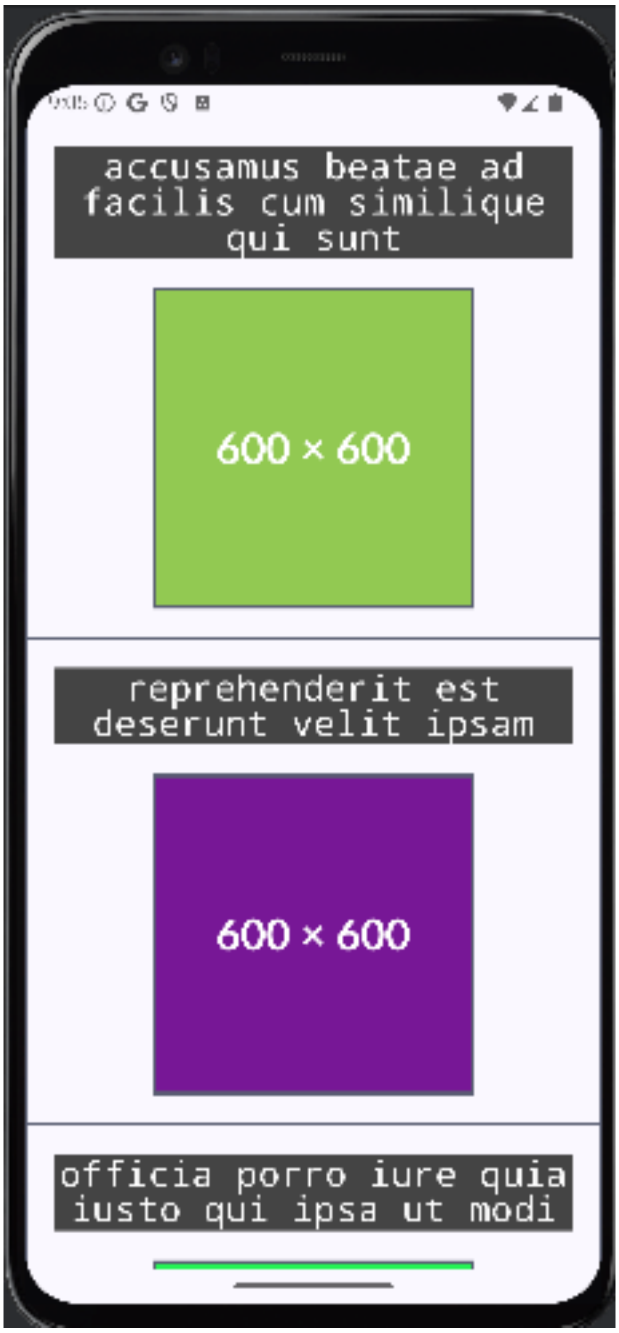
Explanation of Design Choices and Architecture of LBC Project

Author: Duarte Golaio Gonçalves

Date: 19/03/2025


accusamus beatae ad facilis cum similique qui sunt
600 × 600
reprehenderit est deserunt velit ipsam
600 × 600
officia porro iure quia iusto qui ipsa ut modi

Introduction

This document explains the design decisions and architecture implemented in the **LBC Test Album Titles** project. The goal of this project is to fetch album titles from an external API, store them locally using **Room** Database, and display them to the user using **Jetpack Compose**. The project follows the **MVVM (Model-View-ViewModel)** pattern, incorporating **Clean Architecture** principles and modern Android development tools like **Hilt**, **Flow**, and **Coroutines**.

Architecture

Clean Architecture

The project is structured using **Clean Architecture**, which divides the application into different layers with well-defined responsibilities.

1. **Presentation Layer (UI)**

2. **Domain Layer (Business Logic)**

3. **Data Layer (Data Management)**

Why MVVM?

1. **Easier integration with JetPack Compose**

   Allows an reactive UI throught observations in States inside the viewModel.

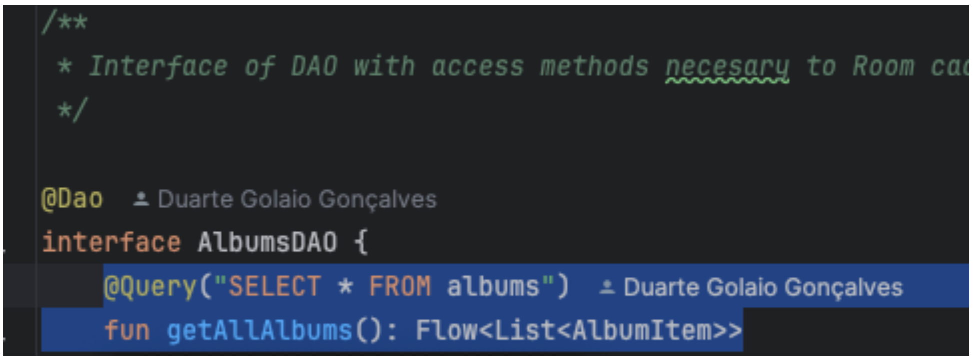
2. **Easier to Prevail state with Configuration Changes (Screen Rotation)**

3. **Better Separation of Responsabillities**

   Better separation between Data, Presentation Layer (UI) and Domain logic.

