Guía de ejercicios - Programación asíncrona en Android (III)

 ¡Hola! Te damos la bienvenida a esta nueva guía de estudio.

¿En qué consiste esta guía?

La siguiente guía de estudio tiene como objetivo practicar y ejercitar los contenidos que hemos visto en clase.

¡Vamos con todo!

Tabla de contenidos

Actividad guiada N°: Job cancelable para controlar estados de UI	**2**
¡Manos a la obra! - Dispatchers y Jobs	**5**
Dispatchers - Soluciones:	**7**
Preguntas de proceso	**9**
Preguntas de cierre	**9**

¡Comencemos!

Actividad guiada N°: Job cancelable para controlar estados de UI

En esta actividad guiada modificaremos nuestro proyecto de coroutines para hacer que la consulta a la base de datos sea cancelable.

Antes de empezar, descarga el archivo **Apoyo guía de ejercicios - Programación asíncrona en Android (III)**

Empecemos!

Actualmente `TaskViewModel` se ve de la siguiente forma:

```
@HiltViewModel
class TaskViewModel @Inject constructor(
    private val repository: TaskRepositoryImp
) : ViewModel() {

    private val _data: MutableStateFlow<List<TaskEntity>> =
        MutableStateFlow(emptyList())
    val taskListStateFlow: StateFlow<List<TaskEntity>> =
        _data.asStateFlow()

    init {
        viewModelScope.launch {
            repository.getTasks().collectLatest {
                _data.value = it
            }
        }
    }
}
```

Lo primero que debemos hacer es crear el job, el scope y el dispatcher que usaremos en la coroutine.

```
private val job = Job()
private val dispatcherIO: CoroutineDispatcher = Dispatchers.IO
private val coroutineScope = CoroutineScope(job + dispatcherIO)
```

Luego movemos la coroutine a una función fuera del init, y obtenemos el valor que retorna, el cual es un job. Recuerda que ya que estamos usando `coroutineScope` debemos crear una suspend function.

```
suspend fun getTasks() {
    val getTasksJob = coroutineScope.launch {
        repository.getTasks().collectLatest {
           _data.value = it
        }
    }
}
```

Ahora se supone que la idea utilizar un job es poder controlar de mejor forma la coroutine, algo que podemos hacer es crear una sealed class la cual pueda manejar estados de UI, como por ejemplo:

```
sealed class UIState {
    data class Success(val state: Boolean) : UIState()
    data class Error(val ex: Throwable) : UIState()
}
```

Luego, en la función `getTasks()` podemos hacer que, dependiendo del resultado de `getTasksJob`, cambiar el estado de UI en el MainActivity.

```
suspend fun getTasks() {
    val getTasksJob = coroutineScope.launch {
        repository.getTasks().collectLatest {
            _data.value = it
        }
    }
  when {
     getTasksJob.isActive -> _uiState.value = UIState.Success(true)
     getTasksJob.isCompleted -> _uiState.value =UIState.Success(true)
     getTasksJob.isCancelled -> _uiState.value =
         UIState.Error(Throwable("Cancelado"))
```

```
    else -> _uiState.value=UIState.Error(Throwable("algo salio mal!"))
  }
}
```

El resultado del job se guarda en la variable `_uiState`, la cual es de tipo mutable, ya que no debemos exponer este tipo de variables desde el viewmodel, creamos una variable que sea visible desde `MainActivity` y que nos permita guardar los cambios de estado de `_uiState`

```
private val _uiState: MutableStateFlow<UIState?> =
MutableStateFlow(null)

val uiState: StateFlow<UIState?> = _uiState.asStateFlow()
```

A continuación podemos leer estos cambios de estados desde MainActivity.

```
private fun getUIState() {
    lifecycleScope.launchWhenCreated {
        viewModel.uiState.collectLatest { state ->
            state?.let {
                 when (it) {
                     is TaskViewModel.UIState.Success ->
                         Toast.makeText(
                            this@MainActivity,
                            "OK",
                            Toast.LENGTH_SHORT
                        ).show()
                     is TaskViewModel.UIState.Error ->
                        Toast.makeText(
                            this@MainActivity, "Error",
                            Toast.LENGTH_SHORT
                        ).show()
                    }
                }
            }
        }
    }
```

En este caso, si el job finaliza con éxito, mostramos un mensaje tipo Toast indicando que está “OK”, en un escenario más real se puede hacer que, en caso de error, se esconda el RecyclerView y se muestre una pantalla de error.

