The MovieDB App

- Pablo Perotti -

Table of Contents

1 - Introduction

The TMDB App is a demo application created for the whole purpose of being used during interviews.

2 - TLDR;

If you are in a rush, this section is for you.

The goal of the app is pretty straightforward: present a list of latest movies to the user in the form of a list, showing each movie in the form of a Card where a poster, a title, a rating and an overview are shown. Once the user chooses a movie, the details are presented. The application changes its layout depending on the device's orientation.

Technically speaking, the app uses Kotlin, Jetpack Compose, Coroutines, Flows, Room, Gson, Retrofit and Coil, it follows UDF principle and Google's Architecture Guidelines.

For further details, see the rest of the document.

3 - Technical Decisions

This section covers the rationale behind the project's technical decisions.

3.1 - Libraries

The project is written 100% in Kotlin since this is the actual standard. No Java code should be used in new features in modern applications.

The project uses the following libraries and key dependencies:

Jetpack Compose	This is the default toolkit currently available to build native UI for Android declaratively. it's worth mentioning that, at this point in time, building the UI using IntelliJ's Compose Multiplatform is also a valid choice, specially if at any point in time in the future we expect to support other platforms than just Android.
Hilt	This library is the official library supported by Google to handle dependency injection. Each class is designed with testing in mind. This means that every class receives its dependencies via the constructor and all classes that belong to the project (in contrast to those from libraries) define its public functionality in an interface. Classes do not depend on classes but their interfaces.
Gson	I chose this library to handle json serialization because it's been in the market for enough time to make it a default option. Other options could have been using Moshi or IntelliJ's serialization library but there was no need to consider them for this project.
Retrofit/OkHttp	The default option to handle networking in an application. OkHttp behind the scenes let us control the granularity of the request's used across the app. In particular, network interceptors are used here to send the Authentication Bearer token in all network requests and to control the logging level. If we have to choose an alternative to this library with multiplatform support in mind, I'd choose Ktor with is default for KMP.
Room	This library is the default option these days to isolate the interaction with the SQlite database.
Coil	This is an efficient library used to perform image handling with support for Jetpack Compose.
Navigation	This library coordinates how the app transitions between screens. Using this library is the current standard to do this task.

3.2 - Project's Structure

The project is organized following Google's guideline (https://developer.android.com/topic/architecture), which means that scope of the application is splitted in different layers where each layer has a very distinct, clear and unique responsibility.

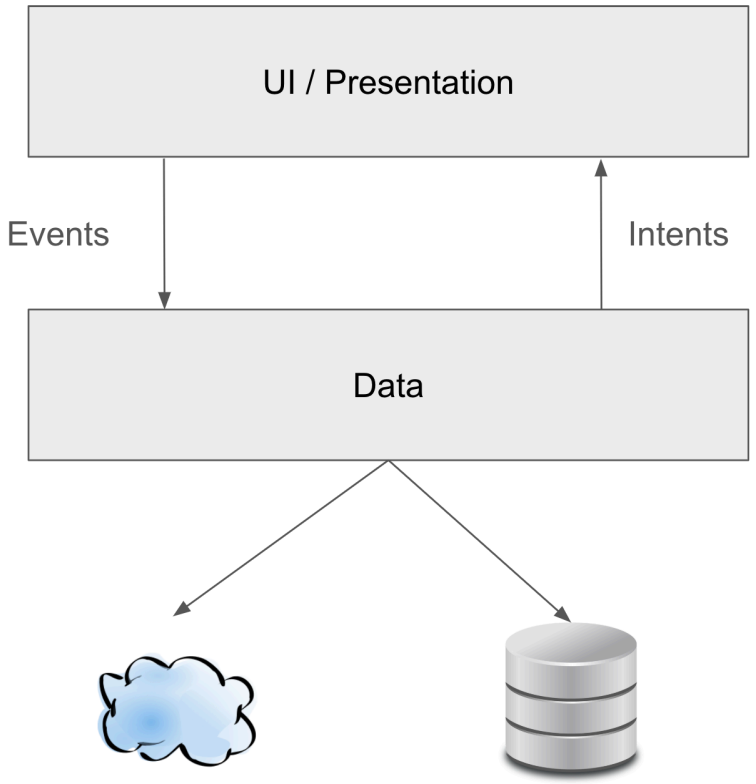
From Google's suggestion, I used 2 layers. They are:

- **Ui/Presentation**: This layer contains all the screens, view models and theme dependencies. All "state holder"'s classes are the ones interacting with the repository from the data layer and transforming that response in the appropriate ui states.

