Guía de ejercicios - Programación asíncrona en Android (II)

 ¡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

¡Comencemos!

Actividad guiada N°: Coroutine + StateFlow (parte 1 - Mostrar lista)

En la siguiente actividad combinaremos conceptos aprendidos durante el curso, en particular Coroutines y StateFlow, también seguiremos el patrón de diseño Repository y los consejos y buenas prácticas sugeridas hasta ahora.

Para empezar, descarga el proyecto **Coroutine + StateFlow (parte 1 - Mostrar lista).zip** ubicado en la sección **Apoyo guía de ejercicios - Programación asíncrona en Android (II)** en el LMS.

El proyecto es una versión simple, un ToDo App, el cual actualmente solo tiene una base de datos con algunos datos precargados.

Lo que haremos es crear una función la cual nos permita leer la lista de tareas desde la base de datos y mostrarlas en un RecyclerView.

¡Empecemos!

El proyecto usa inyección de dependencias (Hilt), pero no es necesario realizar cambios de ese tipo, por lo que podemos ignorarlo.

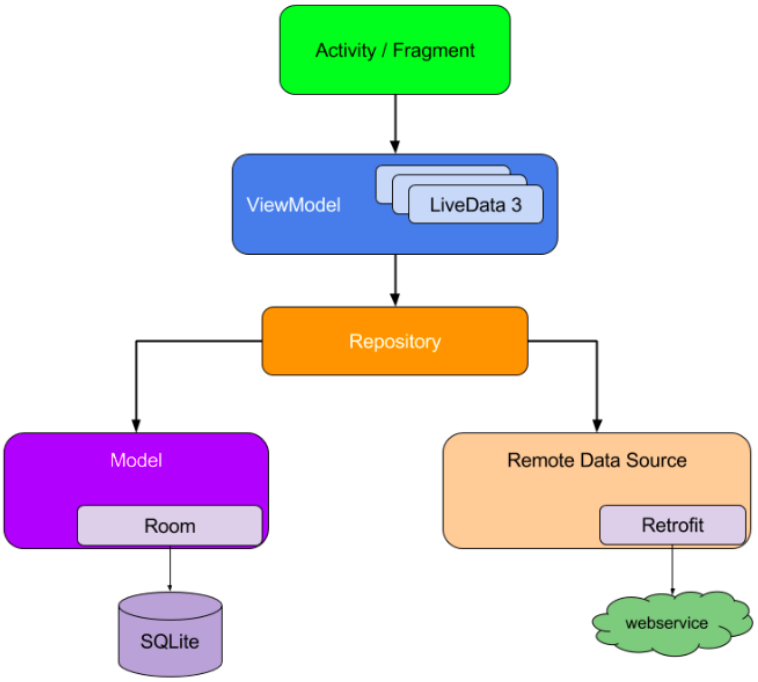
```
lateinit var binding: ActivityMainBinding
private val viewModel: TaskViewModel by viewModels()

@Inject
lateinit var adapter: TaskAdapter
```

El viewModel del proyecto está actualmente vacío, así que primero debemos crear una función en el viewModel la cual nos permite traer los datos desde la base de datos.

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

Podemos ver que se está inyectando una clase repository en el viewmodel, lo que significa que los datos los podemos obtener desde ese repository. Recordemos lo aprendido en patrones de diseño.



En nuestro caso solo tenemos datos locales, por lo que podemos ignorar todo lo relacionado a "Remote Data Source".

Continuemos con `TaskRepositoryImp`, si analizamos el contenido de esa clase podemos ver que la conexión con la base de datos ya existe, así como también algunas funciones:

```
class TaskRepositoryImp @Inject constructor(
    private val taskDao: TaskDao
) : TaskRepository {

    override suspend fun getTasks(): Flow<List<TaskEntity>> =
        taskDao.getTasks()

    override suspend fun addTask(task: TaskEntity) =
        taskDao.addTask(task)

    override suspend fun deleteTask(task: TaskEntity) =
        taskDao.deleteTask(task.id)
}
```

Podemos ver que **`getTasks()`** es una suspend function la cual retorna un Flow de una lista de TaskEntity, esto es ideal, ya que proporciona todo lo necesario para implementar una coroutine que puede leer los datos y mostrarlos en el recyclerview.

Volvamos al viewmodel, ya sabemos que `getTasks()` retorna un flow y queremos que cuando se inicie la app, esta muestre la lista de tasks y, ya que no requiere parámetros adicionales, podemos llamarla en el init del viewmodel.

```
init {
    viewModelScope.launch {
    // aquí va la función
    }
}
```

Puesto que estamos en el viewmodel usamos viewModelScope.

