

Anatomía de una Corrutina

Kotlin Coroutines

The Black Magic of Kotlin



Ignacio Cavallo



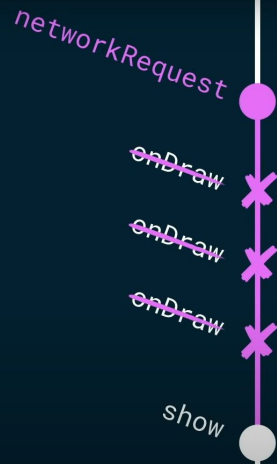
Reflexión

En la clase de hoy se nos propuso que trabajamos en grupos sobre las corrutinas. Al final de cuentas, mi grupo termine siendo yo solo, creo pertinente y por respeto al resto de los compañeros que si no vas a trabajar en grupo, mejor digas que estás de oyente. Más allá de eso, la verdad que estuvo bueno, por que me vi obligado a estudiar y tratar de entender el tema.

Código Sincrónico

Sync blocking

```
fun loadData() {  
    val data = networkRequest()  
    show(data)  
}
```



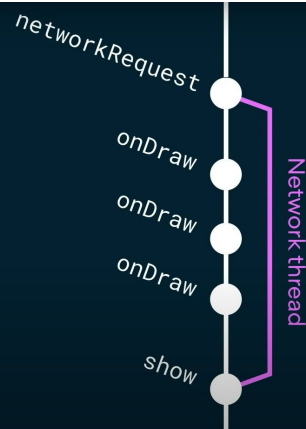
Una función sincrónica, bloqueará el hilo con el cual esté trabajando. Suponiendo que lo haga del hilo de la UI, el usuario experimentará un ANR hasta que termine la ejecución.

Código Asíncrono

Async

with coroutines

```
suspend fun loadData() {  
    val data = networkRequest()  
    show(data)  
}
```



Aunque sospechosamente se parecen la funciones, esta posee una sutil diferencia, el modificador *suspend*.

Suspend



Una corutina puede suspender la ejecución sin bloquear un hilo e incluso moverse a otro hilo. A esto se le llama el punto de suspensión.

Resume



Una vez finalizada el `networkRequest()`, la función `loadData` puede retomar la ejecución.

Suspend

```
suspend fun loadData() {  
    val data = networkRequest()  
    show(data)  
}  
  
suspend fun networkRequest(): Data =  
    withContext(Dispatchers.IO) {  
  
    }
```

Si examinamos la función de `networkRequest()`, podremos observar que es otra función con el modificador `suspend`. Una corrutina solo se puede ejecutar desde otra corrutina.

Suspend function must be called from a coroutine.

```
fun onClicked() {  
    loadData()  
}
```

Suspend fun 'loadData' must be called from a coroutine

```
suspend fun loadData() {  
    val data = networkRequest()  
    show(data)  
}
```

Observemos el siguiente error

Solución...?

```
fun onClicked() {  
    launch {  
        loadData()  
    }  
}  
  
suspend fun loadData() {  
    val data = networkRequest()  
    show(data)  
}
```

Para poder acceder al contenido de una corrutina podemos utilizar launch para acceder a la información. Pero pero....

Es realmente la Solución?

```
fun onClicked() {  
    launch {  
        loadData()  
    }  
}  
  
suspend fun loadData() {  
    val data = networkRequest()  
    show(data)  
}
```

Esto nos plantea las siguientes inquietudes,

Quién puede cancelar esta ejecución?

Sigue algún ciclo de vida?

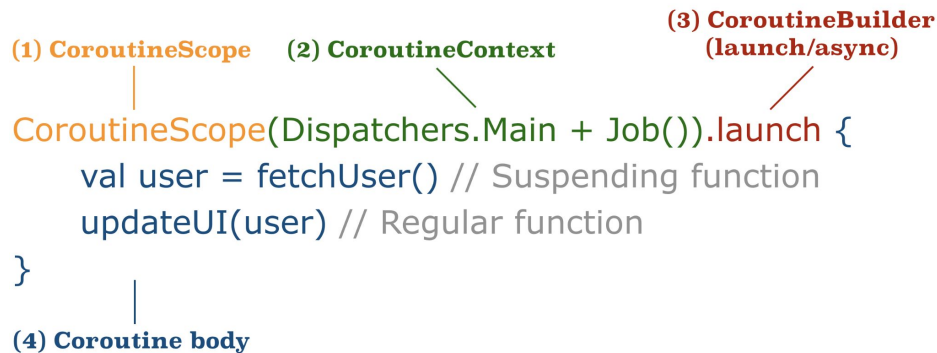
Quién recibe la excepción si falla?

Anatomía de una Corrutina

(1) **CoroutineScope** (2) **CoroutineContext** (3) **CoroutineBuilder**
(launch/async)

```
CoroutineScope(Dispatchers.Main + Job()).launch {  
    val user = fetchUser() // Suspending function  
    updateUI(user) // Regular function  
}
```

(4) **Coroutine body**



Existen 3 Componentes claves:

1- Scope

2- Context

3 - Builder

Scope

SupervisorJob

```
val scope = CoroutineScope(Job())
scope.launch(SupervisorJob()) {
    // coroutine -> can suspend
    launch {
        // Child 1
    }
    launch {
        // Child 2
    }
}
```

Job

Job

Job

Las Corrutinas siguen un patrón de diseño llamado Concurrencia Estructurada, implicando que estas solo pueden ser ejecutadas un Scope determinado definiendo el ciclo de vida de la misma. Por ende el Scope nos permitirá:

Cancelar las Corrutinas

Definir su ciclo de vida.

Notificar Errores.

Context

UI/Non-blocking	.Main
Network & Disk	.IO
CPU	.Default

Una Corrutina siempre se ejecuta dentro de un conexto y este a su vez contiene un dispatcher(despachante), que indica que subprocesso se utilizará. Estos pueden ser:

Dispatchers. Main

Dispatchers. IO

Dispatchers. Default

Builder. Launch/Async

Launch

Creates a new coroutine

Fire and Forget

```
scope.launch(Dispatchers.IO) {  
    loggingService.upload(logs)  
}
```

Async

Creates a new coroutine

Returns a value

```
suspend fun getUser(userId: String): User =  
    coroutineScope {  
        val deferred = async(Dispatchers.IO) {  
            userService.getUser(userId)  
        }  
        deferred.await()  
    }
```

Builder. Launch/Async

Launch

Creates a new coroutine

Fire and forget

Takes a dispatcher

Executed in a scope

Not a suspend function

Async

Creates a new coroutine

Returns a value

Takes a dispatcher

Executed in a scope

Not a suspend function

Habemus Corrutina

```
// MyViewModel.kt

val scope = CoroutineScope(Dispatchers.Main)

fun onClicked() {
    scope.launch {
        loadData()
    }
}
```

Ahora sí que sí, al fin tenemos una corrutina. Esta posee:

- 1- Un Scope
- 2 - Un Contexto con su Dispatcher.Main
- 3 - Un Builder de tipo launch

Resumen Anatomía de una Corrutina