  Classes modelling the UI interacts with a NavController to handle the transition between screens.

- **Data**: Classes here are responsible for retrieving data from the network, persisting the responses in a local storage and advertising their results.

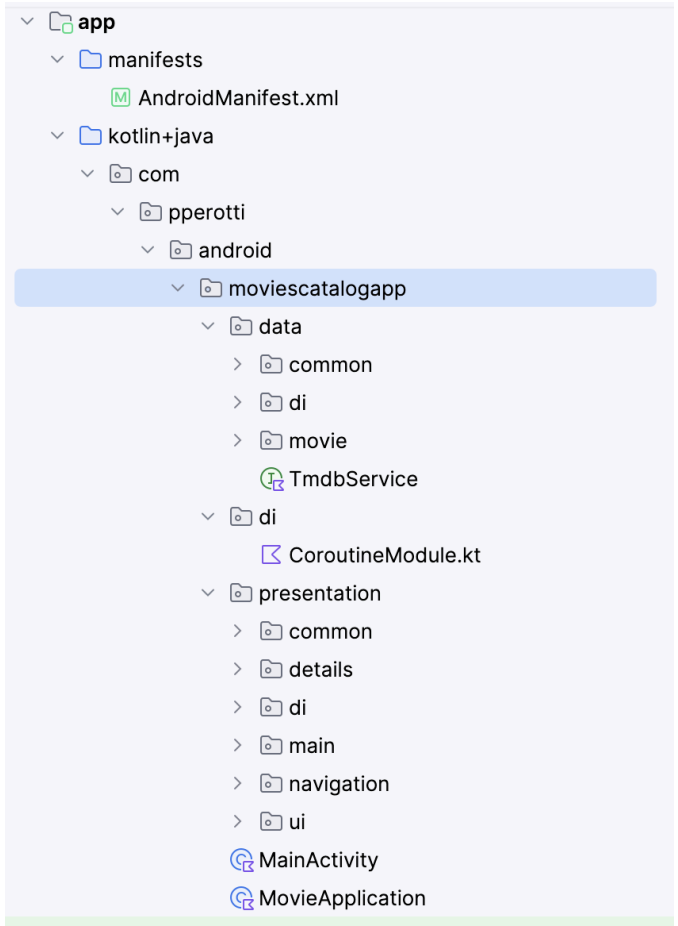
This can be summarized with the following image:



In the image above the key part, on the left, is that "actions" or "system events" that come from the user or activity components are sent to the data layer where the core of the business logic occurs. There, the appropriate repository will decide whether it needs to interact with the network or the local storage, prepare the response and publish it again in the StateFlow used for this purpose.

Layers mentioned above have a 1-1 mapping with how directories exist in the project. Each layer contains more than one directory to split it's dependencies according to its responsibilities.

The project directories looks as follow:

<table>
<tr>
<td> 
</td>
<td>data: Represent the “data” layer. It is the most important since it contains the business logic and it is the one that drives what should happen next.</td>
</tr>
<tr>
<td></td>
<td>data/common: Contains objects that can be reused by more than one repository</td>
</tr>
<tr>
<td></td>
<td>data/di: Contains all the modules and dependencies used by this layer.</td>
</tr>
<tr>
<td></td>
<td>data/movie: Contains the repository and classes required to interact with TMDB service.</td>
</tr>
<tr>
<td></td>
<td>di: Contains dependency that may be shared between presentation & data layers.</td>
</tr>
<tr>
<td></td>
<td>presentation: Contains all UI-related & StateHolder classes. It represents the “presentation” layer as a whole.</td>
</tr>
<tr>
<td></td>
<td>presentation/common: Contains common composables used in more than one screen</td>
</tr>
<tr>
<td></td>
<td>presentation/details: Contains composables & view model associated with the details screen.</td>
</tr>
<tr>
<td></td>
<td>presentation/di: Inject dependencies to classes in this layer.</td>
</tr>
<tr>
<td></td>
<td>presentation/main: Contains composables & view model associated with the main screen.</td>
</tr>
<tr>
<td></td>
<td>presentation/navigation: Contains the component that handles Navigation across different screens.</td>
</tr>
<tr>
<td></td>
<td>presentation/ui/theme: Contains classes associated with improving the theme.</td>
</tr>
</table>

