Programación asíncrona en Android

Threads (Parte II)

{desafío} latam_

Utilizar elementos de la programación asíncrona acorde al lenguaje Kotlin para dar solución a un problema

- Unidad 1: Kotlin para el desarrollo de aplicaciones.
- Unidad 2: Ciclo de vida de componentes.
- Unidad 3: Arquitectura en Android.
- Unidad 4: Programación asíncrona en Android.

Te encuentras aquí

{desafío} latam_

¿Qué aprenderás en esta sesión?

- *Reconoce los aspectos fundamentales del patrón de promesas para la ejecución asíncrona.*

¿Has escuchado los términos “background threads” o “foreground threads”?

/* Código asíncrono y bloqueante */

Kotlin runBlocking

- Ya vimos que se puede crear una función que se ejecuta de forma asíncrona. Esto permite que nuestra app realice varios procesos al mismo tiempo, por ejemplo, enviar y recibir mensajes desde una API y, al mismo tiempo, mostrar esos mensajes en pantalla junto con guardarlos en la base de datos local.

- Sin embargo, es posible que existan casos en los que quieras que uno de esos procesos se ejecute y no permita que nada más se ejecute en el mismo thread hasta que este termine.
- Para esto existe **`runBlocking`**, que es un tipo de función muy especial.

Ejercicio "runBlocking"

Kotlin runBlocking

Copia el siguiente ejercicio y luego ejecuta la aplicación, una vez que la app se esté ejecutando, presiona el botón "**Start**", podrás apreciar como **runBlocking** "bloquea" todo el Main Thread.

```
class MainActivity : AppCompatActivity() {

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

        val btnStart = this.findViewById(R.id.btn_start) as AppCompatButton
        btnStart.setOnClickListener {
            runBlockingCode()
        }
    }

    private fun runBlockingCode() {
        runBlocking {
            delay(10000L)
            Toast.makeText(applicationContext, "Hola Mundo", Toast.LENGTH_LONG).show()
        }
    }
}
```

Explicación: runBlocking

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val btnStart =
this.findViewById(R.id.btn_start) as AppCompatButton
        btnStart.setOnClickListener {
            runBlockingCode()
        }
    }

