Bloque 3 · Tema 5

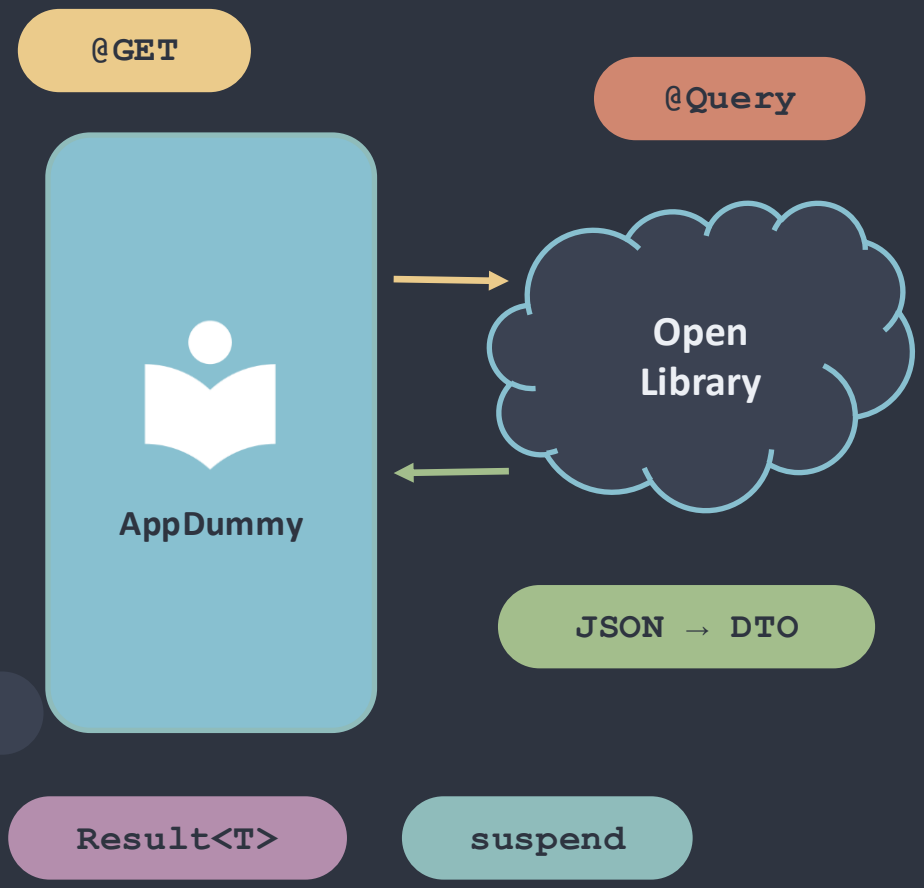
Retrofit2

Consumo de APIs REST con Open Library en Kotlin y Jetpack Compose

6 horas

AppDummy · biblioteca personal

RA2 · RA3



Hoja de ruta del tema

11 apartados + práctica

1 Interfaz anotada → HTTP

¿Qué es Retrofit?

2 Endpoints, límites, fields

API Open Library

3 La de Room, sin cambios

Entidad Libro

4 Forma del JSON → entidad

DTOs y mapeadores

5 @GET, @Query, suspend

Interfaz de la API

6 Timeouts e interceptores

OkHttp + Retrofit

7 Frontera de los DTOs

RemoteDataSource

8 IOException · HttpException

Manejo de errores

9 AsyncImage compartiendo cliente

Carátulas con Coil

10 ISBN, debounce y guardado

Añadir libro

11 ¿Por qué Gson en el curso?

Gson vs kotlinx

12 AppDummy v7

Práctica guiada

¿Qué vamos a evaluar?

RA2 · RA3

RA2

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

RA3

Desarrolla programas que integran contenidos multimedia analizando y empleando las tecnologías y librerías específicas.

RA2-a	Estructura de clases de la app
RA2-b	Ventanas, menús, alertas y controles
RA2-e	Conexiones y comunicaciones HTTP/HTTPS
RA2-g	Pruebas de interacción usuario-app
RA2-i	Documentar el proceso de desarrollo
RA3-c	Conversión de datos multimedia
RA3-d	Procesamiento de datos multimedia
RA3-e	Eventos, tipos de media y excepciones
RA3-h	Depurar y documentar

★ *RA2-e es el criterio central de este tema*

01

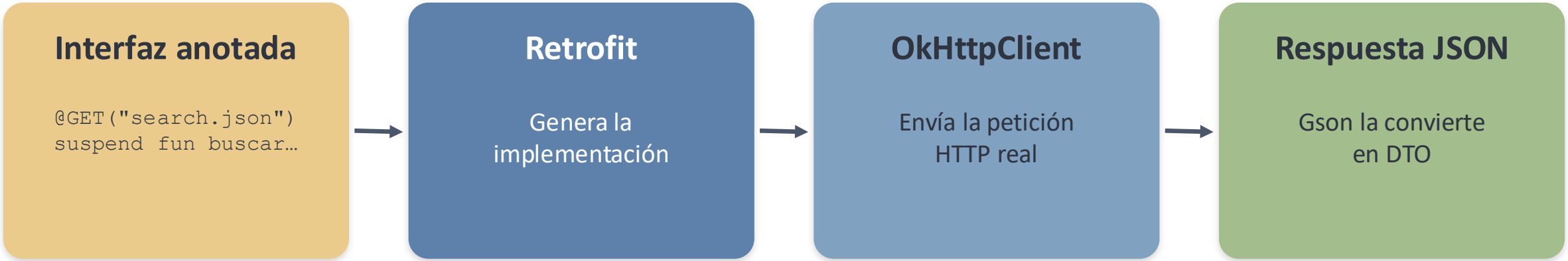
Fundamentos

Qué es Retrofit, dónde encaja y qué API vamos a consumir



Retrofit: de interfaz Kotlin a petición HTTP

Square



Declarativo

Describes la petición con anotaciones; no escribes el código HTTP.

Conversión automática

Un converter (Gson) transforma el JSON en objetos Kotlin.

