Up to date for Android 9, Android Studio 3.2, & Kotlin 1.3

Advanced Android App Architecture

FIRST EDITION

Real-world app architecture in Kotlin 1.3

By Yun **Cheng** & Aldo **Olivares Domínguez**

Advanced Android App Architecture

By Yun Cheng and Aldo Olivares Domínguez

Notice of Rights

Notice of Liability

Trademarks

Dedications

"To my mom, the first software engineer I ever knew."

— *Yun Cheng*

"To my family and friends, for all the support that I got during the writing of this book."

— *Aldo Olivares Domínguez*

About the Authors

Yun Cheng is an author on this book. Yun is a software engineer for the Runkeeper app at ASICS Digital in Boston, MA. If she's not running marathons or facilitating for the Girls Who Code club in Cambridge, MA, you can probably find her setting off the kitchen fire alarm with her cooking. You can also reach out to her on Twitter at @yuncheng13.

Aldo Olivares Domínguez is an author of this book. Aldo is a Software Engineer passionate about creating amazing apps with great user interfaces. He's been an Android Developer since 2012 working primarly as a Freelancer and Instructor. Twitter: @aldominio.

About the Editors

Nick Bonatsakis is a tech editor of this book. Nick is an accomplished software engineer with over a decade of experience in mobile development across both Android and iOS. He is a passionate technologist, musician, father and husband. He currently works as an independent consultant under his own company, Velocity Raptor Inc.

Matei Suica is a tech editor of this book. Matei is a software developer that dreams about changing the world with his work. From his small office in Romania, Matei is trying to create an App that will help millions. When the laptop lid closes, he likes to go to the gym and read. You can find him on Twitter or LinkedIn: @mateisuica

Vijay Sharma is the final pass editor of this book. Vijay is a husband, a father and a senior mobile engineer. Based out of Canada's capital, Vijay has worked on dozens of apps for both Android and iOS. When not in front of his laptop, you can find him in front of a TV, behind a book, or chasing after his kids. You can reach out to him on Twitter and LinkedIn @vijaysharm.

Tammy Coron is an editor of this book. She is an independent creative professional and the host of Roundabout: Creative Chaos. She's also the founder of Just Write Code. Find out more at tammycoron.com.

Manda Frederick is the managing editor of this book. She has been involved in publishing for over ten years through various creative, educational, medical and technical print and digital publications, and is thrilled to bring her experience to the raywenderlich.com family as Managing Editor. In her free time, you can find her at the climbing gym, backpacking in the backcountry, hanging with her dog, working on poems, playing guitar and exploring breweries.

About the Artist

Vicki Wenderlich is the designer and artist of the cover of this book. She is Ray's wife and business partner. She is a digital artist who creates illustrations, game art and a lot of other art or design work for the tutorials and books on raywenderlich.com. When she's not making art, she loves hiking, a good glass of wine and attempting to create the perfect cheese plate.

Table of Contents: Overview

Table of Contents: Extended

What You Need

To follow along with this book, you'll need the following:

- **Android Studio 3.3.2**, available at https://developer.android.com/studio/index.html. This is the environment in which you'll develop the apps in this book.

If you haven't installed the latest versions of Android Studio, be sure to do that before continuing on with the book.

Also, the sample app described in this book makes use of a third party API by the Movie DB to search and retrieve movie information. In order to use the search API, you must first get access to an API key from the Movie DB. To get your API own key, sign up for an account at www.themoviedb.org. Detailed steps will be given in the first chapter of the book.

Book License

By purchasing *Advanced Android App Architecture*, you have the following license:

- You are allowed to use and/or modify the source code in *Advanced Android App Architecture* in as many apps as you want, with no attribution required.
- You are allowed to use and/or modify all art, images and designs that are included in *Advanced Android App Architecture* in as many apps as you want, but must include this attribution line somewhere inside your app: "Artwork/images/designs: from *Advanced Android App Architecture*, available at www.raywenderlich.com."
- The source code included in *Advanced Android App Architecture* is for your personal use only. You are NOT allowed to distribute or sell the source code in *Advanced Android App Architecture* without prior authorization.
- This book is for your personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, coworkers or students; they would need to purchase their own copies.

All materials provided with this book are provided on an "as is" basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this guide are the properties of their respective owners.

B Book Source Code & Forums

This book comes with the source code for the starter and completed projects for each chapter. These resources are shipped with the digital edition you downloaded from store.raywenderlich.com.

We've also set up an official forum for the book at forums.raywenderlich.com. This is a great place to ask questions about the book or to submit any errors you may find.

About the Cover

Birds are, of course, perhaps most well known for their ability to build fantastic nests, and the satin bowerbird, which graces this cover, is no exception.

While many birds craft modest nests of sticks, mud and bits of softness collected here and there, the satin bowerbirds are much more ambitious with their structures. Perhaps one of nature's most creative and serious architects, these birds build "bowers" to attract a mate. They build and decorate with anything from berries to flowers to drinking straws to ballpoint pens. Interestingly, as they mature, they prefer to architect with objects of the color blue.

Like these birds, good engineers understand the importance of good architecture: It is ambitious, structurally sound, attractive and sets your work apart from other simple-nesting developers.

You can learn more about these creative and intelligent birds, here: https://en.wikipedia.org/wiki/Satin_bowerbird.

Section I: Building a Foundation

This section introduces you to topics that will serve as a foundation for your understanding of the UI architecture patterns. You'll get introduced to the sample project, an app named **WeWatch**, that allows users to keep track of movies to watch. You'll also learn concepts that aid architecture, including Android Architecture Components and dependency injection.

- **Chapter 1, Introduction**: This chapter explains what this book is about and its intended audience. By reading this chapter, you'll gain a better understanding of why apps need good architecture. You'll also get an introduction to the sample app that you'll be building throughout this book.
- **Chapter 2, MVC**: The sample project starts off written in the Model View Controller pattern, with the Android Activity serving as both the View and the Controller. In this chapter, you'll learn the history of the MVC pattern as applied to Android, and you'll learn why this UI architecture pattern fails to meet two primary standards for good code: separation of concerns and unit testability.
- **Chapter 3, Testing MVC**: Here, you'll get a quick review on writing unit tests with JUnit, and you'll learn why the MVC pattern results in poor unit testability of code.
- **Chapter 4, Android Architecture Components**: In this chapter, you'll get an overview of the Android Architecture Components and go into detail on the libraries used in the sample project at various points in the book: Room, ViewModel, LiveData and Data Binding.
- **Chapter 5, Dependency Injection**: An important concept in writing testable code is using dependency injection to inject mock objects into code. Here, you'll learn the theory behind dependency injection and create a practical sample project using Dagger 2, a popular dependency injection framework for both Java and Android.
- **Chapter 6, RxJava**: In this chapter, you'll get an overview of RxJava and go into detail on how the library is used in the sample project at various points in the book.

Chapter 1: Introduction

By Yun Cheng

Do you remember when you made your first "Hello World" app on Android? From there, you likely progressed to creating complex user interfaces to display data, made web calls to APIs and managed the persistence of data. As the Android apps you built became more complex, you might have wondered if there were coding best practices available to make your apps more extensible, maintainable and testable. Perhaps you even wondered how to architect your apps so they're best suited to your particular needs.

Given that Google (until very recently) did not provide an opinion on app architecture, Android developers were left to come up with their own. Architecture patterns like **MVC**, **MVP**, **MVVM**, **MVI** and **Viper** are debated passionately among Android developers. So, what are these patterns and which one is the best?

The short answer to the latter question is that it depends on your particular app and its needs. With that in mind, this book aims to guide you to an informed decision by answering the former question in detail.

What is this book?

Throughout this book, you'll work with one sample project named **WeWatch**. You'll build this project multiple times using each of these architecture patterns. During this process, you'll get a hands-on comparison of the patterns and gain a deeper understanding of the theory behind them.

Who is this book written for?

This book is for you if:

- You're a developer who already has a basic understanding of creating Android apps in Kotlin.
- You want to take your apps to the next level with robust architecture.
- You're familiar with unit testing with JUnit and want to write unit tests for your app.

How to use this book

It's not necessary to read the chapters in this book in order. Feel free to jump to the architecture pattern that interests you the most. If there are concepts that are covered in another chapter, you'll be directed to those chapters for more information.

For instance, the sample project uses the following Android Architecture Components at various points in the book: Room, LiveData, ViewModel and data binding, so you may want to read the Android Architecture Components chapter for more information. The project also makes use of RxJava in some chapters, so be sure to check out the RxJava chapter if you need a primer on that library.

Why is app architecture important?

The idea behind the app architecture patterns presented in this book is that they all exist to help you design your app in such a way that allows the app to be maintainable as it scales. Two concepts, in particular, are useful: **separation of concerns** and **unit testing**.

Firstly, separation of concerns deals with separating the components of your app by responsibility. For example, when you update the UI of your app with a fancy new design, you want to do so without having to change any of the other code, such as the underlying data.

As you add more features to your app, you want to do so without having to change too much of your existing code.

Finally, as your app grows, you want to be able to test the app to ensure you didn't break the logic of existing features.

Now that you know the motivation behind app architecture, it's time to get yourself acquainted with the sample project in the book.

Introducing the sample project

WeWatch, the app you'll build in this book, keeps a list of movies you want to watch, allowing you to easily add and delete movies within the app. To add a movie, a user can manually enter the movie or search for movies from a database of movies provided by **The Movie Database** (www.themoviedb.org) API.

The Movie Database API key

Any app that wants access to The Movie Database API must provide an API key in the network call to identify itself to the API. You'll need to obtain your own API key to work with the sample code within this book.

To obtain your API key, sign up for an account at www.themoviedb.org. Then, navigate to your account settings on the website, view your settings for the API and register for a developer API key.

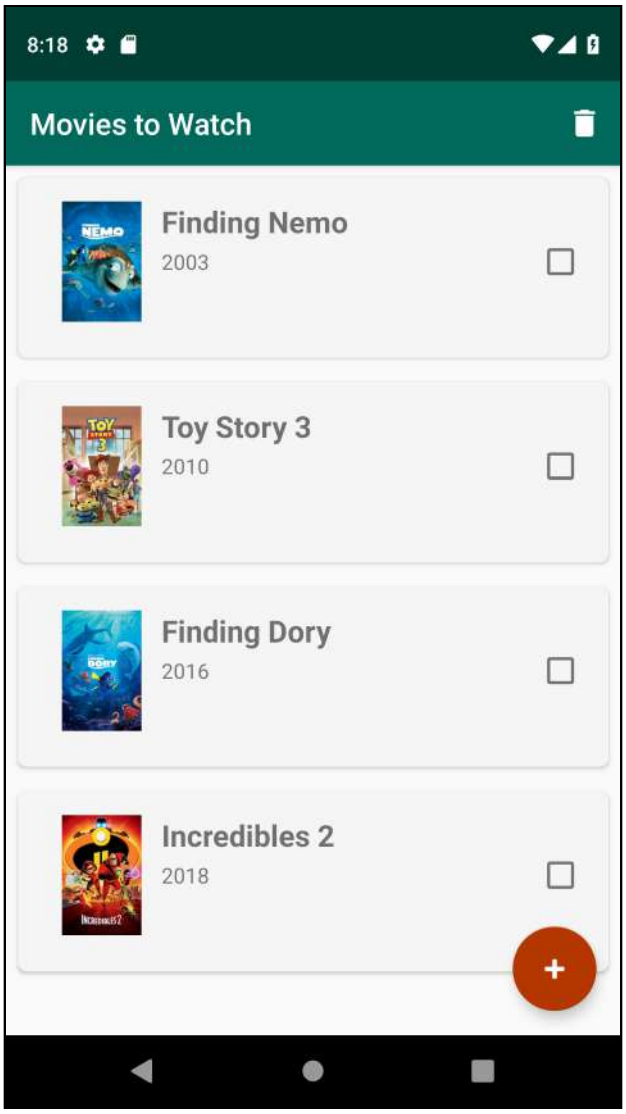
After receiving the API key, open the starter project for this chapter and navigate to **RetrofitClient.kt**. There, you can replace the existing value for `API_KEY` with your new key.

WeWatch sample app walkthrough

From the project resources open the starter project for this chapter in Android Studio. Take a moment to familiarize yourself with the structure of the project. Build and run the app to see what you're working with.

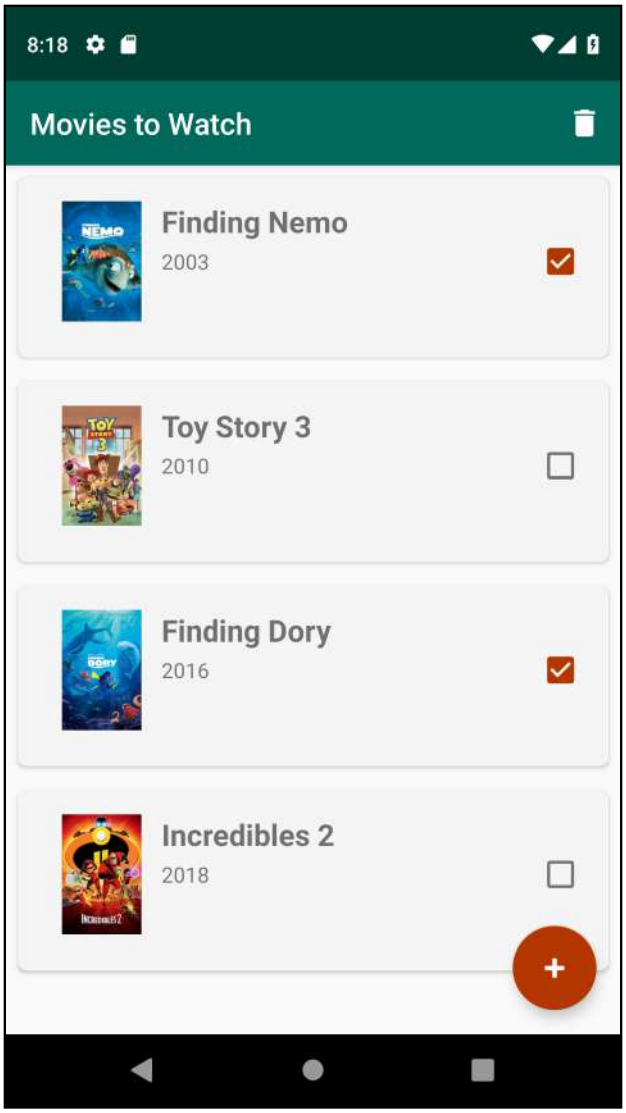
Main screen

The main screen displays the list of movies to watch. You'll find the code for this screen in **MainActivity.kt**.



Main screen of sample app.

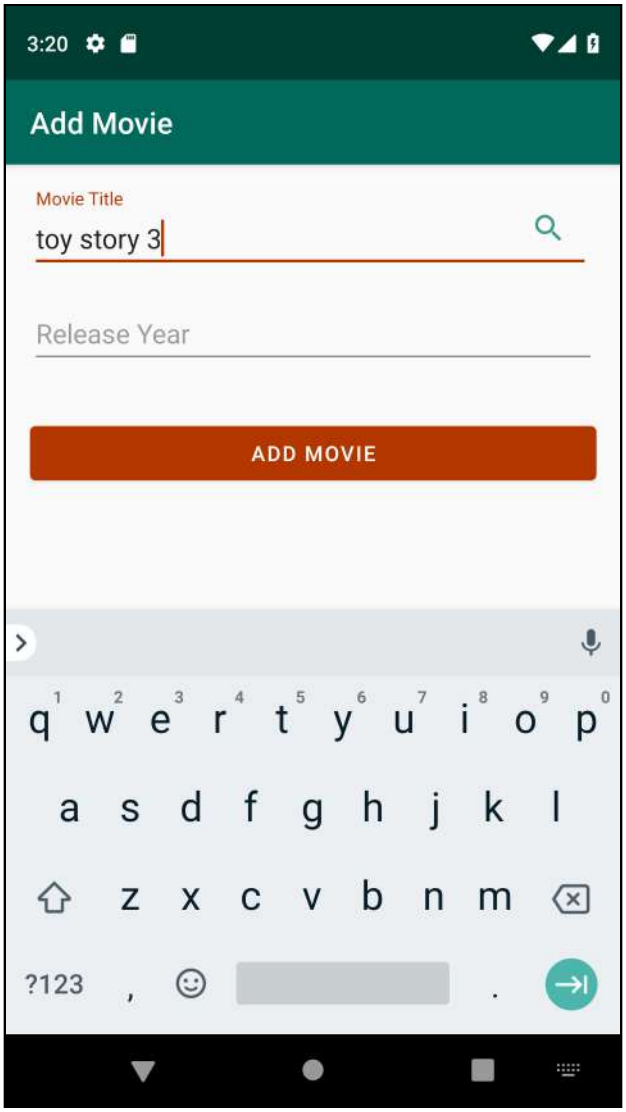
On this screen, you can select movies you want to watch; you can also delete movies you have already watched (or ones that received a terrible rating from rottentomatoes.com) from the list.



Select movies from list to delete.

Add movie screen

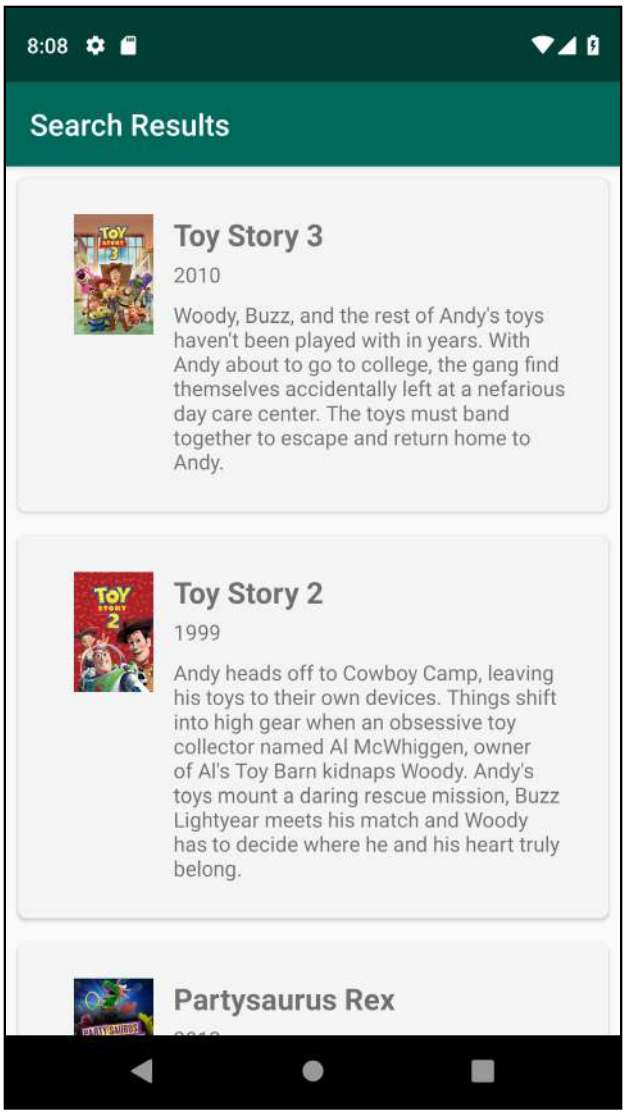
Clicking the floating action button from the main screen brings the user to the add movie screen where they can enter the title and release year of the movie. When they press the Add Movie button, the movie gets added to the to-watch list. You'll find the code for this screen in **AddMovieActivity.kt**.



Add movie screen of sample app.

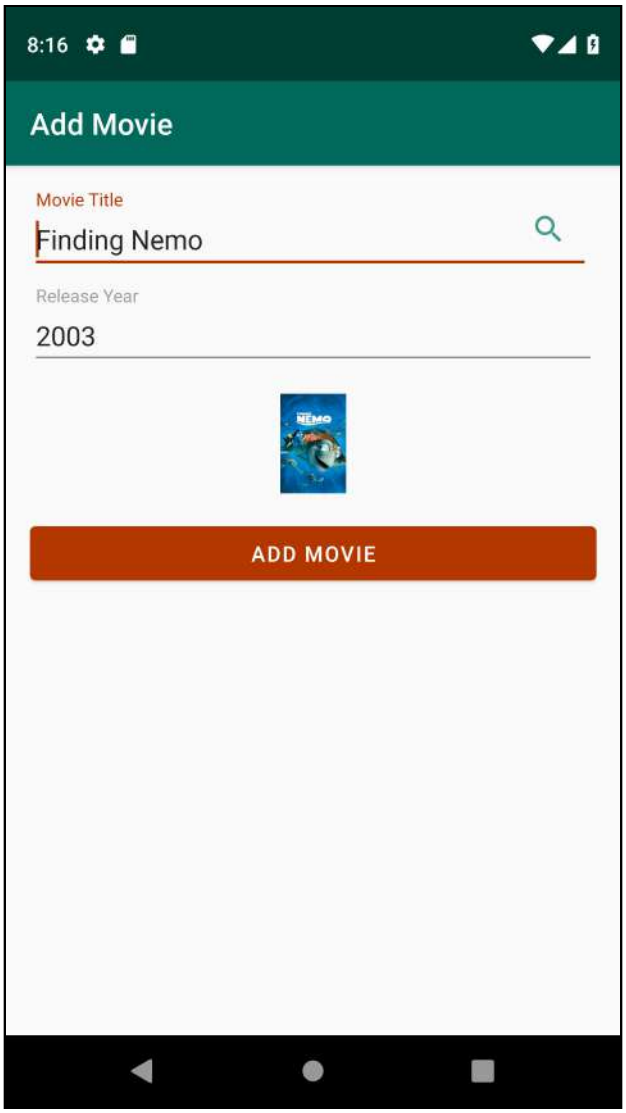
Alternatively, the user can enter the title of the movie and press the Search button to the right, which brings the user to the search movie screen.

Search movie screen



Search results screen of sample app.

For this screen, corresponding to **SearchActivity.kt**, a network call is made to the search endpoint of The Movie Database API, using the movie title passed in as the query. The resulting screen is a list of results matching the movie title. The user can then select a result to return to the add movie screen with the movie information pre-populated.



Add movie screen of sample app with the result from the search.

Where to go from here?

Now that you know the motivation behind this book and had an introduction to the sample project, you're ready to learn the theory behind the Model View Controller architecture. You'll learn how the MVC pattern is ironically not really a pattern in Android at all. This is the default architecture that the sample project uses, at least for now!

Chapter 2: Model View Controller

By Yun Cheng

In this chapter, you will learn about the **Model-View-Controller** pattern and be familiarized with the sample app that you will build throughout this book: **WeWatch**.

The Model-View-Controller pattern

The Model View Controller pattern is a pattern that separates components of a software system, based on responsibilities. In this pattern, three components with distinct responsibilities are defined as follows:

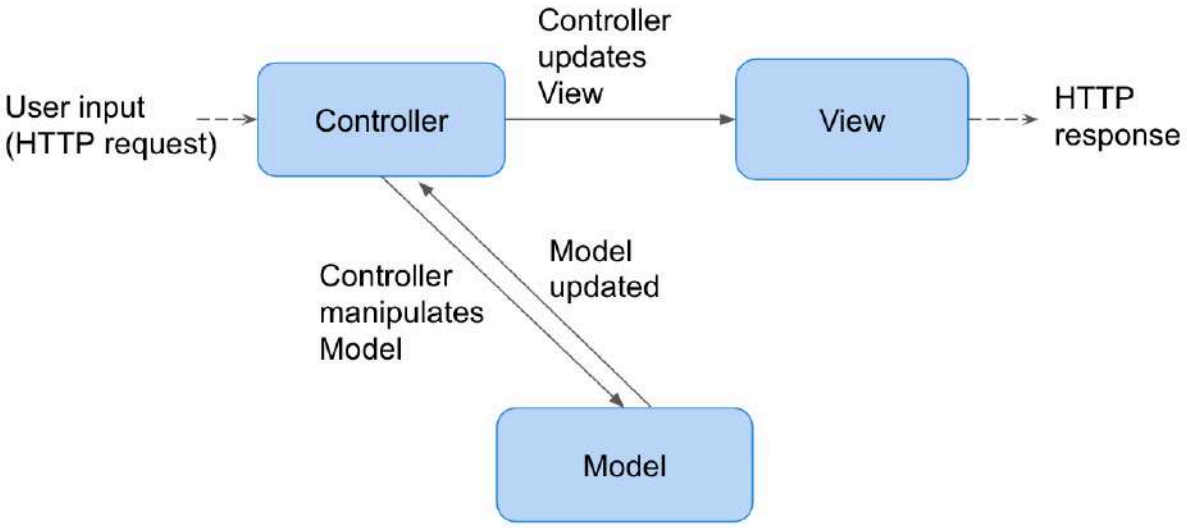
- **Model** is the data layer. This includes the data objects, database classes, and other business logic, concerned with storing the data, retrieving the data, updating the data, etc.
- **View** renders the data using the Model in a way suitable for user interface. It is what gets displayed to the user.
- **Controller** is the brains of the system. It encapsulates the logic of the system, and it controls both the Model and View. The user interacts with the system through this controller.

The general idea is that the **Model** should not be concerned with how it's ultimately displayed to the user. On the flip side, a **View** should not be concerned with the values of the actual data it is displaying, only that it needs to be displayed. The **Controller** then acts as the glue between these two components, orchestrating how the data should be displayed

Organizing a system this way allows for two advantages: **separation of concerns** and **unit testability**. Because components are isolated from each other, each focused on only one responsibility, the system is more flexible and modular. Each component is able to be unit tested on its own and could be swapped out for another version without affecting the other.

For example, the View could be changed out while the underlying data and logic remained the same.

When MVC was originally conceptualized, it was initially applied to desktop and web applications. A typical application of this pattern in the web realm would look like this:

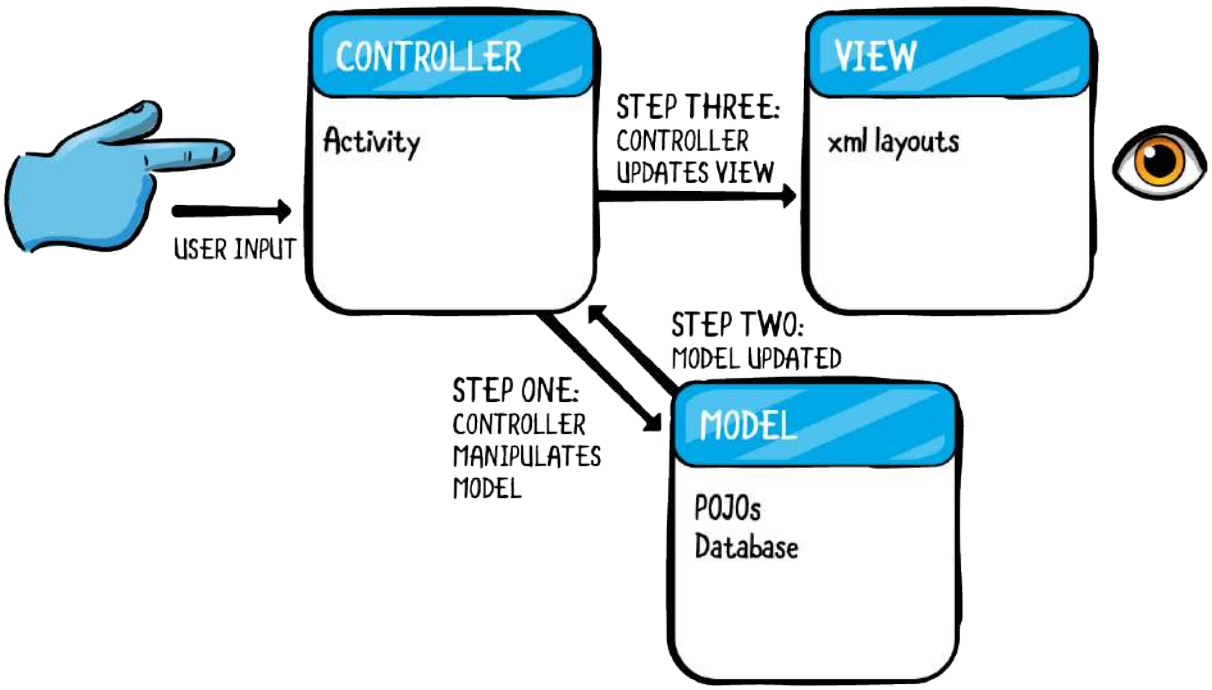


Typical MVC flow in a web application.

- The user inputs are HTTP requests.
- The Controller handles those requests by updating the data in the Model.
- The View returned to the user is a rendered HTML page, displaying the data.