    private fun runBlockingCode() {
        runBlocking {
            delay(10000L)
            Toast.makeText(applicationContext, "Hola
Mundo", Toast.LENGTH_LONG).show()
        }
    }
}
```

En este ejemplo, la función **`runBlockingCode()`** ejecuta un **`runBlocking`** el cual tiene un delay de 3 segundos, esta función se ejecuta cuando el usuario presiona el botón start.

Por la forma en que funciona `runBlocking`, en este caso, bloquea el **MainThread**, lo que impide que cualquier otro proceso se ejecute antes de que este finalice. El usuario percibirá que la aplicación pareciera estar congelada. He ahí la importancia de **`runBlocking`**, se debe ser muy cuidadoso ocupando esta función, especialmente seleccionando el thread correcto.

Una vez que **`runBlocking`** termina su ejecución, mostrará al usuario un Toast con el mensaje “Hola Mundo”

/* Background thread y Main thread */

Background y Main thread

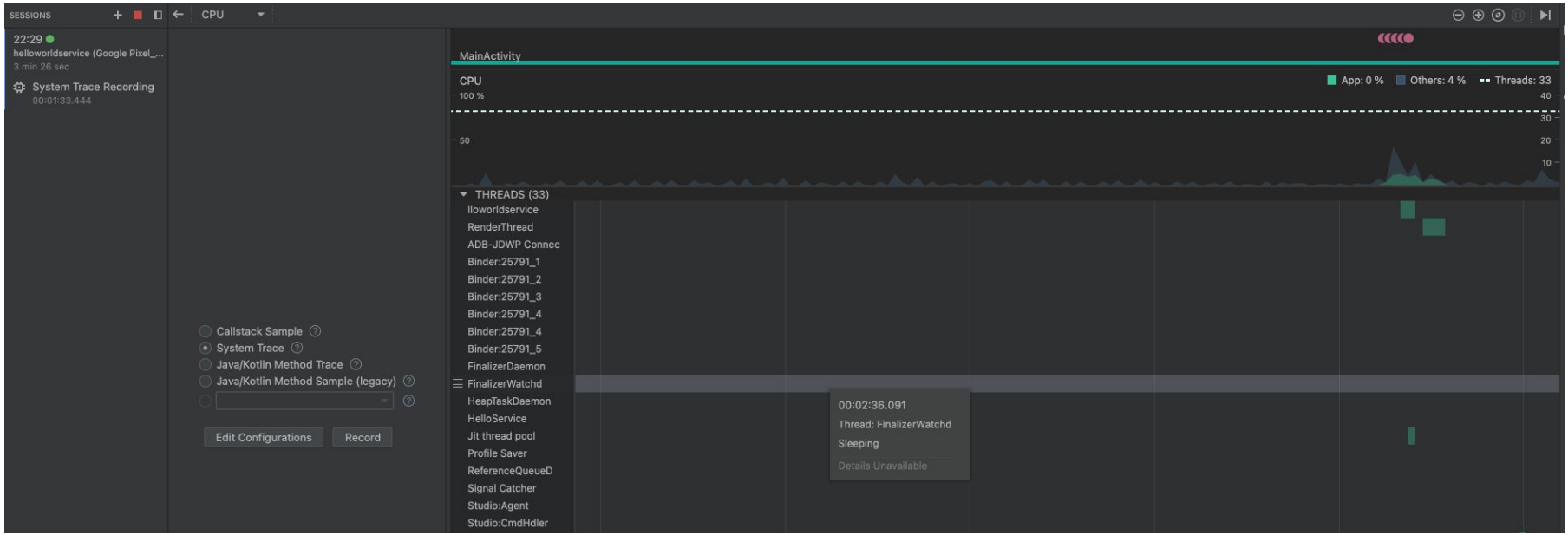
Main thread o **UI Thread** se encarga de administrar todas las interacciones del usuario (inputs y outputs), por lo que se debe evitar hacer cualquier operación que consuma mucho tiempo. Por defecto, Android no permitirá una operación que consuma más de 5 segundos, en caso contrario generará un "Application Not Responding" ANR.

Background Thread es utilizado para ejecutar tareas que pueden tomar tiempo en completar o que puedan no ser completadas y sin bloquear la interfaz de usuario y así sin comprometer la performance de la app.

Estas tareas pueden ser: API o HTTP request, consultas a bases de datos, decodificar imágenes, etc.

Android Studio Tip

Android Studio ofrece una herramienta la cual te permitirá ver todos los threads activos en tu app y otras opciones más. Para esto debes ejecutar el **Profiler**, luego click en “Session”, selecciona tu app, luego cuando ya puedas ver el Profiler ejecutandose, presione en la file “CPU”



{desafío} latam_

/* Uso del background thread */

Background thread y HTTP requests

Podemos utilizar backgrounds threads para hacer request a una Url. En el siguiente ejemplo utilizaremos la librería estándar de Kotlin:

```
val url = "http://www.google.cl"
thread {
        val json = try {
            URL(url).readText()
        } catch (e: Exception) {
            return@thread
        }
        runOnUiThread { displayOrWhatever(json) }
    }
```

Background thread y HttpURLConnection

Similar al ejemplo anterior, en este caso haremos un llamado a una Url la cual retorna un objeto Json, la función es muy simple:

```
private fun testSomething() {
        val url =
URL("https://dummyjson.com/products/1")
        (url.openConnection() as?
HttpURLConnection)?.run {
            requestMethod = "GET"
            val data  =
inputStream.bufferedReader().readText()
            Log.d("MainActivity", "testSomething:
$data")
        }
    }
```

Sin embargo, debemos recordar que Android no permitirá que se ejecuten funciones como estas en el Main Thread, lo que podemos hacer es especificar ejecutaremos la función en un nuevo thread

```
thread {
    testSomething()
}
```

Background thread y Glide

Otro ejemplo de uso de background threads es **Glide**, es una librería para el manejo de imágenes, es muy eficiente, simple y fácil de usar:

```
Glide
    .with(context)
    .load(url)
    .centerCrop()
    .placeholder(R.drawable.loading_spinner)
    .into(myImageView);
```

Background thread y Room

Actualmente, esta es la forma en que se define una base de datos (Room) en una app:

```
@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

Por defecto todo ocurre en background thread, sin embargo, en versiones previas era posible permitir a Room correr en el Main Thread.

```
Room.databaseBuilder(
context.applicationContext,
LocalDatabase::class.java,
"local_db"
).allowMainThreadQueries().build()
```

El uso del background thread permite realizar tareas que no son visibles para el usuario. Sin embargo, recuerda que hay threads con runBlocking los cual pueden congelar la app causando ANR

Próxima sesión...

- *Guía de ejercicios*

{desafío}
latam_

{desafío}
latam_

Academia de
talentos digitales