

# Apuntes

[Descarregar estos apuntes](#)

## Tema 10. Corrutinas i Servicis

### Índice

1. [Corrutines Kotlin](#)
  1. [Abast de les corrutinas](#)
  2. [CoroutineContext](#)
  3. [Builders de corrutines](#)
  4. [ViewModel i Corrutines](#)
2. [Servicis i tasques de llarga duració](#)
  1. [Servicis](#)
  2. [BroadcastReceiver](#)
  3. [WorkManager](#)

# Corrutines Kotlin

Una [corrutina de Kotlin](#) és un conjunt de sentències que realitzen una tasca específica, amb la capacitat de suspendre o resumir la seua execució sense bloquejar un fil. Açò permet que tingues diferents corrutines cooperant entre elles i no significa que existisca un fil per cada corrutina, al contrari, pots executar diverses en un sol. Les corrutinas són part del paquet `kotlinx.coroutines`, per la qual cosa necessites especificar la [dependencia correspondent] (<https://mvnrepository.com/artifact/org.jetbrains.kotlinx/kotlinx-coroutines-android>) en en `build.gradle` . A hores d'ara:

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

Les [corrutines en el desenrotllament d'Android](#), encara que són un element que va a permetre'ns crear codis més senzills, al principi poden paréixer complexes a causa del seu nivell d'abstracció.

## Abast de les corrutinas

Quan creguem corrutines assumim de manera implícita dos aspectes molt importants: L'àmbit (`CoroutineScope` i `GlobalScope`) i el context (`CoroutineContext`) .

L'ús de `GlobalScope` permet crear corrutinas de nivell superior, açò vol dir que tenen la capacitat de viure fins que acabe l'aplicació i és treball del desenvolupador el control per a la seua cancel·lació, per la qual cosa no s'aconsella usar este àmbit.

```
private fun ejemploGlobalScope() {
    GlobalScope.launch(Dispatchers.Main) {
        launch(Dispatchers.IO) {
            delay(8000)
            withContext(Dispatchers.Main) {Toast.makeText(requireActivity(),
                "GlobalScope despues 8 segundos",
                Toast.LENGTH_SHORT).show()}
        }
        Toast.makeText(requireActivity(), "GlobalScope",
            Toast.LENGTH_SHORT).show()
    }
}
```

📌 amb `launch` llancem la corrutina en el context de la Main per a poder mostrar el Toast i dins d'esta llancem una altra per a processos llargs (els tres segons de retard).

Per a reduir este abast, Kotlin ens permet crear els espais on volem que es done la concurrència. Açò ho farem per mitjà de **CoroutineScope** .