Ahora ejecuta el proyecto y ve el resultado, en caso de tener algún error, puedes descargar el archivo: **Solución guía de ejercicios - Programación asíncrona en Android (III)**

¡Manos a la obra! - Dispatchers y Jobs

La siguiente suspend function imprime un mensaje en la terminal indicando en qué thread se ejecutó, luego espera 3 segundos e imprime otro mensaje indicando el thread en el que terminó.

```
suspend fun longRunningTask() {
    println("inicia en...: ${Thread.currentThread().name}")
    delay(3000)
    println("termina en ...: ${Thread.currentThread().name}")
}
```

Preguntas:

1. Usando la coroutine `runBlocking`, crea una función que permita ejecutar la función "`longRunningTask`" en el Main Thread.
2. Repite el paso anterior, pero esta vez ejecuta la misma con `Dispatchers.IO`
3. Repite el paso anterior, pero esta vez ejecuta la misma con `Dispatchers.Default`
4. Repite el paso anterior, pero esta vez ejecuta la misma con `Dispatchers.Unconfined`.

Sugerencia: puedes crear una archivo kotlin y ejecutar directamente desde la función main (`fun main(){ }`)

Encuentra el error:

1. Copia el código, ejecutalo y sigue las instrucciones para poder entender el problema:

```
fun main(): Unit = runBlocking {
    val job = launch {
        longRunningTask()
    }
    println("job status : ${job.status()}")
    delay(1000L)
```

```
    job.cancelAndJoin()
    println("job status : ${job.status()}")
}

suspend fun longRunningTask() {
    println("inicia en: ${Thread.currentThread().name}")
    delay(2000)
    println("termina en: ${Thread.currentThread().name}")
}

fun Job.status(): String = when {
    this.isActive -> "isActive"
    this.isCompleted -> "isCompleted"
    this.isCancelled -> "isCancelled"
    else -> "Nothing"
}
```

Se requiere que entregue el siguiente resultado:

```
job status : isActive
inicia en: main
termina en: main
job status : isCompleted
```

Sin embargo, el resultado actual es el siguiente:

```
job status : isActive
inicia en: main
job status : isCompleted
```

Dispatchers - Soluciones:

1. Ejercicio 1

```
fun main() = runBlocking {
    longRunningTask()
}

suspend fun longRunningTask() {
    println("inicia en...: ${Thread.currentThread().name}")
    delay(3000)
    println("termina en ...: ${Thread.currentThread().name}")
}
```

2. Ejercicio 2:

```
fun main() = runBlocking(Dispatchers.IO) {
    longRunningTask()
}

suspend fun longRunningTask() {
    println("inicia en...: ${Thread.currentThread().name}")
    delay(3000)
    println("termina en ...: ${Thread.currentThread().name}")
}
```

3. Ejercicio 3:

```
fun main() = runBlocking(Dispatchers.Default) {
    longRunningTask()
}

suspend fun longRunningTask() {
    println("inicia en...: ${Thread.currentThread().name}")
    delay(3000)
    println("termina en ...: ${Thread.currentThread().name}")
}
```

4. Ejercicio 4:

```
fun main() = runBlocking(Dispatchers.Unconfined) {
    longRunningTask()
}

suspend fun longRunningTask() {
    println("inicia en...: ${Thread.currentThread().name}")
    delay(3000)
    println("termina en ...: ${Thread.currentThread().name}")
}
```

5. Solucion para “Encuentra el error”:

```
fun main(): Unit = runBlocking {

    val job = launch {
        longRunningTask()
    }
    println("job status : ${job.status()}")
    delay(2100L)
    job.cancelAndJoin()
    println("job status : ${job.status()}")
}

suspend fun longRunningTask() {
    println("inicia en: ${Thread.currentThread().name}")
    delay(2000)
    println("termina en: ${Thread.currentThread().name}")
}

fun Job.status(): String = when {
    this.isActive -> "isActive"
    this.isCompleted -> "isCompleted"
    this.isCancelled -> "isCancelled"
    else -> "Nothing"
}
```

Preguntas de proceso

Reflexiona:

- ¿Te resultó difícil alguno de los ejercicios de esta guía?
- ¿Cual es tu opinión respecto a la programación asíncrona?
- ¿Por qué crees que se inventó la programación asíncrona?
- ¿Te hace sentido el uso de distintos tipos de Dispatchers?

Preguntas de cierre

- ¿Qué pasa si ejecuto varias tareas dentro de un Job() y una de estas tareas falla?
- ¿Cual es la diferencia entre SupervisorJob()?
- ¿Debería usar SupervisorJob() siempre?
- ¿Que sucedería si uso Dispatchers.Main para hacer un HTTP Request?