Applying MVC to Android

Seeing that this pattern worked well in organizing other software systems, developers naturally tried to apply the MVC pattern to the development of Android apps as well when Android was introduced. Here is one way in which the MVC pattern was adapted for Android:



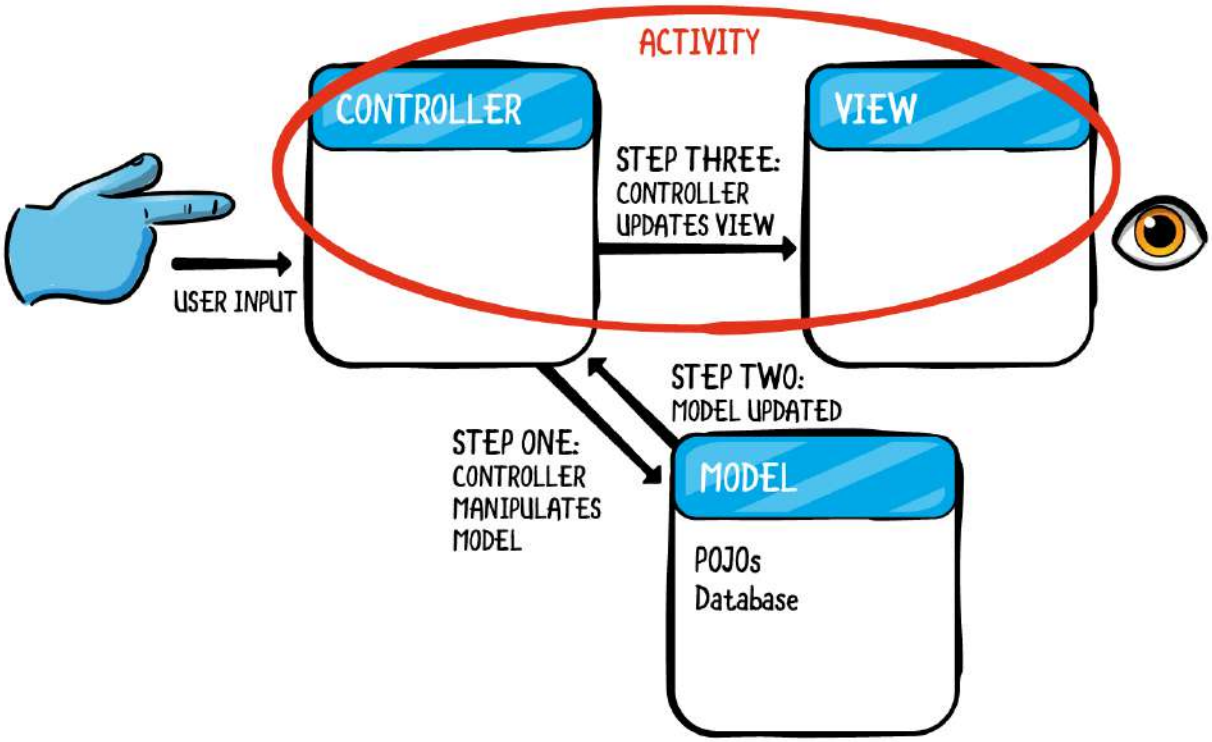
Attempt to apply MVC to Android.

In MVC, the main entry point to the app is through the Controller, so it makes sense to give the role of the Controller to the Android **Activity** where it can take in user inputs, such as a button click, and respond accordingly. The Model consists of the data objects, which, in Android, are just regular Kotlin data classes, as well as the classes to handle data locally and remotely. The **View** consists of the layout files in an Android app.

However, there is a problem with this oversimplified adaptation of MVC to Android.

When the Controller updates the View, the View cannot update if it is merely a static .xml layout. Instead, the Activity must almost always contain some **view logic**, such as showing or hiding views, displaying a progress bar or updating the text on screen, in response to user input. Furthermore, not all layouts are inflated through .xml; what if your Activity dynamically loaded your layouts?

If the Activity must hold references to views and logic for changing them *as well as* all the logic for its Controller responsibilities, then the Activity effectively serves as both the Controller *and* the View in this pattern.



More realistic application of MVC to Android.

Having the Activity act as both the Controller and the View is problematic for two reasons. First, it defeats the goal of MVC, which was to separate responsibilities among three distinct components of a software system. Second, having the Activity as the Controller in MVC poses a problem for unit testing, which you will read more about in the next chapter.

As you will discover, the sample app in this book starts out with precisely those problems described above.

WeWatch MVC code

Like many Android apps out there, WeWatch follows the standard Android architecture, which attempts to follow MVC architecture but falls short due to the nature of the Android framework and the role of the Android Activity. There ends up being a mix of both Controller logic and View logic in the app's Activities, as you'll see in the code for

this app. It will, however, serve as the starting point for the refactoring you will perform in future chapters to address those issues and improve the app.

Model

The Model for WeWatch consists of the following classes:

- The Movie class itself which you'll find in the **Movie.kt** file,
- The classes concerned with the local database are found in **LocalDatabase.kt**, **LocalDataSource.kt**, and **MovieDao.kt**.
- The classes concerned with remote data can be found in **RemoteDataSource.kt** and **TmdbResponse.kt**.

Open these files in the project and familiarize yourself with the classes and their workings.

Because the app gets its search results from *The Movie Database*, the Movie class must contain properties matching those of the movies found in the JSON object returned from the API. If you open **Movie.kt**, you'll see that this class contains far more properties than you'll ever need for this app, such as `originalLanguage` and `genreIds`; these are present in this class to be compatible with the movie JSON object returned from the API.

The actual object returned from The Movie Database is represented by **TmdbResponse.kt**, which includes properties such as number of results, and most importantly, the results themselves, which is a `List<Movie>`.

Next, the app needs a local database to persist the user's list of movies. The app uses the Room Persistence Library, one of the Android Architecture Components. The singleton for the local database is created in **LocalDatabase.kt**. Because Room does not know how to store a `List<Int>`, a `IntegerListTypeConverter` is needed to perform the conversion from `List<Int>` to `String` or `String` to `List<Int>` and is listed as a `TypeConverter` at the top of **LocalDatabase.kt**. The data access object containing the SQL statements for this database is **MovieDao.kt**. Finally, Activities will interact with this Model component through the `LocalDataSource`, which exposes the `insert`, `delete` and `update` functions to the rest of the app.

Networking

The search screen in this app is powered by a call to The Movie Database (www.themoviedb.org) API's search endpoint. `RetrofitClient` and `RetrofitInterface` contain all the functions needed to make this network call succeed.

Main Screen

Open **MainActivity.kt** and **MainAdapter.kt** and familiarize yourself with the code for displaying the user's list of movies to watch. **MainActivity.kt** sets up the views and gets an instance of the `LocalDataSource`. Then it gets the list of movies from the `LocalDataSource` through an `RxJava Observable` and passes the movies to the `MainAdapter`, which displays the movies in the RecyclerView. There is also logic to handle button clicks for adding and deleting a movie.

Add movie screen

Open **AddMovieActivity.kt**. This class sets up the views and gets an instance of the `LocalDataSource`. There are click listeners for the Search button and the add movie button. When the search button is clicked, the `SearchActivity` is started. Upon returning from the `SearchActivity`, the views get updated, including loading an image of the movie poster. When the add movie button is clicked, the movie is inserted into the local database.

Search movie screen

Open **SearchActivity.kt** and **SearchAdapter.kt**. Much of the logic in these classes is similar to those on the main screen. The **SearchActivity.kt** sets up the views and gets an instance of the `RemoteDataSource`. Then it gets the list of search results from the `RemoteDataSource` through an `RxJava Observable` and passes the movies to the `MainAdapter`, which displays the movies in the RecyclerView. There is also logic for clicking on a movie and returning to the previous screen.

Key points

In this chapter you gained an overview of the **Model View Controller** pattern as a way of separating components by responsibility in systems with user interfaces. Unfortunately, it's not easy to apply MVC to Android due to the nature of the Activity class. As you saw in the sample app, when applying MVC to Android, you end up with huge Activity classes, containing both Controller logic and View logic. As you will learn in the next chapter, one of the main drawbacks of this design is lack of unit testability of your codebase.

More specifically:

- **Model View Controller** is a pattern originally used in desktop and web-based applications.

- MVC divides an app into three components with their own responsibilities.
- The **Model** consists of the data and business logic.
- The **View** is responsible for rendering the data in a way that is human readable on a screen.
- The **Controller** is the brains behind the app and communicates with both the Model and the View. The user interacts with the app via the Controller.
- When applying MVC to Android, the Android **Activity** ends up serving as **both the View and Controller**, which is problematic for **separation of concerns** and **unit testing**.

Chapter 3: Testing MVC

By Yun Cheng

Testing software is important to ensure your software's behavior and to catch your bugs before your user catches them. In this chapter, you will learn about testing on Android, understand why this book will focus on unit testing and examine why it is difficult to write unit tests for an MVC app.

Android Testing

Testing in software development is often underestimated and pushed to the later stages of a project, if even at all. Unfortunately, there are some developers, clients and managers who don't believe that serious testing is important. The truth is, mobile apps are becoming larger and more complex. They're used to chat with our friends, play video games, socialize, take pictures, schedule appointments and much more. As our apps and games grow in complexity, so does the code base behind them, making testing even more important.

Android Testing is typically divided into three different types: Small, medium and large. In this section, you'll get an overview of each.

Small tests

Small tests are unit tests that run independently of an emulator or physical device. Small tests generally focus on a single component as all of its dependencies are tested beforehand and are mocked or stubbed with the desired behavior.

When compared against the other two test types, small tests are the fastest because they don't require an emulator or physical device to run. That said, they're also low-fidelity, which makes it difficult to have confidence that your app will function properly in the real world.

On Android, the most commonly used tools for these tests are JUnit and Mockito.

Medium tests

Medium tests are integration tests that help you check how your code interacts with other parts of the Android Framework. Medium tests are typically run after you complete unit tests on your components. That's when you need to verify that things will behave properly together.

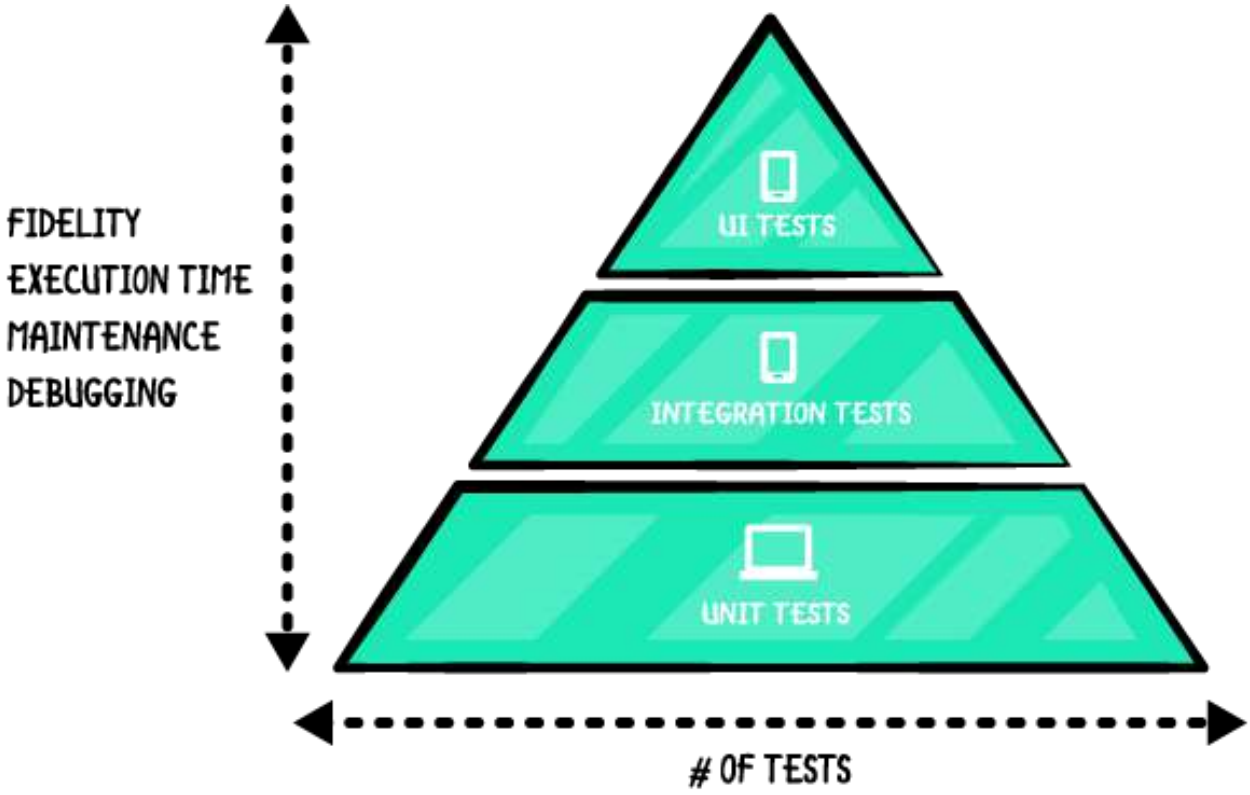
On Android, one of the most common tools to perform integration tests is Roboelectric. Unlike traditional emulator-based Android tests, Robolectric tests run inside a sandbox and don't need a device or emulator. However, it's best to test your app on an emulated device, a real device or using a service like Firebase Test Lab. Device farm services are great because you'll be able to test against different combinations of screen sizes and hardware configurations. This can help you catch bugs specific to some device categories.

Large tests

Large tests are integration and UI tests that emulate user behavior and assert UI results. These tests are the slowest and most expensive because they require the emulator or a physical device.

On Android, the most commonly used tools for UI testing are Espresso and UI Automator.

Overall, Google recommends that you create tests of each category based on your app's use cases according to the following rule: **70 percent small, 20 percent medium, and 10 percent large**. The Testing Pyramid (https://developer.android.com/training/testing/fundamentals) illustrates this.



Focusing on unit tests

Although all three types of testing play a role in ensuring a quality app, this book concerns itself primarily with **unit tests**, for the following reasons:

1. It is much **faster to run** unit tests than it is to run UI or integration tests. UI and integration tests typically run on an emulator, and starting up your app on an emulator and simulating user input takes up significantly more time than just running unit tests on the local JVM.

2. Unit testing **does not require any Android testing libraries**. The idea, here, is that you are testing regular Java/Kotlin code, not code that is Android framework-specific. As a result, using **JUnit**, the testing framework for Java, is all that is needed to write unit tests. If you wish to test code that uses Android-specific classes, you'll need a testing library specifically for Android. Libraries like Robolectric (robolectric.org) and Espresso (https://developer.android.com/training/testing/espresso/) are popular options.

3. Unit tests **make up the foundation** of an app's test suite. You can imagine that individual units of code work behind the scenes to display what the user ultimately sees in the app. The idea behind unit tests is that, if you can test that all the

individual units of code work as expected, you can have high confidence that, added up together, the app as a whole also works as expected.

Unit testing the Movie class

First, you will write your first unit tests for the WeWatch app. In the starter project, open **Movie.kt** and take a look at the `getReleaseYearFromDate()` method:

```
fun getReleaseYearFromDate(): String? {
  return releaseDate?.split("-")?.get(0)
}
```

This method exists because the format of dates returned by The Movie Database API is `YYYY-MM-DD`, but the app is only interested in displaying the year. It's easy to make errors when parsing dates, so it is worthwhile to test that this method returns expected values.

In the starter project, open the file **MovieTests.kt** and add the following test code:

```
class MovieTest {

  @Test
  fun testGetReleaseYearFromStringFormmattedDate() {
    //1
    val movie = Movie(title = "Finding Nemo", releaseDate = "2003-05-30")
    assertEquals("2003", movie.getReleaseYearFromDate())
  }

  @Test
  fun testGetReleaseYearFromYear() {
    //2
    val movie = Movie(title = "FindingNemo", releaseDate = "2003")
    assertEquals("2003", movie.getReleaseYearFromDate())
  }

  @Test
  fun testGetReleaseYearFromDateEdgeCaseEmpty() {
    //3
    val movie = Movie(title = "FindingNemo", releaseDate = "")
    assertEquals("", movie.getReleaseYearFromDate())
  }

  @Test
  fun testGetReleaseYearFromDateEdgeCaseNull() {
    //4
    val movie = Movie(title = "FindingNemo")
    assertEquals("", movie.getReleaseYearFromDate())
  }
}
```

Here are the four test cases tested in this suite:

1. Pass in a `releaseDate` of `2003-05-30`. The expected output would be the year extracted from the date: `2003`.
2. Pass in a releaseDate of `2003`. Even though the input is not in the format `YYYY-MM-DD`, the expected output would still be `2003`.
3. Edge case: pass in an empty `releaseDate` of `""`. In this case, the most appropriate output for this input would be just an empty string.
4. Edge case: pass in a `null releaseDate`. In this case, the best output for this would also just be an empty string.

Run the tests. You should see that only three out of four tests pass! Test case #4, which tests a `null` input, fails. This is because the code for `getReleaseYearFromDate()` does not properly handle a `null` input. Instead of returning an optional `String?`, it is better to return a `String`, even if the `String` returned is just an empty one. Make the following fix in the **Movie.kt** file:

```
fun getReleaseYearFromDate(): String {
  return releaseDate?.split("-")?.get(0) ?: ""
}
```

The addition of the Elvis operator ensures that if the `releaseDate` is `null`, an empty string is returned. Rerun the test with these changes, and you should see that it now passes.

Unit testing an Android Activity

As you just saw, unit testing the logic in data classes — like **Movie.kt** — that make up the Model is pretty straightforward using JUnit. You simply instantiate the object using the class' constructor, like with `val movie = Movie(title = "Finding Nemo", releaseDate = "2003-05-30")`. Then, you call a method on the object, like `movie.getReleaseYearFromDate()`, and assert that the result is what you expect.

But there is an entire rest of the app beyond data models still left to test! What about unit testing all the Android Activities in app?

Unfortunately, testing an Activity is tricky for two reasons:

1. **Android lifecycle**: An Activity is an Android framework-specific class that is bound by the Android **lifecycle**, which is controlled by the operating system. As such, you cannot instantiate an Activity: It has no constructor. Thus, you cannot just create an

object out of that Activity and call methods on it. For example, if you wanted to test the behavior of `displayMovies(movieList: List<Movie>?)` in **MainActivity.kt** given different inputs, you cannot just create an instance of that Activity like `val mainActivity = MainActivity()`, nor can you call methods on that instance like `mainActivity.displayMovies(ArrayList<Movie>())`.

2. **Android dependencies**. A typical Android Activity will contain other Android framework-specific classes in it, such as Context, RecyclerView, Toast, etc. As you will learn in Chapter 5: "Dependency Injection", these dependencies will need to be mocked if you want to test code in isolation. It is easy enough to mock your own classes, but not so easy to mock classes that belong to the Android framework.

Again, there are libraries like Robolectric that can address the above two challenges, but these will run much slower than regular JUnit tests.

Why MVC makes unit testing hard

Recall from the previous chapter that, while the Model View Controller pattern proved useful for other software systems, it didn't quite work for Android apps. Yes, you managed to separate the **Movie** model in a way that allowed it to be tested, however, the Android Activity ended up serving as both the **Controller** and the **View**. The downside with that, as you just learned, is that Activities are not easily tested with unit tests. The result is that all the Controller logic is trapped inside the Activity, and it becomes untestable.

Clearly, the MVC pattern in Android is inadequate for not just separation of concerns but also for unit testing.

Key points

- There are three categories of testing in Android: unit tests, integration tests and UI tests.
- Unit tests are fast to run and don't require external Android test frameworks.
- If you can achieve high unit test coverage of your app, you can be confident that the app as a whole works as expected, because ultimately the app is made up of all those small units all working correctly.
- Unit testing a regular data class using JUnit is easy: Simply create an instance of that class, call a method on it and assert that the output is expected.

- Unit testing an Android Activity is difficult due to the Android lifecycle and the many Android dependencies in an Activity.
- There are external libraries like Robolectric and Espresso that can test classes containing Android framework-specific components, but these are significantly slower to run than unit tests.
- The MVC pattern is not conducive to unit testing because all the Controller logic is stuck inside an Android Activity unable to be tested. (The Model, however, *can* be tested.)

Where to go from here?

It's impossible to cover all of the aspects of Android Testing in a single chapter. If you want to learn the ins and outs of Android Testing, check out the following links:

- Test Driven Development by Victoria Gonda: www.raywenderlich.com/7109-test-driven-development-tutorial-for-android-getting-started
- Android Unit Testing with Mockito by Fernando Sproviero: www.raywenderlich.com/195-android-unit-testing-with-mockito
- UI Testing codelab by the Google developers team: codelabs.developers.google.com/codelabs/android-testing/index.html?index=..%2F..%2Findex#6

In this chapter, you gained an overview of unit testing and saw how a major drawback of the MVC pattern is the lack of unit testability due to the nature of the Android Activity. In future chapters, you will learn about other patterns that address the problems of unit testing logic in Activities — by moving that logic *out* of the Activities.

Chapter 4: Android Architecture Components

By Yun Cheng

For many years, Android developers have had to handle database mappings, lifecycle management issues, asynchronous operations and other tricky challenges that were prone to errors and resulted in boilerplate code. In November 2017, Google introduced the **Android Architecture Components**, a collection of libraries to help with some of these architecture-related pain points. This book uses several of them in the sample project at various points: **Room**, **LiveData**, **ViewModel** and **Data binding**.

This chapter provides an overview of these libraries and goes into detail about using a subset of these in the sample project. While there's no sample project for *this* chapter, you'll view the sample projects from other chapters to see these libraries in action. For the Android Architecture Components that aren't covered in the sample project, links to free tutorials are provided where you can learn more.

Room

The **Room** library handles persistence for your app. Room provides an abstraction layer over SQLite and makes it easy to directly map objects to raw database content, as well as easily define type-safe queries for interacting with data. This is achieved through the use of annotations that generate boilerplate code behind the scenes for you.

Lifecycle

The set of classes in this library offer **lifecycle management**, allowing you to create components that can automatically adjust their behavior based on the current Android lifecycle state.

To see an example using this library, you can read this tutorial: www.raywenderlich.com/164-android-architecture-components-getting-started.

ViewModel

A **ViewModel** provides data between a repository and your View. Unlike Activities, which get destroyed on configuration changes, ViewModels survive configuration changes, keeping your data safe and preserving state.

LiveData

If you've used RxJava before, **LiveData** is a similar library (with scaled down features) that allows Android Views to observe model changes and respond accordingly. LiveData is also lifecycle-aware, so it only tells the UI to update things if it's in the correct lifecycle state. It also gets cleaned up for you, unlike RxJava subscriptions, which you must dispose of yourself.

Data binding

The **data binding** library allows you to observe changes in the ViewModel directly within the View XML. This save you from having to write the boilerplate code that's normally required to bind data to Views in code.

Paging

The **Paging** library helps you to display lists (especially infinite ones) by loading data in small chunks and updating the UI as more data is loaded.

To learn more about this topic, read this tutorial: www.raywenderlich.com/6948-paging-library-for-android-with-kotlin-creating-infinite-lists.

Navigation

The **Navigation** architecture component allows you to specify navigation throughout your app using an XML graph or via a graph editor that's built into the latest version of Android Studio. It reduces the hassle of handling it all yourself through boilerplate code.

This tutorial will help get you started using this library: www.raywenderlich.com/6014-the-navigation-architecture-component-tutorial-getting-started.

WorkManager

The **WorkManager** library manages deferrable background tasks that require guaranteed execution.

Read this tutorial for more information: www.raywenderlich.com/6040-workmanager-tutorial-for-android-getting-started.

Using the Android Architecture Components

Phew! That was a long list of components. Tying them together can be a daunting task, but each one has a very important role to play when talking architecture. In this section, you'll go over how the WeWatch app makes use of some of these components.

Room

Throughout this book, the sample project uses the **Room** library for its database operations.

Open the starter project from the **MVC** chapter 2 and review the code to see how it's implemented. Specifically, open and review the moving entity in **Movie.kt**, the data access object in **MovieDao.kt**, the database class found in **LocalDatabase.kt** and the data source class found in **LocalDataSource.kt**.

In WeWatch, the main data object of interest is `Movie`, and the app persists a user's list of movies by storing each movie as a row in a database table. With Room, it's easy to create a table named `movie_table` representing the `Movie` class in the database.

For any class whose fields you wish to directly map in a database table, use the `@Entity` annotation above the declaration for the class. By default, this creates a table with the same name as the class. You can also give the table a different name by explicitly setting the `tableName`, as is done in **Movie.kt**:

```
@Entity(tableName = "movie_table")
data class Movie(...) {...}
```

By default, the columns for this table have the same names as the fields in the constructor for this class, such as `id`, `title` and `releaseDate`. If you want a column to have a different name from the field name in the class, you can use the `@ColumnInfo` annotation before a field.

To set the primary key for a table, use the `@PrimaryKey` annotation. In the case of the movie table, the primary key is `id`:

```
@PrimaryKey
@SerializedName("id")
@Expose
var id: Int? = null
```

> **Note**: The `@SerializedName` and `@Expose` annotations are from the **Gson** library, not from Room. The app uses Gson to convert JSON data from API web calls to Kotlin data objects. Thus, this single `Movie` class can fluidly convert between JSON to Kotlin object to Room database entity all through the use of annotations.

Next are the operations on this table, such as retrieval, insertion and deletion. The app accesses these operations through **data access objects** (DAO). Generally, each table has its own DAO interface, specified using the `@Dao` annotation.

Open **MovieDao.kt**, the DAO for the movie table, and look at the following CRUD (Create, Read, Update, Delete) operations it has:

```
@Dao
interface MovieDao {
  @get:Query("SELECT * FROM movie_table")
  val all: Observable<List<Movie>>

  @Insert(onConflict = REPLACE)
  fun insert(movie: Movie)

  @Query("DELETE FROM movie_table WHERE id = :id")
  fun delete(id: Int?)

  @Query("DELETE FROM movie_table")
  fun deleteAll()

  @Update
  fun update(movie: Movie)
}
```

Each type of operation for a function is labeled with its own annotation: `@Insert`, `@Query` and `@Update`, with the corresponding SQLite query included to modify the data in the database.

Next, you need to declare the Room database object itself. Open **LocalDatabase.kt** and examine how this is done:

```
@Database(entities = [Movie::class], version = 1)
@TypeConverters(IntegerListTypeConverter::class)
abstract class LocalDatabase : RoomDatabase() {...}
```

`LocalDatabase` is abstract and inherits from `RoomDatabase`. It's marked with a Room `@Database` annotation that lists the Entities that are in this database (corresponding to the *tables* in this database) and the version number.

The only entity in the WeWatch database is the Movie entity, so that's the one that's listed. The version number for the database is a number that you need to increment for database migrations if the schema of the database changes — but don't worry about that right now.

The **LocalDatabase.kt** class then instantiates the Room database, like so:

```
LocalDatabase.INSTANCE =
  Room.databaseBuilder(application, LocalDatabase::class.java,
LocalDatabase.DB_NAME)
  .allowMainThreadQueries()
  .build()
```

A singleton pattern is used to create the Room database only once in the app.

The instantiation of the database occurs in **LocalDataSource.kt**. Open this file, and you'll see that this is where the single instance of the database resides:

```
val db = LocalDatabase.getInstance(application)
```

`LocalDataSource` is the source of data for the app, providing the list of movies from the database to the rest of the app. `LocalDataSource` also acts as the point of contact for the rest of the app to expose the DAO's other database calls. Any time a database call from the DAO is needed somewhere in the app, the calling Activity obtains an instance of `LocalDataSource` and calls `insert()`, `delete()` and `update()`.

LiveData

In some chapters of this book, you'll use the RxJava library to perform asynchronous web calls and database operations on the `Schedulers.io()` thread and to receive responses from them on the `MainThread`. In the **MVVM** chapters, an alternative to RxJava is presented: **LiveData**.

LiveData is a lifecycle-aware component that wraps around objects you wish to emit in a reactive way, much the same way the `Observable` object does in RxJava. LiveData is "live" in the sense that when the underlying data updates, anything observing that data will also receive the updates. For example, when paired with Room, LiveData retrieved from the Room database automatically updates when the data in the database changes.

In the MVVM with Android Architecture Components chapter project, the **MovieDao.kt** class returns a list of movies from the database as LiveData, like so:

```
@Query("select * from movie")
fun getAll(): LiveData<List<Movie>>
```

To learn more about LiveData, read chapter 11: *MVVM with Android Architecture Components*.

ViewModel

One problem with Activities and the Android lifecycle is that if the Activity gets destroyed, data held by that Activity also gets destroyed. So, how can you return the Activity to the state it was in once it is rebuilt? It's possible to save some state through the Activity's `onSaveInstanceState(outState: Bundle?)`, but this solution is cumbersome and doesn't work for more complicated data.

Android Architecture Components now offer a more robust way for data to survive configuration changes via the **ViewModel**. A ViewModel is a lifecycle-aware class used to hold onto the LiveData so that it doesn't get destroyed on configuration changes like screen rotation. The ViewModel keeps its state throughout the Activity's lifecycle, but it's essential to avoid any references to Views or Activities *within* a ViewModel because these throw a `nullPoinerException` when destroyed.