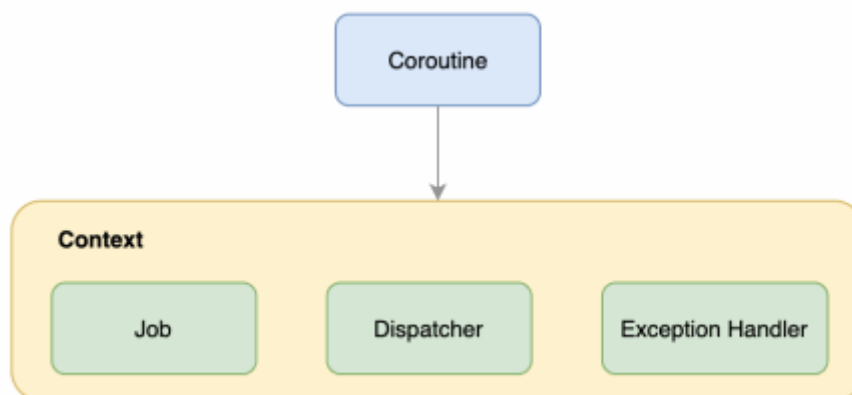
```
var coroutineScope= CoroutineScope(Dispatchers.Main)
private fun ejemploCoroutineScopeConMain()
{
    coroutineScope?.launch {
        while (isActive) {
            delay(3000)
            Toast.makeText(requireActivity(), "[CoroutineScope-Main] I'm alive o
            "${Thread.currentThread().name}!",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

✦ esta corrutina roman activa fins que siga cancel·lada d'alguna forma. Està llançada amb **launch** i en l'abast de la Main, per la qual cosa podem usar el Toast per a mostrar la informació. L'abast personalitzat ho hem aconseguit amb el constructor de **CoroutineScope ()**

Cada vegada que usem un constructor de coroutines en realitat estem fent una crida a una funció que rep com a primer paràmetre un objecte de tipus **CoroutineContext** .

## CoroutineContext

Les coroutines sempre s'executen en algun context que està representat per un valor del tipus **CoroutineContext**.



El context de Coroutine és un conjunt de diversos elements. Els elements principals són el Job de la Coroutine, el seu dispatcher i també el seu Exception handler.

- **Job** D'acord amb la documentació oficial "Un Job és una cosa cancel·lable amb un cicle de vida que culmina amb la seua finalització. Els treballs de coroutines es creen amb `launch coroutine builder`. Executa un bloc de codi especificat i es completa a la finalització d'este bloc.

L'execució d'un treball no produïx un valor de resultat. Hauríem d'utilitzar una interfície `Deferred` per a un treball que produïska un resultat. Un treball amb resultat (`Deferred`), es crea amb el `async coroutine builder` i el resultat es pot recuperar amb el mètode `await()`, que llança una excepció si el `Deferred` ha fallat.

Els jobs tenen un parell de funcions interessants que poden ser molt útils. Però és important entendre que un Job pot tindre al seu torn un altre Job pare. Eixe job pare té un cert control sobre els fills, i ací és on entren en joc estes funcions:

- `job.join` -> Amb esta funció, es pot bloquejar la corutina associada amb el job fins que tots els jobs fills hagen finalitzat. Totes les funcions de suspensió que es criden dins d'una corutina estan vinculades a job, així que el job pot detectar quan finalitzen tots els jobs fills i després continuar l'execució. `job.join()` és una funció de suspensió en si mateixa, per la qual cosa ha de cridar-se dins d'una altra corutina.
- `job.cancel` -> Esta funció cancel·larà tots els seus jobs fills associats. `job.cancel()` esta és una funció normal, per la qual cosa no requereix una corutina per a ser cridada.

👉 Amb `GlobalScope` el pare no esperarà la finalització dels seus fills i una vegada que el pare és cancel·lat, els altres treballs continuaran corrent a banda. És a dir, en este cas és responsabilitat del desenvolupador portar el control del temps de vida de les coroutines perquè no hi ha sincronització amb els treballs fills.

- **Dispatcher** En Kotlin, totes les Coroutines han d'executar-se en un dispatcher inclús quan s'executen en el fil principal. Els Dispatchers són un tipus de contextos de corutina que especifiquen el fil o fils que poden ser utilitzats per la corutina per a executar el seu codi. Hi ha dispatchers que només usen un fil (com `Main`) i altres que definixen un grup de fils que s'optimitzaran per a executar totes les coroutines que reben. Per a especificar on han d'executar-se les coroutines, Kotlin proporciona tres Dispatchers que pots utilitzar:
  - `Dispatchers.Main`. Fil principal en Android, interactua amb la UI i realitza treballs lleugers com: cridar a funcions de suspensió, cridar a funcions de la interfície d'usuari i actualitzar `LiveData`
  - `Dispatchers.IO`. En general, totes les tasques que bloquejaren el fil mentre esperen la resposta. Optimitzat per a l'E/S de disc i xarxa fora del fil

principal: sol·licituds de bases de dades, lectura/escritura d'arxivis, sensors, connexió en xarxa, ...

- **Dispatchers.Default** . Optimitzat per al treball intensiu de la CPU fora del fil principal: ordenar una llista, anàlisi de JSON, utilitats, etc.

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

- **ExceptionHandler**. Este element és opcional del CoroutineContext, i ens permet manejar excepcions no capturades. I d'esta manera millorar l'expericencia d'usuari

## Funcions de Suspensió

Les corrutinas es basen en les **funciones de suspension** que són funcions normals a què se'ls agrega el modificador **suspend** , les funcions de suspensió tenen la capacitat de bloquejar l'execució de la corrutina mentres estan fent el seu treball. Una vegada que acaba, el resultat de l'operació es torna i es pot utilitzar en la següent línia.

D'esta manera s'agreguen dos noves operacions (a més d'invocar/cridar i tornar):

- suspendre: pausa l'execució de la corrutina actual, guardant totes les variables locals. El fil actual pot continuar amb el seu treball, mentres que el codi de suspensió s'executa en un altre fil.
- reprendre: continua una corrutina suspesa des del lloc on es va interrompre quan el resultat està preparat.

Les funcions de suspensió només poden cridar-se o executar-se dins d'una corrutina o dins d'una altra funció de suspensió. Algunes de les funcions usades amb freqüències per a llançar corrutinas són funcions de suspensió en si mateixes, com ara **delay, join, await, withContext, supervisorScope, etc.** .

### withContext

Esta és una funció de suspensió que permet canviar fàcilment el context que s'utilitzarà per a executar una part del codi dins d'una corrutina. **withContext** és una

funció de suspensió en si mateixa.

## Builders de corrutines

Els constructors de corrutines servixen per a iniciar les corrutines. Tenim diferents builders depenent del que vullguem fer, i inclús tècnicament es poden crear personalitzats. Però per a la majoria dels casos són suficients amb els que proporciona la llibreria:

- **runBlocking**, este builder bloqueja el fil actual fins que s'acaben totes les tasques dins d'eixa corrutina. No sol ser una funcionalitat molt usada de les corrutines, ja que precisament el que es pretén amb estes és la sincronització de més d'una tasca.