Corrutinas nativas

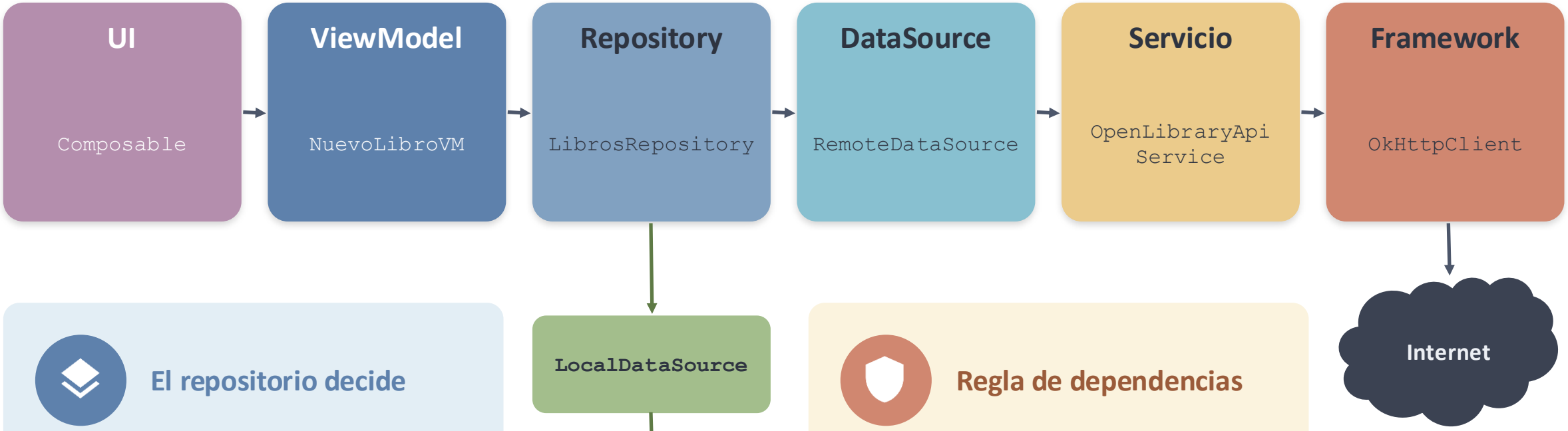
Funciones suspend: sin callbacks ni hilos manuales.

Extensible

Comparte OkHttp: interceptores, timeouts, caché, logs.

¿Dónde vive la red en la arquitectura?

MVVM



El repositorio decide

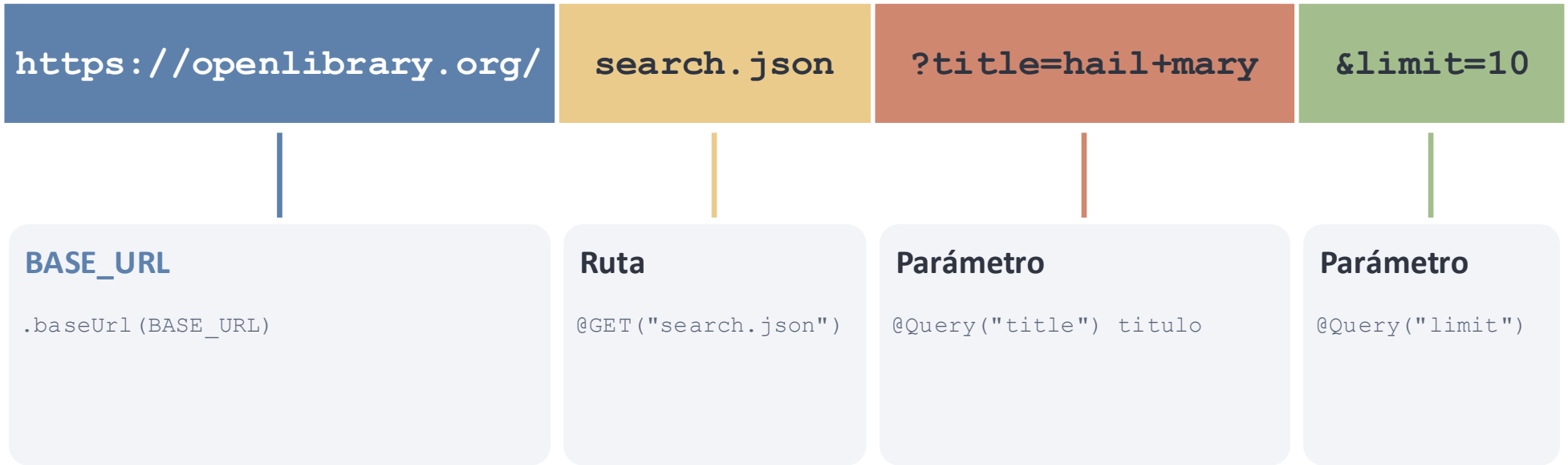
Coordina dos fuentes: la red para buscar libros y Room para guardarlos. La UI no sabe de dónde viene cada dato.

Regla de dependencias

Cada capa solo conoce a la siguiente. Los DTOs y Retrofit no salen de la capa remota.

Anatomía de una petición a Open Library

HTTP GET



HTTP

Método GET

Solo lectura: pedimos datos y el servidor responde con un JSON.

La BASE_URL termina en /

La ruta de @GET se escribe sin barra inicial y se concatena a la base.

Open Library: una API abierta… con límites

`openlibrary.org`



Sin API Key

Catálogo abierto: no hay que registrarse para consultar libros.

1

petición / segundo

Anónimas

3

peticiones / segundo

Con User-Agent y email de contacto

100

cada 5 minutos por IP

Carátulas por ISBN / OCLC / LCCN

Endpoints que usaremos

GET	`search.json?title=…`	Buscar por título
GET	`search.json?q=isbn:9780593135204`	Buscar por ISBN
IMG	`covers.openlibrary.org/b/id/{cover_i}-M.jpg`	Carátula

403 / 429

Respuesta del servidor al superar los límites

El parámetro fields: pide solo lo que usas

`@Query("fields")`

```
search.json?title=hail+mary
// Sin fields: cada doc trae
// muchísimos campos
{
  "key": "/works/OL…W",
  "title": "Project Hail Mary",
  "author_name": ["Andy Weir"],
  "author_key": [ … ],
  "edition_count": …,
  "language": [ … ],
  "publisher": [ … ],
  "subject": [ … ],
  "ia": [ … ],
  …  // y muchos más
}
```

fields=



```
…&fields=key,title,author_name,…
{
  "key": "/works/OL…W",
  "title": "Project Hail Mary",
  "author_name": ["Andy Weir"],
  "first_publish_year": 2021,
  "isbn": ["9780593135204", …],
  "cover_i": …
}
```