To see an example of how a ViewModel works, open **MainViewModel.kt** from the project for the MVVM with Android Architecture Components chapter.

```
class MainViewModel(private val repository: MovieRepository =
MovieRepositoryImpl()): ViewModel() {}
```

Here, `MainViewModel` extends `ViewModel` and takes in a `MovieRepository` in its constructor.

Next, check out how the ViewModel holds onto a list of movies as LiveData:

```
private val allMovies = MediatorLiveData<List<Movie>>()

init {
  getAllMovies()
}

fun getSavedMovies() = allMovies

fun getAllMovies() {
  allMovies.addSource(repository.getSavedMovies()) { movies ->
    allMovies.postValue(movies)
  }
}
```

The `allMovies` list, in this case, is of type `MediatorLiveData`, which is a subclass of regular `LiveData` that allows you to mutate the LiveData and react to updates on the LiveData's value via an `onChanged()` event.

The app can then access `allMovies` in any Activity, such as in **MainActivity.kt**, like so:

```
override fun onCreate(savedInstanceState: Bundle?) {
  ...
  viewModel = ViewModelProviders.of(this).get(MainViewModel::class.java)

  viewModel.getSavedMovies().observe(this, Observer { movies ->
    movies?.let {
      adapter.setMovies(movies)
    }
  })
}
```

Here, the ViewModel is instantiated by using a `ViewModelProvider` to provide the ViewModel. The Activity can then access the list of movies by calling `viewModel.getSavedMovies()` and observing the LiveData returned from that call. In this example, the list of movies is set on the Adapter when observed.

Be sure to read Chapter 11: *MVVM with Android Architecture Components* to see the code in full.

Data binding

The Data Binding library allows you to bind UI components in your Layouts to data sources. Normally, this is done in code — such as setting the text of a `TextView` using `movieTitleTextView.setText = movie.title` — but with the Data Binding library, you'll be able to do it directly in the XML Layout files.

For example, open the project from the chapter MVVM with Data Binding and look at this snippet of **item_movie_main.xml**:

```
<TextView
  android:id="@+id/movieTitleTextView"
  android:text="@={movie.title}"
  ... />
```

You can read more about data binding in chapter 10: *MVVM with Data Binding chapter*.

Key points

- Android Architecture Components are a set of libraries to help with various challenges in dealing with Android architecture.
- Room handles database persistence.
- Lifecycle helps you create components that are aware of the current Android lifecycle state.
- ViewModel holds data and survives configuration changes.
- LiveData provides observable data to views.
- Data binding allows Android Views to observe changes to data in XML.
- Paging handles displaying infinite lists.
- Navigation handles complex navigation.
- WorkManager manages background tasks.

Where to go from here?

Having a good understanding of the Android Architecture Components will be useful as you navigate the various chapters that use them. Next up, you'll learn about another useful concept that's present throughout the book: dependency injection.

Chapter 5: Dependency Injection

By Yun Cheng

In this chapter, you'll learn about dependencies, why they can be problematic and how you can remove them from your code using dependency injection, either manually or by using a dependency injection framework.

The truth is, you may already be using dependency injection, but you don't recognize it as such because you're unfamiliar with the terminology. By gaining a deeper understanding of dependency injection, you'll better deco`uple object dependencies, and as a whole, become a better programmer! Specifically in this book, you'll see the authors make an effort to use dependency injection in MVP, MVVM and MVI architecture.

What is a dependency?

A **dependency** occurs when an object from one class requires an object from another class in order to function properly. These dependencies are often member variables of the class. To give you a better sense of how a dependency might work, consider the following example: A movie theatre requires at least a screen, a projector and a movie, otherwise it won't function.

In this chapter, you'll create reusable, scalable code to support this type of dependency scenario.

Open the **MovieTheatreExample** starter project from the book resources in Android Studio and review **MovieTheatre.kt**:

```
class MovieTheatre {

  var screen: Screen
  var projector: Projector
```

```
    var movie: Movie

    init {
      screen = Screen()
      projector = Projector()
      movie = Movie()
    }

    fun playMovie() {
      System.out.println("Playing movie now")
    }
  }
```

In this initializer block, `MovieTheatre` instantiates its dependencies, which includes a `Screen`, a `Projector` and a `Movie`. In other words, `MovieTheatre` *depends* on these objects to function.

While the instantiation of dependencies in the `init` may seem fine at first glance, there are some problems with this approach.

Why dependencies can be problematic

When you instantiate a class's dependencies in its `init`, you create a situation where the class is tightly **coupled** with its dependencies. In this case, `MovieTheatre` is tightly coupled with its dependencies — `Screen`, `Projector` and `Movie` — so every time you create a new `MovieTheatre` instance, it needs to create a `Screen`, a `Projector` and a `Movie`.

The trouble is, you've given the responsibility of both *creating* and *using* these objects. As a general rule, when creating a class you should separate the creation of an object from the usage of an object.

Consider the following scenario: You have two movie theaters, one with a projector that uses 8mm film, and one with a projector that uses 16mm film. If you were to create two instances of `MovieTheatre`, each with its own `Projector`, you'd have no way to specify the type of film because the current `init` doesn't allow for that kind of flexibility.

To ensure **reusability** of the `MovieTheatre` class, it's better to create two instances of `Projector` elsewhere and pass them into their respective `MovieTheatre` instance. In either case, `MovieTheatre` takes whatever `Projector` it's handed, and then does what it needs to do to show the movie, regardless of the type of film.

Another reason to avoid instantiating dependencies in an object's initializer block is **unit testing**. When unit testing `MovieTheatre`, you're only interested in testing the behavior of that class, not any of its dependencies. For example, a `Projector` may have

complicated inner workings involved in displaying a movie. Assuming you already performed unit testing on `Projector`, you don't need to test it as part of `MovieTheatre`'s unit tests. If, however, you instantiate `Projector` in `MovieTheatre`'s initializer block, you'll effectively be testing the dependency behavior along with the behavior of the object of interest.

To avoid testing dependencies in the `MovieTheatre` unit tests, you'll make **mock** objects out of your dependencies. When testing the `MovieTheatre` code, you pass in these mock objects from the outside, so when the tests execute, `MovieTheatre` will call the methods on the mock object rather than a real object.

Injecting dependencies

It's important to pass dependencies from an external source rather than creating them within a class; this process is known as **dependency injection**. The most common way to inject dependencies is through the constructor.

Rewrite the `MovieTheatre` class using constructor dependency injection:

```
class MovieTheatre(val screen: Screen, val projector: Projector, val
movie: Movie) {

  fun playMovie() {
    System.out.println("Playing movie now")
  }
}
```

Now, `MovieTheatre` is no longer responsible for creating its own dependencies; instead it receives its dependencies as parameters when it's constructed.

`MovieTheatre` is instantiated in `MainActivity`, so open **MainActivity.kt** and pass in the dependencies that `MovieTheatre` is expecting:

```
class MainActivity : AppCompatActivity() {

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    //1 Instantiate dependencies
    val screen = Screen()
    val projector = Projector()
    val movie = Movie()

    //2 Instantiate MovieTheatre, passing in dependencies
    val movieTheatre = MovieTheatre(screen, projector, movie)

    //3 Call methods on the MovieTheatre
```

```
        movieTheatre.playMovie()
    }
}
```

Run the project and confirm that logcat displays the "Playing movie now" message.

Injecting dependencies through the constructor is fine for simple apps like the WeWatch sample app. A more complicated app, however, may have a web of dependencies.

Suppose `Screen` has its own dependencies like a `Curtain` and `Backdrop`, while `Projector` has its own dependencies like a `Lens`, `PowerCord` and `Reel`. Each of these dependencies need to be instantiated and passed into the respective class (`Screen` or `Projector`) upon creation of that class, and then the `Screen`, `Projector` and `Movie` need to be passed in to create a `MovieTheatre` object. The result is a lot of boilerplate code.

If it takes too many lines of code before you can instantiate your main class, it's a sign that a dependency injection framework may help.

Dependency injection frameworks

Because the WeWatch example app is relatively simple, there's no need to use an external framework for dependency injection, which can sometimes make it difficult to understand what's happening. However, if an app has overly complicated dependencies, consider using a third party framework like Koin, Proton, Feather, Tiger, Lightsaber, Transfuse or Dagger 2.

In the following simplified example, you'll briefly explore Dagger 2 to generate the dependency injection boilerplate code for you through the use of **annotations**. The two main annotations you'll use are **@Inject** and **@Component**.

- **@Inject**: This annotation marks which dependencies to inject. You can use it on a constructor, field or method.
- **@Component**: This annotation is used on an interface class from which Dagger will then generate a new class that contain methods that return objects with their dependencies injected.

Open the app's **build.gradle** file and set up the project. Add this plugin to the top of the file:

```
apply plugin: 'kotlin-kapt'
```

This directive applies `kapt`, which is the Kotlin annotation processor. It allows you to use the Dagger annotations in your Kotlin classes.

Next, add these dependencies:

```
//Dagger dependencies
implementation 'com.google.dagger:dagger:2.15'
kapt 'com.google.dagger:dagger-compiler:2.15'
kapt 'com.google.dagger:dagger-android-processor:2.15'
```

Now, open **Projector.kt** and add the `@Inject` annotation to the constructor to expose this class as a dependency to Dagger:

```
class Projector @Inject constructor()
```

Then, open **Screen.kt** and make the same change:

```
class Screen @Inject constructor()
```

Next, open **Movie.kt** and make this change:

```
class Movie @Inject constructor()
```

Finally, open **MovieTheatre.kt** and add the `@Inject` annotation:

```
class MovieTheatre @Inject constructor(val screen: Screen, val projector:
Projector, val movie: Movie) {
  ...
}
```

Create a new file, **MovieTheatreComponent.kt**, and add the following to it:

```
@Component
interface MovieTheatreComponent {
  fun getMovieTheatre() : MovieTheatre
}
```

In this interface, there's a single method, `getMovieTheatre()`, which returns an instance of `MovieTheatre`. Note that the interface is also annotated with `@Component`.

Rebuild the project. Dagger will generate a class for you named `DaggerMovieTheatreComponent` with all of the dependency injection code added using the `@Inject` annotations as its guide.

When you call `DaggerMovieTheatreComponent.create().getMovieTheatre()` now, you'll get an instance of `MovieTheatre` with all of the work of injecting its dependencies done for you.

Open **MainActivity.kt** and modify the code in `onCreate`:

```
//Get the MovieTheatre from the DaggerMovieTheatreComponent
val movieTheatre = DaggerMovieTheatreComponent.create().getMovieTheatre()

//Call methods on the MovieTheatre
movieTheatre.playMovie()
```

The lines of code needed to instantiate `MovieTheatre` is now replaced with a call to get the instance from `DaggerMovieTheatreComponent` instead. Run the project and verify that logcat displays the message "Playing movie now".

Note that the simplified example above demonstrates only the basic features of Dagger 2 and leaves out many other annotations available in the framework.

If you'd like to learn more about Dagger 2, check out the official documentation google.github.io/dagger. You can also checkout a deep dive into dependency injection and Dagger 2 on the site www.raywenderlich.com/262-dependency-injection-in-android-with-dagger-2-and-kotlin. If you're interested in using the pure Kotin library, Koin, check out the this tutorial www.raywenderlich.com/9457-dependency-injection-with-koin on the site.

Key points

- A dependency of class A is any class B that is used by A.
- Generally, a class should not be responsible for both creating and using its dependencies.
- Rather than have a class create its own dependencies, you should create those dependencies outside and pass them into the class via its constructor.
- Dependency injection is a pattern where dependencies are passed into a class from an external entity.
- Injecting dependencies into a class allows for greater reusability of that class.
- Dependency injection is especially important for unit testing a class, as passing in dependencies through a class's constructor allows for mock objects to be passed into that class during unit tests.
- For more complex dependencies in a project, you can use an external dependency injection framework, such as Dagger 2, to generate boilerplate dependency injection code.

Where to go from here?

In this chapter, you learned the importance of using dependency injection to improve reusability and unit testing.

Although the movie theatre example used in this chapter was for demonstration purposes only, keep the concept of dependency injection in mind in the upcoming chapters. You'll make use of it in the WeWatch sample project when you apply various architectures to this project.

Chapter 6: RxJava

By Aldo Olivares Domínguez

In this chapter, you won't be writing any code. Instead, you'll learn the theory behind one of the most popular open source libraries for Java: **RxJava**.

You may be wondering: Why learn about RxJava if this is a book written in Kotlin?

Even though RxJava is a Java library, it'll work with your Kotlin code. This is because Kotlin was designed with Java interoperability built in.

You might also be wondering: I thought this was a book about architecture patterns!

True, this is a book about architecture patterns for Android, but RxJava makes use of important elements that are present amongst most of the architecture patterns you'll see in this book, such as Observables and Reactive Programming. Also, RxJava is popular, and a lot of source code that you'll encounter online use RxJava. So, it's important that you understand how this library works, at least at a high level.

What is the Observer pattern?

Before you dive into the RxJava world, it's critical that you understand the **Observer pattern**. When most Android developers hear the word pattern, they immediately think about **MVC**, **MVP** and **MVVM**. In reality, those are not technically patterns; they're **compound patterns**.

What is a compound pattern?

A compound pattern is a pattern made up of multiple patterns. In other words, a pattern of patterns.

MVC, for example, relies on the **Strategy pattern** to configure the interaction between the Views and the Controller. The controller provides a strategy for the View, and the View is only concerned about displaying the UI. The View also relies on the **Composite pattern** under the hood to manage all the UI widgets that you see on the screen, such as buttons or lists. The models rely on the Observer pattern to make sure the controllers and Views have the latest data updates.

How does the Observer pattern work?

The Observer pattern helps your objects know when something happens — such as a database update or a network response — so they can react accordingly.

To understand the Observer pattern, imagine how a subscription to your favorite YouTube channel works:

- A content creator publishes a new YouTube video every other day.
- You like the channel's content, so you subscribe to the channel for new video notifications.
- As soon as a new video is published, you get a notification. You'll continue to receive notifications as long as you're a subscriber. But that doesn't mean you have to watch the video.
- If you unsubscribe, you no longer receive notifications from that channel.
- As long as the user keeps posting videos, all subscribers to the channel will get a notification when a new video is posted.

The Observer pattern works similarly to the YouTube channel subscription described above. The analogy is that the YouTube channel is the **Observable** and the subscribers as the **Observers**. When the state of the Observable changes, all of the Observers who subscribed to it get a notification about the update, creating a one-to-many relationship between them.

Now, suppose you want to display a list of items as soon as the user presses a button in your app. You can define your button as the Observable and your list as the Observer. As soon as the button changes its state, it emits an event to the Observers. In this case, the list is automatically updated to display a list of items.

In Android, there are many ways to implement the Observer pattern in your apps, one of which is RxJava.

Getting to know RxJava

RxJava is an open-source library for Java and Android that helps you create reactive code. It offers the possibility to implement the Observer pattern for Observable/Observer callbacks and gives you a range of operators that allow you to handle asynchronous and event-based programs.

Programming reactively

RxJava helps you create reactive code. In imperative programming, you often evaluate an expression **linearly** or line-by-line. If you want to calculate the area of a rectangle, your code would look like this:

```
var width = 2
var height = 3
var area = width * height // z is 6
```

In this code example, the area is calculated once at runtime and never changes.

Reactive programming is a different paradigm in which your code dynamically reacts to changes. The changes can be almost anything you want, such as value changes or state changes.

The truth is, you might have done a bit of reactive programming in the past without even noticing it. Every time you add an `onClickListener` to one of your buttons, for example, you did something like this:

```
button.setOnClickListener {
  //Some code
}
```

In this case, you're reacting to a change in the button state. Once the state changes, you can react to it by updating a list, displaying a notification or adding any action to it.

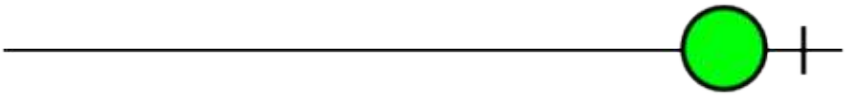
Observing events

RxJava implements the Observer pattern via two main interfaces: `Observable` and `Observer`. One of the most important methods in the `Observable` interface is `subscribe()`.

The `Observer` interface, on the other hand, has three methods that the `Observable` calls when it changes the appropriate state:

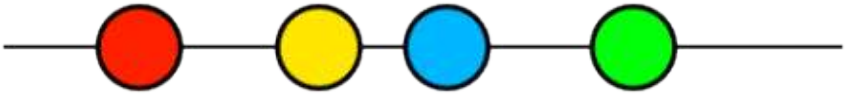
- **`onNext(T value)`**: This gets called by the `Observable` when it emits a new item of type T to the `Observer`.
- **`onComplete()`**: This notifies all of the `Observers` that the `Observable` is done with its task.
- **`onError(Throwable e)`**: This notifies the `Observer` that the `Observable` has experienced an error.

As a rule of thumb, every `Observable` may emit one or more items that can be followed by a completion or an error.



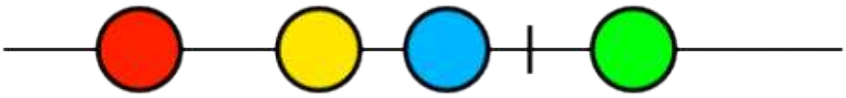
The diagram above represents an event emitted by an `Observable`. The green dot represents an item that was emitted by the `Observable` (such as a new video notification), and the vertical black line represents a completion or an error.

However, there can be continuous events with no completion or errors. Here's how an event caused by a mouse `Observable` looks:



As you can see, there are many events on this diagram that can represent mouse movements: right-clicks, left-clicks, middle-button clicks and much more.

But keep in mind: Events can't be emitted after an `Observable` completes its tasks. Here's an example of an `Observable` that's not following the appropriate flow:



This flow violates the `Observable` contract by emitting an event after signaling completion.

Creating an Observable

There are many libraries in Java that help you create an `Observable` from almost any type of event. But sometimes it's better to create your own. You can create an `Observable` using `Observable.create()`. Here's its signature:

```
Observable<T> create(ObservableOnSubscribe<T> source)
```

That's nice and concise, but what does it mean? What is the source? To understand that signature, you need to know what about `ObservableOnSubscribe`. `ObservableOnSubscribe` an interface, with this contract:

```
public interface ObservableOnSubscribe<T> {
  void subscribe(ObservableEmitter<T> e) throws Exception;
}
```

RxJava's `Emitter` interface is similar to the `Observer`:

```
public interface Emitter<T> {
  void onNext(T value);
  void onError(Throwable error);
  void onComplete();
}
```

An `ObservableEmitter` also provides a means to cancel the subscription.

The best way to understand Observables is with simple examples that illustrate the entire process. Once you understand the basics, the complicated stuff becomes a little easier. It's kind of like solving a jigsaw puzzle: Once you have the corners, everything else fits into place.

To create a simple method that returns an `Observable`, you can use this code:

```
//1
fun createYoutuber(): Observable<String>{
  //2
  return Observable.create{emitter ->
    //3
    emitter.onNext("How to breed llamas")
  }
}
```

The `Observable` emits a string (here, new video title) for its subscribers. Here's a closer look:

1. Declare a method that returns an `Observable` of type `String`. Since `Observables` are generic, you can create an `Observable` for almost anything you want such as **Strings**, **Doubles** or **Network Responses**.
2. Use `create()` to create a new `Observable`.
3. Make the Observable (The Youtuber) emit the new video title to its Observers (The subscribers).

The following example declares a method that creates a new `Observer` object:

```
//1
private fun createSubscriber(): Observer<String> {
  //2
  return object : Observer<String> {
    //3
    override fun onSubscribe(d: Disposable) {
      Log.d(TAG, " Im subscribed")
    }
     //4
    override fun onNext(value: String) {
      Log.d(TAG, "New Video : $value")
    }
    //5
    override fun onError(e: Throwable) {
      //Define onError() Action
    }
    //6
    override fun onComplete() {
      //Define onComplete() Action
    }
  }
}
```

Taking each commented section in turn

1. This method returns an `Observer` of type `String`. Just like the `Observables`, the `Observers` can observe any type of values.

2. Create a new `Observer` of type `String`.
3. `onSubscribe()` declares the action to take when your `Observer` is attached to the `Observable`.
4. `onNext()` declares the action to take when the emitter emits a new value. The parameter represents the value emitted by your `Observable`.
5. `onError()` declares the action to take when the `Observable` emits an error.
6. `onComplete()` declares the action to take when the `Observable` completes its tasks.

With these methods, you can easily create a new `Observable` and subscribe an `Observer`, like so:

```
createYoutuber().subscribe(createSubscriber())
```

When you execute that code, you'll get these logs in the console:

```
D/MainActivity: Im subscribed
D/MainActivity: New Video : How to breed llamas
```

If you want to emit some values to your `Observers`, there's an easier way to create your `Observables` using `just()`.

Here's how you can rewrite `createYoutuber()` to create an `Observable` that emits a single value to its observers:

```
fun createYoutuber(): Observable<String>{
  return Observable.just("How to breed Llamas")
}
```

The resulting log is the same:

```
D/MainActivity: Im subscribed
D/MainActivity: New Video : How to breed llamas
```

Asynchronous tasks

One common misconception about RxJava is that the tasks executed with this library are executed on a background thread. By default, RxJava does all of the tasks in the same thread from which it was called. For an Android app, this means that an `Observable` will usually emit all its data using the UI thread unless told otherwise.

You can, however, execute code on a background thread using `observeOn()` and `subscribeOn()`. `observeOn()` subscribes `Observers` to their `Observable` on the specified scheduler. `subscribeOn()` modifies the `Observable` to emit its events and notifications on the specified scheduler.

Suppose you want the `Observers` from previous examples to subscribe to their `Observable` on a background thread, but you still want to emit notifications on the main thread. This is how you can rewrite the **subscribe/observe** model to do it:

```
createYoutuber()
    // Subscribe on a background thread
    .subscribeOn(Schedulers.io())
    // Observe on the main thread
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(createSubscriber())
```

The resulting output:

```
D/MainActivity: Im subscribed
D/MainActivity: New Video : How to breed llamas
```

Although the result is the same for this example, having the ability to execute different tasks on different threads is useful — especially when you're dealing with background operations such as database updates and network calls.

Operators

YouTube is great, but there are other video streaming services available. Perhaps you'd like to subscribe to more than just a YouTube `Observable`, but apply the same logic to each.

You can create a method that returns a new Netflix `Observable` like this:

```
fun createNetflixChannel(): Observable<String> {
  return Observable.just("House of Cards")
}
```

Then, you can subscribe to both the Youtube `Observable` and the Netflix `Observable`, like so:

```
val subscriber = createSubscriber()
createYoutuber()
  .subscribeOn(Schedulers.io())
  .observeOn(AndroidSchedulers.mainThread())
  .subscribe(subscriber)

createNetflixChannel()
   .subscribeOn(Schedulers.io())
   .observeOn(AndroidSchedulers.mainThread())
```

```
    .subscribe(subscriber)
```

That's a lot of duplicated code! Fortunately, there's an easier way to do this without duplicating the code. You can merge your `Observables` into a single `Observable` using the **merge** operator.

Operators in RxJava are a way to modify your `Observables` and/or their data to facilitate your development workflow without the need to create your own methods. Most operators operate on an `Observable` and return an `Observable`, making it possible to chain them one after the other.

You've already learned about two operators: **SubscribeOn** and **ObserveOn**. Both of them are utility operators that operate on an `Observable` and return an `Observable` — that's why you were able to chain them in the previous examples.

On the other hand, the merge operator lets you merge two different `Observables` into one by using the `merge()` method from RxJava. This is especially useful if you want to apply the same operations to both of them.

You can rewrite the above code using the `merge()` method like this:

```
Observable.merge(createYoutuber(), createNetflixChannel())
  .subscribeOn(Schedulers.io())
  .observeOn(AndroidSchedulers.mainThread())
  .subscribe(createSubscriber())
```

This is much simpler and easier to read. Here, you're creating a Youtuber `Observable`, a Netflix `Observable` and merging them into a new `Observable` using `merge()`.

After that, you can apply the necessary operations to the new `Observable` just like you would to each of them separately.

The result is the following:

```
D/MainActivity:  Im subscribed
D/MainActivity:  Im subscribed
D/MainActivity: New Video : How to breed Llamas
D/MainActivity: New Video : House of Cards
```

Awesome! You just made your code easier to understand.

There are many other useful operators at your disposal in the RxJava library. They're not all covered in this chapter because there are too many. If you're interested in learnig more, check out the official documentation: http://reactivex.io/documentation/operators.html.

Frequently Not Asked RxJava Questions

Q. Can I use the usual listeners/callbacks instead of RxJava?

You certainly can! But using RxJava offers several benefits that can heavily pay off in the long run by reducing the amount of boilerplate code and making it more concise. You can even apply several operations to your Observables to reduce duplicate code.

Q. Are you sure that RxJava is 100% compatible with my Kotlin code?

Yes. No need to worry about compatibility issues, Kotlin was designed with Java interoperability in mind, and the RxJava library is no exception.

Q. I just read about RxKotlin, what is that?

RxKotlin is a lightweight library that takes advantage of Kotlin's extension functions to make RxJava code more Kotlin-y. Of course, you can use RxJava with Kotlin out-of-the-box, but this library makes things even easier.

Key points

- The Observer pattern helps your objects know when something interesting happens so they can react accordingly.
- The Observer pattern defines a one to many relationship between an Observable and the Observers.
- An `Observable` is an object that emits notifications about its state to one or more Observers.
- Reactive Programming is a declarative programming paradigm in which you dynamically react to changes.
- RxJava is an open source library for Java and Android that helps you create reactive code. It's heavily inspired by functional programming.
- RxJava implements the Observer pattern with two main interfaces: `Observable` and `Observer`.
- RxJava executes all tasks on the same thread from which it was called.

Where to go from here?

RxJava is a powerful and popular library that lets you write reactive code in your Android Projects. Even though it was written in Java, it's 100% interoperable with Kotlin. If you want to learn more about RxJava, check out Irina Galata's on the site **Reactive Programming with RxAndroid in Kotlin** raywenderlich.com/384-reactive-programming-with-rxandroid-in-kotlin-an-introduction

Section II: Fundamental UI Architectures

In this section, you'll rewrite WeWatch three different ways, first using MVP, then using MVVM with data binding, and finally using MVVM with Android Architecture Components. Though a user of the app won't be able to see any difference after each of these changes, you as a developer will see how they solve some of the problems of the original WeWatch app written in MVC.

- **Chapter 7, MVP Theory**: The first architecture pattern you'll learn is Model View Presenter. Here, you'll learn the theory behind pulling presentation logic out of the Android Activity and why it results in a significant improvement over the MVC pattern in terms of separation of concerns and testability of code.
- **Chapter 8, MVP Sample**: With theory in tow, you'll then refactor WeWatch to the MVP pattern step-by-step.
- **Chapter 9, Testing MVP**: Now that you've refactored the app to MVP, you can write unit tests on the logic that previously was untestable when it was in the MVC pattern.
- **Chapter 10, MVVM Theory**: The next architecture pattern you'll learn is Model View ViewModel, a pattern which centers on exposing a stream of events to Views. Architecting your app this way helps you write loosely coupled and easily testable code.
- **Chapter 11, MVVM Sample with Data Binding**: In this chapter, you'll refactor WeWatch to the MVVM pattern using the Data Binding library from Android Architecture Components.
- **Chapter 12, MVVM Sample with AAC**: In this chapter, you'll again refactor WeWatch to the MVVM pattern, this time using a combination of the ViewModel and LiveData libraries from Android Architecture Components.
- **Chapter 13, Testing MVVM**: Having completed the refactoring of the app to MVVM, you'll write unit tests on the ViewModel to ensure it behaves as expected.

Chapter 7: Model View Presenter Theory

By Yun Cheng

As you saw in the MVC chapter, although the MVC architecture pattern theoretically allows an app to achieve separation of concerns and unit testability, in practice, MVC didn't quite work on Android. The problem was that the Android Activity, unfortunately, served the role of both View and Controller. Ideally, you would want to somehow move the Controller out of the `Activity` into its own class so that the Controller can be unit testable.

One architecture pattern that achieves this goal is **MVP**, which stands for **Model View Presenter**, and is made up of the following parts:

- **Model** is the data layer, responsible for the business logic.
- **View** displays the UI and listens to user actions. This is typically an `Activity` (or `Fragment`).
- **Presenter** talks to both Model and View and handles presentation logic.

The Model-View-Presenter pattern

With MVP, the Model remains the same as with the MVC architecture. The Model is the data layer, and is responsible for the business logic: retrieving the data, storing the data and changing the data. In your sample project, your business logic revolves around movies, so the Model is composed of the `Movie` data object, the `LocalDataSource`, which interacts with the local database. It is also composed of the `RemoteDataSource`, which interacts with the Movie Database API over the network.

