Bloque 2 · Tema 2

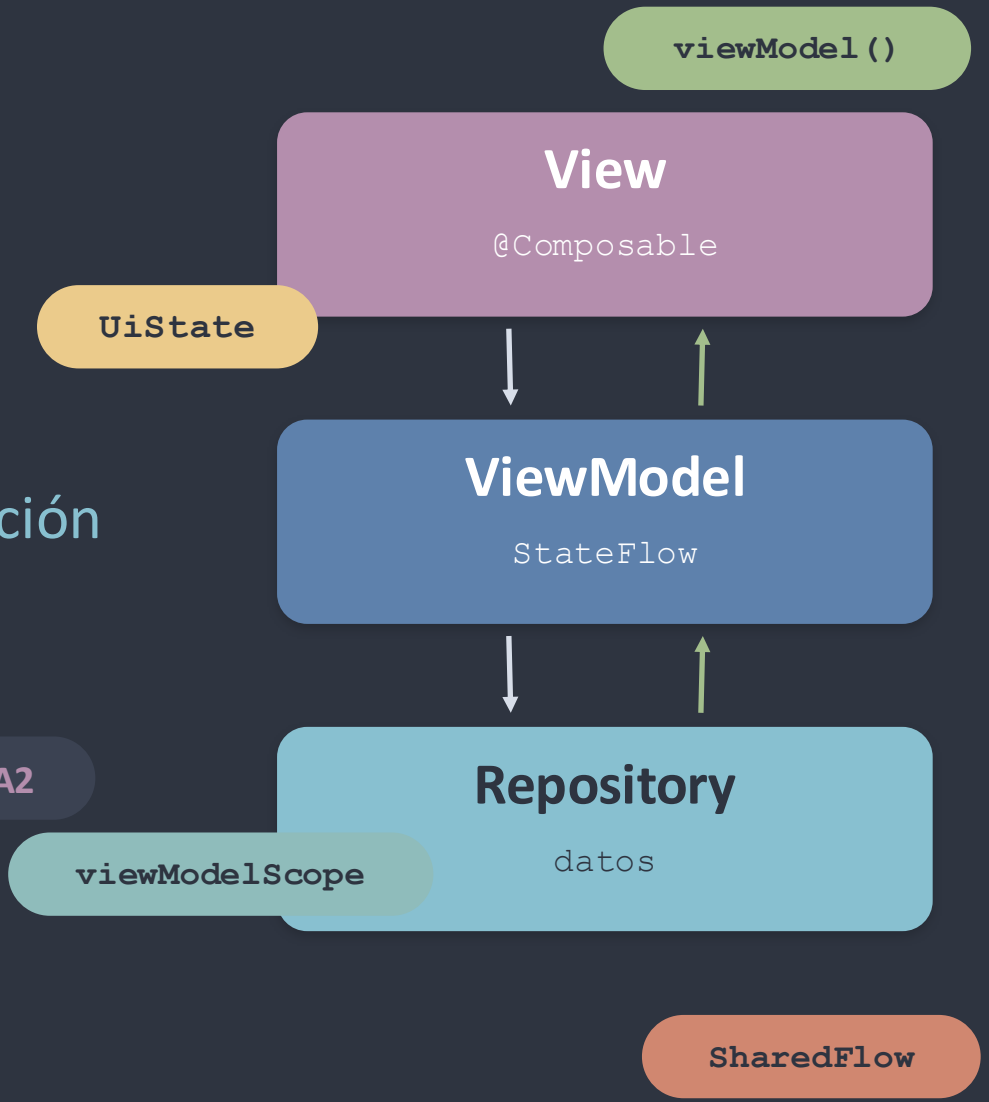
MVVM

y ViewModel: arquitectura de la capa de presentación con Kotlin y Jetpack Compose

2,5 horas

AppDummy · biblioteca personal

RA2



Hoja de ruta del tema

8 apartados

1

El problema

Por qué remember no basta

2

Patrón MVVM

Capas y flujo unidireccional

3

ViewModel

Ciclo de vida y Factory

4

UiState y StateFlow

Estado observable y sealed class

5

Clean Architecture

Dependencias y paquetes

6

SharedFlow

Eventos de un solo disparo

7

Testing

ViewModel sin Android

8

Práctica guiada

Arquitectura de AppDummy

¿Qué vamos a evaluar?

RA2

RA2

Desarrolla aplicaciones para dispositivos móviles analizando y empleando las tecnologías y librerías específicas.



Arquitectura de la capa de presentación

RA2-a — **Estructura de clases de la aplicación**

MVVM, capas y paquetes

RA2-b — **Ventanas, menús, alertas y controles**

PantallaListado, chips y Snackbar

RA2-g — **Pruebas de interacción usuario-aplicación**

Tests unitarios del ViewModel

RA2-i — **Documentar el proceso de desarrollo**

Diagrama de arquitectura de AppDummy

Dependencias necesarias

Gradle KTS

build.gradle.kts (módulo app)

```
// build.gradle.kts (app) — añadir a las del Bloque 1
dependencies {
    implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.11.0")
    implementation("androidx.lifecycle:lifecycle-runtime-compose:2.11.0")
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.10.2")
    testImplementation("org.jetbrains.kotlinx:kotlinx-coroutines-test:1.10.2")
}
```

lifecycle-viewmodel-compose

viewModel() y viewModelFactory { }

lifecycle-runtime-compose

collectAsStateWithLifecycle()

coroutines-android

viewModelScope.launch, delay

coroutines-test

runTest y TestDispatcher

⚠ **Todos los artefactos lifecycle-* deben compartir la misma versión.**

01

Arquitectura

Por qué la necesitamos y cómo se organiza MVVM



El problema: la pantalla lo hace todo

Antipatrón

PantallaListadoMal.kt

```
// ✕ Lógica de negocio mezclada con la UI
@Composable
fun PantallaListadoMal() {
    var libros by remember {
        mutableStateOf<List<String>>(emptyList())
    }
    var cargando by remember {
        mutableStateOf(false)
    }

    LaunchedEffect(Unit) {
        cargando = true
        libros = cargarDatosDeAlgunSitio()
        cargando = false
    }
}
```



Pérdida de estado

Al rotar, Android destruye y recrea la Activity: el estado de remember se reinicia.



Responsabilidades mezcladas

La UI decide qué datos cargar, cómo cargarlos y cómo mostrarlos.