```
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 exemple llancem una corrutina que al seu torn llança altres tres corrutines, la primera amb **runBlocking** bloquejarà el fil principal, per la qual cosa el **Toast** no ocorrerà fins que no acaben totes les subrutines filles de **runBlocking**. **launch** es llança, en este exemple, en un context que no és el Main, per la qual cosa mai podrem introduir cap interacció amb les vistes (per això l'eixida la realitzem per mitjà de log). Cada una de les tres corrutines llançades per launch romandrà 5 segons d'espera, però al realitzar-se al mateix temps, el retard és de poc més de 5 segon. Si el bucle no llançara les tres subrutines amb launch i només deixàrem el **delay** el retard seria d'uns 15 segons aproximadament.

- **launch** , este és el builder més usat. A diferència de `runBlocking`, no bloquejarà el subprocés actual (si s'usen els dispatchers adequats) .`Launch` torna un `Job`, que permetrà realitzar les funcions explicades anteriorment.

Quan **launch** s'usa sense paràmetres, hereta el context (i per tant el dispatcher) del `CoroutineScope` des del que s'inicia. Pot iniciar-se dins d'una altra corrutina ometent-se o no el Dispatchers, però si no està dins d'una corrutina necessitarà obligatòriament especificar l'abast. `Launch` ens torna un `Job`, que permetrà realitzar les funcions explicades anteriorment.

```
...
//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 altre builder també permet executar diverses tasques en segon pla en paral·lel. No és una funció de suspensió en si mateixa, per la qual cosa quan executem `async`, el procés en segon pla s'inicia, però la següent línia s'executa immediatament. **async** sempre ha de cridar-se dins d'una altra corrutina, i torna un job especialitzat que s'anomena **Deferred** .

**Deferred** té una nova funció de suspensió cridada **await ()** que és la que bloqueja. Cridarem a `await()` només quan necessitem el resultat. Si el resultat encara no està llest, la corrutina se suspén en eixe punt. Si ja tenim el resultat, simplement ho tornarà i continuarà.