OpenLibrary.CAMPOS

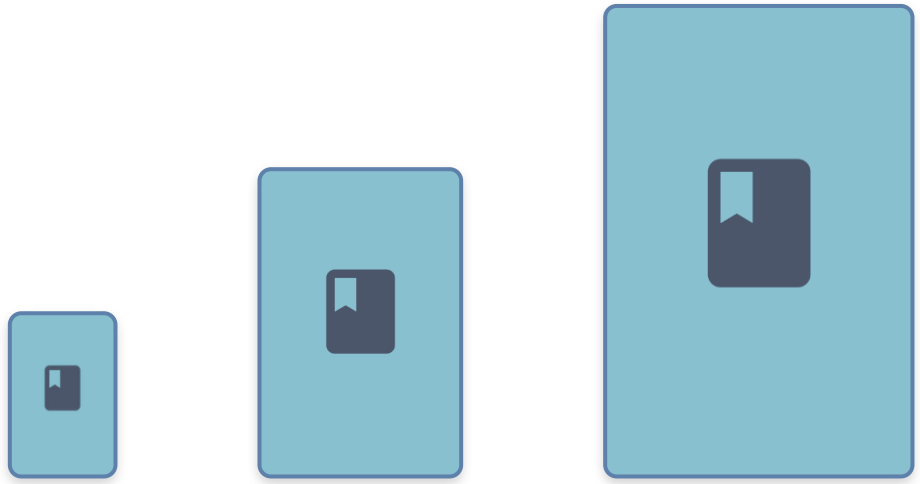
key, title, author_name, first_publish_year, isbn, cover_i

Carátulas: una URL por tamaño

Covers API

```
covers.openlibrary.org/b/id/{cover_i}-M.jpg?default=false
```

o bien …/b/isbn/{isbn}-M.jpg?default=false



S

≈ 90 px

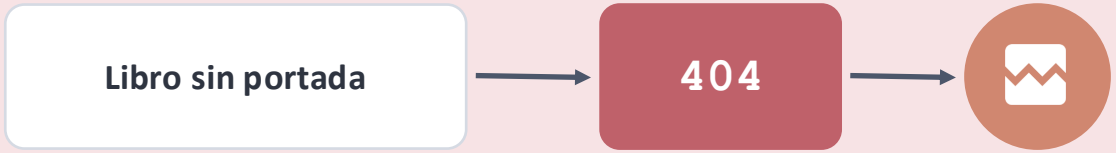
M

≈ 180 px

L

≈ 500 px · por defecto

¿Para qué ?default=false?



Sin el parámetro, el servidor devuelve una imagen en blanco y Coil cree que la carga ha ido bien. Con default=false responde 404 y Coil muestra su imagen de error.

Dependencias y configuración

Gradle KTS

gradle/libs.versions.toml

```
[versions]
retrofit = "3.0.0"
okhttp = "5.4.0"
gson = "2.14.0"

[libraries]
retrofit = { module = "com.squareup.retrofit2:retrofit",
    version.ref = "retrofit" }
converter-gson = { module =
    "com.squareup.retrofit2:converter-gson",
    version.ref = "retrofit" }
# okhttp y logging-interceptor: igual, con
# com.squareup.okhttp3 y version.ref = "okhttp"
```

app/build.gradle.kts

```
android {
    buildFeatures {
        compose = true
        buildConfig = true   // AGP 8+
    }
}
dependencies {
    implementation(libs.retrofit)
    implementation(libs.converter.gson)
    implementation(libs.okhttp)
    implementation(libs.logging.interceptor)
    implementation(libs.coil.compose)
    implementation(libs.coil.network.okhttp)
}
```

AndroidManifest.xml

```
<uses-permission
    android:name="android.permission.INTERNET" />
```

- converter-gson = versión de retrofit
- buildConfig para BuildConfig.DEBUG
- Sin INTERNET → excepción de seguridad

02

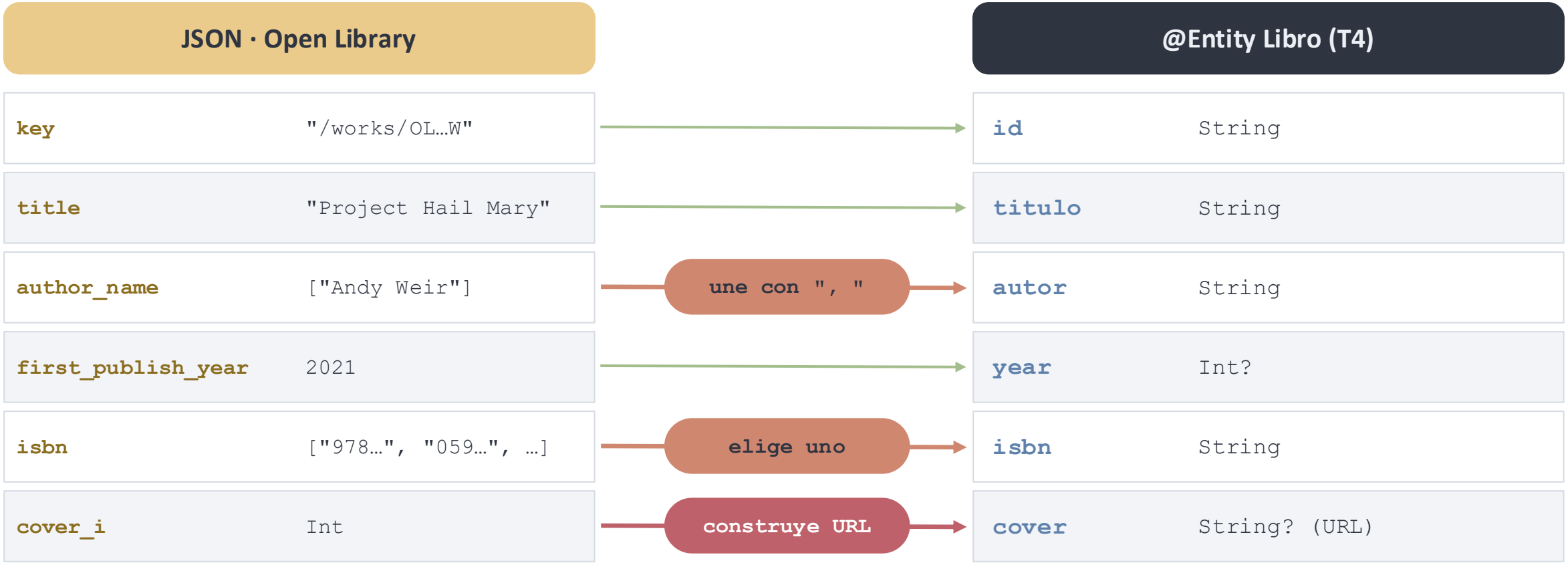
Del JSON a la entidad

Entidad Libro, DTOs, mapeadores y sus tests



El JSON no encaja campo a campo con Libro

Mapeo



Listas: author_name e isbn son arrays

Enteros → URL: cover_i no es una imagen

Campos locales: esFavorito y leido no vienen

DTOs: un espejo del JSON

DTO

data/datasource/remote/dto/LibroDto.kt

```kotlin
data class BusquedaResponseDto(
    @SerializedName("numFound") val numFound: Int? = null,
    @SerializedName("start") val start: Int? = null,
    @SerializedName("docs") val docs: List<LibroDto>? = null
)

data class LibroDto(
    @SerializedName("key") val key: String? = null,
    @SerializedName("title") val title: String? = null,
    @SerializedName("author_name")
    val authorName: List<String>? = null,
    @SerializedName("first_publish_year")
    val firstPublishYear: Int? = null,
    @SerializedName("isbn") val isbn: List<String>? = null,
    @SerializedName("cover_i") val coverId: Int? = null
)
```