`repository.getTasks()` retorna `Flow<List<TaskEntity>>` así que podemos usar `collectLatest { }`

```
init {
    viewModelScope.launch {
        repository.getTasks().collectLatest {
        }
    }
}
```

Con **`collectLatest`** podemos obtener **`List<TaskEntity>`**, ahora solo nos queda tomar esa lista y pasarla a una variable tipo StateFlow.

Creamos una variable tipo `MutableStateFlow` que nos permita guardar la lista proveniente desde la base de datos.

```
val _data: MutableStateFlow<List<TaskEntity>> =
MutableStateFlow(emptyList())
```

Recordemos que una de las buenas prácticas de StateFlow y LiveData dice "*jamás exponer objetos mutables desde el viewmodel*", para evitar eso, creamos una variable tipo StateFlow y le pasamos los datos guardados en "**`_data`**", así que pasamos de:

ANTES:

```
val _data: MutableStateFlow<List<TaskEntity>> =
MutableStateFlow(emptyList())

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

DESPUÉS:

```
private val _data:
    MutableStateFlow<List<TaskEntity>> =MutableStateFlow(emptyList())

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

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

Ahora solo nos falta llamar **taskListStateFlow** desde el activity. Para esto creamos una función que nos permita llamar a una coroutine, ya que estamos en un activity debemos usar `lifecycleScope`.

```
private fun getTaskList() {
    lifecycleScope.launchWhenCreated {
        viewModel.taskListStateFlow.collectLatest {

        }
    }
}
```

Creamos una función para inicializar el recyclerview, la cual recibirá una lista de `TaskEntity`.

```
private fun initRecyclerView(taskList: List<TaskEntity>) {
    adapter = TaskAdapter()
    adapter.taskList = taskList
    binding.rvTask.layoutManager = LinearLayoutManager(this)
    binding.rvTask.adapter = adapter
}
```

Y finalmente llamamos esta función después que recibamos los datos provenientes de la base de datos, y a su vez llamamos a esta función desde el método `onCreate`

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)
    getTaskList()
}

private fun getTaskList() {
    lifecycleScope.launchWhenCreated {
        viewModel.taskListStateFlow.collectLatest {
            initRecyclerView(it)
        }
    }
}
```

Ejecuta el proyecto, si todo está correcto debería mostrar una lista de objetos en el recyclerview.

Puedes descargar el proyecto con el código terminado en la sección **Solución guía de ejercicios - Programación asíncrona en Android (II)** para poder comparar.

¡Manos a la obra! - Coroutine + StateFlow (parte 2 - Borrar item)

En esta actividad deberás continuar trabajando con el proyecto anterior y agregar una funcionalidad.
Se requiere que cuando el usuario haga long press en uno de los ítems, este se borre, no es necesario mostrar una advertencia al usuario.

Descarga el proyecto **Coroutine + StateFlow (parte 2 - Borrar item).zip** en la sección **Apoyo guía de ejercicios - Programación asíncrona en Android (II)** del LMS.

Sugerencias:

Crea una variable en el adapter como la siguiente:

```
var onLongClick: ((TaskEntity) -> Unit)? = null
```

Luego la podrás llamar en el evento `setOnLongClickListener` de la siguiente forma:

```
onLongClick?.invoke(task)
```

Y luego la podrás llamar desde la función `initRecyclerView()`

Recuerda que debes crear una función en el viewModel, la cual a su vez llamará a la función deleteTask en el repository

```
repository.deleteTask(task)
```

Preguntas de proceso

Reflexiona:

- ¿Existió algún ejercicio que se te hizo difícil de comprender?
- ¿Qué opinas de las coroutines hasta ahora?
- ¿Has trabajado anteriormente con funciones asíncronas?
- ¿Puedes nombrar al menos 2 momentos o situaciones en los cuales el contenido aprendido te sea de utilidad?

Preguntas de cierre

- ¿Por qué se recomienda no exponer objetos mutables desde el viewmodel?
- ¿Por qué en el viewmodel y en el activity usan distintos scopes (`lifecycleScope`, `viewModelScope`)?
- ¿Podemos reemplazar los scope con `GlobalScope`?
- ¿Cual es la diferencia entre `launchWhenCreated`, `launchWhenStarted` y `launchWhenResumed` ?