```
private fun ejemploAsyncAwait()
{
    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()
    }
}
```

✂ llancem el **async** dins d'una corrutina launch amb context Main, per a poder pintar cada segon el text que ens interesse (en este cas un comptador d'1 a 5) . Canviem el Dispatcher a IO per a realitzar el retard i tornem una cadena com a resposta del **async** . Quan acabe el treball la corrutina, recuperarem la cadena amb **await** i es llançarà el Toast que la utilitza.

✍ **Còpia els codis dels exemples i completa'ls fins a fer-los funcionar. Per a ajudar-te en la comprensió pots analitzar l'exercici resolt de la Barra de Progrés**

## ViewModel i Corrutines

ViewModel és estés amb una propietat **viewModelScope** que s'encarrega de cancel·lar les corrutinas quan ja no són necessàries.

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

Gràcies al comportament de **CoroutineScope** a través de la propietat **viewModelScope** es realitza un seguiment de totes les corrutinas que es creen en este ambet. Per tant, si es cancel·la un abast, es cancel·len totes les corrutinas creades en ell.

Per exemple, en el següent codi creuem un ViewModel que ens gestionarà un **MutableLiveData** de tipus ArrayList de Strings i que té un mètode que simula una descàrrega de dades que es guardaran en l'objecte:



```

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>()
        //Simulació 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 mètode **descargarDatos** s'usa la propietat **viewModelScope** per a llançar la corrutina que produirà el retard i la llista generada amb elements aleatoris.

L'ús del ViewModel des de les distintes parts de l'aplicació, és igual a què hem vist en temes anteriors.

✍️ **Realitza l'exercici proposat del cronòmetre**

## Servicis i tasques de llarga duració

Les corrutines estan dissenyades per a treballar tasques que han de finalitzar quan l'usuari ix d'un determinat abast o acaba una interacció, però a l'hora de realitzar el que cridem **long-time running tasks**, és a dir tasques que han d'estar durant un gran temps en segon pla i no tenen perquè estar lligades a l'activitat que les ha llançat o a la interacció amb l'usuari, és necessari pensar en altres respostes:


1. Tasques que no necessiten executar-se immediatament i processar-se de forma contínua, inclús quan l'usuari col·loca l'aplicació en segon pla o es reinicia el dispositiu. En este cas el recomanable és l'ús de **WorkManager**.
2. En casos concrets en què es realitzen tasques llargues com la reproducció de contingut multimèdia o la navegació activa, es recomana l'ús de **Servicios** però en primer pla.

## Servicis

Els **Servicis** són components que poden executar-se en background (o segon pla) i que no proporciona una interfície d'usuari, per la qual cosa no estan pensats per a interactuen amb l'usuari.

La plataforma Android ofereix una gran quantitat de servicis predefinits, disponibles regularment a través de la classe `Manager`. D'esta manera, en les nostres activitats podrem accedir a estos servicis a través del mètode `getSystemService ()`.

Si el servici creat realitzarà tasques que tinguen un cost computacional elevat, es poden provocar problemes en l'aplicació (Application Not Responding o ANR), per a evitar situacions com estes, el servici pot executar-se en un fil secundari. Si l'aplicació està orientada al nivell d'API 26 o un nivell superior, el sistema imposa restriccions en l'execució de servicis en segon pla quan l'aplicació mateixa no es troba en primer pla. En estos casos és millor usar una tasca programada, `WorkManager`.

 D'altra banda, si necessitem utilitzar servicis propis, estos han d'anar declarats en l'arxiu `AndroidManifest.xml`.

Tenim diverses opcions perquè nostra activity es comuniqui amb un servici.