Imposible de testear

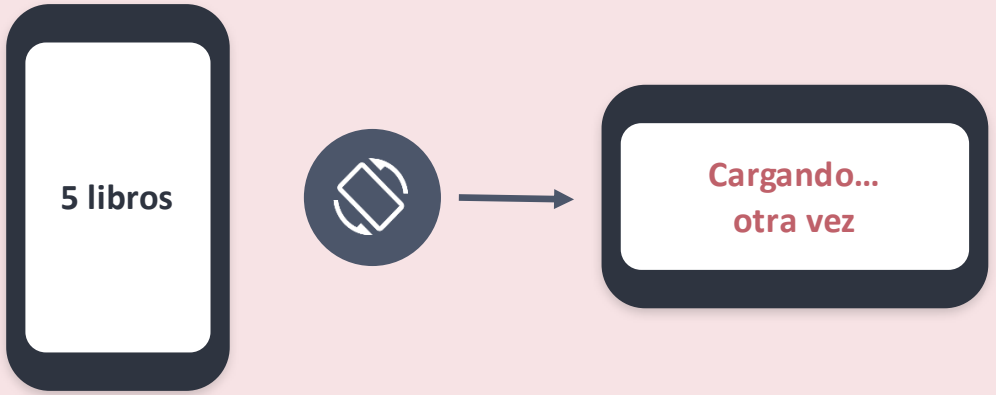
La lógica solo se prueba instanciando una pantalla real de Android.

¿Qué ocurre al rotar la pantalla?

Configuración

remember

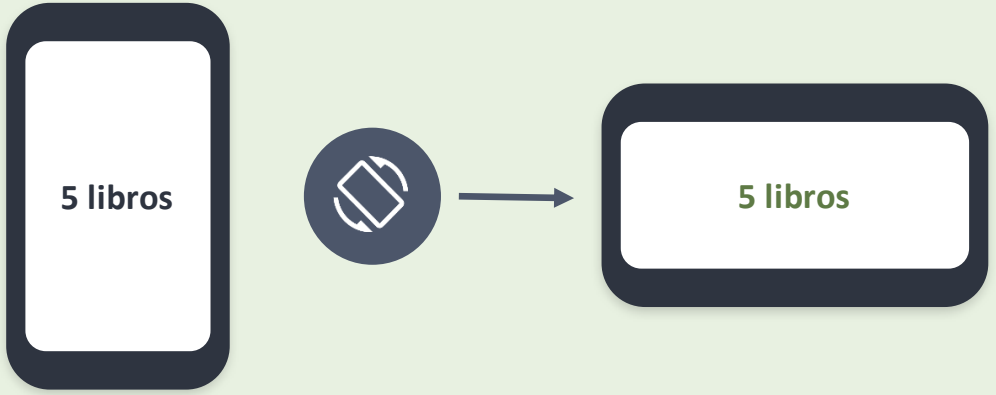
El estado vive dentro del Composable



La Activity se recrea y remember vuelve al valor inicial. Se repite la carga.

ViewModel

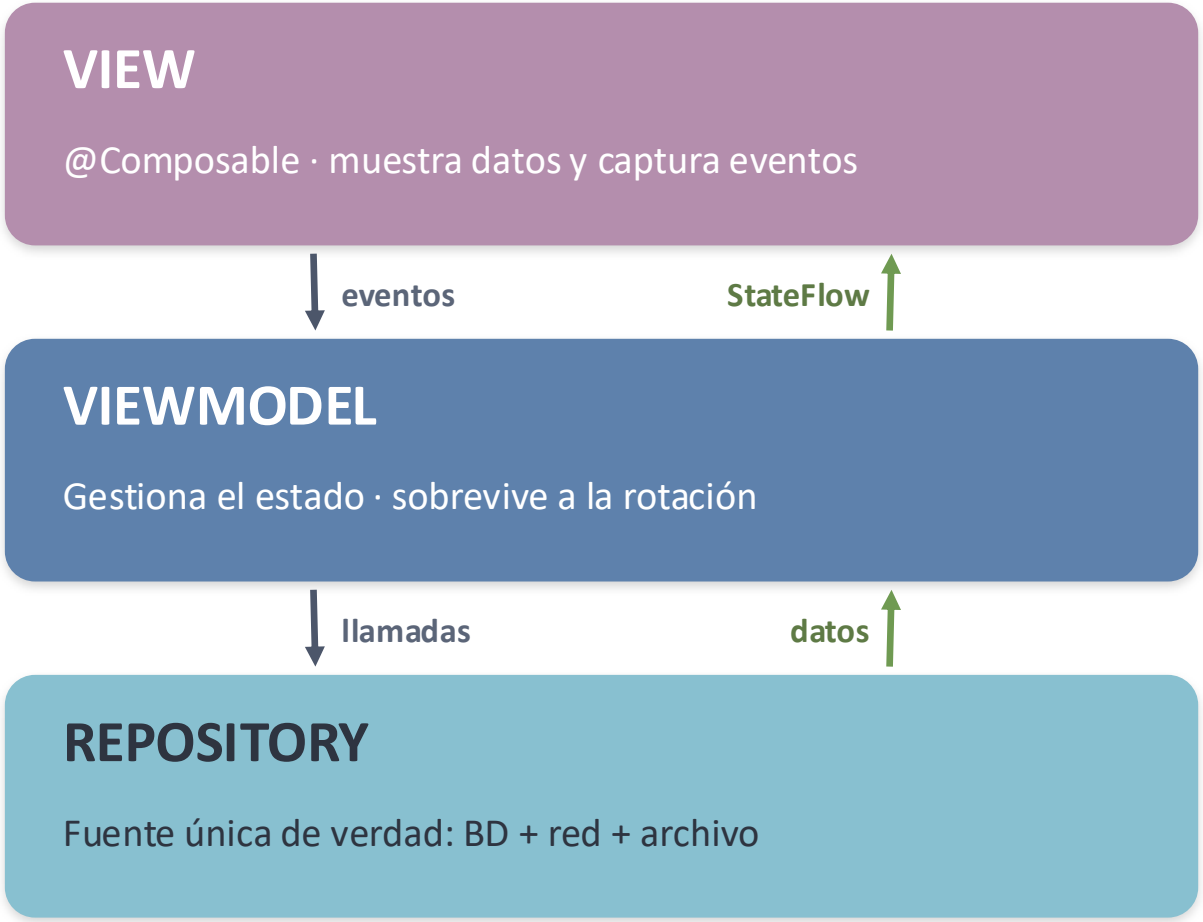
El estado vive fuera de la Activity



La Activity se recrea, pero el ViewModel sigue vivo con sus datos.

El patrón MVVM

MVVM



Flujo unidireccional de datos (UDF)

El estado baja hacia la UI

La View observa un StateFlow y se recompone cuando cambia.

Los eventos suben al ViewModel

Un clic se traduce en una llamada a una función del ViewModel.

Cada capa, una responsabilidad

La View no accede a datos; el ViewModel no conoce la UI.