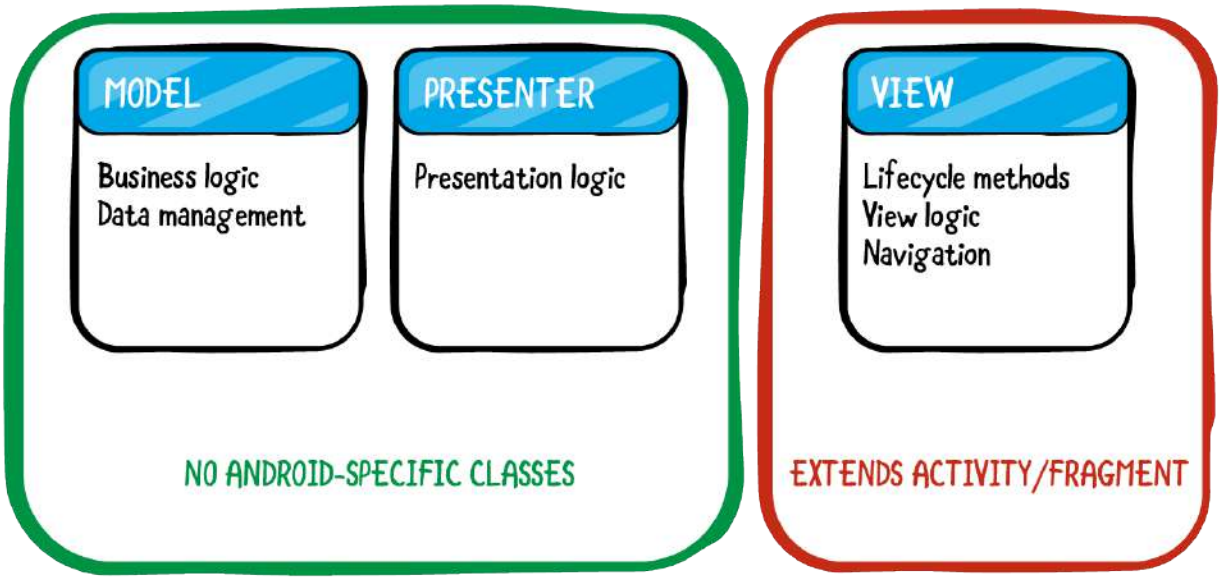
The View is also the same as with the MVC architecture, responsible for displaying the UI, except that, with MVP, this role is specifically designated to an `Activity` or

`Fragment`. The View will hide and display views, handle navigating to other Activities through intents, and listen for operating system interactions and user input.

The Presenter is the glue class that talks to both the Model and View. Any code that does not directly handle UI or other Android framework-specific logic should be moved out of the View and into the Presenter class. For example, while the View will listen for button clicks, the presentation logic of what happens after a user clicks a button should go into the Presenter class. Similarly, when the Model has updated, it is not the responsibility of the Model to know how the View ultimately wants to display the data.

It is the Presenter's job to do any additional mapping or formatting of the data before handing it to the View to display it. This kind of logic is known as presentation logic, and it is handled by the aptly named Presenter.

While the View extends from san `Activity` or `Fragment`, the Model and Presenter do not extend Android framework-specific classes, and, for the most part, they should not contain Android framework-specific classes. In other words, there should be no references to the **com.android.*** package in the Model or Presenter. As you will read later, this rule allows for both the Model and the Presenter to be unit tested with this pattern.



The Presenter should not contain Android framework-specific classes

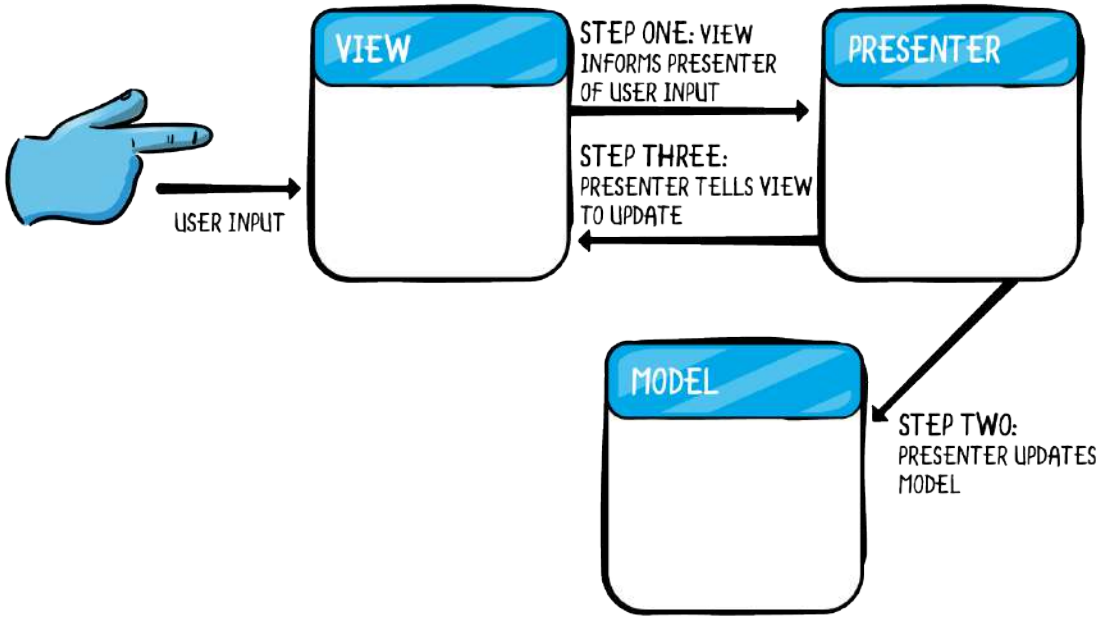
In summary, with the MVP pattern, the roles of the Model and the View are equivalent to those in MVC, except that now the `Activity` or `Fragment` is explicitly designated the role of the View. The Controller has now been replaced with a Presenter class, which, based on the definition above, seems to serve a similar role as the theoretical Controller.

However, there are several key characteristics of the MVP pattern that differentiate it from MVC and allow for it to work in Android: the order in which events occur, the breaking of the ties between View and Model and the use of interfaces.

The MVP flow

To use MVP, you'll need to reimagine the order in which events occur in the flow of an Android app. Unlike in MVC wherein the Controller is supposed to be the main entry point for the app, the main entry point is the View.

If the View serves as the entry point for user input — such as a button tap — that implies that the View must be an `Activity` or `Fragment`, as these Android classes are able to handle user input. The View can then inform the Presenter of the user input, and the Presenter can handle it by updating the Model. Once the Model has been updated, the Presenter can then update the View with the most up-to-date data.



MVC flow

For example, in your movie app, the user adds a new movie to the To-Watch list by entering the title and release year of the movie, and then they tap a button to add. In **Step One**, the `Activity` listens for user inputs and tells the Presenter that the user tapped the button:

```
fun onClickAddMovie(view: View) {
  ...
  presenter.addMovie(title, releaseDate, posterPath)
}
```

In **Step Two**, the Presenter updates the Model (here, served by the `localDataSource`) by giving it the new Movie to insert into the database:

```
override fun addMovie(title: String, releaseDate: String, posterPath:
String) {
  val movie = Movie(title, releaseDate, posterPath)
  localDataSource.insert(movie)
  ...
}
```

Once the data in the Model has been updated, the Presenter then updates the View in **Step Three**. In this particular example of adding a new movie, the Presenter simply instructs the View to finish and return to the main screen:

```
override fun addMovie(title: String, releaseDate: String, posterPath:
String) {
  val movie = Movie(title, releaseDate, posterPath)
  dataSource.insert(movie)
  view.returnToMain()
}
```

Separation of concerns and unit testing

As the example above demonstrates, the Presenter serves as a middleman between the Model and the View, shuttling information and updates between the two classes. Because the Presenter handles these responsibilities, the Model and the View do not need to be aware of each other. By breaking the direct tie between the Model and the View, MVP allows for better separation of concerns.

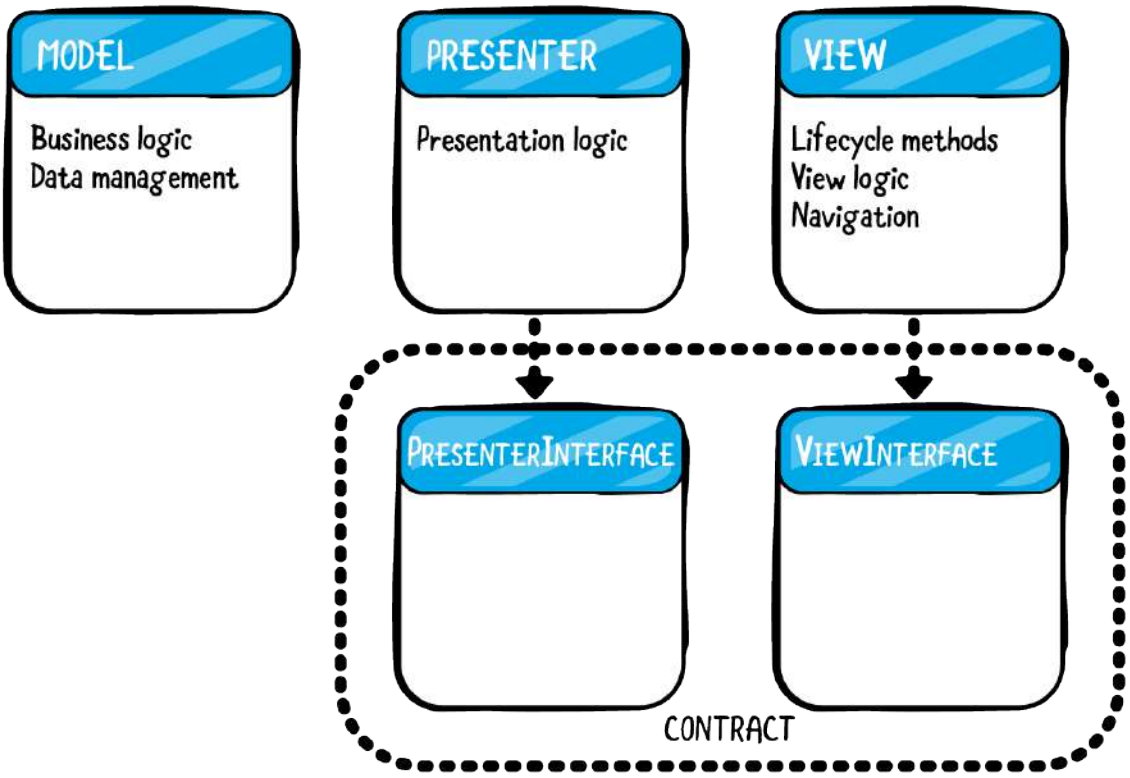
Furthermore, with this pattern, the Presenter can now be unit tested. Because it does not extend any class specific to the Android framework, the Presenter can be instantiated through a constructor like `var presenter = Presenter()`, allowing us to call methods on an instance of the Presenter like `presenter.onDeleteTapped(moviesToDelete)` to test that it behaves as expected in unit tests.

Using interfaces

With the exception of Android Architecture Component classes, which are allowable in Presenters, the Presenter should not contain references to any Android framework-specific classes, such as `Context`, `View`, or `Intent`. This rule allows you to write regular JUnit tests on the Presenter. However, the Presenter, of course, needs to talk to the View, which means it needs a reference to the Activity, an Android framework-specific class. How can you get around this reference to an Android class when you write your unit tests?

The way to resolve this problem is to create interfaces for the Presenter and View, keeping the interfaces in a single class called the Contract class. The Presenter class implements the `PresenterInterface` and the `Activity` implements the `ViewInterface`.

The Presenter class will then hold a reference to an instance of the ViewInterface interface, rather than a reference directly to the `Activity`. This way, the Presenter and View interact with each other through interfaces rather than actual implementations, which is, in general, a good software engineering principle that allows decoupling of the two classes. In this case, it also has the additional benefits of removing the Android-specific framework class `Activity` from the Presenter and allows you to mock the ViewInterface in the Presenter for unit testing.



Create interfaces for the Presenter and View

While it is necessary to create an interface for the View, strictly speaking, the same is not true of the Presenter interface. It would be perfectly fine to allow the View to interact with the actual implementation of the Presenter, especially since it is unlikely to have changing implementations of the Presenter. However, you choose to have a Presenter interface in this project so that the Contract class neatly documents the relationship between the Presenter and the View, clearly outlining the interactions.

MVP advantages and concerns

By dividing an `Activity` into separate Model, View, and Presenter classes with interfaces, you are able to achieve separation of concerns as well as unit-testable Models and Presenters. These are certainly considerable achievements, even if it means having to navigate through interfaces, which can be confusing at first. In Android Studio, `CMD` + `OPTION` + `B` on Mac or `CTRL` + `ALT` + `B` on PC is the keyboard shortcut for finding your way to actual implementations of interface methods.

Additionally, it is important to give some thought to what happens when the operating system destroys the `Activity`. When your `Activity` is destroyed, you must be sure to have your Presenter destroy any subscriptions or `AsyncTasks` or else that will cause problems when the task completes and the `Activity` is no longer there.

You'll also need to think about what happens to the Presenter upon destruction of the `Activity`. If you want the memory held by the Presenter to be freed and garbage-collected when the `Activity` is destroyed, you must make sure there are no references to the Presenter in any other classes that will continue to live in memory.

If you do want your presenter survive `Activity` destruction, you can try to save some state through the `onSaveInstanceState`, or use loaders, which survive configuration changes. In our example project, we will allow the Presenter to be destroyed when the `Activity` is destroyed.

Key points

- MVP stands for **Model-View-Presenter**.
- MVP is an architecture pattern whose main objective is the separation of concerns and increased unit testability.
- Unlike MVC wherein the main entry point is the Controller, in MVP the main entry point is the **View**.

- The **Model** is the data layer that handles business logic.
- The View displays the UI and informs the **Presenter** about user actions.
- The View extends `Activity` or `Fragment`.
- The Presenter tells the Model to update the data and tells the View to update the UI.
- The Presenter should not contain Android framework-specific classes.
- The Presenter and the View interact with each other through interfaces.

Where to go from here?

In the next chapter, you will apply your knowledge by rewriting the Movies app using MVP. The exercise will set you up for writing more unit tests for the app that previously were not possible without this architecture pattern.

Chapter 8: Model View Presenter Sample

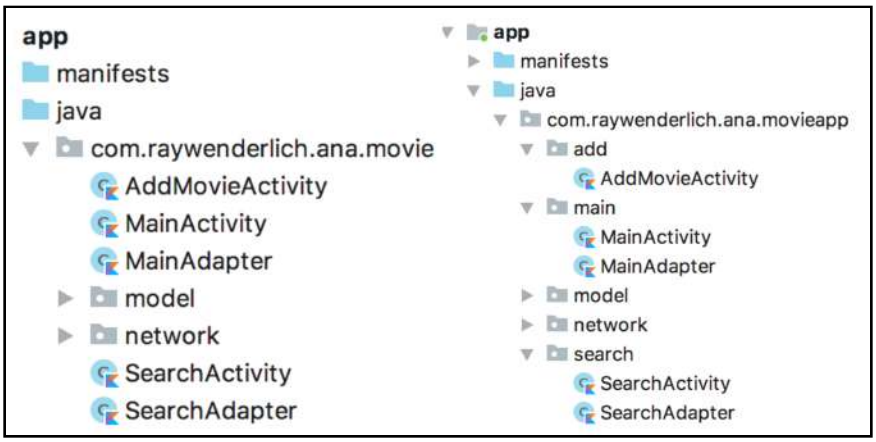
By Yun Cheng

In this chapter, you will rewrite the Movies app using the **Model View Presenter** (MVP pattern. Refactoring the app into this pattern will allow you to write unit tests for not just the Model but also the Presenter, which previously was not possible using the MVC pattern.

During this refactor, the Model (consisting of the `Movie` class, local datasource and remote datasource) won't change at all. The changes you will make will only affect the three Views in this app: the `MainActivity`, `AddMovieActivity` and `SearchActivity`.

Getting started

Before you start coding, you will reorganize your project folder structure to group together classes for each screen of the app. Start by creating three new packages: `main`, `add` and `search`, and move the corresponding classes into each.



Before and after directory structure.

Applying MVP to the Movies app

For each of the three screens, you need to do the following:

1. Create a new Presenter class and connect it to the View and Model via dependency injection.
2. Create a new Contract class that contains the PresenterInterface and ViewInterface.
3. Move all presentation logic into the Presenter.

Moving the presentation logic out of the View and into the Presenter class will achieve a greater degree of separation of concerns. You will reduce the role of the View to be that of displaying the data, listening for user input and handling navigation. Any logic that does not fall under one of those categories should be moved out into the Presenter so that the logic can be unit tested.

For each of the Views, you will also create a corresponding Contract class that contains interfaces for the Presenter and View. It is necessary for the Presenter to interact with an interface of the View because you must keep the Presenter free of any Android framework-specific classes if you wish to write unit tests on the Presenter.

Although the Presenter doesn't strictly need an interface, you will also create one for it anyway and treat the Contract class as a document that describes communication between the View and Presenter.

The Main screen

Recall that the main screen of the app displays the user's list of movies to watch, with a Delete icon in the toolbar and an Add floating action button. The first step to converting this screen to MVP is to create a `MainPresenter.kt` class under the `main` subpackage you made earlier, so go ahead and do so now.

> **Note**: Whenever you add code, make sure to import the appropriate packages by pressing **Alt+Enter** on Windows or **Option+Enter** on Mac.

```
class MainPresenter(private var view: MainActivity, private var
dataSource: LocalDataSource) {

  private val TAG = "MainPresenter"
  ...
}
```

This Presenter will need to interact with both the View and the Model, so the `MainActivity` and the `LocalDataSource` must be passed into the constructor. As you learned in Chapter 5: "Dependency Injection," it is important to inject dependencies like the View and the Model in through the constructor for unit testing purposes. When you write unit tests for `MainPresenter` you will pass in mock objects for the View and the Model; this way you will unit test the `MainPresenter` class itself and not its dependencies.

Now, create a `MainContract.kt` class under the `main` subpackage:

```
class MainContract {

  interface PresenterInterface {
    //TODO: add interface methods for Presenter
  }

  interface ViewInterface {
    //TODO: add interface methods for View
  }
}
```

For now, this class will contain empty definitions for a `PresenterInterface` and a `ViewInterface`. You will add methods to these interfaces later, as you build out the Presenter.

Now that you have the interfaces set up for the View and the Presenter, open **MainActivity.kt** and update the `MainActivity` to implement the `MainContract.ViewInterface` like the following:

```
class MainActivity : AppCompatActivity(), MainContract.ViewInterface {
  ...
}
```

Next, open **MainPresenter.kt** and update the `MainPresenter` class so that it implements the `MainContract.PresenterInterface` and also holds a reference to a `MainContract.ViewInterface` for its View rather than a reference to the direct `MainActivity` implementation:

```
class MainPresenter(
    private var viewInterface: MainContract.ViewInterface,
    private var dataSource: LocalDataSource) :
MainContract.PresenterInterface {

  private val TAG = "MainPresenter"
  ...
}
```

Throughout this Presenter class, the Presenter will interact with the `MainContract.ViewInterface` rather than the `MainActivity` implementation directly. This is done to avoid having Android framework-specific classes like `Activity` in the Presenter. `MainActivity` extends `AppCompatActivity`, which is specific to the Android framework and cannot be mocked, making it difficult to write unit tests.

Now that you have created the `MainPresenter` class, you need to instantiate the `MainPresenter` inside the `MainActivity`.

Open **MainActivity.kt** again, and in the `MainActivity` class, add the following code:

```
private lateinit var mainPresenter: MainContract.PresenterInterface

private fun setupPresenter() {
  val dataSource = LocalDataSource(application)
  mainPresenter = MainPresenter(this, dataSource)
}
```

Here, you instantiate the `MainPresenter`, pass in the `Activity` itself using the `this` keyword and a local instance of the `LocalDataSource`.

Next add a call to `setupPresenter()` inside the `MainActivity`'s `onCreate()` method:

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)
  setupPresenter()
  setupViews()
}
```

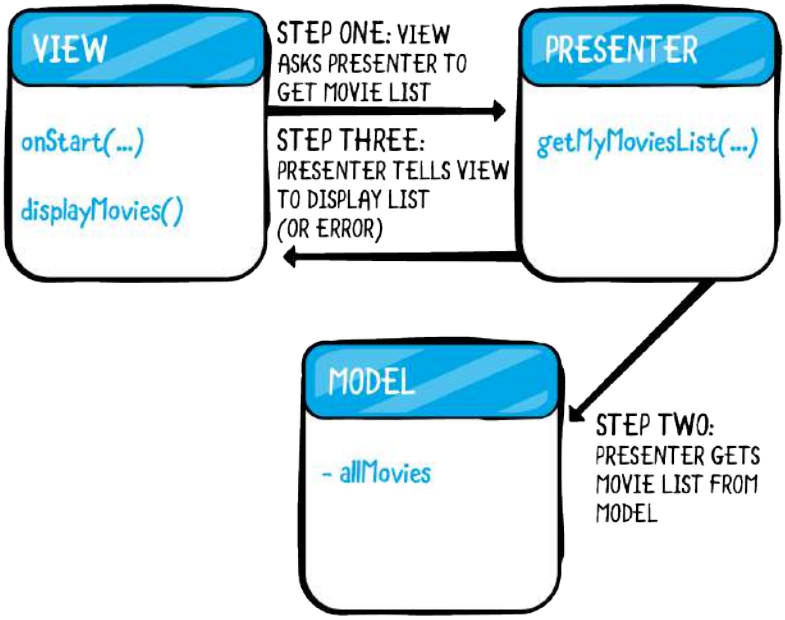
Great! You've wired up a basic Presenter, but it doesn't really do much yet. In the next section, you'll add code for the Presenter to retrieve movies.

Fetching movies

Now that you have connected the Presenter and View, you can begin to move the presentation logic out of the View and into the Presenter. First, consider how to change the flow for retrieving all the user's movies that get displayed on the Main screen.

Instead of having the View (`Activity`) perform both the retrieving and displaying of the movies, you can move the retrieving logic into the Presenter.

The following diagram breaks down the sequence of steps.



Flow for retrieving movies.

You can see that this is a three step process:

- **Step One**: The view asks the Presenter to get the movie list.
- **Step Two**: The Presenter gets the movie list from the Model. The Model returns a list of `Movie` objects to the Presenter or an error in the event that the Model cannot return the list.
- **Step Three**: The Presenter then uses the `ViewInterface` to tell the View how to display the error. If the Presenter receives an error from the Model, it will tell the `ViewInterface` to present an error state to the user.

Step One delegates the responsibility of retrieving movies the Presenter, so start by opening **MainContract.kt** and adding the `getMyMoviesList()` method to the Presenter's interface as follows:

```
interface PresenterInterface {
  fun getMyMoviesList()
}
```

Then, implement **Step One** of the flow by opening **MainActivity.kt** and replacing the contents of the `onStart()` method of `MainActivity` as follows:

```
override fun onStart() {
  super.onStart()
  mainPresenter.getMyMoviesList()
}
```

As soon as the View starts, it asks the Presenter to retrieve the list of movies.

Next, you'll implement **Step Two** and **Step Three** of the flow, getting and displaying the list of movies.

Open **MainPresenter.kt** and add the following code, moved over from the `MainActivity`:

```
private val compositeDisposable = CompositeDisposable()

//1
val myMoviesObservable: Observable<List<Movie>>
  get() = dataSource.allMovies

//2
val observer: DisposableObserver<List<Movie>>
  get() = object : DisposableObserver<List<Movie>>() {

    override fun onNext(movieList: List<Movie>) {
      if (movieList == null || movieList.size == 0) {
        viewInterface.displayNoMovies()
      } else {
        viewInterface.displayMovies(movieList)
      }
    }

    override fun onError(@NonNull e: Throwable) {
      Log.d(TAG, "Error fetching movie list.", e)
      viewInterface.displayError("Error fetching movie list.")
    }

    override fun onComplete() {
      Log.d(TAG, "Completed")
    }
  }

//3
override fun getMyMoviesList() {
  val myMoviesDisposable = myMoviesObservable
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribeWith(observer)

  compositeDisposable.add(myMoviesDisposable)
}
```

That's a lot of code, so let's go through it one step at a time:

1. The `myMoviesObservable` is an observable for the list of `Movie` objects from the `dataSource`. This code is the same as before, except now the Presenter will be the one interacting with the `dataSource` to get movies, which is why you passed in the `dataSource` to the Presenter through the constructor.

2. Inside the `observer`, the Presenter determines how to consume the list of movies it receives, depending on the size of the list and whether an error has occurred. What has changed in this code is that we must interact with the View by calling methods on the `viewInterface` property to respond to each of these scenarios. The Presenter does not know how to display lists or errors, so it delegates those tasks to the View. If there are no movies to display, the Presenter asks the View to display no movies. If there are movies to display, the Presenter asks the View to display the list of movies.

3. The `getMyMoviesList()` method connects the `observer` to the `myMoviesObservable` so that it can begin observing. The method is exactly the same as before, except now it belongs to the Presenter.

As you can see, in **Step Two** of the retrieval flow occurs in `getMyMoviesList()`, where the Presenter gets the list of movies from the Model by subscribing to an observable of the `LocalDataSource`'s list of `allMovies`. Although the Presenter contains the logic that decides *what* to display, the actual job of displaying content belongs to the View.

As part of **Step Three** in the flow, the Presenter will ask the View to display the list of movies; otherwise, it will display a view that indicates that there are no movies to display or else display an error message if an error occurred.

Next, the code inside the `MainActivity` will need to be slightly refactored to handle these scenarios. But, first, you need to define the View interface.

Open **MainContract.kt** and add the following methods to `MainContract.ViewInterface`:

```
interface ViewInterface {
  fun displayMovies(movieList: List<Movie>)
  fun displayNoMovies()
  fun displayMessage(message: String)
  fun displayError(message: String)
}
```

Now, open up **MainActivity.kt** again, remove the existing implementation of `displayMovies()`, and add the following methods:

```
//1
override fun displayMovies(movieList: List<Movie>) {
  adapter.movieList = movieList
  adapter.notifyDataSetChanged()

  moviesRecyclerView.visibility = VISIBLE
  noMoviesTextView.visibility = INVISIBLE
}

//2
override fun displayNoMovies() {
  Log.d(TAG, "No movies to display.")

  moviesRecyclerView.visibility = INVISIBLE
  noMoviesTextView.visibility = VISIBLE
}
```

Let's take the above code in turn:

1. In this scenario in which there is a list of movies to display, the adapter's `movieList` gets updated to the list passed in, and the adapter calls `notifyDataSetChanged()` to update the data. The `RecyclerView` is shown and the `TextView` hidden.

2. When there are no movies to display, the method hides the `RecyclerView` and shows a `TextView` indicating to the user that there are no movies to display.

Also notice that the `MainAdapter` remains an instance variable of the `MainActivity`.

By definition, the Adapter class serves as a bridge between the view and the data source, so it is somewhat unclear whether the Adapter belongs to the View or Presenter. But because the RecyclerView undoubtedly belongs in the View, and a RecyclerView requires an associated Adapter, you will keep the Adapter inside the View.

Next, although the `MainActivity` already contains the methods `displayMessage(message: String)` and `displayError(message: String)`, you want to override them as part of the `ViewInterface` methods. This way, the Presenter will be able to call those methods on its instance of `ViewInterface`, such as when it calls `viewInterface.displayError("Error fetching movies")` when it fails to retrieve any movies.

Inside `MainActivity`, simply prepend the `override` keyword to each of the two methods like so:

```
override fun displayMessage(message: String) {
  Toast.makeText(this@MainActivity, string, Toast.LENGTH_LONG).show()
}

override fun displayError(message: String) {
  displayMessage(message)
}
```

You're nearly done, but before we can move on, we should implement a means to clean up our `Observable` subscriptions. Even though the `compositeDisposable` now lives in the Presenter rather than the Activity, you will still need to clear the `CompositeDisposable` when the `Activity` is stopped. In the Activity's `onStop()`, be sure to tell the Presenter to do that work by changing that method to look like this:

```
override fun onStop() {
  super.onStop()
  mainPresenter.stop()
}
```

Now, open **MainContract.kt** and add the `stop()` method to the `PresenterInterface` as follows:

```
interface PresenterInterface {
  ...
  fun stop()
}
```

Then open **MainPresenter.kt** and implement the method like this:

```
override fun stop() {
  compositeDisposable.clear()
}
```

This method clears the `CompositeDisposable` so that it is no longer observing once the `Activity` has stopped.

Build and run the app on a device or emulator. It should function the same as before.

Though it may seem like all the changes you've made so far haven't accomplished much, you've actually done quite a bit to increase the separation of concerns for this part of the app! Just open up the `MainContract` to see evidence of this:

```
interface PresenterInterface {
  fun getMyMoviesList()
  fun stop()
}
```

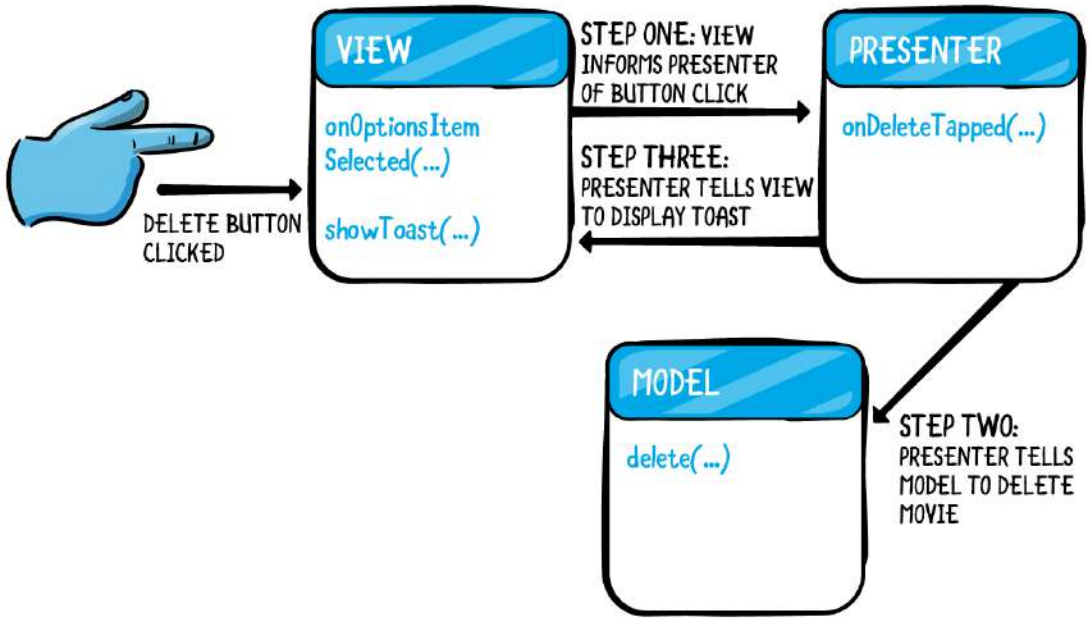
```
interface ViewInterface {
  fun displayMovies(movieList: List<Movie>)
  fun displayNoMovies()
  fun displayMessage(message: String)
  fun displayError(message: String)
}
```

The methods in this contract clearly delineate the roles of the Presenter and View: While the Presenter is responsible for interacting with the Model to get the movie list, the View is only responsible for hiding and displaying UI as directed by the Presenter.

Note: To easily navigate to actual implementations of interface methods in Android Studio, use `CMD` + `OPTION` + `B` on Mac or `CTRL` + `ALT` + `B` on Windows.

Deleting movies

Next, consider the flow for deleting movies. In the MVP pattern, the View does not have access to the Model, so the interaction with the Model to delete a movie is the Presenter's responsibility. Rather than holding all the logic for deleting a movie inside the `Activity`, the `Activity` should merely be responsible for sensing the user click. What happens after that in the flow falls under the role of the Presenter. The new flow should look like this:



Flow for deleting movies.

Once again, the flow breaks down into three steps:

- **Step One**: The View notifies the Presenter through the `PresenterInterface` that the user would like to delete a given `Movie`.
- **Step Two**: The Presenter tells the Model to delete a given `Movie`.
- **Step Three**: If the `Movie` was successfully deleted, the Presenter notifies the View of the deletion through the `ViewInterface`.

To implement **Step One**, open **MainActivity.kt** and in the `MainActivity`'s `onOptionsItemSelected`, replace the contents of the method with the following:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
  if (item.itemId == R.id.deleteMenuItem) {
    mainPresenter.onDeleteTapped(adapter.selectedMovies)
  }

  return super.onOptionsItemSelected(item)
}
```

In the above snippet, the View simply informs the Presenter that the Delete button has been tapped, and it passes in the `HashSet` of movies to be deleted.