- Usar **intent**: és un escenari simple on no es requereix comunicació directa ni notificacions de processos. El servici rep les dades via intent i realitza la tasca que siga necessària sense tornar cap tipus de dada. Un escenari on podem utilitzar açò és quan necessitem actualitzar un content provider. El servici rep les dades via intent, actualitza el content provider i este notifica a la nostra app que el contingut està actualitzat, sense que es requereisca alguna acció extra pel nostre servici.
- Usar **receivers**: un altre mètode és emetre esdeveniments i registrar receptors (receivers, classe `BroadcastReceiver`) per a establir la comunicació. Per exemple, una activity registra dinàmicament un receiver per a un esdeveniment, i el servici és l'encarregat d'emetre eixe esdeveniment. Este escenari s'usa quan és necessari que l'escenari notifique que va acabar de fer alguna tasca.

### Construint un Servici

Per a crear un Servici haurem d'estendre de `Service` pel que tindrem que redefinir i conèixer 4 mètodes principals:

- **onCreate**: De manera semblant a un Activity, s'executa la primera vegada que es crega el servici i servix per a inicialitzar variables.

- **onStartCommand** : S'executa quan el servici rep un intent llançat per mitjà del comando startService. Si el servici ja està iniciat, els intents successius no tornaran a generar un **onCreate** si no que passaren directament ací. És important tindre en compte que si es creen fils en este mètode, ha de comprovar-se que el fil no estiga ja funcionant o podrien accidentalment crear-se fils nous cada vegada que es passa per ací.
- **onBind** : Este comando és cridat quan s'executa un bindService () des d'un Activity. Servix per a tornar una referència a manera d'IBinder a l'Activity de la instància del servici. \* **onDestroy** : Quan s'anomena al comando **stopService()** o dins del mateix service al mètode.
- **stopSelf()** es passa per este mètode, on han d'acabar-se fils o tasques que puguen estar executant-se.


Exemple d'un servici senzill, podem analitzar el seu funcionament visualitzant les esbosses 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()
    }
}
```

🚩 Cal destacar que este tipus de servicis cal parar-los explícitament, encara que el sistema també podria parar-los si es trobara escàs de memòria. Si en el mètode onStartCommand s'ha tornat **START\_STICKY**, el sistema podrà reviuire el servici quan torne a tindre memòria disponible, encara que cal tindre en compte que llavors l'intent que rebrà onStartCommand serà nul (null).

Una manera d'inicialitzar este servici, és des d'una activitat i de forma molt pareguda 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 s'ha de declarar els servicis usats en el manifest.

```
<service android:name=".MyService"/>
```

Com ja s'ha indicat, tot este codi funcionaria en primer pla, si desitgem crear un fil per a realitzar tasques en segon pla, caldria fer-ho 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()
    }
}

```



### Prova els dos exemples anteriors i mira el resultat en el Logcat

Hi ha dos tipus de servicis. Aquell que s'inicia explícitament a través del mètode `startService()` cridat **Started Service** i el que es lliga a un component amb `bindService()` cridat **Bound Service**.

- Started: Una vegada iniciat el servici, s'executa de forma indefinida, inclús si el component que ho va iniciar no està en primer pla.
- Bound: En este cas, el servici oferix una interfície de tipus client-servidor a aquells components que s'associen o enllacen al dit servici. Els components associats o enllaçats (bound) al servici poden enviar peticions, aconseguir resultats i respostes o, inclús, dur a terme tasques de comunicació amb altres processos. En este cas, el servici només s'executa quan hi ha algun component associat o enllaçat a ell.