Todo nullable con null por defecto

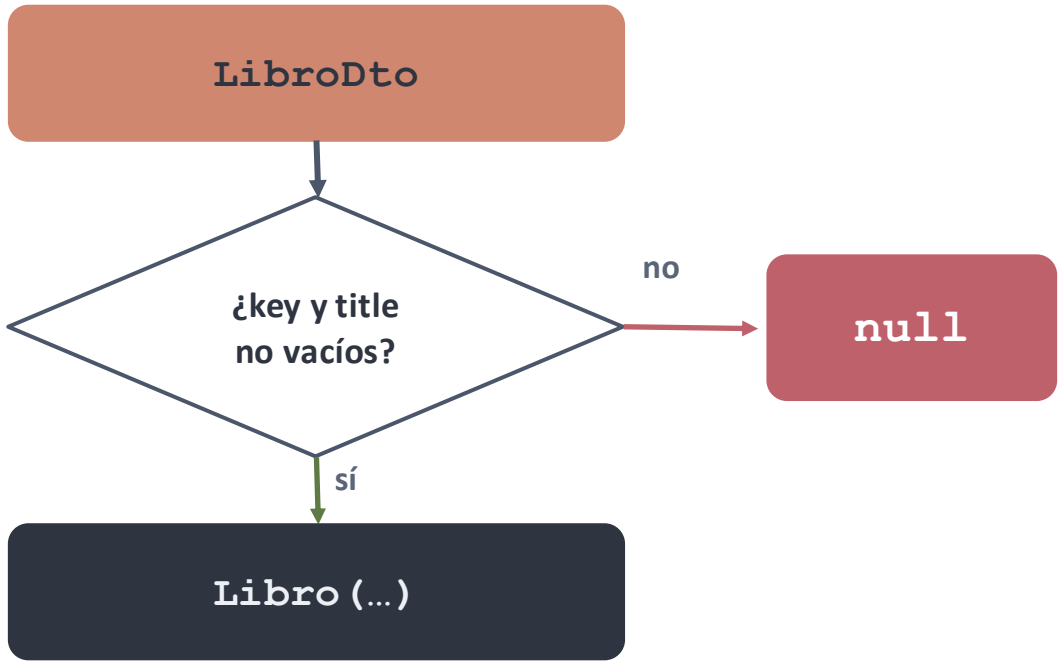
Gson ignora la nulabilidad de Kotlin: si falta un campo, deja null aunque el tipo diga String.

@SerializedName

Separa el nombre JSON (snake_case) del nombre Kotlin (camelCase).

El mapeador toLibro(): reglas de traducción

Mapper



toLibros() aplica toLibro() a cada doc y descarta los null (mapNotNull).

`id`	key, tal cual (/works/OL…W)
`autor`	Autores no vacíos unidos con ", " · si no hay: "Autor desconocido"
`isbn`	El buscado → el primero de 13 dígitos → el primero → ""
`cover`	URL por cover_i → si no, URL por ISBN-13 → si no, null
`esFavorito / leido`	No se tocan: valores por defecto de la entidad

Tests unitarios del mapeador: sin red, sin emulador

JUnit

app/src/test/…/LibroMapperTest.kt

```kotlin
class LibroMapperTest {

    @Test
    fun sinTituloDevuelveNull() {
        val dto = LibroDto(key = "k", title = null)
        assertNull(dto.toLibro())
    }

    @Test
    fun variosAutoresSeUnenConComa() {
        val dto = LibroDto(key = "k", title = "T",
            authorName = listOf("Ana", "Luis"))
        assertEquals("Ana, Luis", dto.toLibro()?.autor)
    }

    @Test
    fun sinAutoresUsaAutorDesconocido() { … }
}
```

test/ JVM local · milisegundos

Mapeadores, validación ISBN

androidTest/ Emulador o dispositivo

DAO de Room (T4)



Una función pura (DTO → Libro) es lo más fácil de probar del proyecto.

03

La capa remota

ApiService, OkHttpClient, Retrofit y RemoteDataSource



OpenLibraryApiService: el contrato HTTP

`@GET · @Query`

data/datasource/remote/OpenLibraryApiService.kt

```kotlin
interface OpenLibraryApiService {

    @GET("search.json")
    suspend fun buscarPorTitulo(
        @Query("title") titulo: String,
        @Query("fields") campos: String = OpenLibrary.CAMPOS,
        @Query("limit") limite: Int = 10,
        @Query("page") pagina: Int = 1
    ): BusquedaResponseDto

    @GET("search.json")
    suspend fun buscar(
        @Query("q") consulta: String,   // "isbn:978…"
        @Query("fields") campos: String = OpenLibrary.CAMPOS,
        @Query("limit") limite: Int = 10,
        @Query("page") pagina: Int = 1
    ): BusquedaResponseDto
}
```

Anotaciones de Retrofit

`@GET`	Petición GET a la ruta
`@POST`	Petición POST
`@PUT`	Petición PUT
`@DELETE`	Petición DELETE
`@Path`	Sustituye {nombre} en la URL
`@Query`	Añade ?nombre=valor
`@Body`	Objeto como cuerpo JSON
`@Header`	Añade una cabecera

suspend frente a Call<T>

Corrutinas

Call<T> · estilo clásico

Requiere enqueue() con callbacks o execute() bloqueante

```
@GET("search.json")
fun buscar(@Query("q") q: String):
    Call<BusquedaResponseDto>

api.buscar(q).enqueue(object :
    Callback<BusquedaResponseDto> {
    override fun onResponse(…) { … }
    override fun onFailure(…) { … }
})
```

suspend · recomendado

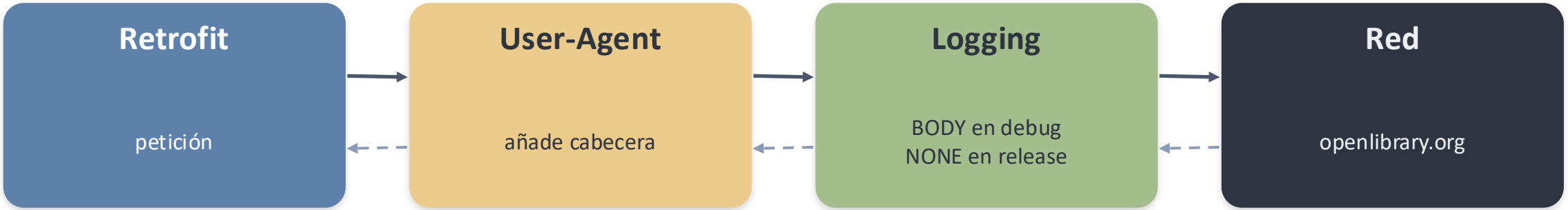
Se llama desde una corrutina y devuelve el resultado o lanza excepción

```
@GET("search.json")
suspend fun buscar(@Query("q") q: String):
    BusquedaResponseDto

viewModelScope.launch {
    val r = api.buscar(q)  // hilo de I/O
    …
}
```