Next, open **MainContract.kt** and add the interface method `fun onDeleteTapped(selectedMovies: HashSet<*>)` to `MainContract.PresenterInterface`. Then, implement the method in the Presenter like so:

```
override fun onDeleteTapped(selectedMovies: HashSet<*>) {
  for (movie in selectedMovies) {
    dataSource.delete(movie as Movie)
  }
  if (selectedMovies.size == 1) {
    viewInterface.displayMessage("Movie deleted")
  } else if (selectedMovies.size > 1) {
    viewInterface.displayMessage("Movies deleted")
  }
}
```

Upon receiving the `HashSet` of movies, the Presenter performs **Step Two** of the flow, interacting with the Model to delete each movie in the `HashSet`. Then, there is some logic to determine what the View should show in a `Toast`. Depending on the number of deleted movies, the Presenter performs **Step Three** of the flow which is to notify the View of the deletion.

At this point, you can clean up the `MainActivity` by removing the instance variable `dataSource`, as the View does not interact with the Model in the MVP pattern.

There should be no more references to the `dataSource` instance within `MainActivity`, because all interaction with the Model, including retrieval and deletion, has been moved into the `MainPresenter`.

Scroll through the methods in `MainActivity` at this point and notice that the methods that remain only handle displaying views, listening to clicks, and navigating to a new screen. Thus, there is not much value in writing tests for the View, as you can assume the actual act of displaying a `Toast` or opening a new screen will work as expected.

Any logic that *is* worth testing is moved into the Presenter, which you will thoroughly unit test in the next chapter. Build and Run your app; deletion should work just the same as before.

The Add Movie screen

The Add Movie screen displays two text inputs for the user to fill out with the movie information, with an Add Movie button to submit the movie data. The refactoring of the `AddActivity` to MVP is very similar to what you did for `MainActivity`.

Create an `AddMoviePresenter.kt` class and an `AddMovieContract.kt` class under the `add` subpackage you made earlier. Add these interfaces to the `AddMovieContract`:

```
class AddMovieContract {
  interface PresenterInterface {
    fun addMovie(title: String, releaseDate: String, posterPath: String)
  }

  interface ViewInterface {
    fun returnToMain()
    fun displayMessage(message: String)
    fun displayError(message: String)
  }
}
```

Then, create a new **AddMoviePresenter.kt** file and add the following:

```
class AddMoviePresenter(
    private var viewInterface: AddMovieContract.ViewInterface,
    private var dataSource: LocalDataSource) :
AddMovieContract.PresenterInterface {

  override fun addMovie(
        title: String,
        releaseDate: String,
        posterPath: String) {
  }
}
```

The `AddMoviePresenter` class implements the `AddMovieContract.PresenterInterface` and takes in a reference to the `AddMovieContract.ViewInterfaceand` a reference to the Model, the `LocalDataSource` class.

The reason for injecting these two dependencies into the constructor is to unit test `AddMoviePresenter` more easily. During unit tests you will pass in mock objects into the constructor to make it easier to unit test just the `AddMoviePresenter` class itself rather than its dependencies. For now, you'll leave the `addMovie` method empty; you'll flesh that out in a few steps.

Next, open **AddMovieActivity.kt**, and have the class implement the `AddMovieContract.ViewInterface` like so:

```
class AddMovieActivity : AppCompatActivity(),
AddMovieContract.ViewInterface {
  ...
}
```

Because `AddMovieActivity` now implements `AddMovieContract.ViewInterface`, prepend the `override` keyword to the `displayMessage(message: String)` and `displayError(message: String)` methods of `AddMovieActivity`:

```
override fun displayMessage(message: String) {
  Toast.makeText(this@AddMovieActivity, string, Toast.LENGTH_LONG).show()
}

override fun displayError(message: String) {
  displayMessage(message)
}
```

Then, add the following to `AddMovieActivity`:

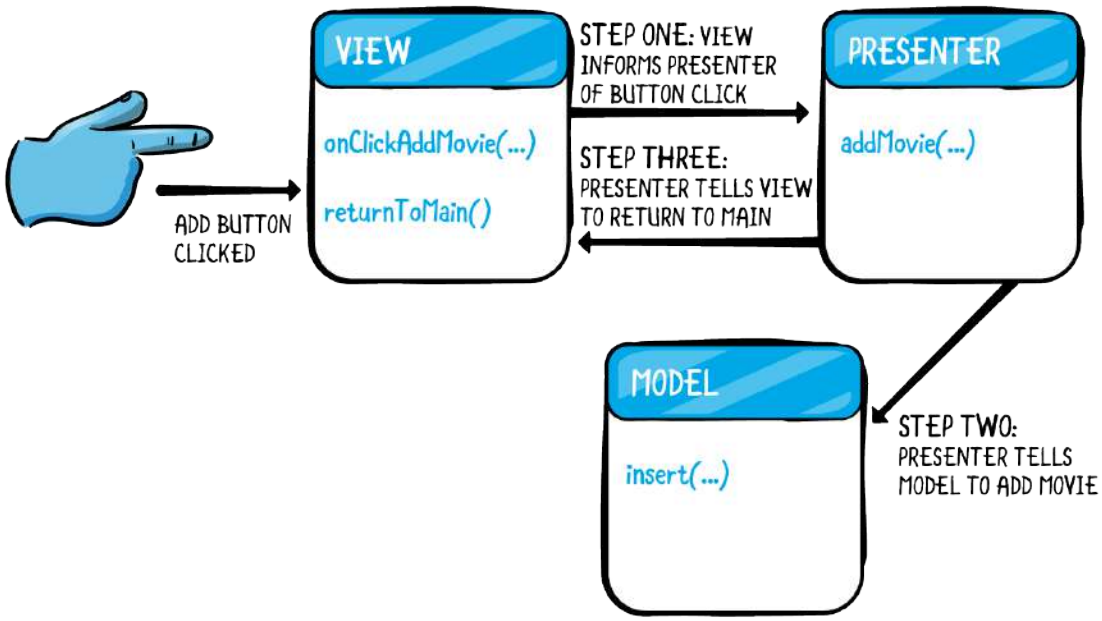
```
private lateinit var addMoviePresenter: AddMoviePresenter

fun setupPresenter() {
  val dataSource = LocalDataSource(application)
  addMoviePresenter =  AddMoviePresenter(this, dataSource)
}
```

Call this `setupPresenter()` method inside the `AddMovieActivity`'s `onCreate()` to instantiate the Presenter and pass a local instance of the Model to the Presenter.

Adding movies

Now, you are ready to move the presentation logic surrounding the adding of movies out of the View and into the Presenter. Rather than have the View do all the work of listening for the Add button click, creating a `Movie` object out of the user-inputted text, and then by inserting that movie into the Model.

The new flow should move the non-UI-related responsibilities to the Presenter to look like this:



Flow for adding movies.

In the same three step process you've seen before, the flow breaks down as follows:

- **Step One**: The View notifies the Presenter through the `PresenterInterface` that the user would like to add a new `Movie` to their list.
- **Step Two**: The Presenter inserts the `Movie` directly into the Model. In the use case where the data that is passed from the View to the Presenter for insertion, the Presenter might use this opportunity to perform validation on the supplied data.
- **Step Three**: The Presenter notifies the View through the `ViewInterface` that the `Movie` has been added, and that it should transition back to the `Main` View.

To implement **Step One**, open `AddMovieActivity` and replace the `onClickAddMovie(view: View)` method with this one:

```
  fun onClickAddMovie(view: View) {
    val title = titleEditText.text.toString()
    val releaseDate = releaseDateEditText.text.toString()
    val posterPath = if (movieImageView.tag != null)
movieImageView.tag.toString() else ""

    addMoviePresenter.addMovie(title, releaseDate, posterPath)
  }
```

The View gathers all the input from the user, including the title, release date and poster path, and it passes them to the Presenter. Then, the `AddMoviePresenter` determines what to do with the inputted data.

Open **AddMoviePresenter.kt** and fill out the `addMovie()` method like this:

```
override fun addMovie(title: String, releaseDate: String, posterPath:
String) {
  //1
  if (title.isEmpty()) {
    viewInterface.displayError("Movie title cannot be empty")
  } else {
    //2
    val movie = Movie(title, releaseDate, posterPath)
    dataSource.insert(movie)
    viewInterface.returnToMain()
  }
}
```

1. If the user did not input the movie title, then show an error message.
2. Otherwise, if at least the title of the movie was provided, the Presenter will create a new `Movie` object and ask the Model, the `LocalDataSource`, to insert that movie into the local database in **Step Two** of the flow. After that, in **Step Three**, the Presenter asks the View to finish the `Activity` and return to the main screen.

Open **AddMovieActivity.kt** once again, and add this `returnToMain()` method to the `AddMovieActivity` class:

```
override fun returnToMain() {
  setResult(Activity.RESULT_OK)
  finish()
}
```

Because only a class that extends `Activity` knows how to `finish()` itself, the handling of navigation is therefore left to the View.

As a final step, remove the `LocalDataSource` instance variable initialization and instantiation from the `AddMovieActivity` to break the connection between the View and the Model, as is required in the MVP pattern.

Build and run the app to verify that everything still works properly.

The Search Movie screen

Recall that the Search Movie screen displays the list of search results for the movie title query that was passed in through the Intent. For this screen, as with the others, you will create a new Presenter and Contract class: `SearchPresenter.kt` and `SearchContract.kt`. First add the Contract class:

```
class SearchContract {

  interface PresenterInterface {
    fun getSearchResults(query: String)
    fun stop()
  }

  interface ViewInterface {
    fun displayResult(tmdbResponse: TmdbResponse)
    fun displayMessage(message: String)
    fun displayError(message: String)
  }
}
```

The Contract class between the Presenter and the View clearly differentiates the roles of the two interfaces. The role of the View is simply to display views, while the responsibility of communicating with the Model to fetch the search results falls on the Presenter.

Open **SearchPresenter.kt** and add the following class:

```
class SearchPresenter(
    private var viewInterface: SearchContract.ViewInterface,
    private var dataSource: RemoteDataSource) :
SearchContract.PresenterInterface {
  private val TAG = "SearchPresenter"
  ...
}
```

The `SearchPresenter` needs references to both the ViewInterface and the `RemoteDataSource` passed in so that it can interact with the `RemoteDataSource` to fetch results and then tell the View what to display afterward. By injecting these dependencies through the `SearchPresenter`'s constructor, it will allow for mock objects to be passed in for the View and RemoteDataSource when writing unit tests later.

Next, open **SearchActivity.kt** and have it implement `SearchContract.ViewInterface` like so:

```
class SearchActivity : AppCompatActivity(), SearchContract.ViewInterface
{
  ...
}
```

Just like you did with the other screens, prepend the search screen's `displayResult(tmdbResponse: TmdbResponse)` and `displayMessage(message: String)` methods with the `override` keyword in the `SearchActivity` so that the `SearchActivity` conforms to the contract for `SearchContract.ViewInterface`.

Next, add a `searchPresenter` instance variable to the `SearchActivity` and add a method that sets up the Presenter:

```
private lateinit var searchPresenter: SearchPresenter

private fun setupPresenter() {
  val dataSource = RemoteDataSource()
  searchPresenter = SearchPresenter(this, dataSource)
}
```

To instantiate the `SearchPresenter`, the `SearchActivity` is passed in through `this` keyword, and the RemoteDataSource is also passed in. Be sure to call `setupPresenter()` in the `SearchActivity`'s `onCreate()`. Now that your Presenter and View are properly connected, you can move the logic of fetching search results out of the View and into the Presenter. Here is the MVP way to execute the flow:

To implement **Step One** in this flow, add a call to the Presenter's `searchPresenter.getSearchResults()` to `onStart()` in `SearchActivity` like this:

```
override fun onStart() {
  super.onStart()
  progressBar.visibility = VISIBLE
  searchPresenter.getSearchResults(query)
}
```

Next, open **SearchPresenter.kt** and add the following, moved from `SearchActivity`:

```
private val compositeDisposable = CompositeDisposable()

//1
val searchResultsObservable: (String) -> Observable<TmdbResponse> =
{ query -> dataSource.searchResultsObservable(query) }

//2
val observer: DisposableObserver<TmdbResponse>
  get() = object : DisposableObserver<TmdbResponse>() {

    override fun onNext(@NonNull tmdbResponse: TmdbResponse) {
      Log.d(TAG, "OnNext" + tmdbResponse.totalResults)
      viewInterface.displayResult(tmdbResponse)
    }

    override fun onError(@NonNull e: Throwable) {
      Log.d(TAG, "Error fetching movie data.", e)
      viewInterface.displayError("Error fetching movie data.")
    }
```

```
      override fun onComplete() {
        Log.d(TAG, "Completed")
      }
    }

  //3
  override fun getSearchResults(query: String) {
    val searchResultsDisposable = searchResultsObservable(query)
      .subscribeOn(Schedulers.io())
      .observeOn(AndroidSchedulers.mainThread())
      .subscribeWith(observer)

    compositeDisposable.add(searchResultsDisposable)
  }
```

1. The `searchResultsObservable` is an observable for the `TmdbResponse` from the `dataSource`, taking a movie title query as an input. This code is the same as before, except now the Presenter will be the one interacting with the `dataSource` to get the API response, which is why we passed in the `dataSource` to the Presenter through the constructor.

2. Inside the `observer`, the Presenter determines how to consume the response it receives, depending on if it was a success or error. What has changed in this code is that we must call the `viewInterface` to perform responses to each of these scenarios. The Presenter does not know how to display the response or errors so it delegates those tasks to the `viewInterface`. If there is a successful response to display, the Presenter asks the `viewInterface` to display the result. If there is an error, the Presenter asks the `viewInterface` to display an error `Toast`.

3. The `getSearchResults(query: String)` connects the `observer` to the `searchResultsObservable` so that it can begin observing. The method is exactly the same as before, except now it belongs to the Presenter.

In the code above, the Presenter performs **Step Two** and **Step Three** of the flow diagram, setting up the observable to get the search results from the Model and, then upon receiving them, instructing the View to display the results.

Again, the `CompositeDisposable` must be stopped when the `Activity` stops, so in the Presenter, remember to add this method:

```
  override fun stop() {
    compositeDisposable.clear()
  }
```

Open **SearchActivity.kt** again and call the `stop()` method of the Presenter in the activity `onStop()` like this:

```
override fun onStop() {
  super.onStop()
  searchPresenter.stop()
}
```

Finally, clean up the `SearchActivity` by removing the instance variable `dataSource`. There should be no more references to the `dataSource` instance within `SearchActivity`, because all interaction with the Model to get search results has been moved into the `SearchPresenter`.

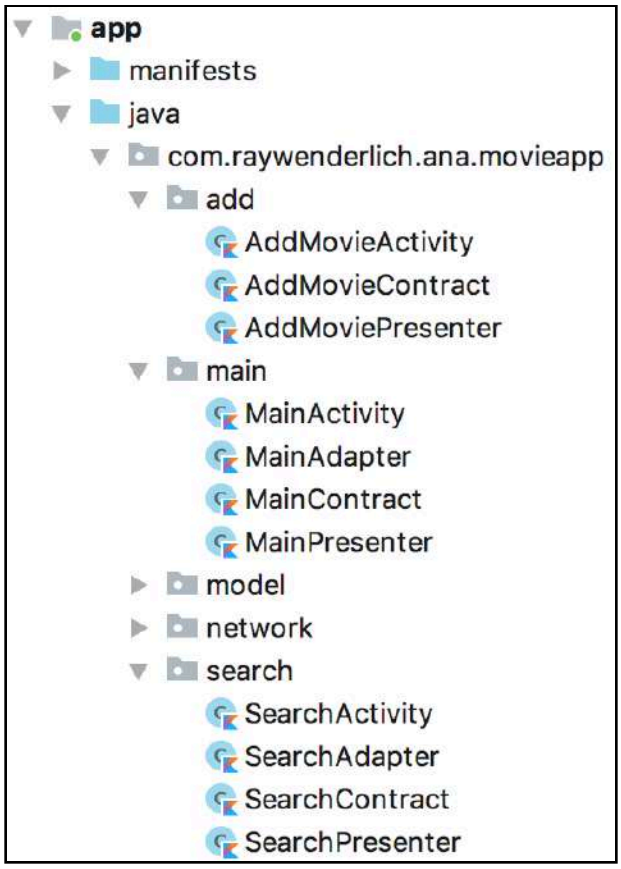
Build and run your app to confirm that searching for a movie still works as expected.

Key points

- In the **Model View Presenter** pattern, each View interacts with an associated Presenter class.
- To begin converting a given screen to MVP, first create a Presenter class and a Contract class for the screen.
- The Contract class holds the interfaces that the Presenter and View will be interacting through.
- In the View's `onCreate()`, call a method to setup the Presenter.
- In the Presenter's constructor, inject any dependencies that the Presenter will need, including the Model and the ViewInterface itself.
- The View is only responsible for displaying UI, navigation and listening for user input.
- Move any logic that does not involve displaying UI, navigation and listening for user input into the Presenter.
- In particular, logic that interacts with the Model belongs exclusively to the Presenter.
- Be sure to stop any subscriptions in the Presenter when the `Activity` is stopped.

Where to go from here?

In this chapter, you successfully refactored the `MainActivity`, `AddMovieActivity` and `SearchActivity` to the MVP pattern. After adding Presenter and Contract files to each of your Views, here is what your project directory should look like now:



As you were building and running your project throughout his chapter, you confirmed that the app ran exactly the same as before. A user would not notice any difference with the app now refactored into MVP, as the functionality has not changed. What *has* changed in each of the screens of the app by refactoring to MVP is the extent to which each type of class is now focused on their own designated tasks, achieving a greater degree of separation of concerns, as well as greater decoupling.

However, the biggest change that converting to MVP has had on this project has yet to be demonstrated: improved testability. As you will see in the next chapter, keeping the Android framework-specific code out of the Presenter allows the Presenter to be completely unit testable. Now you are ready to write some tests!

Chapter 9: Testing MVP

By Yun Cheng

Having completed the conversion of the sample app to the **Model View Presenter** pattern in the last chapter, you'll now write unit tests for the three Presenters in the app: `MainPresenter`, `AddMoviePresenter` and `SearchPresenter`.

Getting started

Before you can write your tests, there are some housekeeping steps you need to complete:

1. Create a base test class to wrap the `capture()` functionality for Mockito `ArgumentCaptors`.
2. Create a custom TestRule for testing your RxJava calls.

Getting to know Mockito

This book will, for the most part, will use Mockito to help with testing. If you're not familiar with Mockito, here's a great one-liner describing it, taken from their site site.mockito.org

> The Mockito library enables mock creation, verification and stubbing.

If you want a deep dive into the library, its highly recommended to check out the tutorials on the Ray Wenderlich site, specifically **Android Unit Testing with Mockito** www.raywenderlich.com/195-android-unit-testing-with-mockito.

However, a high level description of some of the features we'll be using is given below:

- `mock()`/`@Mock`: create mock optionally specify how it should behave via Answer/ MockSettings
- `when()`/`given()` to specify how a mock should behave If the provided answers don't fit your needs, write one yourself extending the Answer interface
- @InjectMocks: automatically inject mock fields annotated with @Mock
- `verify()`: to check methods were called with given argumentscan use flexible argument matching, for example any expression via the any()or capture what arguments were called using @Captor instead

The starter project should already have Mockito added as a dependency, however if you're following along in your own project, be sure to add the following libraries to the dependencies in your build.gradle file.

```
testImplementation 'org.mockito:mockito-core:2.2.5'
testImplementation('com.nhaarman:mockito-kotlin-kt1.1:1.5.0', {
        exclude group: 'org.jetbrains.kotlin', module: 'kotlin-stdlib'
})
```

The first dependency pull Mockito in for unit testing, and the second dependency simply provides some helper functions to work with Mockito and Kotlin.

Mockito can be really helpful when writing unit tests, however, there's more boiler plate stuff you'll go over in the next section in order to get Mockito to work with Kotlin.

Wrapping Mockito ArgumentCaptors

Sometimes, the mock objects in your unit tests will make use of Mockito `ArgumentCaptors` in method arguments to probe into the arguments that were passed into a method. Using Mockito's `capture()` method to capture an `ArgumentCaptor` is fine in Java, but when you write your unit tests in Kotlin, you'll get the following error:

```
java.lang.IllegalStateException: classCaptor.capture() must not be null
```

This error is due to a difference in the way Java and Kotlin handle null safety. With Mockito's `ArgumentCaptor`, `capture()` returns `null`. If the method you're stubbing is expecting a non-null parameter, Kotlin enforces that rule and throws an error when you try to use an `ArgumentCaptor`.

If you still want the ability to use this feature in your Kotlin code, you must write a wrapper for `ArgumentCaptor`'s `capture()` method in a class, and then have your test classes extend that base class.

Inside **test**, create a new file named **BaseTest.kt** and add the following:

```
open class BaseTest {
  open fun <T> captureArg(argumentCaptor: ArgumentCaptor<T>): T =
argumentCaptor.capture()
}
```

This `BaseTest` class contains the method `captureArg()`, making use of Generics to cast the null object returned by Mockito into the specific class object. Your test classes will extend `BaseTest` and have access to this modified version of `capture()` when you want to use them.

Adding a TestRule for RxJava Schedulers

Recall that by design, the Presenters in an MVP app do not have references to Android framework specific classes such as `Context`. This rule is what allows you to write JUnit tests on the Presenters. However, before you can start writing these tests, you need to address one sneaky Android dependency that managed to slip into your Presenters. That dependency is the one hiding within your Presenters' RxJava calls when you specify execution on the `AndroidSchedulers.mainThread()`.

If you try to write unit tests for the various RxJava calls you have in the Presenters, you'll see this exception thrown:

```
Caused by: java.lang.RuntimeException: Method getMainLooper in
android.os.Looper not mocked. See http://g.co/androidstudio/not-mocked
for details.
```

When retrieving movies from the local database and getting search results from your web service, the observables produce results on the `Schedulers.io()` thread to avoid blocking the main thread, and the results are observed on the `AndroidSchedulers.mainThread()`.

To fix this, you'll need to tell your test code to use different schedulers from the ones in the production code you're testing. You can achieve this using `RxAndroidPlugins` hooks and apply to your tests via a custom `TestRule`.

> **Note**: A TestRule is an alteration in how a test method, or set of test methods, is run and reported. A TestRule may add additional checks that cause a test that would otherwise fail to pass, or it may perform necessary setup or cleanup for tests. TestRules can do everything that could be done previously with methods annotated with @Before or @After, but they are more powerful, and more easily shared between projects and classes. Multiple TestRules can be applied to a test or suite execution.

Inside **test**, create a new file named **RxImmediateSchedulerRule.kt** and add the following:

```
class RxImmediateSchedulerRule : TestRule {

  override fun apply(base: Statement, d: Description): Statement {
    return object : Statement() {
      @Throws(Throwable::class)
      override fun evaluate() {
        //1
        RxJavaPlugins.setIoSchedulerHandler {
          Schedulers.trampoline()
        }
        RxJavaPlugins.setComputationSchedulerHandler {
          Schedulers.trampoline()
        }
        RxJavaPlugins.setNewThreadSchedulerHandler {
          Schedulers.trampoline()
        }
        RxAndroidPlugins.setInitMainThreadSchedulerHandler {
          Schedulers.trampoline()
        }

        try {
          //2
          base.evaluate()
        } finally {
          //3
          RxJavaPlugins.reset()
          RxAndroidPlugins.reset()
        }
      }
    }
  }
}
```

Diving into the code:

1. `TestRule`, when applied to tests, specifies the `Schedulers.trampoline()` scheduler to your test code, overriding whatever scheduler was specified by the production code.
2. Call `base.evaluate()` to evaluate the statement.
3. Reset the hook afterward.

Testing the MainPresenter

Now you're ready to create your test classes, starting with the test class for `MainPresenter`. Because **MainPresenter.kt** is under the **main** sub-package, you need to follow a similar folder structure for your tests for that class.

Inside **test**, create a **main** sub-package. Then, in that sub-package, create a new file named **MainPresenterTests.kt** and add the following:

```
@RunWith(MockitoJUnitRunner::class)
class MainPresenterTests : BaseTest() {
  @Rule @JvmField var testSchedulerRule = RxImmediateSchedulerRule()
}
```

This code sets up some rules for `MainPresenterTests`, which is a subclass of `BaseTest`. Having it run with `MockitoJUnitRunner` will validate framework usage after each test method and initialize mocks annotated with `@Mock`, while adding the `RxImmediateSchedulerRule` will address the RxJava Schedulers issue as explained in the previous section.

Next, you'll set up the instances of the View, the Model and the Presenter. The Presenter is the class you're testing, while the View and the Model are dependencies that you can mock using the `@Mock` annotation.