Responsabilidades de cada capa

MVVM

Capa	Clase típica	Hace	No hace
View	`@Composable`	✓ Mostrar UI y capturar eventos	✗ Lógica de negocio, acceso a datos
ViewModel	`: ViewModel()`	✓ Gestionar estado, coordinar lógica	✗ Conocer la UI, acceder a BD o red
Repository	`clase normal`	✓ Coordinar fuentes de datos	✗ Conocer la UI, exponer detalles
DataSource	`DAO, API Service`	✓ Acceder a una fuente concreta	✗ Mezclar fuentes, presentación
Model	`data class`	✓ Representar los datos	✗ Cualquier lógica

02

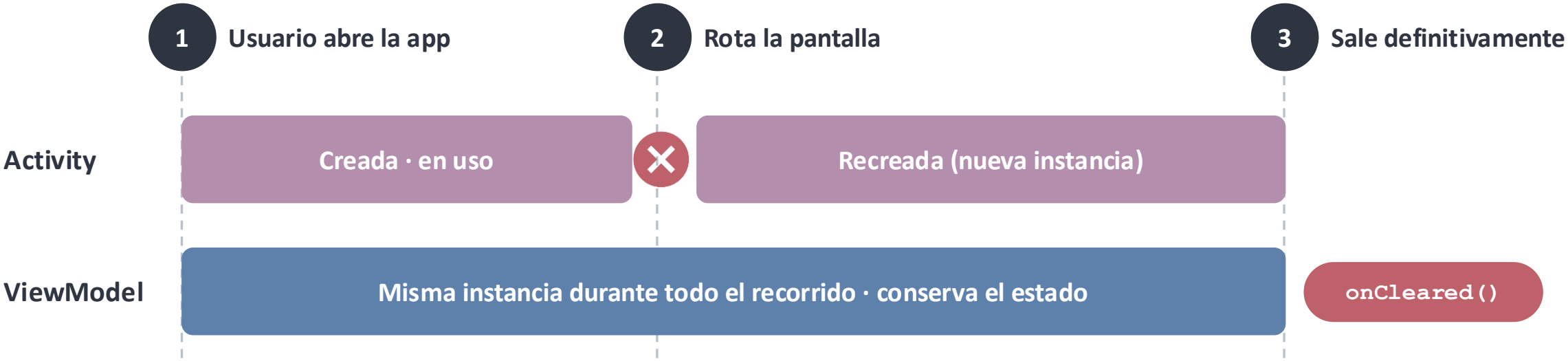
ViewModel

Ciclo de vida, viewModel() y ViewModelProvider.Factory



Ciclo de vida: el ViewModel sobrevive a la Activity

`Lifecycle`



Creación

viewModel() crea la instancia la primera vez que se solicita.

Reutilización

Tras rotar, viewModel() devuelve el mismo objeto con sus datos.

Destrucción

Al salir, se llama a onCleared() y se cancela viewModelScope.

viewModel(), nunca el constructor

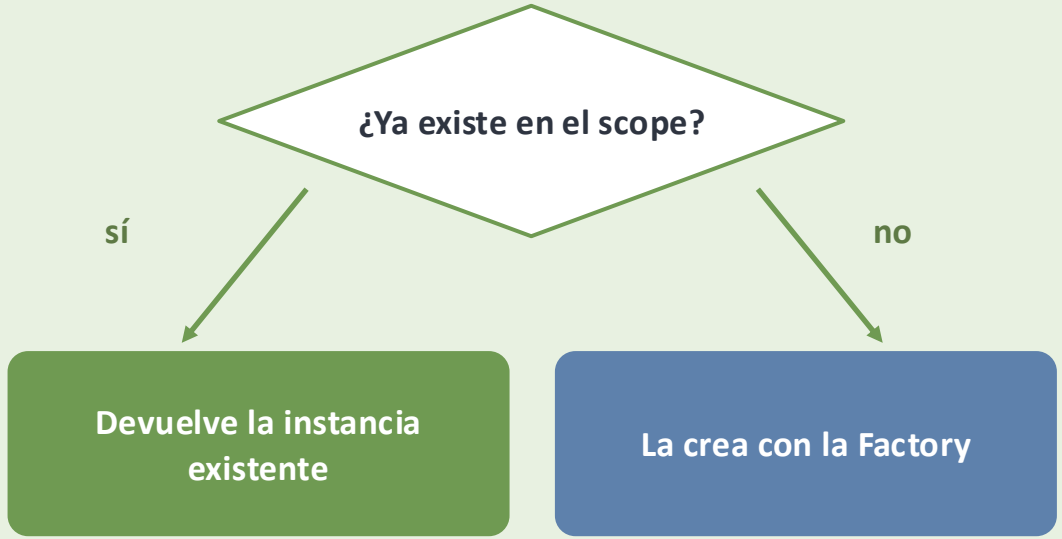
`viewModel()`

```
@Composable
fun PantallaListado() {
    // ✕ Instancia directa
    val viewModel = LibrosViewModel(...)
}
```

```
@Composable
fun PantallaListado(
    viewModel: LibrosViewModel =
        viewModel(factory =
LibrosViewModel.Factory)
) { /* ... */ }
```

Constructor directo

- Se crea una instancia nueva en cada recomposición.
- No sobrevive a la rotación: el estado se pierde.
- Jetpack no gestiona su ciclo de vida.



Factory sin inyección de dependencias

Factory

`viewModel(…)` → Composable

`Factory` → viewModelFactory { }

`initializer { }` → receta de creación

`LibrosViewModel(LibrosRepository())`

screens/listado/LibrosViewModel.kt

```
class LibrosViewModel(
    private val repository: LibrosRepository
) : ViewModel() {
    companion object {
        val Factory: ViewModelProvider.Factory =
            viewModelFactory {
                initializer {
                    LibrosViewModel(LibrosRepository())
                }
            }
    }
}
```



¿Por qué una Factory?

viewModel() no sabe pasar parámetros al constructor: la Factory le explica cómo.



En el Bloque 3

El repositorio vendrá de un AppContainer con ROOM y Retrofit2.

03

Estado de la UI

StateFlow, backing property y UiState como sealed class



StateFlow: un flujo que siempre tiene valor

`StateFlow`



Cuando su valor cambia, Compose lo detecta y recompone solo los elementos afectados.

Aspecto	LiveData	StateFlow
Lenguaje	Java / Kotlin	**Kotlin puro**
Corrutinas	Adaptadores externos	**Integración nativa**
Valor inicial	Opcional	**Obligatorio**
Observar en Compose	`observeAsState()`	**`collectAsStateWithLifecycle()`**
Recomendación	Proyectos legacy	**Proyectos nuevos**

El patrón backing property

StateFlow




```
class LibrosViewModel : ViewModel() {
    // Mutable y PRIVADO: solo el ViewModel lo modifica
    private val _uiState =
        MutableStateFlow<LibrosUiState>(LibrosUiState.Cargando)
    // Inmutable y PÚBLICO: los Composables solo lo leen
    val uiState: StateFlow<LibrosUiState> = _uiState.asStateFlow()
}
```

Sin este patrón…

cualquier Composable podría modificar el estado y se rompería el flujo unidireccional.