OkHttpClient: la cadena de interceptores

OkHttp 5



→ petición ⇠ respuesta

```kotlin
RetrofitClient.kt

val okHttpClient = OkHttpClient.Builder()
    .connectTimeout(30, TimeUnit.SECONDS)
    .readTimeout(30, TimeUnit.SECONDS)
    .writeTimeout(30, TimeUnit.SECONDS)
    .addInterceptor { chain ->
        chain.proceed(chain.request().newBuilder()
            .header("User-Agent", "AppDummy/1.0 (tu_correo@correo.com)")
            .build())
    }
    .addInterceptor(HttpLoggingInterceptor().apply {
        level = if (BuildConfig.DEBUG) Level.BODY else Level.NONE
    })
    .build()
```

30 s connect · read · write

Nunca BODY en release: expone datos en Logcat

Un solo OkHttpClient para Retrofit y Coil

AppContainer

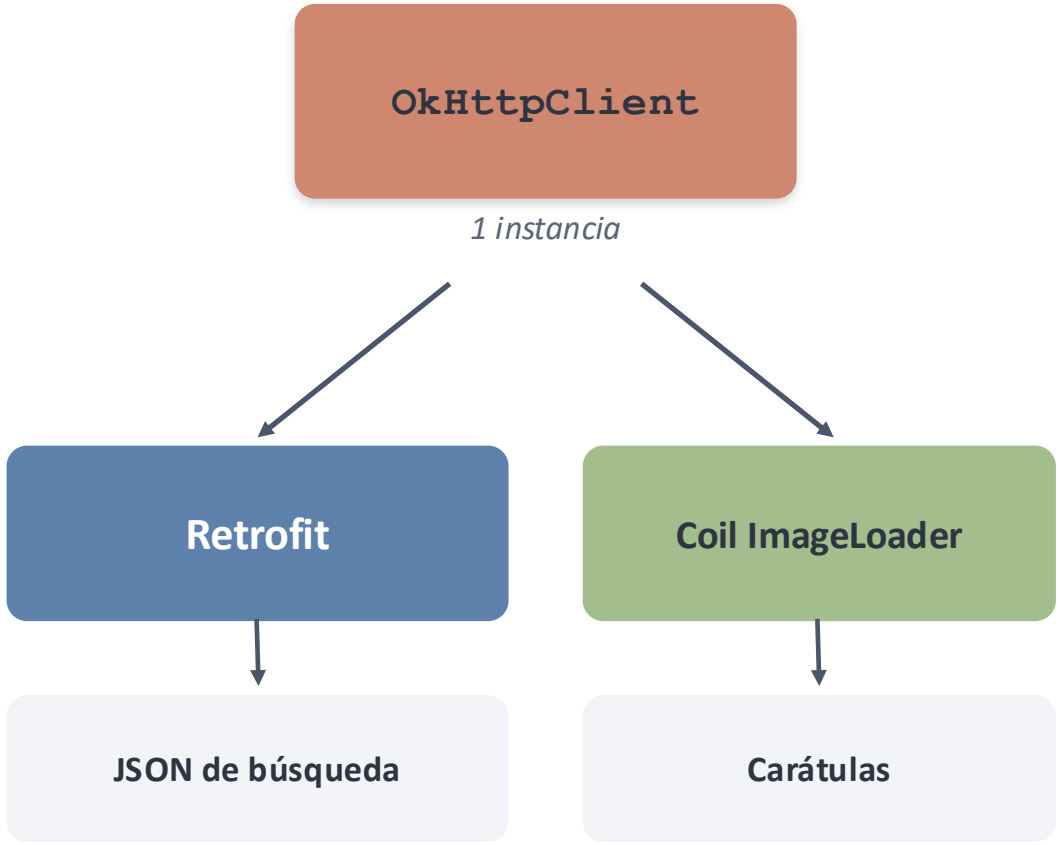
```
data/datasource/remote/RetrofitClient.kt

object RetrofitClient {
    val okHttpClient: OkHttpClient by lazy { … }

    val openLibraryApiService: OpenLibraryApiService by lazy {
        Retrofit.Builder()
            .baseUrl(OpenLibrary.BASE_URL)
            .client(okHttpClient)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
            .create(OpenLibraryApiService::class.java)
    }
}
```

```
data/di/AppContainer.kt

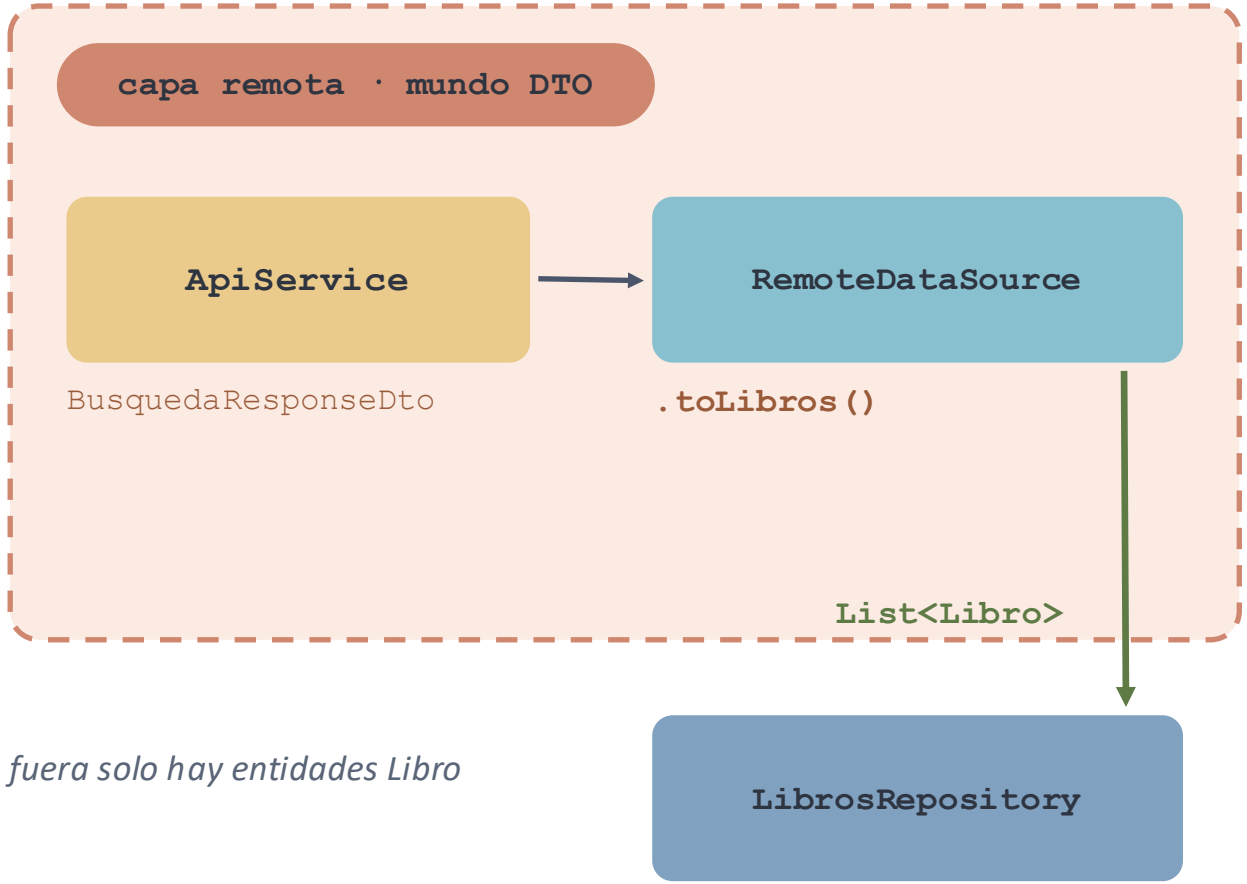
override val okHttpClient by lazy {
    RetrofitClient.okHttpClient }
private val remoteDataSource by lazy {
    RemoteDataSource(RetrofitClient.openLibraryApiService) }
override val librosRepository by lazy {
    LibrosRepository(localDataSource, remoteDataSource) }
```