## **Aplicació exemple per a crear un servici per a poder consumir-ho des d'una activitat**

Crearem un servici perquè des d'una activitat podem consumir-ho. L'objectiu del servici serà recuperar dades de forma periòdica perquè d'esta manera l'activitat puga desplegar-los a través d'una ListView, sempre podent demanar les dades més actualitzats d'este servici. Enllaçarem el servici amb l'activitat (Bound) . Tenint el concepte clar, passem a codificar-ho:

1. Creguem un nou projecte cridat MiServicio.
2. Creguem una nova classe crida MiService que és on escriurem tot el codi corresponent al que fa el servici. A continuació es proporciona el codi que haurà de tindre esta classe:

```

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 lloc podem observar que esta classe hereta de **Service** . Dins de la classe MiServicio declarem un **Timer** que ens ajudarà a retardar les actualitzacions de la informació que proveirà el servici junt amb la constant estàtica UPDATE\_INTERVAL que fixem amb un valor de 5 segons. Tenim un ArrayList i un arreglament de cadenes que ens serviran per a jugar

amb la informació del servici; i un IBinder que permet interactuar amb un objecte de forma remota.

Dins dels mètodes que estem creant en el nostre servici es troba

**pollForUpdates()** en el que definim la tasca l'execució de la qual es repetirà segons el temps que hàgem establert, per mitjà del mètode de la classe Timer **scheduleAtFixedRate()** , **Línea 15**.

Respecte a l'ús d'IBinder, la documentació d'Android ens diu que no hem d'implementar-la de manera directa, i que per a efectes pràctics hem d'utilitzar una classe que estenga de Binder. És per això que creuem una classe interna crida MiBinder **Líneas 36-39**. La resta és incloure els mètodes anul·lats

**onDestroy()** i **onBind()** , per a cancel·lar el servici o per a tornar-ho respectivament.

1. Ara passem a codificar l'activitat que consumirà este servici. Per a la quina definim la interfície gràfica amb un botó per a refrescar la informació del servici i una ListView per a desplegar les dades del mateix.



```

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()
        }
    }
}

```



Declarem un ArrayList i un objecte de tipus MiServicio. Per a poder treballar amb el servei serà indispensable crear un objecte de tipus **ServiceConnection** **Línea 19 - 30** , per mitjà del qual realitzarem les funcions necessàries perquè nostra activitat pugui connectar-se amb el servei creat i podem així, accedir a la informació que ens proveïx. Este objecte serà inicialitzat a través del mètode

`bindService()` **Línea 32**, que és cridat en el `onCreate()` de l'activity. Com omplirem una ListView amb estes dades, és important crear un adaptador. Després, en el mètode propi `showServiceData()` actualitzem la llista amb les dades rebuts del servici **Línea 37**.

1. IMPORTANT, una vegada que hem acabat la lògica de l'aplicació ens faltará donar d'alta el servici creat, en l'arxiu AndroidManifest.xml

 **Reconstruïx l'exemple per a entendre el funcionament de la comunicació per mitjà de Bind.**

## BroadcastReceiver

Un broadcast receiver és el component que permet rebre i respondre davant d'esdeveniments del sistema (avís de bateria baixa, un SMS recibido,...) o esdeveniments produïts per altres aplicacions o servicis. Alguns esdeveniments són d'accés privilegiat i cal establir-los en l'arxiu de manifest de l'aplicació amb l'etiqueta `<uses_permission>`.

Per a crear un broadcast receiver simplement hem de crear una classe que herete de `BroadcastReceiver` i anul·lar el mètode `onReceive()`, que és el mètode que s'executarà cada vegada que es produïska l'esdeveniment a què se subscriga el nostre broadcast receiver (semblant a l'onCreate() d'una activity).

La classe BroadcastReceiver és capaç de rebre intents a través del mètode `sendBroadcast()`.

Altra de les classe útils és `PendingIntent`, que és un token que li donem a una altra aplicació a través d'un Notification Màner, Alarm Màner o altres aplicacions de tercers, que els permeten a estos components utilitzar els permisos de la nostra aplicació per a executar un tros de codi predefinit.

### Aplicació exemple per a activar la vibració i l'alarma del dispositiu

Definirem un broadcast receiver que escolte els canvis d'estat del telèfon. Volem tindre en la pantalla de la nostra aplicació un EditText per a introduir els segons que passaran abans de que ens vibre el telèfon i un botó per a iniciar el conteo. El layout de l'activitat principal s'omet per la seua senzillesa. Passem a explicar la classe que hereta 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))
    }
}

```

✦ Com veiem implementem el mètode **onReceive()** que rep el context des d'on és cridat i l'intent que el dispararà. Visualitzem un toast per a indicar que el temps d'espera va finalitzar i que el mode vibració s'activarà. A continuació activem la vibració durant huit segons. Per a això definim un objecte de tipus **Vibrator** que ho inicialitzem amb el servici propi del sistema i cridem al mètode **vibrate()** del dit objecte.