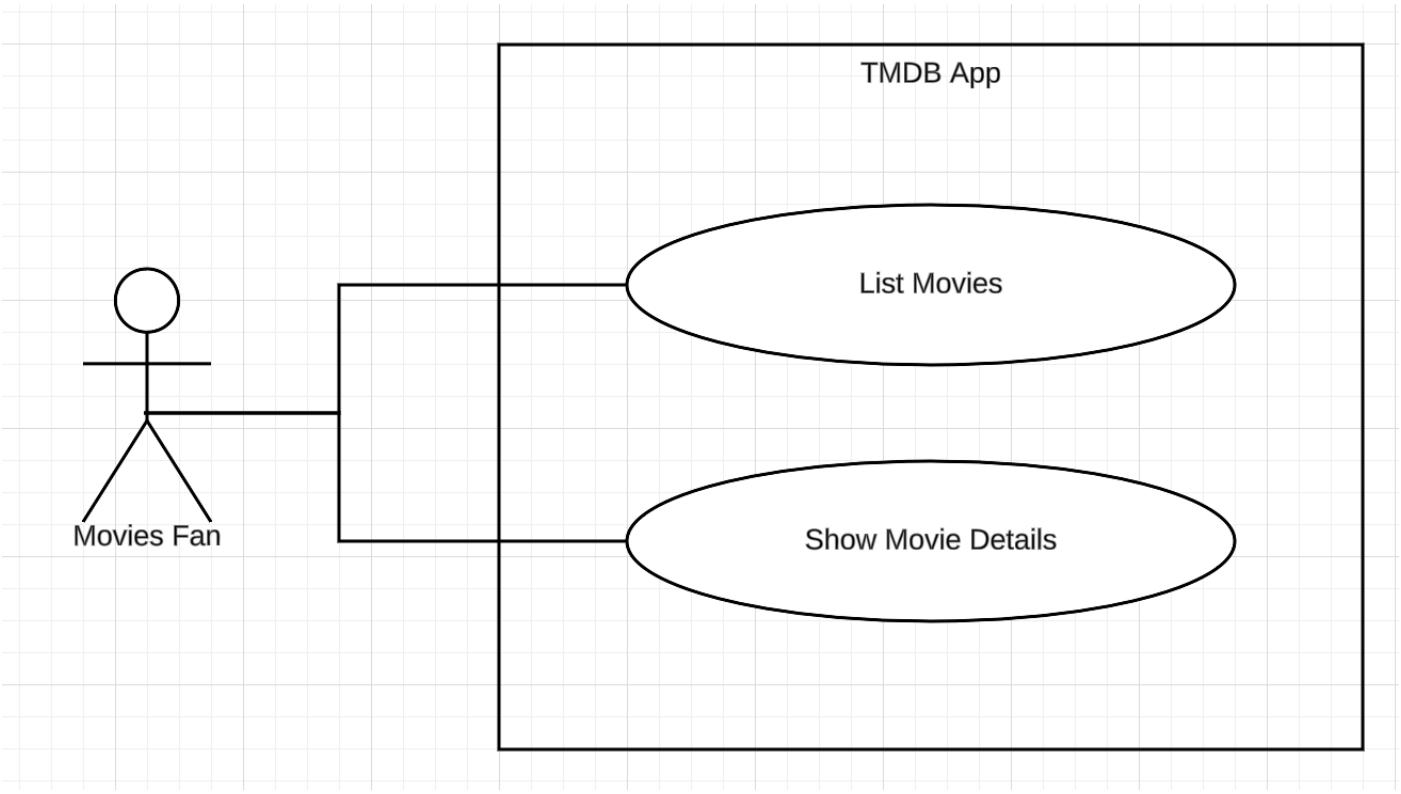
Outside of these classes, there is one directory for language, one for English (values) and one for Spanish (values-es). Each of these folders contains a file name strings.xml with the required per language.

4 - Architectural Diagrams

This section will contain different UML diagrams. I use UML as a Blueprint as defined in the book “Learning UML 2.0 from O’Reilly”. Briefly, this means that UMLs diagram should be used to clarify design decisions but at a high level. Sometimes, understanding the big-picture is not obvious to everyone when only code is available. Using diagramming before starting to code helps a team a lot by agreeing on the approach to follow. This practice usually ends up saving time later when it is time to do code reviews. .