Mismo pool de conexiones, mismo User-Agent y mismos logs

RemoteDataSource: frontera de los DTOs

DataSource



fuera solo hay entidades Libro

remote/RemoteDataSource.kt

```kotlin
class RemoteDataSource(
    private val api: OpenLibraryApiService
) {
    suspend fun buscarPorTitulo(
        titulo: String,
        limite: Int = 10
    ): List<Libro> =
        api.buscarPorTitulo(titulo,
                limite = limite)
            .docs.orEmpty()
            .toLibros()

    suspend fun buscarPorIsbn(
        isbn: String
    ): Libro? =
        api.buscar("isbn:$isbn", limite = 1)
            .docs.orEmpty()
            .firstNotNullOfOrNull {
                it.toLibro(isbn) }
}
```

04

Errores e imágenes

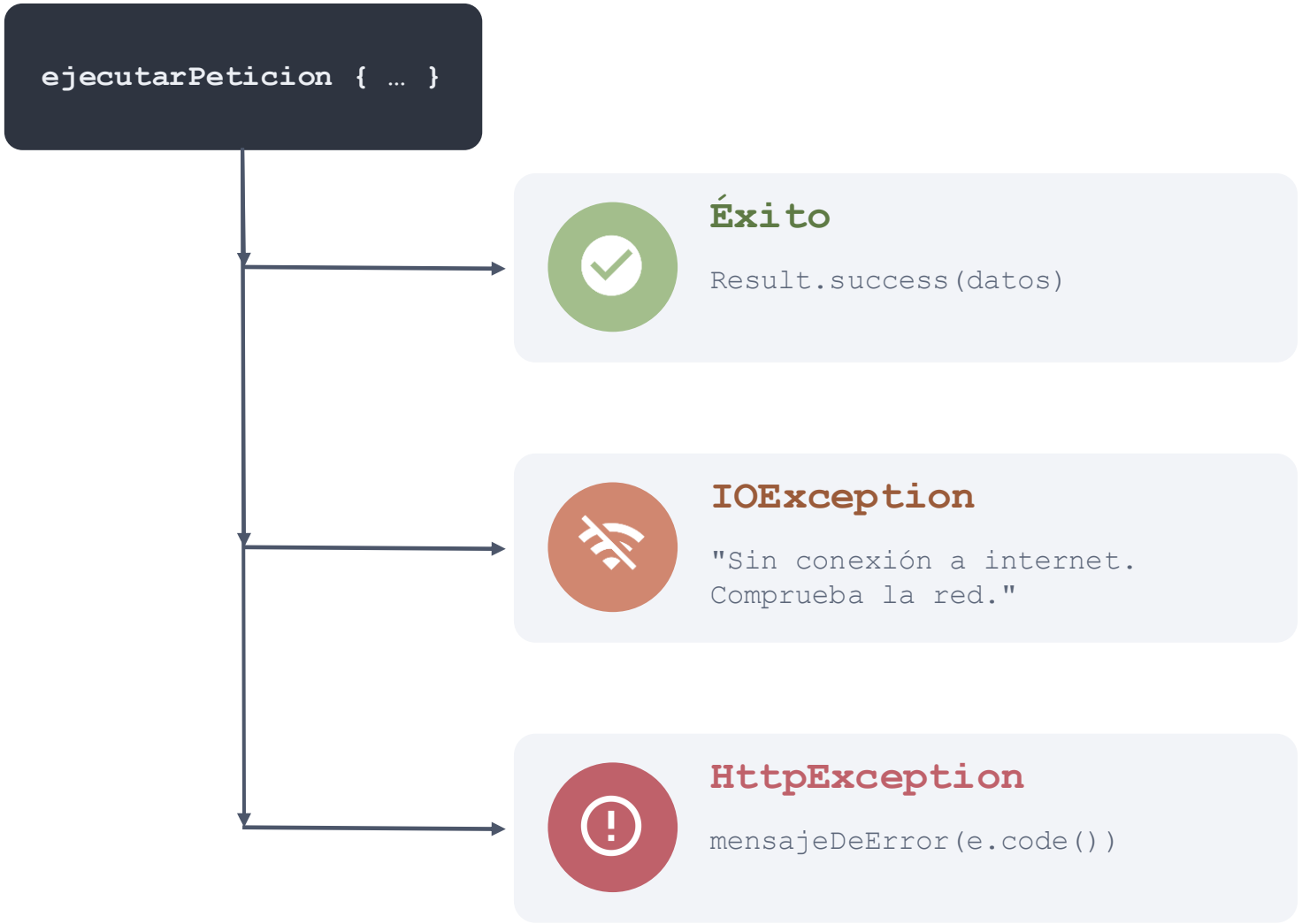
Result<T> en el repositorio y carátulas con Coil



¿Qué puede salir mal en una petición?