UiState como sealed class

sealed class



Cargando

`data object`

Sin datos asociados



Exito

`data class (libros)`

Lleva List<Libro>



Error

`data class (mensaje)`

Lleva el mensaje de error

screens/listado/LibrosUiState.kt

```
// Estados posibles de la pantalla de listado
sealed class LibrosUiState {
    data object Cargando : LibrosUiState()
    data class Exito(val libros: List<Libro>)
        : LibrosUiState()
    data class Error(val mensaje: String)
        : LibrosUiState()
}
```

¿data object u object?

object

`LibrosUiState$Cargando@3a2b1c`

data object

Cargando

toString() legible: depurar es más fácil.

Un estado, una pantalla: when exhaustivo

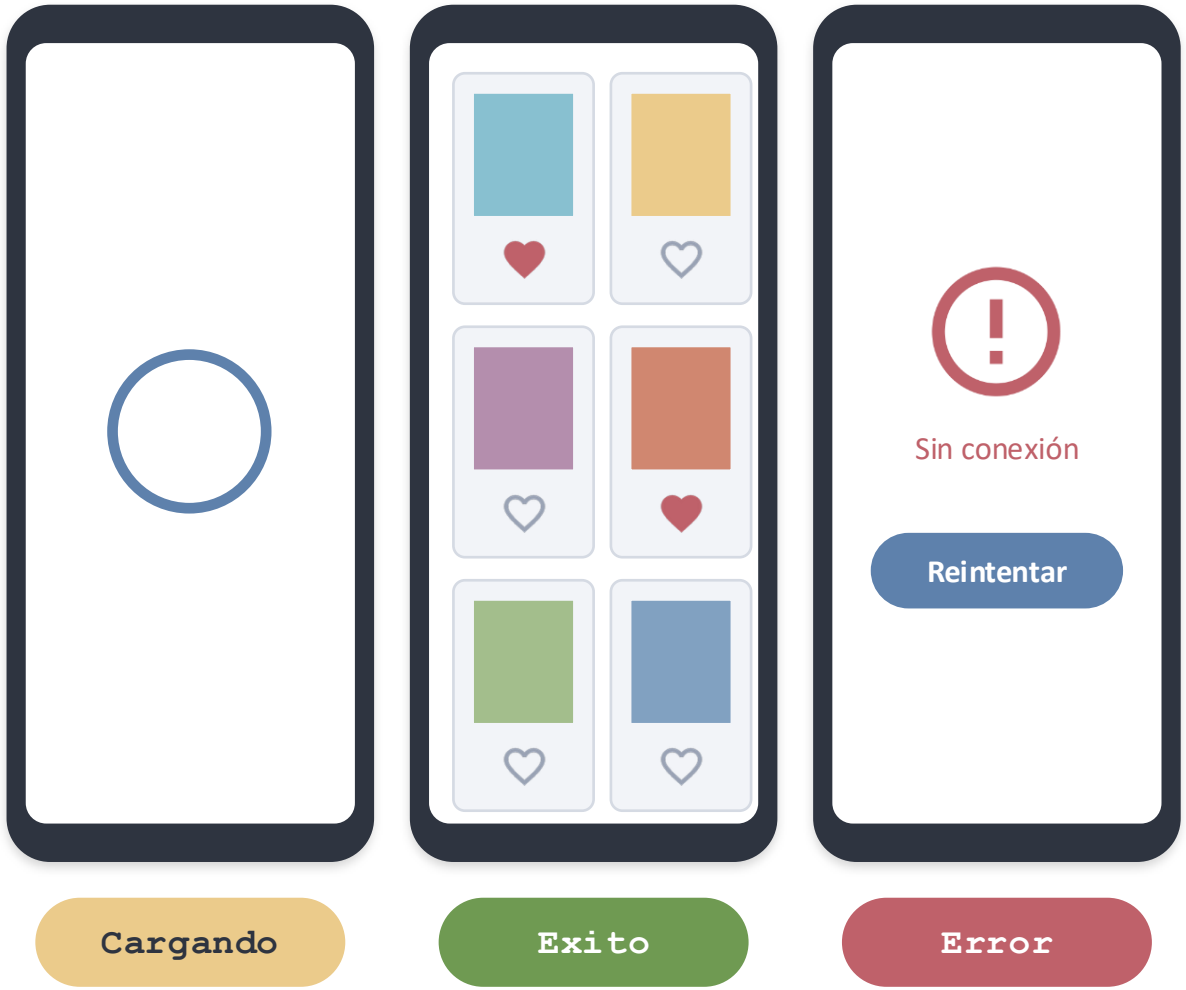
when

```
PantallaListado.kt (simplificado)
when (val estado = uiState) {
    is LibrosUiState.Cargando ->
        CircularProgressIndicator()

    is LibrosUiState.Exito ->
        LazyVerticalGrid { /* libros */ }

    is LibrosUiState.Error ->
        Text(estado.mensaje)
}
```

Sin else: si falta un estado, el código no compila.



Modelo y repositorio de AppDummy

Capa de datos

data/model/Libro.kt

```kotlin
data class Libro(
    val id: Int = 0,
    val titulo: String = "",
    val autor: String = "",
    val year: Int? = 1900,
    val isbn: String = "",
    val cover: String = "",
    val esFavorito: Boolean = false,
    val leido: Boolean = false
)
```

Se reutilizará en el Bloque 3 con ROOM y Retrofit2.

LibrosRepository 5 libros estáticos · simula la latencia de red

`suspend getLibros()`	800 ms	→ `List<Libro>`
`suspend getAutores()`	300 ms	→ `List<String>`
`suspend getLibroPorId(id)`	300 ms	→ `Libro?`

suspend: la función puede pausarse sin bloquear el hilo principal.