Continue by adding the following code to `MainPresenterTests`:

```
@Mock
private lateinit var mockActivity : MainContract.ViewInterface

@Mock
private lateinit var mockDataSource : LocalDataSource

lateinit var mainPresenter : MainPresenter
```

You'll link these components together in a `setUp()` method that runs before every test, as denoted by the `@Before` annotation:

```
@Before
fun setUp() {
  mainPresenter = MainPresenter(viewInterface = mockActivity, dataSource
= mockDataSource)
}
```

Here, the `MainPresenter` is instantiated, passing in the mock objects for the View and Model.

Now that the basic test infrastructure is ready to go, it's time to start writing some tests.

Testing movie retrieval

The first tests you'll write for the `MainPresenter` verifies the `getMyMoviesList()` method. Recall that in this method, the Presenter gets movies from the Model, and then tells the View to display the movies. To facilitate the testing of this method, create a dummy list of movies:

```
private val dummyAllMovies: ArrayList<Movie>
get() {
  val dummyMovieList = ArrayList<Movie>()
  dummyMovieList.add(Movie("Title1", "ReleaseDate1", "PosterPath1"))
  dummyMovieList.add(Movie("Title2", "ReleaseDate2", "PosterPath2"))
  dummyMovieList.add(Movie("Title3", "ReleaseDate3", "PosterPath3"))
  dummyMovieList.add(Movie("Title4", "ReleaseDate4", "PosterPath4"))
  return dummyMovieList
}
```

Next, add this test for getting a non-empty list of movies:

```
@Test
fun testGetMyMoviesList() {
  //1
  val myDummyMovies = dummyAllMovies

Mockito.doReturn(Observable.just(myDummyMovies)).`when`(mockDataSource).a
llMovies

  //2
  mainPresenter.getMyMoviesList()

  //3
  Mockito.verify(mockDataSource).allMovies
  Mockito.verify(mockActivity).displayMovies(myDummyMovies)
}
```

Reviewing the code step-by-step:

1. Set up the test by stubbing the method `mockDataSource.allMovies` to return the `dummyAllMovies` list of movies you created, instead of the default data source behavior that would hit the database.
2. Invoke the `getMyMoviesList()` method that you're testing.
3. Verify that the Presenter calls on the Model to get the movies and calls on the View to display the movies.

Run the test by right-clicking the **MainPresenterTests.kt** tab and clicking **Run "MainPresenterTests"**. Confirm that it passes. If so, you just wrote your first passing test — but don't spend too much time basking in the green glory, you have more tests to write.

To continue, add this test for getting an empty list of movies:

```
@Test
fun testGetMyMoviesListWithNoMovies() {
  //1

Mockito.doReturn(Observable.just(ArrayList<Movie>())).`when`(mockDataSour
ce).allMovies

  //2
  mainPresenter.getMyMoviesList()

  //3
  Mockito.verify(mockDataSource).allMovies
  Mockito.verify(mockActivity).displayNoMovies()
}
```

Taking this code in turn:

1. Set up the test by stubbing the method `mockDataSource.allMovies` to return the empty list of movies, again, to override the default data source behavior of accessing the database.
2. Invoke the `getMyMoviesList()` method that is under test.
3. Verify that the Presenter calls on the Model to get the movies and calls on the View handle the displaying of no movies.

Run the tests again, and make sure you still see green.

Congratulations! You can now feel confident that your activity will call use the model to retrieve movies, and will pass the results on to the view.

Testing deleting movies

Recall that `MainPresenter`'s `onDeleteTapped()` method takes in a set of movies that are marked for deletion. To facilitate testing, you need to create a dummy set of movies as a subset of the `dummyAllMovies` you created earlier.

Add the following code to `MainPresenterTests`, just below the other class properties:

```
private val deletedHashSetSingle: HashSet<Movie>
  get() {
    val deletedHashSet = HashSet<Movie>()
    deletedHashSet.add(dummyAllMovies[2])

    return deletedHashSet
  }

private val deletedHashSetMultiple: HashSet<Movie>
  get() {
```

```
    val deletedHashSet = HashSet<Movie>()
    deletedHashSet.add(dummyAllMovies[1])
    deletedHashSet.add(dummyAllMovies[3])

    return deletedHashSet
  }
```

You'll use these static lists in the tests you're about to write.

In your tests, you'll verify that the Presenter calls on the Model to delete the right movies and then calls on the View to display a message upon finishing the deletion.

Add the following code for testing deleting a single movie:

```
@Test
fun testDeleteSingle() {

  //1
  val myDeletedHashSet = deletedHashSetSingle
  mainPresenter.onDeleteTapped(myDeletedHashSet)

  //2
  for (movie in myDeletedHashSet) {
    Mockito.verify(mockDataSource).delete(movie)
  }

  //3
  Mockito.verify(mockActivity).showToast("Movie deleted")
}
```

Here's the code breakdown:

1. Invoke `onDeleteTapped`, passing in the HashSet of movies for the Presenter to delete.
2. Iterate through the set of movies and verify that they've been deleted by checking that the appropriate calls were executed against the mock data source.
3. Verify that the correct Toast message is displayed for a single movie deleted.

Run this test and confirm it passes. Then, add the test for deleting multiple movies, which looks similar:

```
@Test
fun testDeleteMultiple() {

  //Invoke
  val myDeletedHashSet = deletedHashSetMultiple
  mainPresenter.onDeleteTapped(myDeletedHashSet)

  //Assert
  for (movie in myDeletedHashSet) {
    Mockito.verify(mockDataSource).delete(movie)
```

```
    }

    Mockito.verify(mockActivity).showToast("Movies deleted")
  }
```

Run the tests again and make sure everything passes before moving on to the next section.

Testing the AddMoviePresenter

Next, you'll write tests for `AddMoviePresenter`. Because **AddMoviePresenter.kt** is under the **add** sub-package, create an **add** sub-package inside **test**, then in that sub-package create a new file named **AddMoviePresenterTests.kt**. Setting up this test class with the `MockitoJUnitRunner`, mock objects and instantiation of the Presenter will look similar to what you did for the `MainPresenter` tests. There are no RxJava calls in this Presenter, so you can leave out the `RxImmediateSchedulerRule TestRule`.

Start by adding the following code to **AddMoviePresenterTests.kt**:

```
//1
@RunWith(MockitoJUnitRunner::class)
class AddMoviePresenterTests : BaseTest() {

  //2
  @Mock
  private lateinit var mockActivity : AddMovieContract.ViewInterface

  @Mock
  private lateinit var mockDataSource : LocalDataSource

  lateinit var addMoviePresenter : AddMoviePresenter

  @Before
  fun setUp() {
    //3
    addMoviePresenter = AddMoviePresenter(viewInterface = mockActivity,
dataSource = mockDataSource)
  }
}
```

Walking through the code, you:

1. Annotate the test class to run with `MockitoJUnitRunner` to specify that the test should use the mock test runner as opposed to the standard JUnit runner.
2. Initialize the class properties, including mock objects for the View and Model.
3. Inject the mock View and mock Model into the Presenter's constructor when instantiating the `addMoviePresenter`.

Testing adding movies

Recall that at a minimum, the user must enter a movie title to add a movie to their to-watch list. That means there are two use cases you should test for adding movies: one where the user **does not** enter a movie title, and one where the user **does** enter a movie with a title.

In **AddMoviePresenterTests.kt**, add the following for the first test:

```
@Test
fun testAddMovieNoTitle() {
  //1
  addMoviePresenter.addMovie("", "", "")

  //2
  Mockito.verify(mockActivity).displayError("Movie title cannot be
empty")
}
```

Taking the code in turn, you:

1. Invoke the Presenter's `addMovie()` method, passing in empty strings for the movie's title, release date and poster path.
2. Verify that the View will display an error in this situation.

Run this test and confirm that it passes.

Now, add the second test:

```
//1
@Captor
private lateinit var movieArgumentCaptor: ArgumentCaptor<Movie>

@Test
fun testAddMovieWithTitle() {

  //2
  addMoviePresenter.addMovie("The Lion King", "1994-05-07", "/
bKPtXn9n4M4s8vvZrbw40mYsefB.jpg")

  //3
  Mockito.verify(mockDataSource).insert(captureArg(movieArgumentCaptor))
  //4
  assertEquals("The Lion King", movieArgumentCaptor.value.title)

  //5
  Mockito.verify(mockActivity).returnToMain()
}
```

This is similar to the previous test, with some extras. Here's the breakdown:

1. Create an ArgumentCaptor that captures the movie object passed into the Model's `insert()` method.
2. Invoke the Presenter's `addMovie()` method, passing in proper values for a movie's title, release date and poster path.
3. Verify that the Presenter asks the Model to insert the movie into the database. Meanwhile, you use `BaseTest`'s custom `captureArg()` method to capture the movie object that was passed into the Model.
4. Verify using the ArgumentCaptor the nature of the movie instance that was passed into the Model's `insert()` method. In this case, you confirm that the title matches what the Presenter received.
5. Verify that the Presenter asks the View to navigate back to the main screen afterward.

Run this test and confirm that it passes.

Testing the SearchPresenter

You'll wrap up this chapter by creating some tests for `SearchPresenter`.

Create a new sub-package named **search** inside **test**. Then, create a file named **SearchPresenterTests.kt** and add the following code:

```
@RunWith(MockitoJUnitRunner::class)
class SearchPresenterTests : BaseTest() {
  @Rule
  @JvmField var testSchedulerRule = RxImmediateSchedulerRule()

  @Mock
  private lateinit var mockActivity : SearchContract.ViewInterface

  @Mock
  private val mockDataSource = RemoteDataSource()

  lateinit var searchPresenter: SearchPresenter

  @Before
  fun setUp() {
    searchPresenter = SearchPresenter(viewInterface = mockActivity,
dataSource = mockDataSource)
  }
}
```

The code is nearly the same as it was for the previous test classes, so a belabored explanation isn't needed here.

To test `SearchPresenter`'s searching functionality, you'll test two use cases: when the TMDB API call successfully returns a list of movies, and when an error occurs with the TMDB response. Rather than making the web call, you'll stub the response returned with a dummy list of movies.

Add the following helper code:

```
private val dummyResponse: TmdbResponse
  get() {
    val dummyMovieList = ArrayList<Movie>()
    dummyMovieList.add(Movie("Title1", "ReleaseDate1", "PosterPath1"))
    dummyMovieList.add(Movie("Title2", "ReleaseDate2", "PosterPath2"))
    dummyMovieList.add(Movie("Title3", "ReleaseDate3", "PosterPath3"))
    dummyMovieList.add(Movie("Title4", "ReleaseDate4", "PosterPath4"))

    return TmdbResponse(1, 4, 5, dummyMovieList)
  }
```

Next, add the test for a successful API call:

```
@Test
fun testSearchMovie() {
  //1
  val myDummyResponse = dummyResponse
Mockito.doReturn(Observable.just(myDummyResponse)).`when`(mockDataSource)
.searchResultsObservable(anyString())

  //2
  searchPresenter.getSearchResults("The Lion King")

  //3
  Mockito.verify(mockActivity).displayResult(myDummyResponse)
}
```

Taking this code step-by-step:

1. Set up the test by having the Model return the `dummyResponse` as its response from the API call.
2. Invoke the Presenter's `getSearchResults`, passing in the search query.
3. Verify that because the Model returned a valid response, the Presenter then asks the View to display the results.

Run this test and confirm that it throws no errors.

Now, test the use case where the API call fails and throws an error by adding the following:

```
@Test
fun testSearchMovieError() {
  //1
  Mockito.doReturn(Observable.error<Throwable>(Throwable("Something went
wrong"))).`when`(mockDataSource).searchResultsObservable(anyString())

  //2
  searchPresenter.getSearchResults("The Lion King")

  //3
  Mockito.verify(mockActivity).displayError("Error fetching Movie Data")
}
```

There are a few new things in this code:

1. Set up the test by having the Model return an error with the message "Something went wrong" instead of a proper response from the API call.
2. Invoke the Presenter's `getSearchResults` method, passing in the search query.
3. Verify that upon receiving this error the Presenter asks the View to display an appropriate error message.

Run this test. It passes, but it throws an error in the output. Don't worry; this is expected behavior. If you read the error, you'll notice it's the error you created with the message: "Something went wrong".

Key points

- The **Model View Presenter** pattern makes it possible to verify the behavior of the Presenter to ensure that it sticks to the contract expected between it and the View and Model
- Use Mockito's `ArgumentCaptors` to test Kotlin code, you must override the `capture()` method with your own custom version — one that can get around Kotlin's `null` safety requirements.
- Create `TestRules` to test code containing RxJava's `AndroidSchedulers`. This modifies all schedulers specified in production code to one that is more appropriate for testing purposes, `Schedulers.trampoline()`.
- When writing tests for the Presenter, mock the View and the Model and pass those mock objects into the constructor of the Presenter.

- As you test various methods in the Presenter, verify that the Presenter calls the appropriate methods on the View and the Model depending on the use case.
- Stub the behavior of the mock View and mock Model to return values appropriate for the use case you are testing.

Where to go from here?

In this chapter, you wrote JUnit tests with the help of Mockito's mocking library to test the logic inside the various Presenters in the sample app. Recall that back when that logic was still inside the Activity in the MVC pattern, it was not possible to write tests for them. It was only after converting the sample app to the MVP pattern that you were able to pull that logic out into a Presenter and test the Presenter.

Using the MVP pattern is only one way to achieve testability in your app. In the next chapter, you'll learn other patterns that allows you to unit test your app. Choosing what pattern to use ultimately comes down to what pattern is the best fit for your particular app, though the end goals are still the same: separation of concerns and unit testability throughout the app.

Chapter 10: Model-View-ViewModel Theory

By Aldo Olivares Dominguez

In this chapter you will learn about a distant relative of MVP — the **MVVM Architecture Pattern**.

First, you will explore how MVVM works at a high level. You will learn about each of its layers and how they communicate between each other. You will also learn how MVVM improves the testability of your apps by providing a clear level of abstraction to your code.

Finally, you will understand the advantages and limitations of MVVM to know when and how to apply it properly.

Ready? Let's get started!

The Model-View-ViewModel pattern

MVVM stands for **Model-View-ViewModel**. MVVM is an architectural pattern whose main purpose is to achieve separation of concerns through a clear distinction between the roles of each of its layers:

- **View** displays the UI and informs the other layers about user actions.
- **ViewModel** exposes information to the View.
- **Model** retrieves information from your datasource and exposes it to the ViewModels.

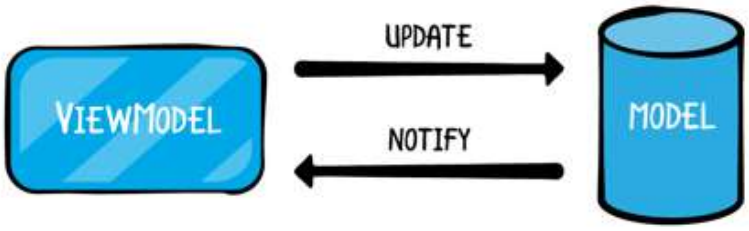
At first glance, MVVM looks a lot like the **MVP** and **MVC** architecture patterns from the last chapters.

The main difference between MVVM and those patterns is that there is a strong emphasis that the ViewModel *should not contain any references to Views*. The ViewModel only provides information and it is not interested in what consumes it. This makes it easy to create a one-to-many relationship wherein your Views can request information from any ViewModel they need.

Also of note regarding the **MVVM** architecture pattern is that the ViewModel is also responsible for **exposing events** that the Views can observe. Those events can be as simple as a new user in your Database or even an update to a whole list of a movie catalog. Now, you will explore how each of the **MVVM** layers work one by one.

The Model

The Model, better known as DataModel, is in charge of exposing relevant data to your ViewModels in a way that is easy to consume. It should also receive any events from the ViewModel that it needs to create, read, update or delete any necessary data from the backend.



In Android, you usually create Models as Kotlin data classes that represent the information that you obtain from your data source, such as an API or a database. For example, say you have an app that displays information about the latest movies. You would surely create a Movie class that contains data such as the title, description, time and release date of the movie.

When following this architecture pattern, you should strive to stick to the single-responsibility principle of software design, creating a Model for each logical object in your domain. This will make it much easier for you to create the necessary ViewModels later on.

Since the Model implementation does not change much from the previous patterns, you won't dive deeper into this layer, here. However, if you want to learn more, review the Models section of the MVC architecture pattern chapter.

The ViewModel

The ViewModel retrieves the necessary information from the Model, applies the necessary operations and exposes any relevant data for the Views.

The Android platform is responsible for managing the lifecycle events of the classes that handle the UI, such as activities and fragments. The operating system can destroy or re-create your activities at any time in response to certain user actions or events.

The problem is that, if Android destroys or re-creates an activity or fragment, all data contained within those components is lost. For example, your app may include a list of movies in one of its activities. If the activity is destroyed and re-created, the list of movies will have to be retrieved again. This may slow down your app if the list is housed in an external database or API.

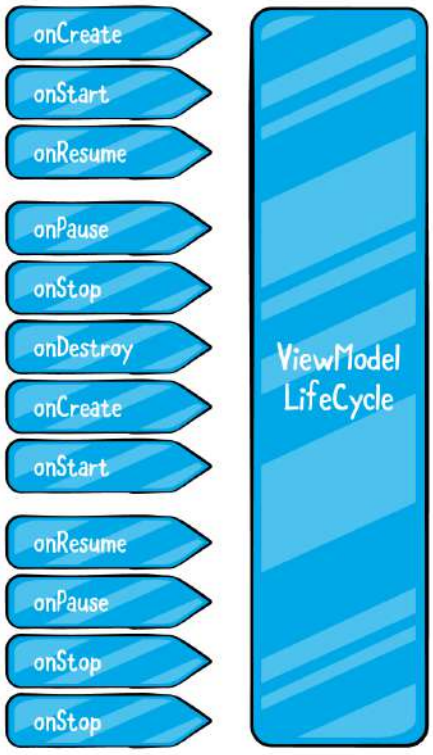
The most common solution to this problem is to save your data in your `onSaveInstanceState()` bundle and restore it later in your `onCreate()` method. But this approach only works for primitive data such as Integers or simple classes that can be serialized and deserialized.

Thanks to Google's new Architecture Components, you now have a special class to build your ViewModels called **ViewModel**.

The **ViewModel** class is specially designed to manage and store information in a lifecycle-aware manner. This means that the data stored inside it can survive configuration/lifecycle changes like screen rotations.

The **ViewModel** remains in memory until the lifecycle object to which it belongs has completely terminated. This behavior applies to activities when they finish and in fragments when they are detached.

In the next illustration, you can see how the **ViewModel** remains active and retains information through the whole lifecycle of an activity, even when it is destroyed:



> **Note**: You don't need to use Android's architecture components to implement your own **ViewModels**. It is just a component that Google provides to make development easier and reliable.

To communicate changes in the data, ViewModels can expose events that the Views can observe and react accordingly. Those events can be as simple as a new user having been created in the database or an update to an entire movies catalog. This way, ViewModels don't need to have any reference to Activities, Fragments or Adapters.

You will learn much more about Google's ViewModel class in the next chapter, so don't worry if something seems confusing at the moment.

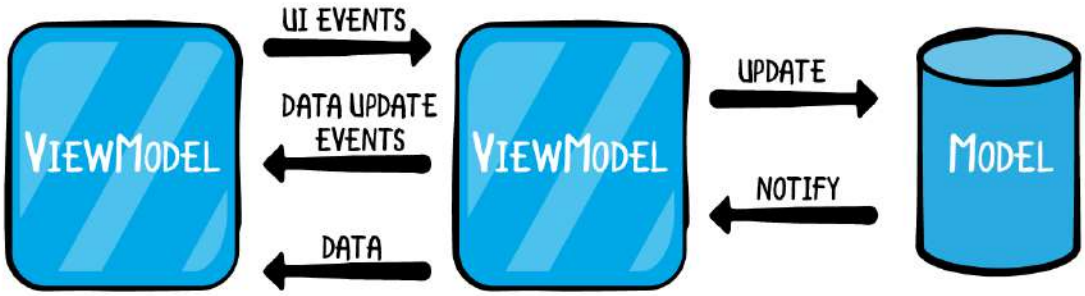
The View

The **View** is what most of us are already familiar with, and it is the only component that the end user really interacts with. The **View** is responsible for displaying the interface, and it is usually represented in Android as **Activities** or **Fragments**. Its main role in the MVVM pattern is to observe one or more ViewModels to obtain the necessary information it needs and update the UI accordingly.

The View also informs ViewModels about user actions. This makes it easy for the View to communicate to more than one Model. Views can have a reference to one or more ViewModels, but ViewModels can never have any information about the Views.

In Android, you will usually communicate the data between the **Views** and the **ViewModels** with **Observables**, using libraries such as **RxJava**, **LiveData** or **DataBinding**.

You can see how the interaction between each layer works, below:



Note: One little trick that will help you know if your Views and your ViewModels are properly detached is to verify that there is no reference to any **com.android.*** package in your ViewModels. There are only a few exceptions to this rule, like the Android Architecture Components package: **com.android.arch.***

MVVM by example

The next two chapters will cover practical examples of MVVM. You will learn how to rewrite the Movies app with two different approaches: Using architecture components and using Data Binding.

To better understand the theory, let's dig into a basic example that shows you how you would connect a View to a ViewModel in a TODO list app.

There's no need to type this code out any where, the code is presented here as an example. Keep reading and we'll concretely break down the pieces that make MVVM.

```
class MainViewModel: ViewModel() {

  //1
  private var items: LiveData<List<Item>>? = null
  //2
  fun getItems(): LiveData<List<Item>> {
    if (items == null) {
      return db.itemDao().getAll()
    }
    return items ?: emptyList()
  }
}

class MainActivity: AppCompatActivity() {

  //3
  private lateinit var mainViewModel: MainViewModel

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    //4
    mainViewModel =
ViewModelProviders.of(this).get(MainViewModel::class.java)

    //5
    recyclerView.layoutManager = LinearLayoutManager(this,
LinearLayoutManager.VERTICAL, false)
    val adapter = ItemAdapter()
    recyclerView.adapter = adapter

    //6
    mainViewModel.getItems().observe(this, Observer {
      if (it != null) {
        adapter.list.clear()
        adapter.list.addAll(it)
        adapter.notifyDataSetChanged()
      }
    })
}
```

> **Note**: The Model code has been omitted for brevity.

Taking each commented section in turn:

1. The **ViewModel** declares a property that will contain a **LiveData** list of items. The **LiveData** class allows any View to observe for any changes on the list and update the UI.
2. **getItems()** is an accessor method that returns the list of TODO items. If the list of items is null, you call the **getAll()** method of your **ItemDao** interface to retrieve them from your database.
3. The **View** holds a reference to your **ViewModel**. The **ViewModel** property is defined as a `lateinit var` so that the compiler knows it won't be initialized until after class initialization.
4. In the `onCreate()` method, you should initialize every reference to the ViewModels you will need. In this case, to the MainViewModel.
5. Next, you configure the recycler view layout and provide an adapter.
6. Finally, you call the `observe()` method of your **LiveData** list of **Items**. If there is any change, you can act accordingly to update the necessary UI elements. In this case, you are updating the list property of the adapter to update your RecyclerView.

As you can see, it's fairly straightforward to implement your ViewModel along with your Views. Once you master them, you will see how they help to make your code easy to test.

MVVM advantages and concerns

One problem that the MVC architecture patterns have in common is that the Controllers and the Presenters are sometimes very hard to test due to their close relationship with the View layer. By handling all data manipulation to ViewModels, unit testing becomes very easy since they don't have any reference to the Views.

One problem present in some architectures, MVC in particular, is that the business logic is quite difficult to test due to a lack of separation from the View logic. By confining all data manipulation to the ViewModel, and by keeping it free of any View code, the business logic becomes unit testable, as it can be executed without requiring the Android runtime.

Another problem with the MVC pattern is that there is usually confusion as to which code goes where. Sometimes, when code doesn't fit in the Model or the View, it is put in the Controller. This often leads to a common problem known as **fat controllers**, whereby the controller classes become overly large and difficult to maintain.

MVVM solves the fat controller issue by providing a better separation of concerns. Adding ViewModels, whose main purpose is to be completely separated from the Views, reduces the risk of having too much code in the other layers.

MVVM vs. MVC vs. MVP

You might be wondering why you would want to use **MVVM** over **MVC** or **MVP**. After all, **MVC** and **MVP** are among the most common Android architecture patterns and are both very easy to understand. There has been endless debate on which approach is best, but the answer largely boils down to personal preference.

As we usually say in the development world, there is no silver bullet to solve every software design issue. And although MVVM is a very useful development pattern, it also has some disadvantages.

The main disadvantage of this architecture pattern is that it can be too complex for applications whose UI is rather simple. Adding as much level of abstraction in such apps can result in boiler plate code that only makes the underlying logic more complicated.

At the end of the day, it is up to each developer to decide which is the best architecture pattern for each development project.

Key points

- MVVM stands for **Model-View-ViewModel**.
- MVVM is an architecture pattern whose main objective is the separation of concerns.
- **Views** display the UI and inform about user actions.
- The **ViewModel** gets the information from your Data Model, applies the necessary operations and exposes the relevant data to your Views.
- The ViewModel exposes backend events to the Views so they can react accordingly.
- The **Model**, also known as the **DataModel**, retrieves information from your backend and makes it available to your ViewModels.

- MVVM facilitates Unit Testing of your code.
- MVVM may be too complex for applications with simple UI.

Where to go from here?

There are several patterns that you could use to build your Android Apps. The Model-View-ViewModel architecture pattern is just one of the many tools that helps you write clear and concise code. But MVVM combines the advantages of the MVP and MVC architecture patterns with other useful features such as DataBinding. It improves the testability of your code by providing a greater level of abstraction and reducing the amount of boiler plate code in your projects.

In the next chapter, you will apply your knowledge by re-writing the Movies app using MVVM.

Chapter 11: MVVM Sample with data binding

By Aldo Olivares

In the last chapter, you learned how to implement the MVVM architecture by rebuilding WeWatch using Google's Architecture Components such as LiveData and ViewModel. In this chapter, you'll learn how to further improve your app's architecture by using the **Data Binding** library to decouple the XML layouts from the activities.

Along the way you'll learn:

- How to use data binding with XML layouts.
- How to use data binding with RecyclerView adapters.
- How to implement one-way data binding.
- How to implement two-way data binding.
- How to create observable fields.
- How to use data binding to observe ViewModels.

What is data binding?

Before implementing data binding in your Android projects you first need to understand what data binding can do for you. According to the official documentation developer.android.com/topic/libraries/data-binding:

> The Data Binding Library is a support library that allows you to bind UI components in your layouts to data sources in your app using a declarative format rather than programmatically.

Simply put, data binding lets you display the values of variables or properties inside your XML layouts such as ConstraintLayouts or RecyclerViews.

Typically (and without data binding), when you want to change or display a value inside your XML layout, you first need to get a reference to the View by using `findViewById()`. Once you have that, you can apply your changes:

```
findViewById<TextView>(R.id.text_view).apply {
    text = viewModel.userName
}
```

While this approach isn't bad, it leads to a lot of boilerplate code that can create a high level of coupling between your layouts and your activities or fragments. Developers are usually forced to create many set up methods that get called immediately after the initial creation of their Views. With data binding, you can keep the UI updated by assigning the variables directly into your layout files:

```
<TextView
    android:text="@{viewmodel.userName}" />
```

In this example, the `@{}` syntax lets you display a property from `viewmodel` within the assignment expression. We'll refer to this syntax as the **One-way data binding** syntax. This approach helps you remove boilerplate code from your activities and fragments. And because you can assign default values, it also prevents memory leaks and `NullPointerExceptions`.

In the next few sections, you'll learn how to use data binding in the WeWatch app from the previous chapter.

Getting Started

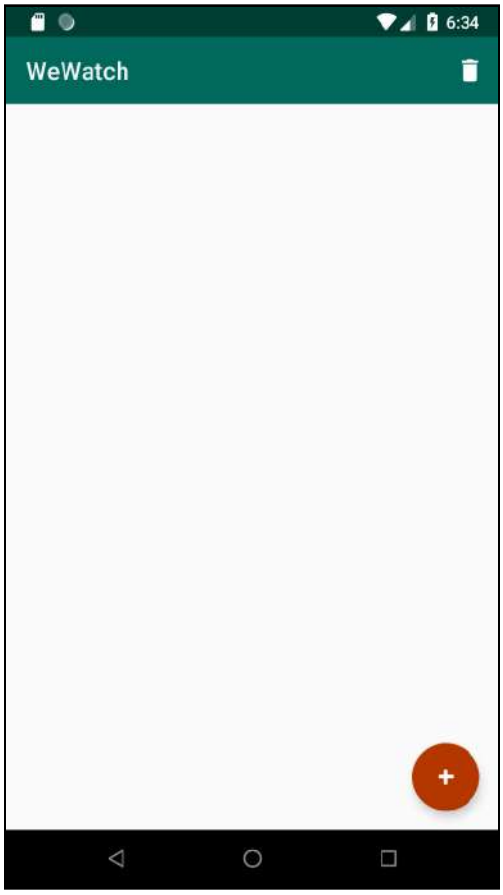
Start by opening the starter project for this chapter. You can also use your own project from the previous chapter.