`Result<T>`

`ejecutarPeticion { … }`



`mensajeDeError(codigo)`

`403`	Límite / acceso denegado
`404`	Recurso no encontrado
`429`	Demasiadas peticiones
`5xx`	Error del servidor
`otro`	"Error HTTP $codigo"

El ViewModel nunca recibe excepciones de red

Del repositorio al ViewModel con Result<T>

Result<T>

LibrosRepository.kt

```kotlin
private suspend fun <T> ejecutarPeticion(
    bloque: suspend () -> T
): Result<T> = try {
    Result.success(bloque())
} catch (e: IOException) {
    Result.failure(Exception(
        "Sin conexión a internet. Comprueba la red."))
} catch (e: HttpException) {
    Result.failure(Exception(mensajeDeError(e.code())))
}

suspend fun buscarEnApiPorIsbn(isbn: String)
    : Result<Libro?> =
    ejecutarPeticion { remote.buscarPorIsbn(isbn) }
```

NuevoLibroViewModel.kt

```kotlin
fun buscarPorIsbn() {
    viewModelScope.launch {
        _uiState.update { it.copy(buscando = true) }
        repository.buscarEnApiPorIsbn(isbn)
            .onSuccess { libro -> … rellenarCon(libro) }
            .onFailure { e ->
                _uiState.update {
                    it.copy(mensaje = e.message) }
            }
        _uiState.update { it.copy(buscando = false) }
    }
}
```

.onSuccess

Rellena el formulario

.onFailure

Mensaje en Snackbar

Coil 3: un ImageLoader para toda la app

Coil

```
AppDummyApplication.kt
class AppDummyApplication : Application(),
    SingletonImageLoader.Factory {

    lateinit var container: AppContainer

    override fun onCreate() {
        super.onCreate()
        container = DefaultAppContainer(this)
    }

    override fun newImageLoader(context: PlatformContext) =
        ImageLoader.Builder(context)
            .components {
                add(OkHttpNetworkFetcherFactory(
                    callFactory = { container.okHttpClient }))
            }
            .crossfade(true)
            .build()
}
```



`SingletonImageLoader.Factory`

Coil pide el ImageLoader a la Application la primera vez que carga una imagen.



`OkHttpNetworkFetcherFactory`

Las descargas usan el mismo OkHttpClient que Retrofit.



`crossfade(true)`

Fundido suave al aparecer la imagen.

Dependencias: `io.coil-kt.coil3:coil-compose` · `io.coil-kt.coil3:coil-network-okhttp`

AsyncImage y los estados de una carátula

`@Composable`



Cargando
placeholder

Éxito
la carátula

Error / 404
error

CaratulaLibro encapsula AsyncImage: listado, detalle y formulario la reutilizan.

screens/componentes/CaratulaLibro.kt

```kotlin
@Composable
fun CaratulaLibro(
    url: String?,
    modifier: Modifier = Modifier
) {
    AsyncImage(
        model = url,
        contentDescription = "Carátula",
        placeholder = painterResource(
            R.drawable.ic_cargando),
        error = painterResource(
            R.drawable.ic_sin_portada),
        contentScale = ContentScale.Crop,
        modifier = modifier
    )
}
```

05

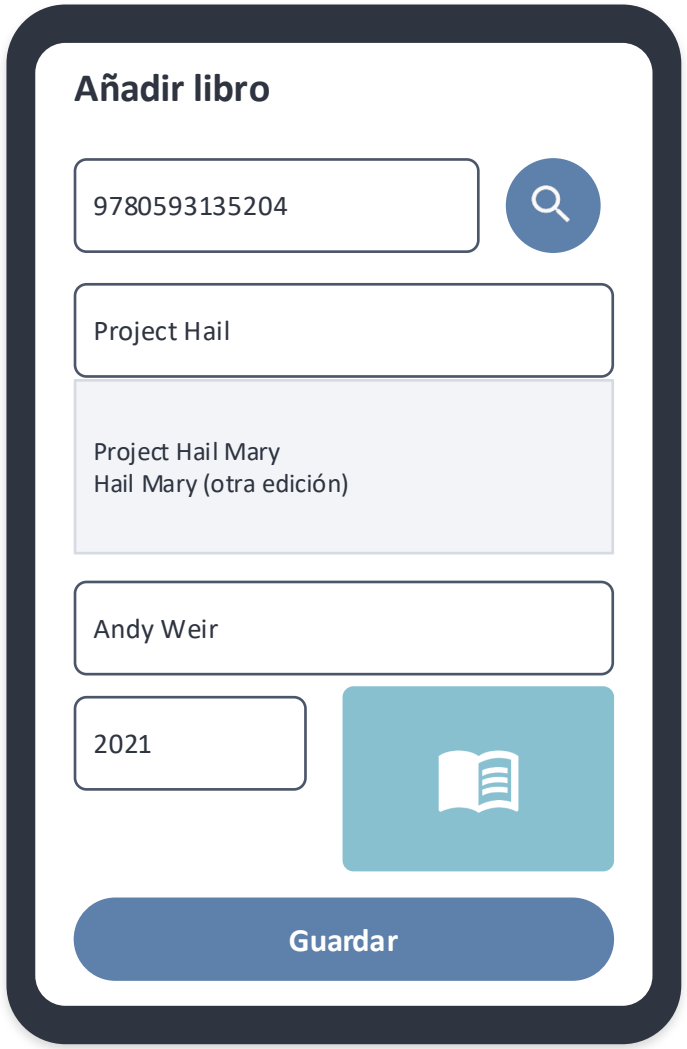
Pantalla «Añadir libro»

Formulario, validación de ISBN, debounce y guardado en Room



La pantalla NuevoLibro de un vistazo

`screens/nuevo`



1

Búsqueda por ISBN

Valida el dígito de control y llama a buscarEnApiPorIsbn().

2

Sugerencias por título

Tras 500 ms sin teclear (≥ 3 caracteres) consulta la API.

3

Datos rellenables

Autor, año, ISBN y carátula se completan al elegir sugerencia.

4

Guardar en Room

Activo solo si puedeGuardar; inserta y vuelve al listado.

