 una biblioteca de procesamiento en segundo plano 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 elimine 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.