4.1 - Supported Use Cases

4.1.1 - Use Cases Diagram

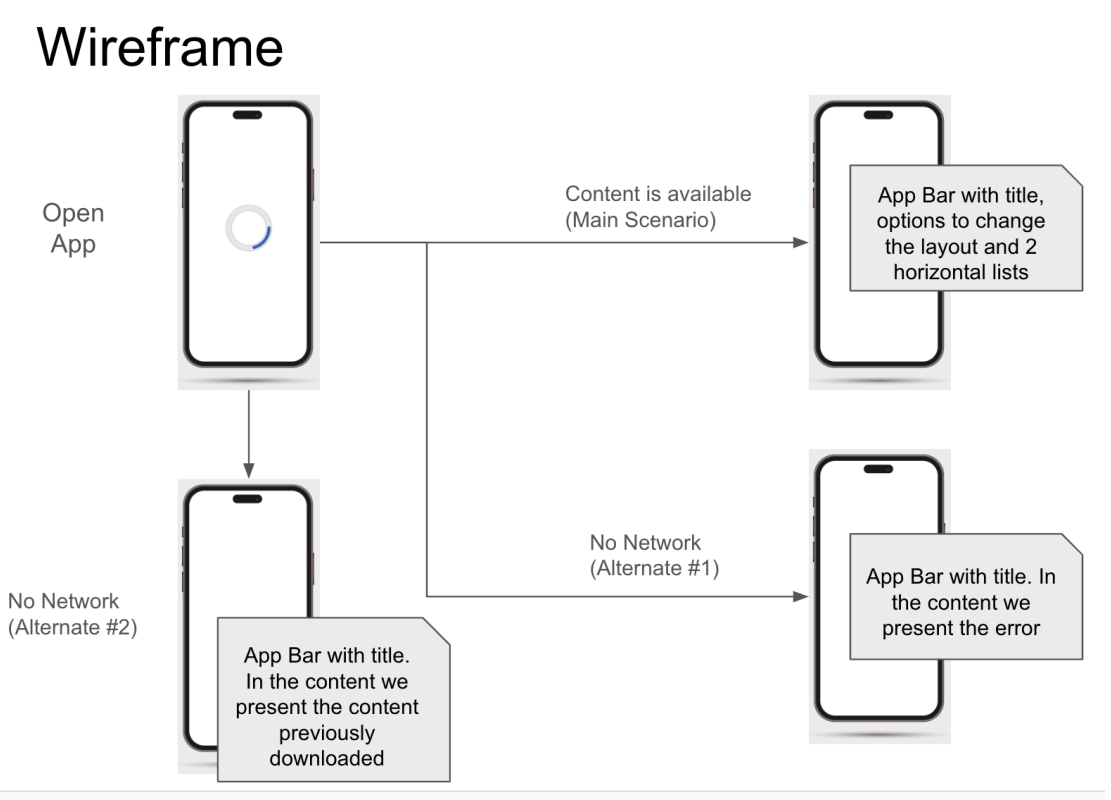


In the diagram above you can observe one actor and two main use cases. We call this actor “Movies Fan” since it represents the nature of a user of this type of application.

The following two sections will show the details about these use cases.