Validar un ISBN-13 antes de llamar a la API

`utils/Isbn.kt`



Σ = 100

`100 % 10 == 0  →  ISBN válido`

1. Quitar lo que no sea dígito (guiones, espacios)
2. Exigir exactamente 13 dígitos
3. Pesos alternos 1 y 3, empezando por 1
4. La suma debe ser múltiplo de 10

utils/Isbn.kt

```kotlin
fun esIsbn13Valido(s: String): Boolean {
    val d = s.filter { it.isDigit() }
    if (d.length != 13) return false
    val suma = d.mapIndexed { i, c ->
        val peso = if (i % 2 == 0) 1 else 3
        c.digitToInt() * peso
    }.sum()
    return suma % 10 == 0
}
```

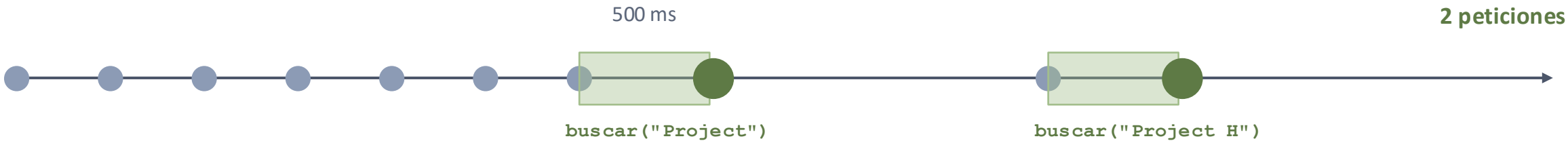
Debounce: esperar a que el usuario deje de teclear

Flow

Sin debounce



Con debounce(500 ms)




```
NuevoLibroViewModel · init

consultaTitulo
    .debounce(500.milliseconds)
    .distinctUntilChanged()
    .filter { it.length >= 3 }
    .collectLatest { buscarSugerencias(it) }
```

debounce
Espera 500 ms de silencio

distinctUntilChanged
Ignora el mismo texto repetido

filter ≥ 3
No busca con menos de 3 letras

collectLatest
Cancela la búsqueda anterior

NuevoLibroUiState: una sola fuente de verdad

UDF

```
data class NuevoLibroUiState
```

Formulario

`titulo` `autor` `year` `isbn` `cover`

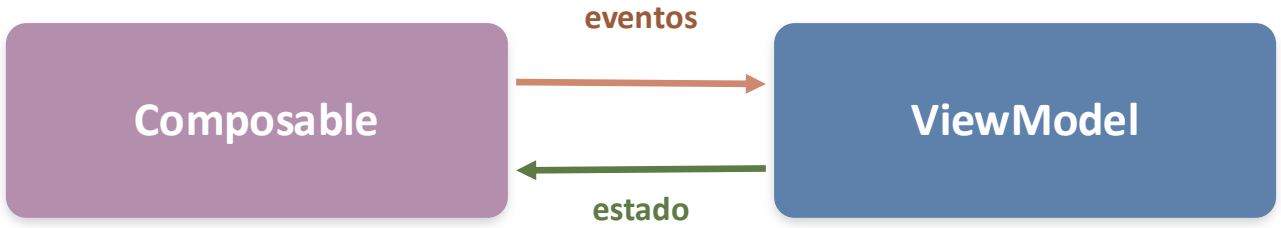
Búsqueda

`idOpenLibrary` `sugerencias` `buscando`

Resultado

`mensaje` `guardado`

`val puedeGuardar: Boolean`



Funciones públicas del ViewModel

- `actualizarTitulo()`
- `actualizarAutor()`
- `actualizarYear()`
- `actualizarIsbn()`
- `buscarPorIsbn()`
- `seleccionarSugerencia()`
- `guardar()`
- `limpiarMensaje()`

Privadas: `buscarSugerencias()` · `rellenarCon()`

Gson frente a kotlinx.serialization

Serialización

Aspecto	Gson	kotlinx.serialization
Configuración	✅ Solo dependencia	❌ Plugin + dependencia
Rendimiento	❌ Reflexión (más lento)	✅ Código generado (más rápido)
Nulabilidad y valores por defecto	❌ Los ignora	✅ Los respeta
Multiplataforma (KMP)	❌ No	✅ Sí
@Serializable de Navigation	❌ No	✅ Sí
DTOs nullable obligatorios	❌ Sí	✅ No
Complejidad para empezar	✅ Baja	❌ Media

¿Por qué Gson en el curso? Menor curva de entrada y ya usado en los TypeConverters de T4.

06

Práctica guiada

AppDummy v7: estructura final y casos límite



Estructura de paquetes al final de T5

AppDummy

```
com.ejemplo.appdummy/
├─ AppDummyApplication.kt
├─ data/
│  ├─ datasource/local/  (T4)
│  ├─ datasource/remote/
│  │  ├─ dto/  LibroDto · BusquedaResponseDto
│  │  ├─ OpenLibraryApiService.kt · RetrofitClient.kt
│  │  ├─ RemoteDataSource.kt · LibroMapper.kt
│  ├─ repository/LibrosRepository.kt
│  ├─ di/AppContainer.kt
├─ screens/componentes/CaratulaLibro.kt
├─ screens/nuevo/  (pantalla + ViewModel)
├─ utils/Isbn.kt
```

```
app/src/test/…/LibroMapperTest.kt
```

Nuevo en T5

Capa remota, componentes, utilidades

Modificado

Application, repositorio, contenedor

Sin cambios

Capa local de Room (T4)

La entidad Libro de T4 no cambia: el DTO absorbe la forma del JSON.

Desarrollo práctico guiado: de v6 a v7

Práctica

1 **Configurar dependencias**

libs.versions.toml, buildConfig e INTERNET

2 **Crear la capa remota**

DTOs, mapeador, ApiService, RetrofitClient, DataSource

3 **Tests del mapeador**

Escribirlos y ejecutarlos en app/src/test

4 **Eliminar la lista estática**

Los libros llegan de Open Library

5 **Formulario «Añadir libro»**

Ruta, ViewModel con debounce y guardado

6 **Sustituir Patterns.WEB_URL**

Según indica la documentación del tema

Probar casos límite

- ISBN inválido
- Sin conexión
- Libro sin portada

Punto de partida: AppDummy v6 (final del Tema 4). La práctica completa tiene 12 tareas: aquí se agrupan las principales.

Ideas clave

Interfaz anotada

@GET y @Query describen la petición; Retrofit la implementa.

DTO ≠ entidad

El DTO copia el JSON; el mapeador produce Libro.

Result<T>

IOException y HttpException no llegan al ViewModel.

Un OkHttpClient

Compartido por Retrofit y Coil, con User-Agent.

Respeta la API

fields, debounce y validación local del ISBN.

?default=false

Sin portada → 404 → imagen de error en Coil.

Referencias

Fuentes

Retrofit (Square)
square.github.io/retrofit

OkHttp (Square)
square.github.io/okhttp

Open Library Search API
openlibrary.org/dev/docs/api/search

Open Library Covers API
openlibrary.org/dev/docs/api/covers

Coil: carga de imágenes para Compose
coil-kt.github.io/coil/compose

Capa de datos (guía de arquitectura de Android)
developer.android.com/topic/architecture/data-layer

Tema 5. Retrofit2: consumo de APIs REST (apuntes)
documentation.javiercarrasco.es/pmdm/B3/B3-T5_Retrofit_OpenLibrary