Anatomía de LibrosViewModel

ViewModel

Pantalla Listado

`@Composable`

observa

llama

LibrosViewModel(repository)

Estado (StateFlow)

`uiState`	`LibrosUiState`
`busqueda`	`String`
`autores`	`List<String>`
`autorSeleccionado`	`String`

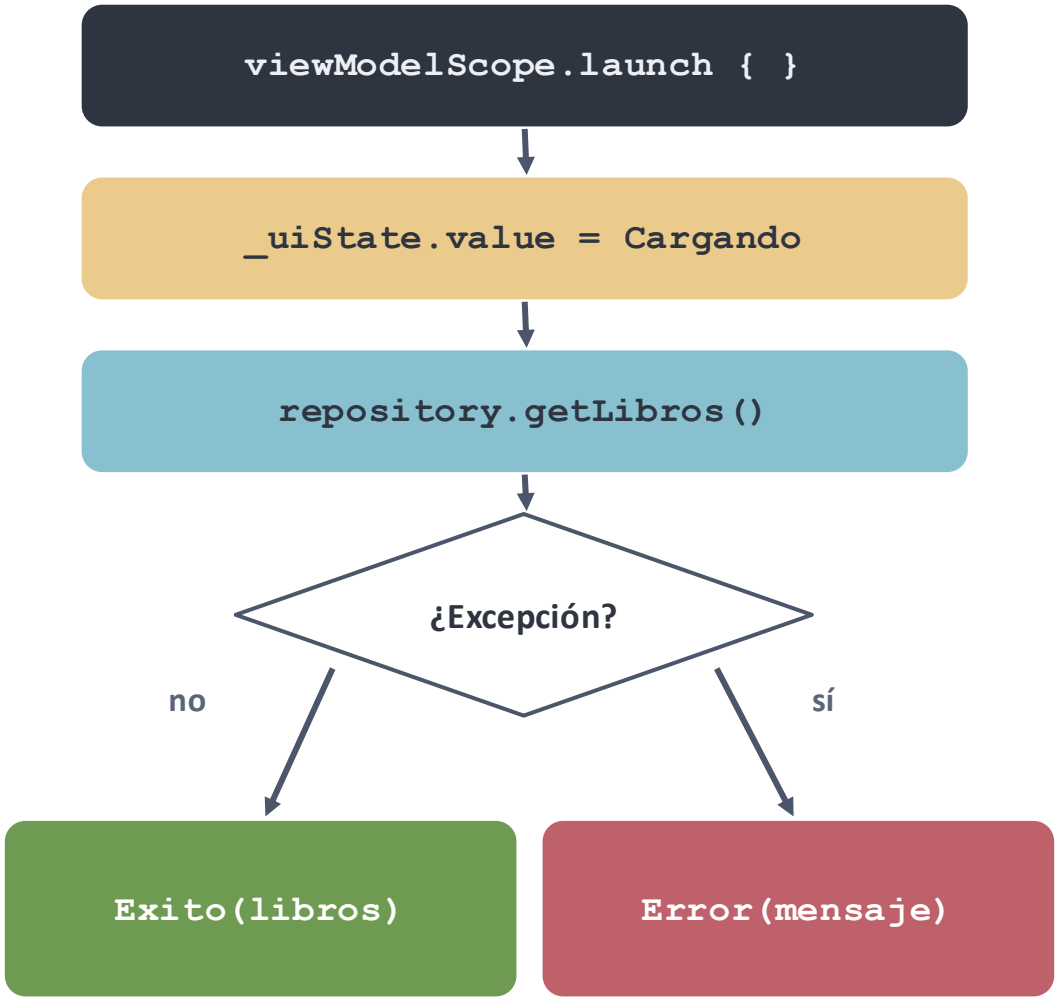
Eventos (funciones públicas)

- `cargarLibros()`
- `cargarAutores()`
- `actualizarBusqueda(texto)`
- `actualizarAutorSeleccionado(autor)`
- `toggleFavorito(id)`
- `toggleLeido(id)`

```
init { cargarLibros(); cargarAutores() }
```
← carga automática al crearse

cargarLibros(): corrutina en viewModelScope

viewModelScope



screens/listado/LibrosViewModel.kt

```kotlin
fun cargarLibros() {
    // Corrutina ligada al ViewModel:
    // se cancela al destruirlo
    viewModelScope.launch {
        _uiState.value = LibrosUiState.Cargando
        try {
            val libros = repository.getLibros()
            _uiState.value =
                LibrosUiState.Exito(libros)
        } catch (e: Exception) {
            _uiState.value = LibrosUiState.Error(
                e.message ?: "Error desconocido")
        }
    }
}
```

update { }: nuevo estado de forma atómica

`update { }`

`Exito(libros)`

- Proyecto Hail Mary ♥
- Juego de tronos ♥
- Festín de cuervos ♡

`toggleFavorito(3)`

→ copy() → objeto nuevo

`Exito(libros')`

- Proyecto Hail Mary ♥
- Juego de tronos ♥
- Festín de cuervos ♥

```
fun toggleFavorito(id: Int) {
    _uiState.update { estado ->
        if (estado is LibrosUiState.Exito) {
            estado.copy(libros = estado.libros.map {
                if (it.id == id)
                    it.copy(esFavorito = !it.esFavorito)
                else it
            })
        } else estado
    }
}
```



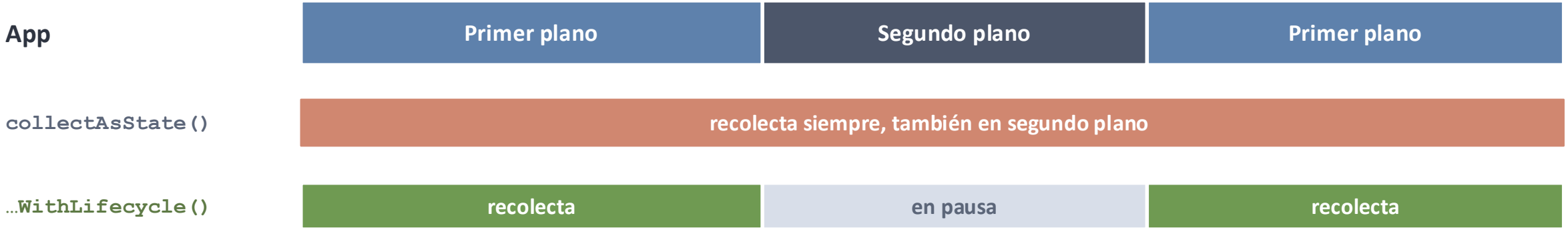
Atómico y thread-safe
Lee y escribe en una sola operación.



Inmutable
Nueva lista y nuevo libro: Compose detecta el cambio.