4.1.2 - Use Case - List Movies

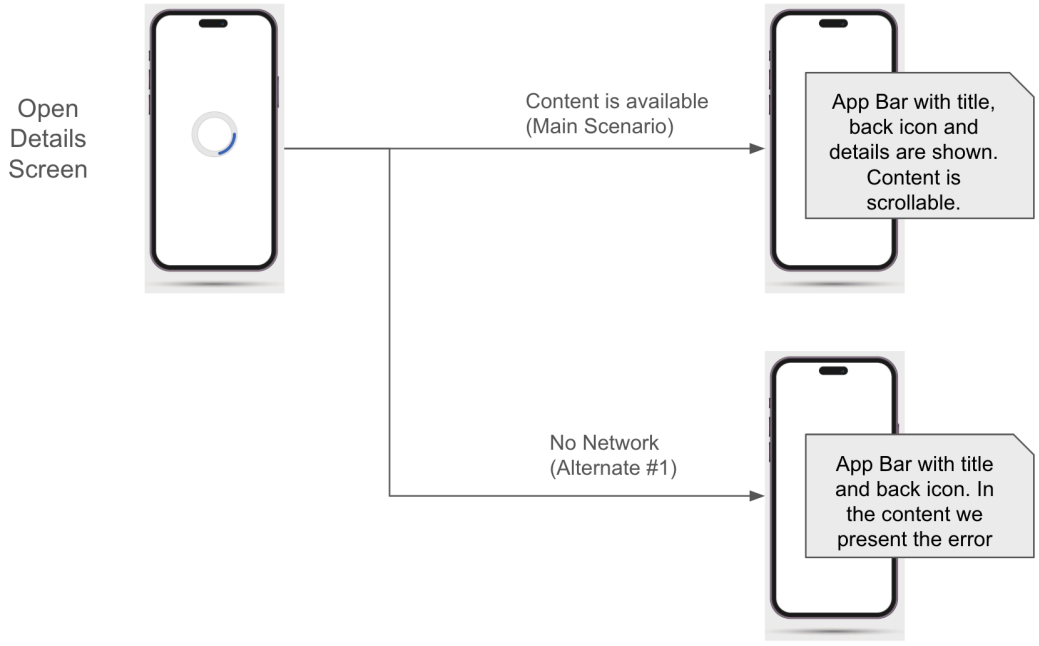
Description	List Movies	
Preconditions	App hasn't been started yet	
Related Actors	Movies Fan	
Successful End Criteria	Main Screen is shown and the user	
Failed End Criteria	Content cannot load and an error screen is presented	
Trigger	"Movies Fan" starts the app.	
Main Flow		
1	App Starts	1 - Main Screen is presented with a loading indicator. 2 - The screen is populated with a list of items.
Secondary Flow		
	Alternate Path #1	
Step 2	1 - Network is not available and the list hasn't yet shown any list before.	A error will be presented to 'Movies Fan'
	Alternate Path #2	
Step 2	2 - Network is not available and list has been loaded previously.	The content is displayed
Associated Wireframe		



4.1.2 - Use Case - Show Movie Details

Description	Show Movie Details	
Preconditions	App is showing the main screen with the list of movies.	
Related Actors	Movies Fan	
Successful End Criteria	Details Screen is loaded	
Failed End Criteria	An error occur and it displayed in the screen	
Trigger	“Movies Fan” selects and item from the main screen	
Main Flow		
1	Select Movies	1 - Details Screen is shown with a loading indicator presented as information is downloaded from the network. 2 - Details are displayed on the screen.
Secondary Flow		
Alternate Path #1		
Step 2	1 - Network is not available and the list hasn’t yet shown any list before.	A error will be presented to ‘Movies Fan’
Associated Wireframe		