En l'activity principal només hem d'implementar la pulsació del botó, s'arreglarà la informació introduïda en l'EditText, es programarà una alarma i quan esta acabe s'invocarà al broadcast receiver definit anteriorment. Vegem el codi:

```

0  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()
13     val intent = Intent(this, BroadcastAlarma::class.java)
        val pendingIntent = PendingIntent.getBroadcast(this,
15                                     REQUESTCODE, intent, 0)
16     val alarmManager = getSystemService(ALARM_SERVICE) as AlarmManager
        alarmManager.set(AlarmManager.RTC_WAKEUP,
                        System.currentTimeMillis() + tiempo * 1000,
19                        pendingIntent)

        Toast.makeText(this, "Has fijado la alarma en "+tiempo+"segundos",
                        Toast.LENGTH_LONG).show()
    }
    companion object {
        val REQUESTCODE=1111
    }
}

```

📌 La primera cosa que fem és recuperar la referència a l'EditText i parseamos el contingut a Integer. Visualitzem un toast per a informar que es va a establir el compte arrere en els segons especificats en l'EditText.

!!**Lo interessant!!** Definim un intent que apunta al nostre broadcast receiver. Després fem ús d'un PendingIntent en el que s'arregla el context en què este objecte s'executarà quan siga atés, un codi d'identificació, l'intent on es defineix l'acció i un flag. Nosaltres ho fem amb **PendingIntent.getBroadcast()**, perquè el que hem definit o qui va atendre esta petició està en un BroadcastReceiver

**Líneas 13 - 15.** Usaríamos **nombrePendingIntent.getActivity()** si iniciàrem una activity i **nombrePendingIntent.getService()** per a un servicio.\*

Finalment creguem un objecte AlarmManager inicialitzant-ho amb el servici propi del sistema (ALARM\_SERVICE). Amb el mètode set() definim el temps en què serà llançada l'alarma (i segons després de l'actual) i el PendingIntent que resol la lògica de la nostra aplicació i que s'ha definit anteriorment **Líneas 16 - 19**.

⚠️ Per a provar esta aplicació és necessari un terminal real i no l'emulador, ja que este últim no pot reproduir la vibració.

 **Prova l'exemple del BroadcastReceiver**

 **Seguix l'exercici resolt que mostra un Toast indicant si s'ha connectado/desconnectado el dispositiu a l'alimentació**

## WorkManager

Android [WorkManager](#) és una biblioteca de processament que s'utilitza per a executar tasques en segon pla que haurien d'executar-se de forma garantida, però no necessàriament de forma immediata. Amb WorkManager podem posar en cua nostre processament en segon pla inclús quan l'aplicació no s'està executant o el dispositiu es reinicia per alguna raó. WorkManager també ens permet definir les restriccions necessàries per a executar la tasca, per exemple, la disponibilitat de la xarxa abans d'iniciar la tasca en segon pla.

Android WorkManager és part d'Android Jetpack i és perfecta per a usar quan la tasca:

1. No necessita executar-se en un moment específic
2. Pot ajornar-se la seua execució
3. Es necessita que s'execute inclús després que es tancament l'aplicació o es reinicie el dispositiu.
4. Ha de complir amb limitacions com el subministrament de bateria o la disponibilitat de la xarxa abans de l'execució.