Observar desde la UI: collectAsStateWithLifecycle()

View




```kotlin
// PantallaListado.kt
val uiState by
    viewModel.uiState.collectAsStateWithLifecycle()
val busqueda by
    viewModel.busqueda.collectAsStateWithLifecycle()
val autores by
    viewModel.autores.collectAsStateWithLifecycle()
val autorSeleccionado by
    viewModel.autorSeleccionado.collectAsStateWithLifecycle()
```

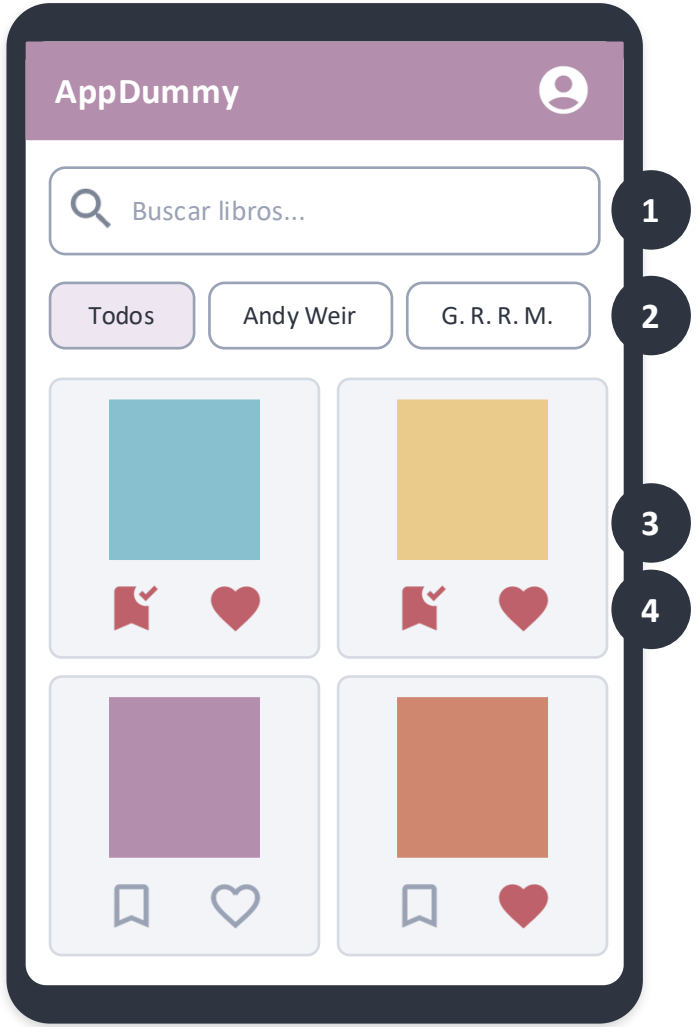
Ahorra recursos

Pausa la recolección cuando la app no está visible y la reanuda al volver.

`lifecycle-runtime-compose`

PantallaListado conectada al ViewModel

View



1 Barra de búsqueda

`OutlinedTextField`

lee `value = busqueda`

evento `onValueChange = viewModel::actualizarBusqueda`

2 Chips de autores

`LazyRow + FilterChip`

lee `items(autores) · autorSeleccionado`

evento `onClick → actualizarAutorSeleccionado(autor)`

3 Rejilla de libros

`LazyVerticalGrid`

lee `when (uiState) → Exito, filtrado`

evento `onClickItem → onNavegaADetalle(id) · T3`

4 Iconos de cada libro

`ItemLibro`

lee `libro.leido · libro.esFavorito`

evento `toggleLeido(id) · toggleFavorito(id)`

El ciclo completo al pulsar ♥

UDF



04

Clean Architecture

Regla de dependencias y estructura de paquetes



Regla de dependencias

Clean Arch.

conoce a →



- **El Composable conoce al ViewModel.**
- El ViewModel no sabe qué pantalla lo usa.

- **El ViewModel conoce al Repository.**
- El Repository no sabe quién le pide datos.

- **El Repository coordina los DataSource.**
- Los DataSource no se conocen entre sí.

Estructura de paquetes de AppDummy

AppDummy

```
com.ejemplo.appdummy/
├─ MainActivity.kt
├─ screens/
│   ├─ listado/
│   │   ├─ PantallaListado.kt
│   │   ├─ LibrosViewModel.kt
│   │   ├─ LibrosUiState.kt
│   ├─ detalle/
│   │   ├─ PantallaDetalle.kt · DetalleViewModel.kt
│   ├─ navegacion/AppNavigation.kt
├─ ui/theme/  Color · Type · Theme
├─ data/
│   ├─ model/Libro.kt
│   ├─ repository/LibrosRepository.kt
│   ├─ datasource/LocalDataSource.kt
│   ├─ datasource/RemoteDataSource.kt
```

- Tema 2 (ahora)
- Tema 3 · Navigation
- Bloque 3 · ROOM y Retrofit2

Capa → paquete

View	screens/*/Pantalla*
ViewModel	screens/*/*ViewModel
Model	data/model
Repository	data/repository
DataSource	data/datasource