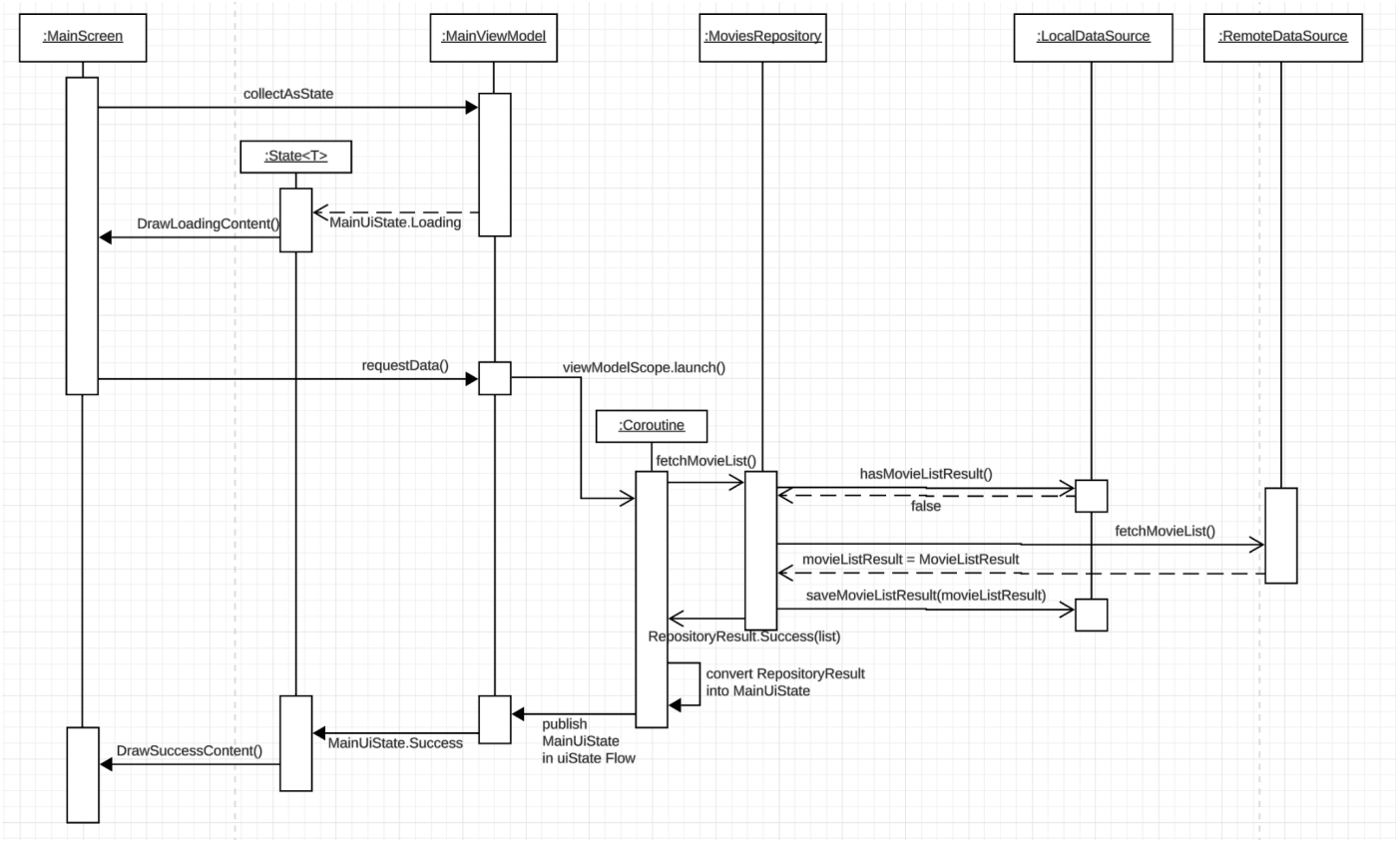
Wireframe



4.2 - Sequence Diagrams

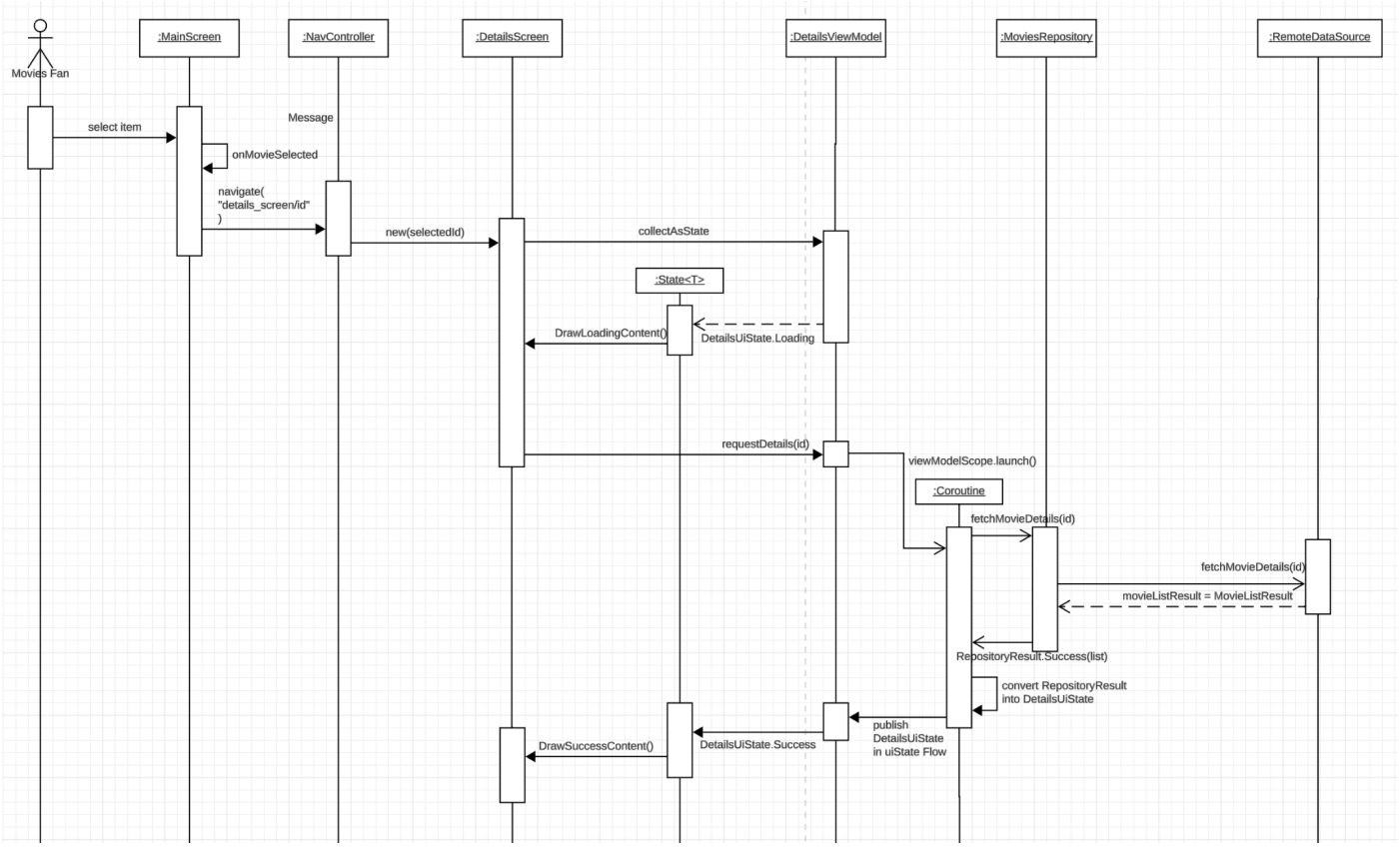
This section is where I only include relevant elements in the sequence diagram for the sake for simplicity. I'll add the appropriate notes in order to facilitate the understanding.

4.2.1 - Loading Main Screen



The key concepts to understand here is that MainScreen starts "collecting/observing" the view model so it starts receiving the next state that best describes what needs to be rendered. Initially when the screen is rendered for the first time, it receives MainUiState.Loading state immediately. Once the data is received from either data source, the Repository publishes its data in the form of RepositoryResult. Now, the view model "transforms" the RepositoryResult data into the next MainUiState and publishes it so MainScreen can then re-render/recompose properly.

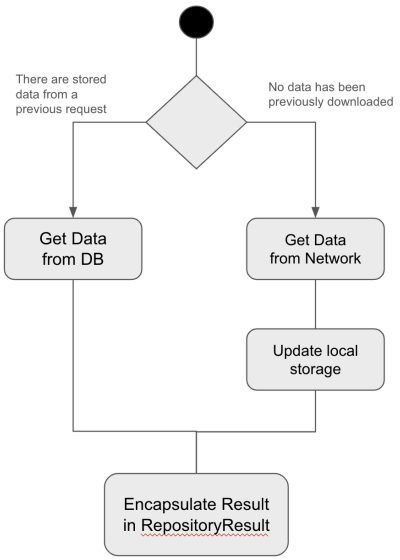
4.2.2 - Get Movie Details



This use case is very similar to the previous one in the sense that the details screen collects ui states from its view model and requests data to the MoviesRepository (which goes only to the network for this case). Once the data (or error) is received, it is encapsulated in a RepositoryResult. Once the ViewModel receives the result, it transforms it to the appropriate DetailsUiState and publishes it to the uiState Flow. Finally, once a new uiState is received by the DetailsScreen, it renders it accordingly.

4.3 - Activity Diagrams

4.3.1 - Choosing Between different data sources



This algorithm is pretty simple. Once the data has been downloaded, you just use that data.

if you haven't persisted any information yet, the app will try to download it, then update the database and then return the result.

A more efficient approach may be using a timestamp, hash or similar so the app can be smarter in determining when it is a good moment to update the stored data.

5 - Summary

The application presents the reviewer with enough information to understand why and how major components interact with each other and why the main decisions were made.

It is worth mentioning that the app and the documentation only attempt to present a modern solution using the latest tools available.