Per a poder treballar amb **WorkManager** serà necessari incloure la implementació següent:

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

### Crear un treball

Per a crear un treball d'este tipus, s'haurà d'implementar una classe que herete de **Worker** i anul·lar el seu mètode **doWork ()** on afegirem el codi que vullguem que s'execute en segon pla (a diferència dels servicis, no caldrà crear cap fil ja que s'encarrega el sistema)

```

0 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 exemple crea una treball que realitza un retard de 5 segons i llança un Intent per a obrir una altra aplicació. Com es pot veure tot el codi està implementat en el mètode `doWork()` .

## Iniciar un treball

Este tipus de treballs permet definir si volem el fil com a tasca única o periòdica cada **x** temps. Per a això s'usen dos classes distintes `OneTimeWorkRequestBuilder()` i `PeriodicWorkRequestBuilder()` . Vegem exemples de la creació de treballs dels dos tipus:

```

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

```

En el cas de tasca única no hi ha res a explicar, en canvi de la periòdica podem dir que li entra com a primer argument l'interval que ha de passar fins que es torne a realitzar la següent iteració i el segon argument especifica les unitats de l'anterior (en este cas la tasca es repetirà cada 16 minuts) .

👉 Alguns punts importants a tindre en compte al treballar amb **WorkManager** son:

- La primera execució ocorre immediatament després que es complixen les restriccions mencionades per la sol·licitud de treball
- El treball es pot encadenar amb altres sol·licituds de treball.
- Si no es cancel·la es continuarà executant fins que acabe.

Además de l'anterior en cas de treballs de tipus periòdics, **PeriodicWorkRequest** , també es tindrà en compte que:

- El treball s'executa diverses vegades.
- Si no es cancel·la es continuarà executant periòdicament.
- La següent execució ocorre després que transcorrega el període de temps indicat en els arguments.
- L'interval mínim de repetició és de *15 minutos*.
- El treball no es pot encadenar amb altres sol·licituds de treball.
- L'interval de temps entre 2 intervals periòdics pot diferir segons les restriccions d'optimització del sistema operatiu.

Una vegada creat l'objecte, per a executar la sol·licitud de treball necessitem cridar al mètode **enqueue()** des d'una instància de **WorkManager** i passar el **WorkRequest** :

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

El mètode **enqueue()** posa en cua una o més **WorkRequests** perquè s'executen en segon pla.

## Funcionalitat de WorkManager

Les instàncies de **WorkManager** proporcionen una sèrie de funcionalitats per a manejar-nos amb les tasques. Per exemple, podem controlar si les tasques estan cancel·lades o cancel·lar-les si ens interessa, 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()
```

Quan cancel·lem un treball, este no es cancel·la automàticament sinó que és el sistema el que s'encarrega d'informar de la cancel·lació. Pel que serà tasca del programador atendre a la informació del sistema per a parar el treball. Açò es fa des de la classe que hereta de **WorkManager**, per exemple de les manera següent:

```
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()  
    }  
}
```

## Restriccions de WorkManager

Una de les avantatges que proporciona esta classe, és la de permetre [restringir](#) la funcionalitat de la tasca fins que es complisquen una sèrie de condicions que se li hagen associat. Logicamente, estes restriccions s'imposaran al ser necessaris alguns requisits per a la funcionalitat de l'app. Per exemple, si es necessita Internet per a realitzar una descàrrega de dades. En el següent codi s'han afegit totes les restriccions possibles:

```
//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())
```



## Comunicació amb WorkManager

Si volem comunicar informació a la tasca, ja siga com a dades d'entrada abans de la seua execució o com a dades d'eixida a l'acabar, s'utilitza la classe `Data.Builder`. La informació s'afeg a l'objecte per mitjà del ja conegut mètode del mapa clau-valor. Per a accedir a estes dades d'entrada dins del Worker, s'usa la propietat `inputData` dins del mètode `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()
    }
}
```

 **Exercici resolts nombres primers.**

 **Exercici proposat temporitzador amb vibració.**