05

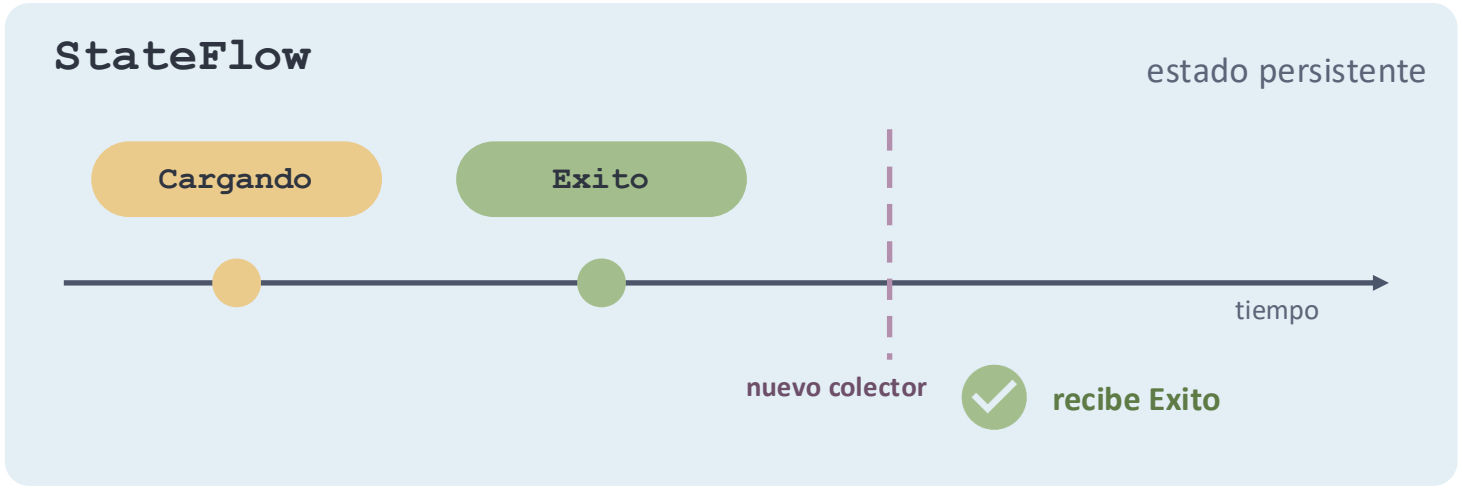
Eventos y testing

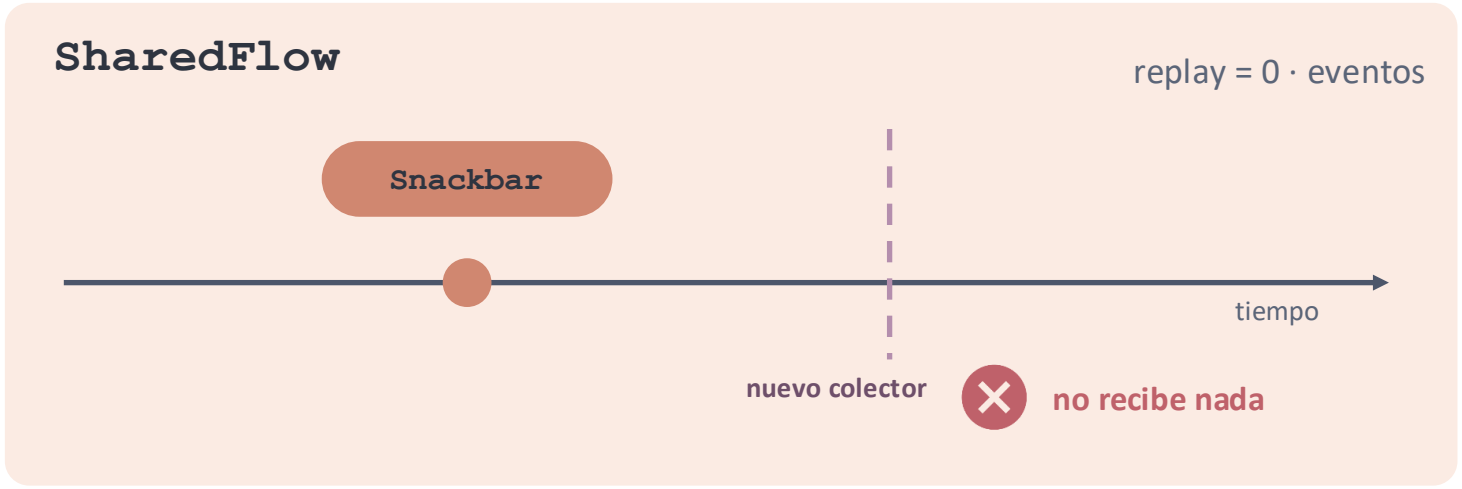
SharedFlow para eventos de un solo disparo y pruebas del ViewModel



Estado frente a evento: StateFlow y SharedFlow

SharedFlow

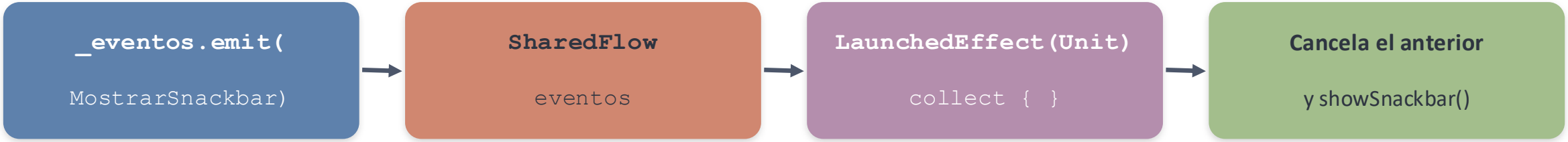




	StateFlow	SharedFlow
Valor actual	✓	✗
Se repite al suscribirse	✓	✗
Uso típico	Estado de la UI	Eventos únicos
Ejemplo	Lista, carga	Snackbar, navegación

SharedFlow: Snackbar al marcar favorito

SharedFlow



ViewModel

```
sealed class LibrosEvento {
    data class MostrarSnackbar(val mensaje: String)
        : LibrosEvento()
    data class NavegarADetalle(val id: Int)
        : LibrosEvento()
}

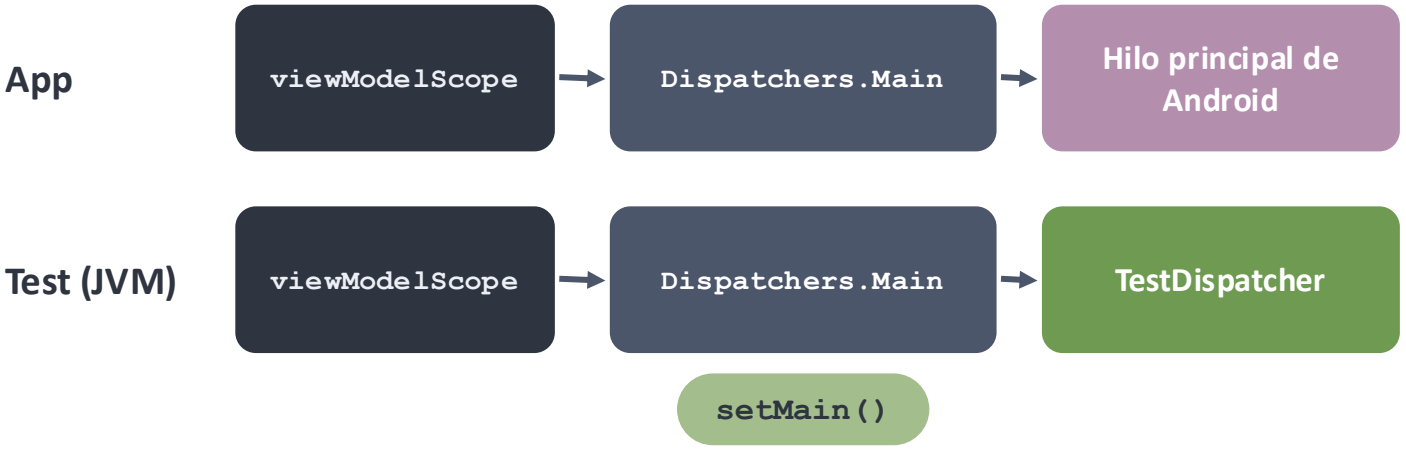
// En LibrosViewModel
private val _eventos = MutableSharedFlow<LibrosEvento>(
    replay = 0,              // no se repiten
    extraBufferCapacity = 1) // emit no se suspende
val eventos: SharedFlow<LibrosEvento> =
    _eventos.asSharedFlow()
```

PantallaListado.kt

```
LaunchedEffect(Unit) {
    viewModel.eventos.collect { evento ->
        when (evento) {
            is LibrosEvento.MostrarSnackbar -> {
                snackbarJob?.cancel()
                snackbarHostState.currentSnackbarData
                    ?.dismiss()
                snackbarJob = scope.launch {
                    snackbarHostState
                        .showSnackbar(evento.mensaje)
                }
            }
            is LibrosEvento.NavegarADetalle -> {} // T3
        }
    }
}
```

Testing del ViewModel: sin Activity, sin Compose

Testing



test/MainDispatcherRule.kt

```
class MainDispatcherRule(
    val testDispatcher: TestDispatcher =
        UnconfinedTestDispatcher()
) : TestWatcher() {
    override fun starting(description: Description) =
        Dispatchers.setMain(testDispatcher)
    override fun finished(description: Description) =
        Dispatchers.resetMain()
}
```

UnconfinedTestDispatcher

Ejecuta las corrutinas de inmediato (eager). Ideal para que las corrutinas del init terminen antes de las aserciones.