Why Flow?

Flow allows the UI to automatically receive updates whenever data changes in the database (Room) or when new data is fetched from the API.



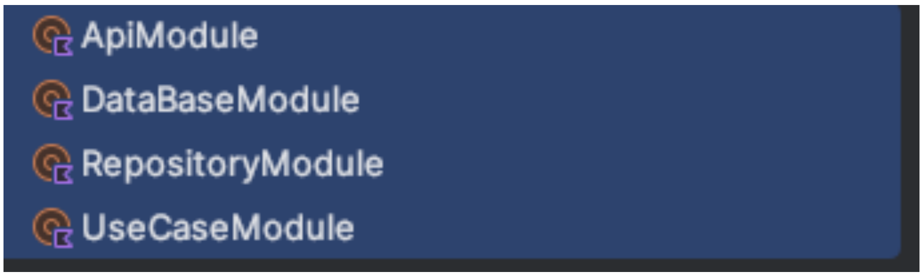
```
/**
 * Interface of DAO with access methods necesary to Room cac
 */

@Dao   Duarte Golaio Gonçalves
interface AlbumsDAO {
    @Query("SELECT * FROM albums")   Duarte Golaio Gonçalves
    fun getAllAlbums(): Flow<List<AlbumItem>>
```

It allows also to collect and Handle Asynchronous Data Streams in an efficient way.

Why HILT?

Hilt simplifies dependency injection , I managed to reduce boilerplate Code.

I have created a separation of Modules to Inject of different compartments of the App



ApiModule
DataBaseModule
RepositoryModule
UseCaseModule

In this way I could provide some of the utilities and instantiate some Classes directly.

Example: I could Inject directly the UseCase to the viewModel

```
 @HiltViewModel
class AlbumTitlesViewModel @Inject constructor(
    private val getAlbumItemUseCase: GetAlbumItemUseCase
) : ViewModel() {
```

Why Coroutines?

To run asynchronously the executions and at the same time to prevent Memory leaks.

Example:

```
viewModelScope.launch {
    try {
            _isRefreshing.value = true
            getAlbumItemUseCase.getAlbumItems().collect { albumList ->
             _albumItems.value = albumList
             delay(200) //added this just for the visual efect,
because the info is downloaded fast
             _isRefreshing.value = false
        }
    } catch (e:Exception) {
        e.printStackTrace() // TODO: Manage posible error
    }
```

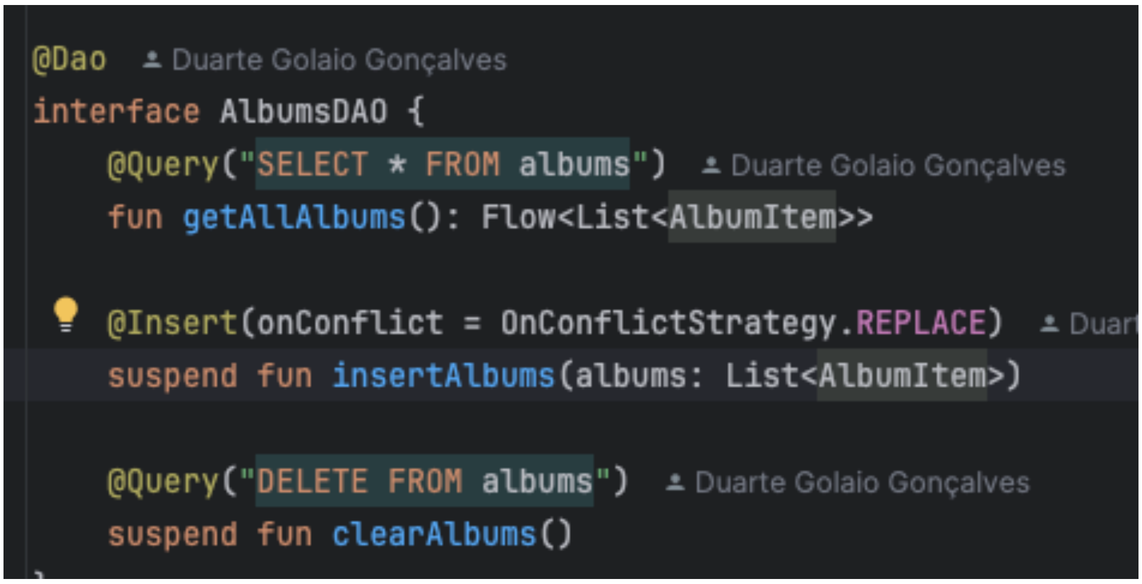
In this project it was utilized the viewModelScope. Once the ViewModel ceases to exist, the coroutine ceases to exist also without any manual cancelation

Why Room?

It was used because it's the state of the art in terms of SQLite management libraries.

It allows to store a local db making it optimal to the cache, or offline Mode of the app.

