Programación asíncrona en Android

Buenas prácticas para la elaboración de coroutines

{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?

- *Utilizar Coroutines para el manejo asíncrono acorde al lenguaje Kotlin*

¿Ya sabes crear coroutines?

Ahora veremos cómo hacerlo correctamente

/* Coroutines para el manejo asíncrono acorde al lenguaje Kotlin */

Buenas prácticas de Coroutines

- Suspend functions deben ser seguras para llamarlas desde el hilo principal
- El ViewModel debería crear coroutines.
- No expongas tipos mutables
- La capa de datos y lógica de negocio debería exponer suspend functions y Flows
- Inyectar Dispatchers
- Evita usar GlobalScope
- Haz que tu coroutine sea cancelable
- Cuidado con las excepciones

Suspend functions deben ser seguras para llamarlas desde el hilo principal

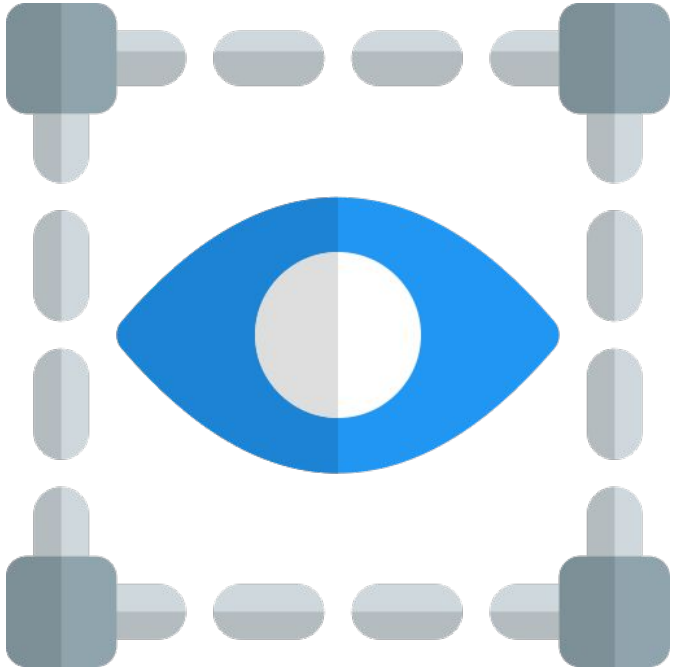
Suspend functions deben ser seguras para el sistema principal, lo que significa que es seguro llamarlas desde el main thread.

Si una clase realiza operaciones de bloqueo de ejecución prolongada en una rutina, está a cargo de mover la ejecución fuera del subproceso principal mediante withContext.

Esto se aplica a todas las clases de su aplicación, independientemente de la parte de la arquitectura en la que se encuentre la clase.

El ViewModel debería crear coroutines

Las clases de ViewModel deberían preferir crear corrutinas en lugar de exponer funciones de suspensión para realizar la lógica comercial. Las funciones de suspensión en ViewModel pueden ser útiles si, en lugar de exponer el estado mediante un flujo de datos, solo se necesita emitir un único valor.



No expongas tipos mutables

Prefiere exponer tipos inmutables a otras clases. De esta manera, todos los cambios en el tipo mutable se centralizan en una clase, lo que facilita la depuración cuando algo sale mal.

No exponer tipos Mutable

```
class LatestNewsViewModel : ViewModel() {
    val uiState = MutableStateFlow(LatestNewsUiState.Loading)
    /* ... */
}

class LatestNewsViewModel : ViewModel() {
    private val _uiState = MutableStateFlow(LatestNewsUiState.Loading)
    val uiState: StateFlow<LatestNewsUiState> = _uiState
    /* ... */
}
```

La capa de datos y lógica de negocio debería exponer suspend functions y Flows

Las clases en las capas de datos y lógica de negocio generalmente exponen funciones para realizar llamadas únicas o para recibir notificaciones de cambios de datos a lo largo del tiempo. Las clases en esas capas deben exponer suspend functions para llamadas únicas y flujo para notificar sobre cambios de datos.

Para las clases en la capa de datos o lógica de negocio que necesitan crear coroutines por diferentes motivos, existen diferentes opciones.

Si el trabajo a realizar en esas corrutinas es relevante solo cuando el usuario está presente en la pantalla actual, debe seguir el ciclo de vida de la persona que llama. En la mayoría de los casos, la persona que llama será ViewModel. En este caso, se debe usar **`coroutineScope`** o **`supervisorScope`**.

Un ejemplo de lo mencionado anteriormente es el siguiente:

```
suspend fun getBookAndAuthors(): BookAndAuthors {
    return coroutineScope {
        val books = async(defaultDispatcher) {
            booksRepository.getAllBooks()
        }
        val authors = async(defaultDispatcher) {
            authorsRepository.getAllAuthors()
        }
        BookAndAuthors(books.await(), authors.await())
    }
}
```

Inyectar Dispatchers

No codifique (hardcode) despachadores cuando cree nuevas coroutines o llames a withContext.

Esto no está bien:

```
class NewsRepository {
  suspend fun loadNews() =
withContext(Dispatchers.Default) {}
}
```

Esto está bien:

```
class NewsRepository(
  private val defaultDispatcher:
CoroutineDispatcher = Dispatchers.Default)
{
  suspend fun loadNews() =
withContext(defaultDispatcher) {}
}
```

Evita usar GlobalScope

Similar a “Inyectar Dispatchers”. Al usar GlobalScope, está codificando el `CoroutineScope` que usa una clase, lo que trae algunas desventajas:

- Promueve valores de codificación rígida (hardcode). Si codifica `GlobalScope`, es posible que también esté codificando Dispatchers.
- Hace que las pruebas sean muy difíciles, ya que su código se ejecuta en un ámbito no controlado, no podrá controlar su ejecución.
- No puede tener un `CoroutineContext` común para ejecutar para todas las coroutines integradas en el ámbito mismo.

En su lugar, considera inyectar un `CoroutineScope` para el Job que debe sobrevivir al alcance actual.

Haz que tu coroutine sea cancelable

La cancelación en las coroutines es cooperativa, lo que significa que cuando se cancela el trabajo de una coroutine, la coroutine no se cancela hasta que se suspende o verifica la cancelación. Si realiza operaciones de bloqueo en una rutina, asegúrese de que la coroutine sea cancelable.

Por ejemplo, si estás leyendo varios archivos del disco, antes de comenzar a leer cada archivo, verifique si se canceló la coroutine. Una forma de comprobar la cancelación es llamando a la función **`ensureActive()`**.

Haz que tu coroutine sea cancelable - Ejemplo

Casi auto explanatorio, **`ensureActive()`** revisa si la coroutine está aún activa:

```
someScope.launch {
    for(file in files) {
        ensureActive() // Check for cancellation
        readFile(file)
    }
}
```

Cuidado con las excepciones

Las excepciones no controladas lanzadas en las coroutines pueden hacer que tu app se bloquee. En caso de ser probable que ocurran excepciones, entonces trata de controlarlas en el cuerpo de cualquier coroutine creada con `viewModelScope` o `lifecycleScope`.

```
fun login(username: String, token: String) {
    viewModelScope.launch {
        try {
            loginRepository.login(username, token)
        } catch (exception: IOException) {
            // Maneja la excepción aquí
        }
    }
}
```

Recuerda que las buenas prácticas pueden parecer un poco tediosas, pero si las sigues desde el principio te pueden ahorrar mucho tiempo cuando tu proyecto empieza a crecer

Próxima sesión...

- *Guía de ejercicios*

{desafío}
latam_

{desafío}
latam_

Academia de
talentos digitales