StandardTestDispatcher

Encola las corrutinas: hay que avanzarlas con advanceUntilIdle(). Útil para controlar el orden de ejecución.

Cuatro pruebas para LibrosViewModel

`runTest`

`estado inicial es Cargando`

Cargando (o Exito) justo tras crear el ViewModel

`cargarLibros produce Exito`

advanceUntilIdle() → Exito con libros

`actualizarBusqueda`

busqueda.value pasa a ser "Dune"

`toggleFavorito invierte`

esFavorito cambia tras la llamada

test/LibrosViewModelTest.kt

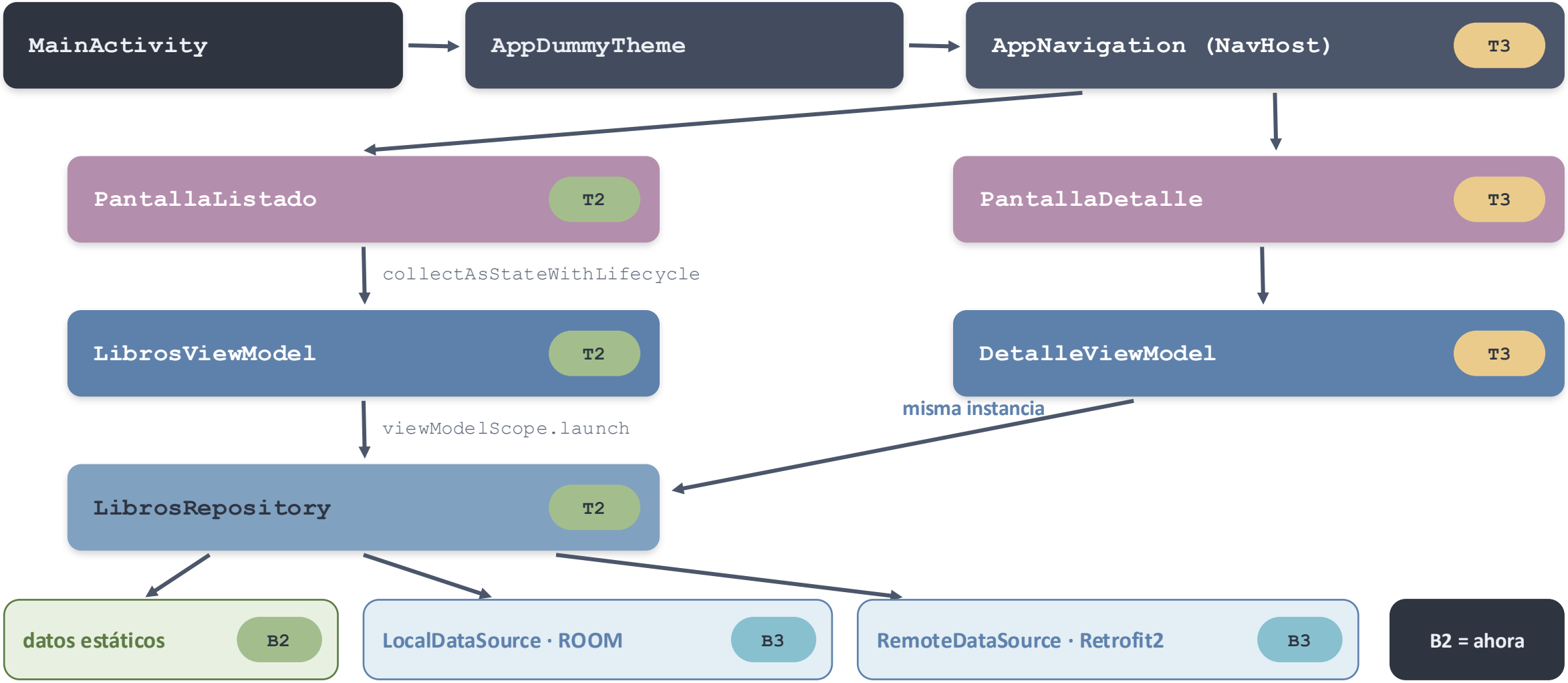
```kotlin
@get:Rule
val mainDispatcherRule = MainDispatcherRule()

@Before
fun setup() {
    viewModel = LibrosViewModel(LibrosRepository())
}

@Test
fun `toggleFavorito invierte favorito`() = runTest {
    advanceUntilIdle()
    val libro = (viewModel.uiState.value
        as LibrosUiState.Exito).libros.first()
    viewModel.toggleFavorito(libro.id)
    val estadoFinal = viewModel.uiState.value
        as LibrosUiState.Exito
    assertNotEquals(libro.esFavorito,
        estadoFinal.libros.first { it.id == libro.id }
            .esFavorito)
}
```

Práctica guiada: arquitectura de AppDummy

Práctica



Ideas clave



MVVM y UDF

Estado hacia la UI, eventos hacia el ViewModel.



ViewModel sobrevive

Obtenlo con viewModel() y su Factory.



Backing property

_uiState privado y mutable; uiState público.



UiState sellado

Cargando, Exito y Error con when exhaustivo.



SharedFlow

Para eventos únicos: Snackbar o navegación.



Testable

MainDispatcherRule y runTest, sin Android.

Referencias

Fuentes

Guía de arquitectura de apps
developer.android.com/topic/architecture

Descripción general de ViewModel
developer.android.com/topic/libraries/architecture/viewmodel

ViewModels con dependencias
developer.android.com/.../viewmodel/viewmodel-factories

StateFlow y SharedFlow
developer.android.com/kotlin/flow/stateflow-and-sharedflow

Capa de la IU
developer.android.com/topic/architecture/ui-layer

Probar corrutinas de Kotlin en Android
developer.android.com/kotlin/coroutines/test

Tema 2. Arquitectura MVVM y ViewModel (apuntes)
documentation.javiercarrasco.es/pmdm/B2/B2-T2_MVVM_ViewModel