In this case It was created a table with the corresponding Album Item and it was stored locally when some new info was fetched.



```
@Dao   Duarte Golaio Gonçalves
interface AlbumsDAO {
    @Query("SELECT * FROM albums")   Duarte Golaio Gonçalves
    fun getAllAlbums(): Flow<List<AlbumItem>>

    @Insert(onConflict = OnConflictStrategy.REPLACE)   Duar
    suspend fun insertAlbums(albums: List<AlbumItem>)

    @Query("DELETE FROM albums")   Duarte Golaio Gonçalves
    suspend fun clearAlbums()
```

It Facilitates the access modes to the DataBase and also integrates well with Kotlin Flow.

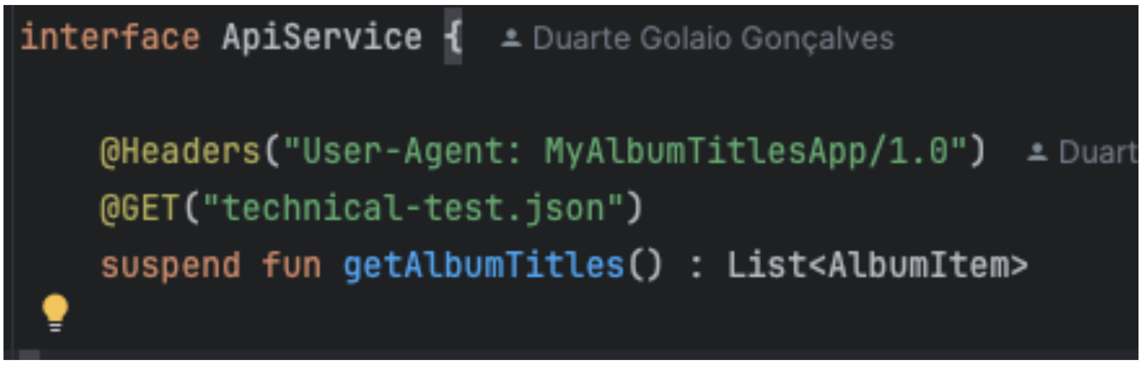
Why Retrofit?

It shares a little bit of the same logic of facilitation of Room but applied to the API Calls.

Makes super simple and readable API Calls throught interfaces with the requests and headers and params (the example of the User-Agent)

Allows to Pass directly a Json Converter into an Object.

It was uses **Moshi** as the Json Converter, for no reason in particular, just being state of the art opposing GSON, for example.



```
interface ApiService {   Duarte Golaio Gonçalves

    @Headers("User-Agent: MyAlbumTitlesApp/1.0")   Duart
    @GET("technical-test.json")
    suspend fun getAlbumTitles() : List<AlbumItem>
```

Why Mockito?

Allows to mock, as the name indicates, Classes in the Test Environments and verify and assert functions and values

Mocking Classes:

```
apiService = mock(ApiService::class.java)
albumsDAO = mock(AlbumsDAO::class.java)
```

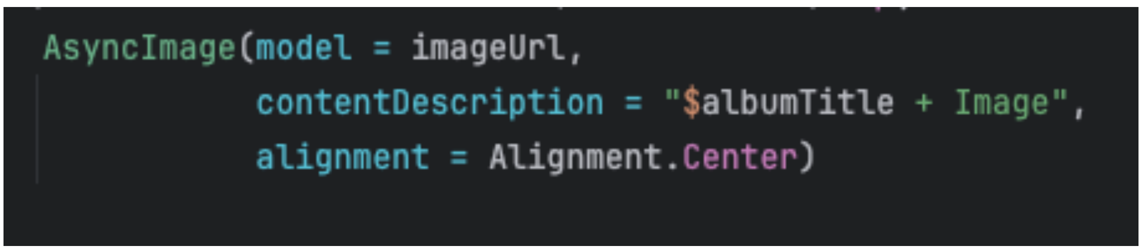
verifying calling of functions

```
verify(albumsDAO, never()).clearAlbums()
verify(albumsDAO, never()).insertAlbums(anyList())
```

Why Coil?

The Coil library was used in order to directly Pass the URL to an AsyncImageComposable that arrived from the API and present the image.

It was also used because it stores displayed images in caches, allowing to the app to present those images in offline Mode



```kotlin
AsyncImage(model = imageUrl,
           contentDescription = "$albumTitle + Image",
           alignment = Alignment.Center)
```

Why coroutines-test ?

This library was used to the ViewModel Tests, in order to test coroutines functions.

It was used to construct a test Dispatcher to execute in the @Before as condition to execute these coroutine Functions.

```kotlin
@Before
fun setup() {
    // has to put a test Dispatcher since,
    // some functions of the VM call Coroutines
    Dispatchers.setMain(testDispatcher)



...



//"Advances time" of the coroutin inside the viewModel
// since it is a mocked environment with a test dispacher
testDispatcher.scheduler.advanceUntilIdle()
```

Why the use of **accompanist-swiperefresh** ?

This library was used to provide a refresh Function to the LazyColumn that contained the items.

```kotlin
SwipeRefresh(
    state = rememberSwipeRefreshState(isRefreshing = isRefreshing),
    onRefresh = { viewModel.loadAlbumTitles() }
) {
```