If you haven't done so already, take some time to familiarize yourself with the code, paying special attention to the classes inside **viewmodel** and **view**.

> **Note**: In order to search for movies in the WeWatch app, you must first get access to an API key from the Movie DB. To get your API own key, sign up for an account at www.themoviedb.org. Then, navigate to your account settings on the website, view your settings for the API, and register for a developer API key. After receiving your API key, open the starter project for this chapter and navigate to

RetrofitClient.kt. There, you can replace the existing value for `API_KEY` with your own.

Build and run the app to see it in action.



Great! All is well. Now it's time to implement data binding!

Implementing data binding

By default, data binding is not enabled. To use the Data Binding library, open **build.gradle** and add the following lines inside the Android block:

```
dataBinding {
  enabled = true
}
```

Click **Sync Now**, and wait until Android Studio finishes syncing your project.

> **Note**: Because the Data Binding library is a relatively new library, you should download the latest Android Plugin for Gradle and make sure you're using Android Studio 3.3 or higher.

In WeWatch, there are three activities that use different XML layouts which can take advantage of data binding:

- **MainActivity**: Consists of **activity_main.xml**, which contains a `RecyclerView` that displays a list of your favorite movies retrieved from the Room database.
- **AddMovieActivity**: Consists of **activity_add.xml**, which contains a `ConstraintLayout` that communicates with `AddViewModel`.
- **SearchMovieActivity**: Similar to `MainActivity`, it contains a `RecyclerView` that displays a list of movies retrieved from the TMDB API.

In this chapter, you'll implement both one-way data binding and two-way data binding. You'll start with `MainActivity`, which is the perfect place to learn about one-way data binding. You'll then move on to two-way data binding in `AddMovieActivity`. Once you're done with that, you'll add data binding to `SearchMovieActivity` during the challenge as it's quite similar to `MainActivity`.

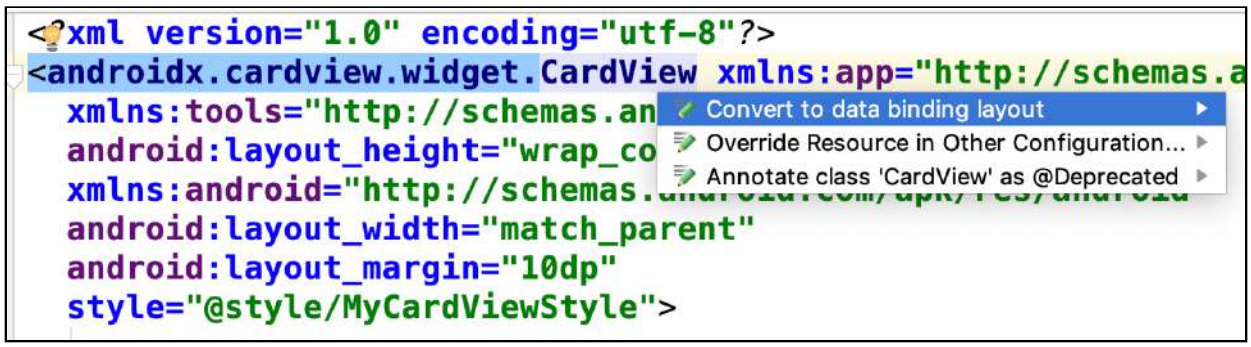
Adding data binding to MainActivity

There are four steps to implement data binding in your Views:

1. Convert your regular layouts into data binding layouts.
2. Add a `data` tag with variables bound to your data source.
3. Use binding expressions to handle events emitted by your Views.
4. Bind your data source to your XML layouts.

Because `MainActivity` contains a `RecyclerView`, you need to work directly with the layout that your `RecyclerView` is using to display a movie item; in this case, **item_movie_main.xml**.

Open **item_movie_main.xml** and select the root `CardView`. Click **Alt-Enter** and select **Convert to data binding layout**:



After selecting the option, your layout should look as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android">

    <data>

    </data>

    <androidx.cardview.widget.CardView
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:layout_margin="10dp"
        style="@style/MyCardViewStyle">
```

Effectively, Android Studio wrapped the root `CarView` element in a new `layout` element and added a `data` element (which you'll learn more about in the next section). This completes Step 1.

Data binding layouts start with a root tag of `layout`, followed by a `data` element. For the `data` element, you need to specify your own data source, which is usually a `ViewModel`, but can sometimes be a `Model`. The data source for this layout is the `movie` object returned by `MovieListAdapter`.

To add it to your layout, add the following `variable` tag inside the `data` element:

```
<data>
    <variable
      name="movie"
      type="com.raywenderlich.wewatch.data.model.Movie"/>
</data>
```

The **name** property of the variable element is how you'll call the data source inside the layout. The **type** property indicates the class that will be used; in this case, `Movie`. You can now use the expression language to display the data using the object inside the layout using the `@{}` syntax.

Scroll down to the `TextView` with an ID of `movieTitleTextView`. Replace the text assignment with the following:

```
android:text="@{movie.title}"
```

This displays the `title` of the `movie` object that was passed in as a parameter.

Scroll down to `movieReleaseDateTextView` and add the following text assignment:

```
android:text="@{movie.releaseDate}"
```

This displays the `releaseDate` of the `movie` object that was passed as a parameter.

The first three steps are complete, so it's on to the fourth: binding the data source to this layout.

You typically want to bind your data source inside the Activity or Fragment that controls the layout, which in this case is `MainActivity`. However, since you're working with a `Recyclerview`, the actual class that controls the list of movie items shown is `MovieListAdapter`.

Inside **view/adapter**, open **MovieListAdapter.kt**. Remove the body inside the `MovieHolder` inner class. and modify the constructor parameters to match this:

```
inner class MovieHolder(val binding: ItemMovieMainBinding) :
RecyclerView.ViewHolder(binding.root)
```

`MovieHolder` is now expecting an `ItemMovieMainBinding` as a constructor parameter.

Wait a minute, you didn't create an `ItemMovieMainBinding` class!

That's what's great about the Data Binding library — it automatically generates the classes required to bind your layouts with your data objects. Here's how it works:

- A single class is generated for each layout file.
- Using the name of the layout file, the library names this new class using Pascal Case and the "Binding" suffix.

For example, if your layout's name is **item_movie_main.xml**, the corresponding class it generates is `ItemMovieMainBinding`.

You're almost ready to use the generated class to manage the information that's displayed in your `RecyclerView`. But first, you need to bind it to the View.

Modify `onCreateViewHolder()` to match this:

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
MovieHolder {
  val layoutInflater = LayoutInflater.from(parent.context)
  val binding =
DataBindingUtil.inflate<ItemMovieMainBinding>(layoutInflater,
R.layout.item_movie_main, parent, false)
  return MovieHolder(binding)
}
```

Here, instead of calling the traditional `LayoutInflater`, you're using `DatabindingUtil`, which allows you to inflate your View and bind `ItemMovieObject` to it by calling `inflate()`.

> **Note**: You should create your binding objects as soon as your layouts are inflated. With an Activity, this means adding some code to `onCreate()`; With an Adapter, you need to add code to `onCreateViewHolder()`.

Now that `ItemMovieMainBinding` is bound to the layout, you only need to tell the object which movie to display using `onBindViewHolder()`.

Modify `onBindViewHolder()` method to match this:

```
//1
override fun onBindViewHolder(holder: MovieHolder, position: Int) {
  //2
  val movie = movies[position]
  //3
  holder.binding.movie = movie
  //4
  holder.binding.checkbox.setOnCheckedChangeListener{ checkbox, isChecked
->
    if (!selectedMovies.contains(movie) && isChecked) {
      selectedMovies.add(movies[position])
    }else{
      selectedMovies.remove(movies[position])
    }
  }
  //5
  holder.binding.checkbox.isChecked = selectedMovies.contains(movie)
}
```

Here's what's happening:

1. Similar to any regular `onBindHolder()`, this method takes `holder` and the current `position` in the list.

2. You get a movie object from the list of movies that should be displayed.
3. Here's where the magic happens. Remember the `movie` variable that you created inside **item_movie_main.xml**? This line tells the layout which movie object is going to be bound to your layout and therefore in all your assignment expressions, such as: `android:text="@{movie.title}"`.
4. This method sets the listener needed to add the watched movies to your list in case they need to be deleted.
5. Finally, this line assigns the correct state to your checkbox to either true or false.

The only step left is to display the appropriate thumbnail in the list using `BindingAdapter`.

Inside the **root** package, open **Extensions.kt** and add the following methods:

```
@BindingAdapter("imageUrl")
fun ImageView.setImageUrl(url: String?) {
  Picasso.get().load(url).into(this)
}

@BindingAdapter("imageUrl")
fun ImageView.setImageUrl(int: Int) {
  this.setImageDrawable(resources.getDrawable(int,null))
}
```

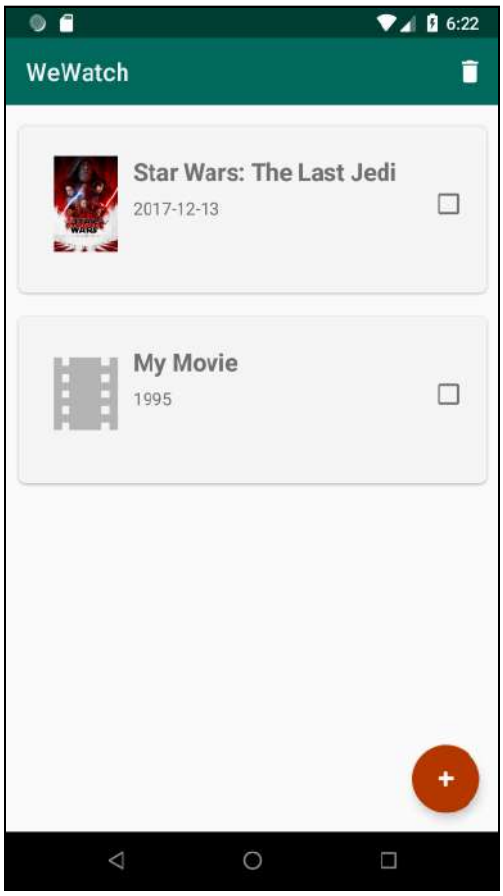
`BindingAdapter` objects are useful for changing the way the traditional bindings behave because they override the traditional adapters provided by the Android Framework. In this case, you're using the power of Kotlin's extension functions to call `setImageUrl()` on any `ImageView` to load a specific image using Picasso. The first method loads an image from a URL and the second from a drawable resource.

Now, in **MovieListAdapter.kt**, you need to call `setImageUrl()`, which you've just extended inside `onBindViewHolder()`:

```
if (movie.posterPath != null) {
  holder.binding.movieImageView.setImageUrl(
        RetrofitClient.TMDB_IMAGEURL + movie.posterPath)
} else {
  holder.binding.movieImageView.setImageUrl(
        R.drawable.ic_local_movies_gray)
}
```

That's it! `MainActivity` is ready for use.

Build and run the app to verify that `MainActivity` is still working properly.



Adding data binding to AddMovieActivity

To implement data binding in `AddMovieActivity`, you need to follow similar steps, with one key difference: You'll use two-way data binding.

One-way data binding lets you set a value in one of your layout's attributes, but you can also react to a change in that attribute by setting a listener. In the snippet below, you can see that `onCheckedChanged` attribute is set with a callback to watch for changes to the `Checkbox`:

```
<CheckBox
    android:id="@+id/checkbox"
    android:checked="@{viewmodel.watched}"
    android:onCheckedChanged="@{viewmodel.watchedChanged}"
/>
```

Similarly, **Two-way data binding**, lets you consolidate setting those two attributes by allowing you to set values and react to changes at the same time:

```
<CheckBox
    android:id="@+id/checkbox"
    android:checked="@={viewmodel.watched}"
/>
```

Using `@={}` syntax, there's no need to override `onCheckedChanged()` anymore. We'll refer to this syntax as the **Two-way data binding** syntax. It's worth noting the tiny little = that separates this syntax from the one way syntax.

You'll use two-way data binding to get and set the movie title and movie release date of the `AddViewModel` using the values entered by the user into the text fields.

Open **activity_add.xml** and convert the layout into a data binding layout by selecting `ConstraintLayout`, and then clicking **Alt-Enter**. When prompted, select **Convert to data binding layout**.

Add the following `variable` inside the `data` tag:

```
<variable
  name="viewModel"
  type="com.raywenderlich.wewatch.viewmodel.AddViewModel"/>
```

In this case, you create a `viewModel` variable that's bound to `AddViewModel`.

Next, you need to create the appropriate properties in `AddViewModel` that will get updated whatever the user types into the `EditText` fields. To create these properties, you have two options:

- Wrap `AddViewModel` properties in `ObservableField`.
- Make `AddViewModel` extend from `BaseObservable`, which can handle all of the properties at once.

Because `AddViewModel` already extends the architecture component `ViewModel`, you cannot also extend from `BaseObservable`. Instead, you'll use an `ObservableField` on any properties that need updating using data binding.

Inside **viewmodel**, open **AddViewModel.kt** and add the following properties:

```
var title = ObservableField<String>("")
var releaseDate = ObservableField<String>("")
```

This creates two `ObservableField` properties, both of type `String`.

Finally, delete the old `saveMovie()` and replace it with the following code:

```
//1
private val saveLiveData = MutableLiveData<Boolean>()
//2
fun getSaveLiveData(): LiveData<Boolean> = saveLiveData
//3
fun saveMovie() {
  if (canSaveMovie()) {
    repository.saveMovie(Movie(title = title.get(), releaseDate =
releaseDate.get()))
    saveLiveData.postValue(true)
  } else {
    saveLiveData.postValue(false)
  }
}
//4
fun canSaveMovie(): Boolean {
  val title = this.title.get()
  title?.let {
    return title.isNotEmpty()
  }
  return false
}
```

Here's what's going on:

1. `saveLiveData` is the property you'll use to signal your Activity if a movie has been saved or not.
2. `getSaveLiveData()` returns a `LiveData` boolean that's updated with the values in `saveLiveData`.
3. `saveMovie()` uses `saveMovie()` of your repository to save a movie in the database using the `title` and `releaseDate` that the user inserts. If the movie successfully saves, `saveLiveData` is `true`, otherwise it's `false`.
4. Since `title` and `releaseDate` are automatically updated with whatever the user types into the `EditText` fields, it's now the responsibility of `viewModel` (and not the Activity) to verify that the title is not empty.

With the properties ready, you can now use them inside the layout.

Open **activity_add.xml**, scroll down to the `EditText` element with an ID of `titleEditText`, and add the following property:

```
android:text="@={viewModel.title}"
```

Now, scroll down to the second `EditText` element, identified with `yearEditText`, and add the following property:

```
android:text="@={viewModel.releaseDate}"
```

It's essential to use two-way binding syntax (`@={}`) whenever you need two-way data binding as it keeps your `Observable` fields up-to-date. If you use the one-way syntax (`@{}`), the `title` and `releaseDate` properties in `AddViewModel` won't get updated.

Finally, scroll down to `addMovieButton` and replace the `android:onClick` attribute with the following:

```
android:onClick="@{()-> viewModel.saveMovie()}"
```

Not only is data binding used to display or update properties, you can also use it to call methods in your data source. In this case, when the user taps the button, you call `saveMovie()` of `viewModel`.

You're nearly done; you just have to create a binding object once the layout is inflated. For that, you need to modify `AddMovieActivity`.

Open **AddMovieActivity.kt** and modify `onCreate()`:

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  //1
  val binding = DataBindingUtil.setContentView<ActivityAddBinding>(this,
R.layout.activity_add)
  //2
  viewModel = ViewModelProviders.of(this).get(AddViewModel::class.java)
  //3
  binding.viewModel = viewModel
}
```

Here's what the code does:

1. Similar to what you did for `MovieListAdapter`, you use `DatabindingUtil` to inflate the layout and get a reference to `binding`.
2. This creates a reference to `AddViewModel`.
3. This assigns `viewModel` to `binding`.

Since you now call `saveMovie()` of `viewModel` directly from the layout (thanks to data binding!), you can now delete `addMovieClicked()` from the Activity.

There's one last piece of plumbing left. You need to add an observer to the ViewModel when a movie gets saved. Add the following method:

```
private fun configureLiveDataObservers() {
  viewModel.getSaveLiveData().observe(this, Observer { saved ->
    saved?.let {
      if (saved) {
        finish()
      } else {
        showMessage(getString(R.string.enter_title))
      }
    }
  })
}
```

This code sets up an observer on `saveLiveData` of `viewModel` to close the the activity. If the movie hasn't been saved, it displays a message to the user indicating that the title cannot be empty.

Finally, call `configureLiveDataObservers()` as soon as your Activity is created by adding the following code to the end of `onCreate()`:

```
configureLiveDataObservers()
```

That's it!

Build and run your app, and try adding some movies to see it in action:

Challenge

You now know how to use data binding to improve your MVVM architecture. It's time to put that knowledge into practice by refactoring `SearchMovieActivity`.

Your mission, should you choose to accept it: Change the **item_movie_search.xml** layout to use data binding and make `SearchAdapter` add a `Movie` as a data source for the layout as you did for **item_movie_main.xml**.

This challenge uses the steps as the ones you followed in the *Adding data binding to MainActivity* section of this chapter:

1. Convert the regular layouts into data binding layouts.
2. Add a data tag with variables bound to your data source.
3. Use binding expressions to handle events emitted by your Views.
4. Bind your data source to your XML layouts.

If you get stuck, review the challenge folder included with this chapter. But remember… practice makes perfect, so do your best to complete this challenge on your own.

Key points

- Data binding lets you display the values of variables or properties inside XML layouts.
- Data binding is not enabled by default; you need to activate it in the app-level **build.gradle**.
- Two-way data binding lets you set values and react to changes at the same time.
- The two-way binding syntax `@={}`, lets you update the appropriate values in the `ObservableFields`.
- The one-way binding syntax `@{}`, lets you display a certain property from the `viewmodel` in the assignment expression.

Where to go from here?

The Data Binding library works well with other Android Architecture Components such as the `ViewModel`. Also, data binding makes your code easy-to-read and maintain by providing a reliable way to bind your XML Layouts, thus reducing boilerplate.

In this chapter, you only learned the basics of data binding. If you want to learn more, check out these great resources:

- The official Android documentation: https://developer.android.com/topic/libraries/data-binding/.
- The expression language documentation: https://developer.android.com/topic/libraries/data-binding/expressions where you can find very interesting operators.
- This MVVM video course: https://www.raywenderlich.com/8984-mvvm-on-android.

Chapter 12: MVVM Sample with Android Architecture Components

By Aldo Olivares Dominguez

In the previous chapter, you learned how MVVM works by understanding how the Model, View and ViewModel interact with each other, and about their responsibilities and limitations.

In this chapter, you are going to use your newly acquired knowledge to rebuild your Movies app to use MVVM by integrating the **ViewModel** and **LiveData** components from the **Android Architecture Components** or **AAC**.

By the end of this chapter, you will have learned:

- How to migrate an app from the MVC architecture to the MVVM architecture.
- How to integrate the ViewModel with the View layer of your apps.
- How to integrate your Models with your ViewModels.
- How to create a centralized repository for your data sources.
- How to use LiveData to work with asynchronous responses from webservices or APIs.
- And much more!

Getting started

Start by opening the starter project for this chapter. If you haven't done so already, take some time to familiarize yourself with the code.

> **Note**: You may notice that the starter project looks a little different than the one from previous chapters. Don't worry, this is intended to give you a head start for this chapter and we will explore it shortly.

The **data** package has three packages related to the backend of your app:

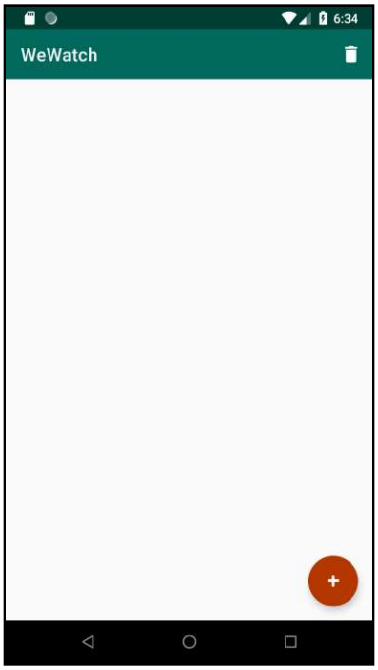
- The **db** package contains the files required for your Room database: the **MovieDatabase.kt** and the **MovieDao.kt** files.
- The **model** package contains the models for your app: the **Movie** model and the **MovieResponse** model.
- The **net** package contains the files required by Retrofit to communicate with the **TMDB** web service: **MoviesAPI.kt** and **RetrofitClient.kt**.

> **Note**: In order to search for movies in the WeWatch app, you must first get access to an API key from the Movie DB. To get your API own key, sign up for an account at www.themoviedb.org. Then, navigate to your account settings on the website, view your settings for the API, and register for a developer API key. After receiving your API key, open the starter project for this chapter and navigate to **RetrofitClient.kt**. There, you can replace the existing value for `API_KEY` with your own.

The **view** package contains three packages related to the front end of your app such as the Activities and Adapters.

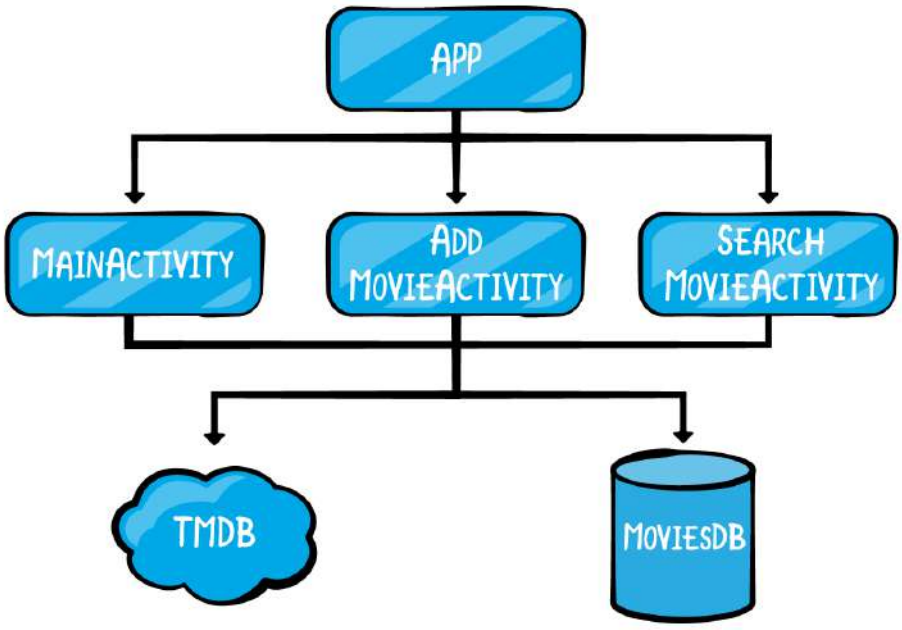
Take all the time you need to familiarize yourself with the project. You will be spending a lot of time on each of the files.

Once you are ready, **build** and **run** the app on a device or emulator to see it in action. You should now see the basic app running.



Current architecture layers

Before making any change to the code of your app, take a quick look at the current architecture, just to refresh:



It's pretty easy to see what's going on: Activities and Fragments communicate with the data store directly. While this architecture is quite easy to understand (and works just fine), there are some disadvantages:

- Your views are individually interacting directly with the TMDB API and with your Room database. There should be a centralized repository to get information from all your backends, including your APIs and your DB.
- Your current structure is violating the single responsibility principle. Views are doing too much work; they should only be in charge of displaying the UI and receiving events from the user.

Your mission, should you choose to accept it, will be to fix each of these flaws with the MVVM architecture pattern by adding ViewModels and LiveData to the mix. At the end, you will compare the old architecture with the new one to see how things have improved.

Creating a movie repository

You will start by creating a centralized repository to retrieve movies for your app, both from the **TMDB API** and from your **Room** database.

Start by opening **build.gradle** in your **app** directory. Add the following line inside the dependencies block:

```
def lifecycle_version = '2.0.0-rc01'
implementation "androidx.lifecycle:lifecycle-extensions:
$lifecycle_version"
implementation "androidx.lifecycle:lifecycle-viewmodel:
$lifecycle_version"
```

The code above adds the lifecycle part of the Android Architecture Components to your dependencies, including `ViewModel` and `LiveData`.

Click the **Sync Now** button that should have appeared at the top of the editor and wait until Android Studio has finished syncing your project.

Under the **data** package, create a new Kotlin class and name it **MovieRepositoryImpl**.

Replace the autogenerated class with the following:

```
class MovieRepositoryImpl : MovieRepository {
  //1
  private val movieDao: MovieDao = db.movieDao()
  private val retrofitClient = RetrofitClient()
  private val allMovies: LiveData<List<Movie>>
```

```
  //2
  init {
    allMovies = movieDao.getAll()
  }
  //3
  override fun deleteMovie(movie: Movie) {
    thread {
      db.movieDao().delete(movie.id)
    }
  }
  //4
  override fun getSavedMovies() = allMovies
  //5
  override fun saveMovie(movie: Movie) {
    thread {
      movieDao.insert(movie)
    }
  }
  //6
  override fun searchMovies(query: String): LiveData<List<Movie>?> {
    val data = MutableLiveData<List<Movie>>()

    retrofitClient.searchMovies(query).enqueue(object :
Callback<MoviesResponse> {
      override fun onFailure(call: Call<MoviesResponse>, t: Throwable) {
        data.value = null
        Log.d(this.javaClass.simpleName, "Failure")
      }

      override fun onResponse(call: Call<MoviesResponse>, response:
Response<MoviesResponse>) {
        data.value = response.body()?.results
        Log.d(this.javaClass.simpleName, "Response: $
{response.body()?.results}")
      }
    })
    return data
  }
}
```

Note: Whenever you add code, make sure to import the appropriate packages by pressing **Alt + Enter** on Windows or **Option + Enter** on Mac.

Taking each commented section, in turn:

1. Here, you create the `movieDao` and `retrofitClient` properties. The `retrofitClient` instance is initialized immediately since it does not require the app context.
2. Next, you initialize the `movieDao` property by creating a `MovieDatabase` instance.
3. `getSavedMovie()` returns a list of all the movies stored in your Room database.

4. `saveMovie()` takes a movie as a parameter and uses the `insert()` method of the `movieDao` to save it in your database.
5. `updateMovie()` takes a movie parameter and uses the `updateMovie()` method of your `movieDao` to update the appropriate record in your database.
6. Finally, `deleteWatchedMovies()` deletes all the movies in your database whose `watched` attribute is equal to `true`.

You may notice that the `saveMovie()` and `updateMovie()` methods are using the `thread()` method from Kotlin's standard library to create a separate thread and execute database tasks. This is needed since database operations can take a long time to run and may block the main thread in which your app is executing. Since UI operations happen on the main thread, bogging it down with non-UI workloads will make the UI stutter and become unresponsive.

> **Note**: In the above code, you use Kotlin's `thread` method to create a separate thread rather than an async task for simplicity. This way, you don't have to make your **MovieRepository** class extend the AsyncTask class and implement different methods.

The last step in creating the movie repository is to make it available to your other classes with a single-entry point.

Open **App.kt** inside the root directory of your app and add the following method just below `onCreate()`:

```
fun getMovieRepository(): MovieRepository = MovieRepository(this)
```

The above method returns an instance of your `MovieRepository`. Since your `MovieDatabase` needs a reference to your application context, the **App** file is the best place to create an accessor method.

And that's it! Now that you have created your movie repository and made it available through your `getMovieRepository()` method, it is time to create your ViewModels.

Creating ViewModels

While it is possible to access the movie repository from your views, it is generally considered bad practice to have your Activities or Fragments communicate directly to your backend.

Creating your movie repository was just the first part of migrating your app. In the **MVVM** architecture, **ViewModels** are in charge of receiving requests from your **Views**, communicating those requests to your **Models** and updating your backend accordingly.

You can create your own **ViewModel** classes from scratch. In fact, this is what developers used to do before Google introduced the Android Architecture Components. But now, you have an easy and consistent way of creating the ViewModels for our Views: **The ViewModel Architecture Component**.

According to the official documentation:

> The **ViewModel** class is designed to store and manage UI-related data in a lifecycle conscious way. The **ViewModel** class allows data to survive configuration changes such as screen rotations.

No way! This is exactly what you need for your app.

Create a new package under the root directory and name it **viewmodel**

Create a new class under the **viewmodel** package and name it **AddViewModel**.

Replace the autogenerated class with the following code:

```
//1
class AddViewModel(private val repository: MovieRepository =
MovieRepositoryImpl()): ViewModel()  {
  //2
  fun saveMovie(movie: Movie) {
    repository.saveMovie(movie)
  }
}
```

Taking each commented section in turn:

1. `AddViewModel` extends from the `ViewModel` class. If you were to take a look at the documentation for `ViewModel`, you'd find that there is no need to override any method to make your data survive configuration changes. Everything is already taken care of for you. The `MovieRepository` is immediately initialized using the constructor.
2. `saveMovie()` uses your movie repository to save the movie passed as a reference to the database.

> **Note**: The only difference between the `ViewModel` and the `AndroidViewModel` class is that the latter depends on your app's context. This is useful when working with

other libraries, such as Room, but it also makes your app harder to test. We will talk more about this in the MVVM Testing chapter.

Now that your ViewModel is ready, it's time to use it.

Open **AddMovieActivity.kt** and add the following attribute to store a reference to an instance of `AddMovieViewModel`:

```
private lateinit var viewModel: AddViewModel
```

Once you have your attribute add the following code inside the `onCreate()` method:

```
viewModel = ViewModelProviders.of(this).get(AddViewModel::class.java)
```

`ViewModelProviders` is a special class that returns an existing `ViewModel` or creates a new one while the scope of a given Activity/Fragment is alive. In this case, since you are passing a reference to your `AddMovieActivity`, it will create a new `AddViewModel` that will stay alive during the whole lifecycle of your `AddMovieActivity` activity.

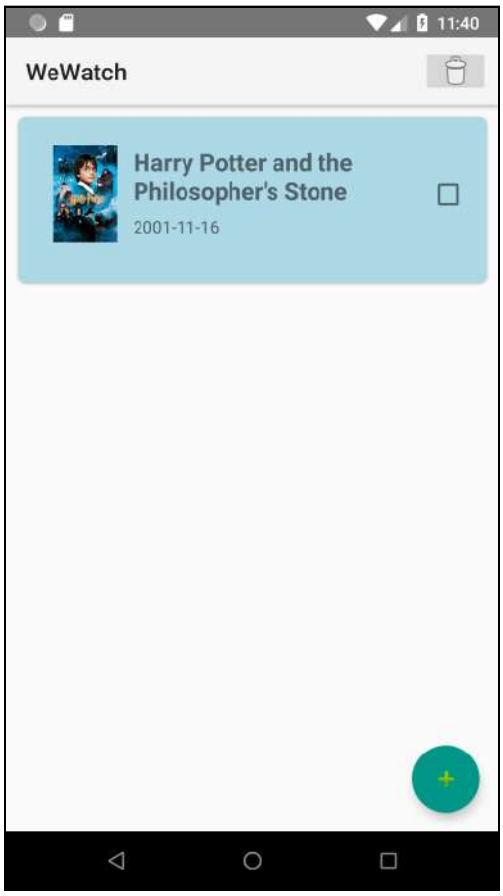
The only step left is to use your **ViewModel** to save a movie when the user presses the **Save Movie** button.

Locate the `addMovieClicked()` method and add the following code:

```
fun addMovieClicked(view: View) {
  if (titleEditText.text.toString().isNotBlank()) {
    viewModel.saveMovie(Movie(
      title = titleEditText.text.toString(),
      releaseDate = yearEditText.text.toString()))
    finish()
  } else {
    showMessage(getString(R.string.enter_title))
  }
}
```

As you can see, there is no need to create a thread inside your Fragment since your `MovieRepository` is already creating one each time `saveMovie()` is called. This helps you avoid code duplication inside your Views.

Build and Run the app. Try adding a movie to verify everything is working properly:



Before creating the next ViewModels, you will need to learn about LiveData.

Using LiveData with ViewModels

In an earlier section, you added a method named `searchMovie()` to your repository which returned a LiveData list of movies, but what is LiveData?

LiveData is a data holder class, just like a List or a HashMap, that can be observed for changes within a given lifecycle. This basically means that you can attach an **Observer** that will be notified about any modification on the wrapped data.

For example, say that you want to retrieve a list of users from your database with a method like the following:

```
fun getUsers(): List<User> {
  return userDao().getAll()
}
```

There are two problems with the above approach. First, this method is passive; it only retrieves the list of all users in the database when it is explicitly called upon to do so. So, if you were to use it to back a list UI, you would have to call it every time you added, inserted or deleted a user.

Second, the method is synchronous; it blocks the calling thread until the database query is finished. if you execute long-running tasks in the UI thread, your app could be stopped by the operating system and the user would get an **Application Not Responding Error** or **ANR**

To solve this problem you could use LiveData to wrap your list of users:

```
fun getUsers(): LiveData<List<User>> {
  return users
}
```

Then, observe for any changes with an **Observer** like below:

```
getUsers().observe(this, Observer { users ->
  //Update UI with list of users
})
```

This approach is much better since the observer will notify any consumers of data changes as they happen, removing the need to respond to those changes manually.

With the above in mind, you will use LiveData's powers to observe your Room database and your Retrofit callbacks.

Create a new class under the **viewmodel** package and name it **MainViewModel**.

Replace the code inside with the following:

```
class MainViewModel(private val repository: MovieRepository =
MovieRepositoryImpl()) : ViewModel() {
  //1
  private val allMovies = MediatorLiveData<List<Movie>>()
  //2
  init {
    getAllMovies()
  }
  //3
  fun getSavedMovies() = allMovies
  //4
  private fun getAllMovies() {
    allMovies.addSource(repository.getSavedMovies()) { movies ->
      allMovies.postValue(movies)
    }
  }
  //5
  fun deleteSavedMovies(movie: Movie) {
    repository.deleteMovie(movie)
```

```
    }
}
```

Step by step:

1. First, you create the `repository` and the `allMovies` properties. You may have noticed that the `allMovies` property is a `MediatorLiveData` type. `MediatorLiveData` is a subclass of `LiveData` that can hold data from different sources. It can also react to `onChanged` events from `LiveData` objects.
2. Next, you call the `getAllMovies()` method as soon as the `MainViewModel` class is initialized.
3. `getSavedMovies()` returns a LiveData list of movies stored in your `allMovies` property.
4. `getAllMovies()` sets the datasource of `allMovies` from `MovieRepository`. It fetches the list of movies by executing `repository.getSavedMovies()` and posting the value to `allMovies`.
5. `deleteSavedMovies()` deletes the movie passed as a parameter using the `deleteMovie()` method of your `MovieRepository`.

Your ViewModel is now ready to be used inside your Activity.

Open **MainActivity.kt** and add the following attribute to hold a reference to your `MainViewModel`:

```
private lateinit var viewModel: MainViewModel
```

Initialize it inside the `onCreate()` method just like you did in `AddMovieViewModel`:

```
viewModel = ViewModelProviders.of(this).get(MainViewModel::class.java)
```

Now that your viewModel has been initialized, it's time to use it.

Add the following code below the line you just added inside `onCreate()`:

```
showLoading()
viewModel.getSavedMovies().observe(this, Observer { movies ->
  hideLoading()
  movies?.let {
    adapter.setMovies(movies)
  }
})
```

The above code uses the `getSavedMovies()` method from your ViewModel to retrieve a LiveData list of movies. The **observe()** method attaches the Observer passed as a

parameter to the observers list of your LiveData object. Once the movies have been retrieved from your database, your observer's callback is executed and the adapter receives the list of movies to be displayed in your recyclerView.

You might also notice that the `observe` method passes your Activity, an instance of `LifecycleOwner`, as the first parameter. By doing so, the observer is bound to the `Lifecycle` object associated. This basically means three things:

1. After the `Lifecycle` object is destroyed, the observer is automatically destroyed.
2. If the `Lifecycle` is inactive, the observer isn't called, even if your list changes.
3. `LiveData` objects, just like your `ViewModel` objects, are lifecycle-aware. You can share data between your Activities, Fragments or even services.

Now, add the following inside `deleteMoviesClicked()`:

```
for (movie in adapter.selectedMovies) {
  viewModel.deleteSavedMovies(movie)
}
```

The MovieList screen of your app is now ready!

Build and run to see it in action:

Creating the SearchViewModel

Before creating your new ViewModel open the **MovieRepository.kt** file and analyze the `searchMovies()` method bit by bit:

```
override fun searchMovies(query: String): LiveData<List<Movie>?> {
  //1
  val data = MutableLiveData<List<Movie>>()
  //2
  retrofitClient.searchMovies(query).enqueue(object :
Callback<MoviesResponse> {
    //3
    override fun onFailure(call: Call<MoviesResponse>, t: Throwable) {
      data.value = null
      Log.d(this.javaClass.simpleName, "Failure")
    }
    //4
    override fun onResponse(call: Call<MoviesResponse>, response:
Response<MoviesResponse>) {
      data.value = response.body()?.results
      Log.d(this.javaClass.simpleName, "Response: $
{response.body()?.results}")
    }
  })
  return data
}
```

Briefly, here's what's going on:

1. First, you create an empty `MutableLiveData` list of movies.
2. Next, you call the `searchMovies()` method of your retrofitClient to retrieve movies that match the given query.
3. The `onFailure()` method is triggered if there is a problem with your call to the TMDB API. Appropriate error handling has been omitted here for simplicity.
4. Once there is a successful response from the TMDB API, your movie list is set by using the `setValue()` method of your `MutableLiveData` class.

You might notice that you are using `MutableLiveData` instead of LiveData. `MutableLiveData` is a `LiveData` subclass that exposes two methods: `setValue()` and `postValue()`:

- `setValue()`: Sets the value of your data from the main thread.
- `postValue()`: Adds a task to the main thread to set the value of your data.

In short: Use the `setValue()` method if you are on the main thread and the `postValue()` method if you are on a background thread.

Now, it's time to create your **ViewModel**.

Create a new class under the **viewmodel** package and name it `SearchViewModel`.

Replace the auto-generated code inside the class with the following:

```
class SearchViewModel(private val repository: MovieRepository =
MovieRepositoryImpl()): ViewModel()  {

  fun searchMovie(query: String): LiveData<List<Movie>?> {
    return repository.searchMovies(query)
  }

  fun saveMovie(movie: Movie) {
    repository.saveMovie(movie)
  }
}
```

The code above creates your `repository` and creates two methods to save and retrieve movies: the `searchMovie()` method and the `saveMovie()` method.

Now that your **ViewModel** is ready, the only thing left is to use it inside your Activity.

Open **SearchMovieActivity.kt** and add a property to store an instance of your `SearchViewModel` class:

```
private lateinit var viewModel: SearchViewModel
```

Initialize your attribute inside `onCreate()`:

```
viewModel = ViewModelProviders.of(this).get(SearchViewModel::class.java)
```

Create a new `searchMovie()` method:

```
private fun searchMovie() {
  showLoading()
  viewModel.searchMovie(title).observe(this, Observer { movies ->
    hideLoading()
    if (movies == null) {
      showMessage()
    } else {
      adapter.setMovies(movies)
    }
  })
}
```

The above uses the `searchMovie()` method of your **ViewModel** to retrieve the list of movies from the TMDB API. Once the list of movies have been retrieved, your observer's callback is executed and the adapter is attached to your **RecyclerView**.

Go to the `displayConfirmation()` method and add the following code inside the snackbar action:

```
viewModel.saveMovie(movie)
```

The code above uses the `saveMovie()` method of your `MovieRepository` to save the movie passed as a parameter and returns to the `MainActivity`.

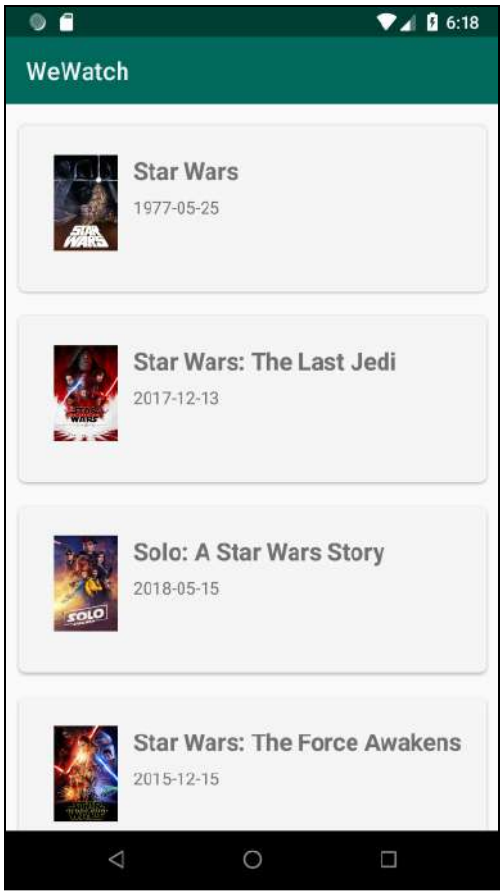
Finally, call your `searchMovie()` method as soon as the `SearchMovieActivity` is created by adding the following line inside `onCreate()`:

```
searchMovie()
```

And also use it inside the `showMessage()` method by adding this code inside your snackbar action:

```
searchMovie()
```

Build and Run your app one last time to test your changes:

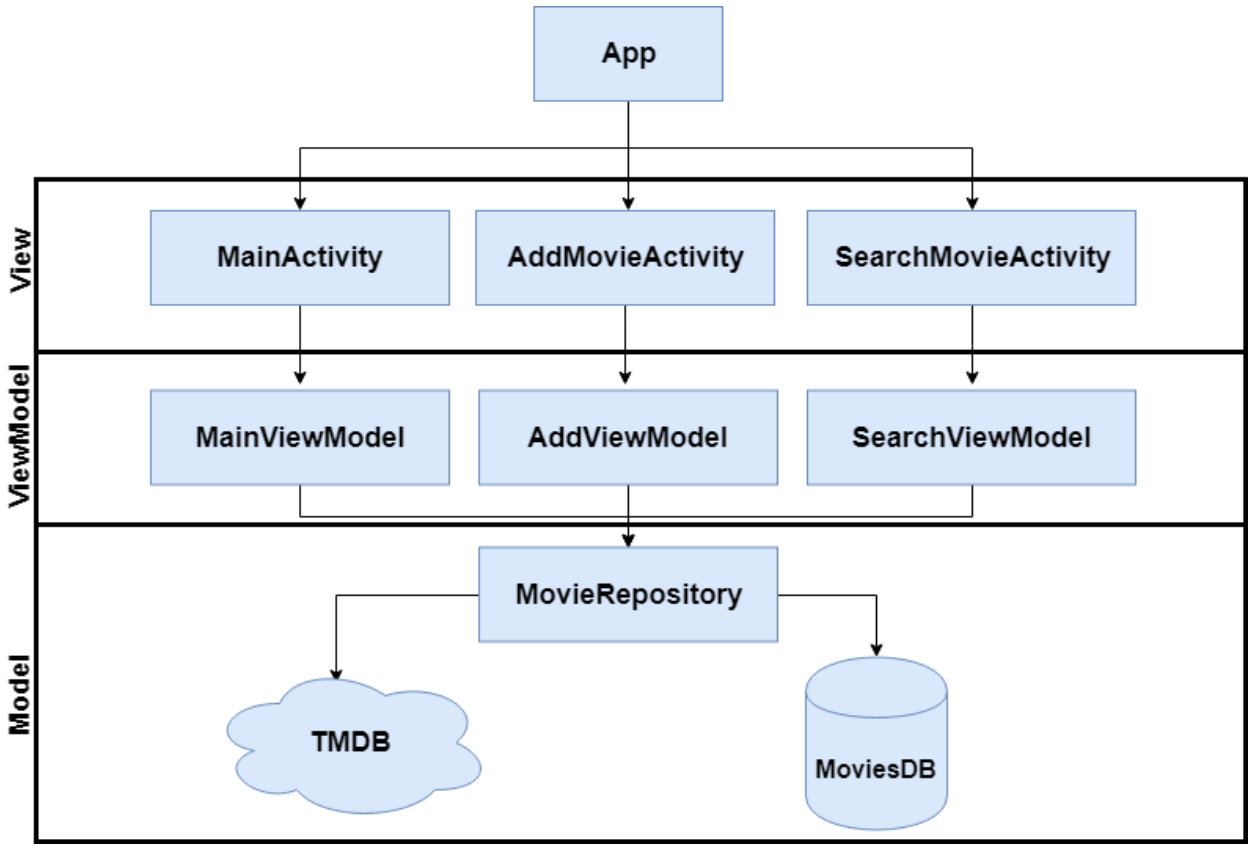


MVVM architecture

At the beginning of this chapter, you saw the current architecture of your app without MVVM.

Your new architecture has the following advantages:

- Your Views only have one job: interacting with the user and displaying the UI.
- Your ViewModels handle all the interaction between your Models and your Views.
- You now have a centralized repository for your two different backend endpoints: your local database and your external API.
- All your classes respect the single responsibility principle.



Key points

- The `ViewModel` class is designed to store and manage UI-related data in a lifecycle-aware way.
- The `ViewModel` class allows data to survive configuration changes, such as screen rotations.
- `LiveData` is a data holder class, just like a List or a HashMap, that can be observed for any changes within a given lifecycle.
- Having a robust architecture like MVVM makes your code scalable and easy to maintain.

Where to go from here?

Although refactoring an app might seem like a daunting task at first, it pays off in the long run. Having a robust architecture like MVVM makes your code scalable and easy to maintain.

In the next chapter, you will learn how to further improve your code by integrating the Data Binding library to bind your UI components in your layouts to your data sources.

Chapter 13: MVVM Testing

By Aldo Olivares

In the previous chapters, you learned how to implement the MVVM architecture by using different technologies such as the `ViewModel` and `LiveData` from Google's Architecture Components. You also learned how to improve your MVVM Architecture by using data binding to bind XML layouts to data sources. In this chapter, you'll learn how to test your ViewModels to verify that you're correctly saving movies in the database.

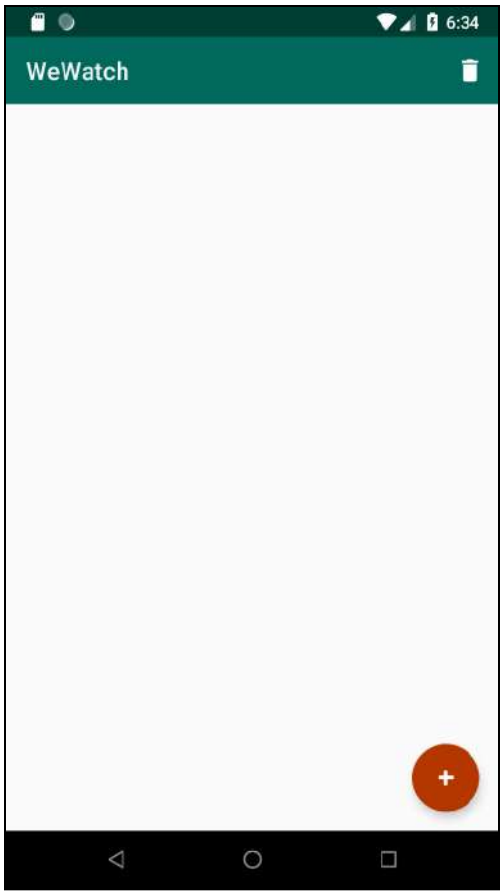
Getting started

Start by opening the starter project for this chapter. This starter project is the same as the final project of the MVVM sample chapter and contains the following packages:

- **data**: Contains the Room database components, including the data access objects, movies database, models and `RetrofitClient`.
- **viewmodel**: Contains the ViewModels.
- **view**: Contains the Activities and Adapters.

> **Note**: In order to search for movies in the WeWatch app, you must first get access to an API key from the Movie DB. To get your API own key, sign up for an account at www.themoviedb.org. Then, navigate to your account settings on the website, view your settings for the API, and register for a developer API key. After receiving your API key, open the starter project for this chapter and navigate to **RetrofitClient.kt**. There, you can replace the existing value for `API_KEY` with your own.

Build and run the app to verify everything is working.



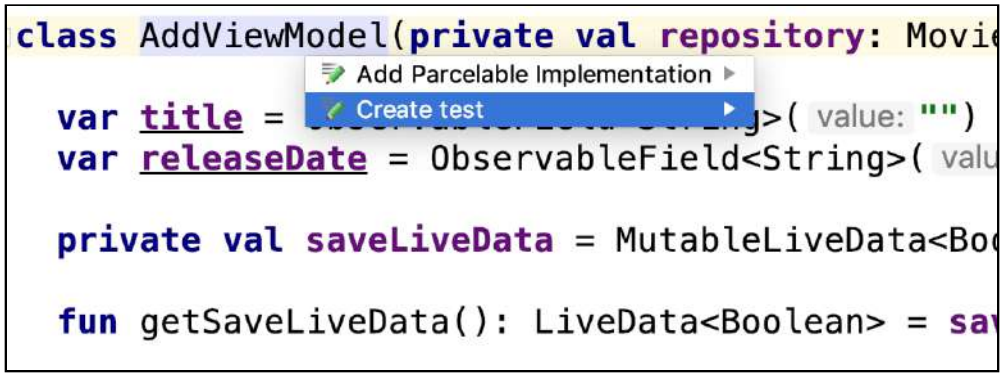
Creating unit tests

WeWatch has three ViewModels:

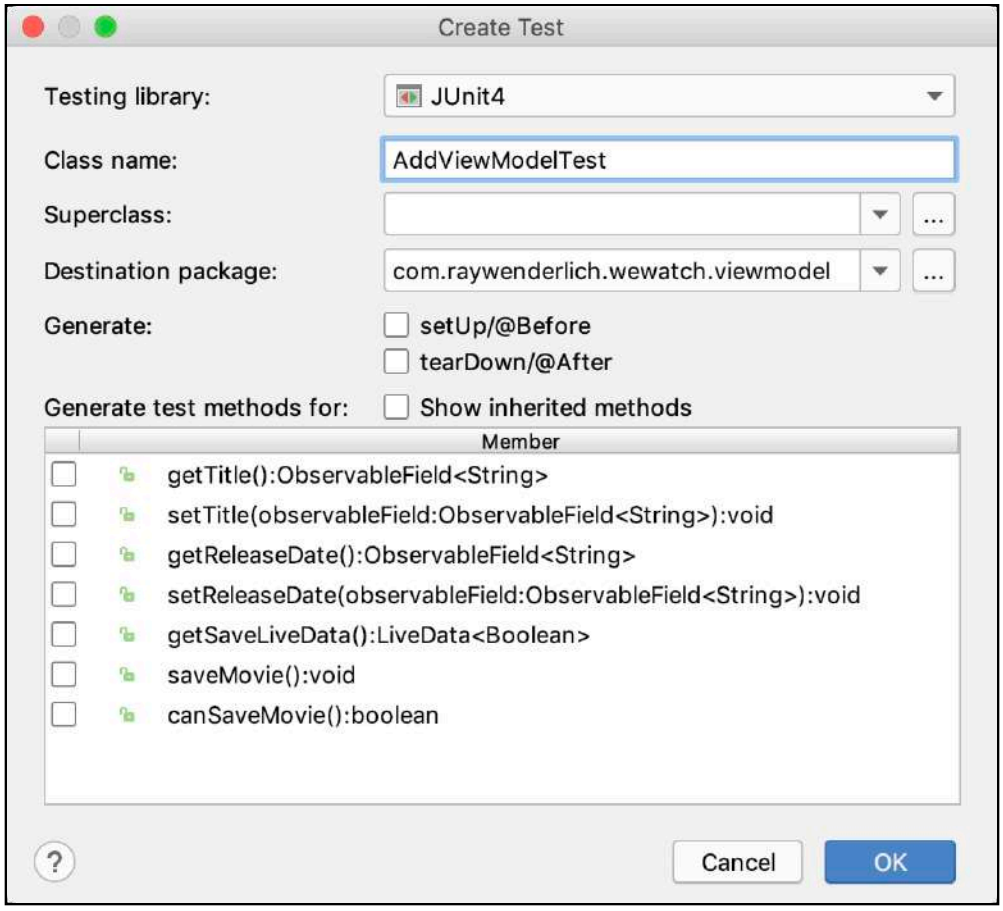
- **`MainViewModel`**: Uses `MovieRepository` to retrieve a list of saved movies that are stored in the Room database.
- **`AddViewModel`**: Uses `MovieRepository` to store a new movie in the database if it meets the appropriate criteria.
- **`SearchViewModel`**: Uses `MovieRepository` to retrieve a list of movies that match the given query parameter from the TMDB API.

Because the only functionality of `MainViewModel` and `SearchViewModel` is to retrieve movies from a different data source, you'll focus on `AddViewModel` to create unit tests.

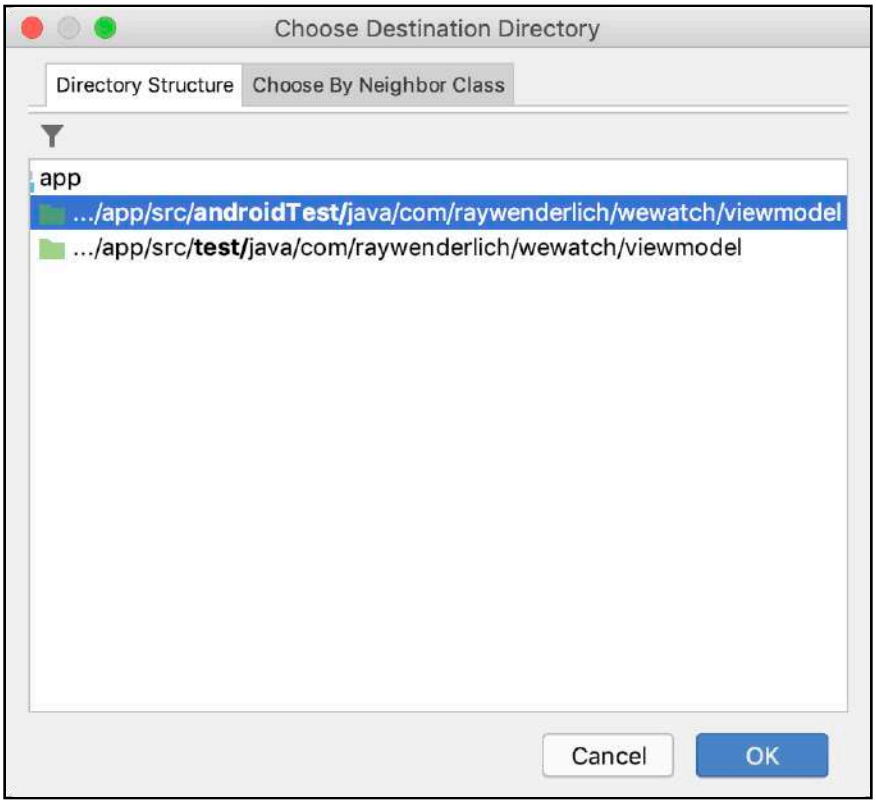
Inside **viewmodel**, open **AddViewModel.kt** and select `AddViewModel`. Then, click **Alt-Enter** and select **Create test**:



On the next screen, use the default values and click **OK**:



You may get another window pops up which looks as follows:



Select the option that includes **test** in the folder path and click **OK**; do not select **androidTest**. This should immediately create a new `AddViewModelTest` class inside the **viewmodel** package of the **test** directory.

Annotate the `AddViewModelTest` to use the `MockitoJUnitRunner`, and a `AddViewModel` property:

```
@RunWith(MockitoJUnitRunner::class)
class AddViewModelTest {
  private lateinit var addViewModel: AddViewModel
}
```

Since you'll be testing `AddViewModel`, you need to create an instance of it. You might remember that when writing unit tests, you need to assume that all of your dependencies are working. Since you're focusing on a single component, you can use some tools to help — this is where mocking frameworks come in handy!

Mocking frameworks allow you to create a dummy implementation of an interface. With this dummy interface, you can easily configure it to produce the expected inputs an outputs.

So, instead of running the code in a dependency, you substitute that dependency with an object you can control: the mock.

Although it's perfectly acceptable to create your own mocks, it's more convenient to use a well-tested framework like Mockito. Mockito is a popular mocking framework for Java and Android that you can use along with JUnit to simplify the testing process of your classes.

Coming back to WeWatch: Because `AddViewModel` needs an instance of `MovieRepository` to save movies, you need to add the following property:

```
@Mock
lateinit var repository: MovieRepository
```

`@Mock` indicates that this is a mock object that will get initialized later.

Now, add the following method:

```
//1
@Before
fun setup() {
  //2
  addViewModel = AddViewModel(repository)
}
```

Here's what's happening:

1. When creating tests, it's common to create methods that initialize your objects and configure your mocks. `@Before` indicates that this method needs to get executed before each test is run.

2. This line initializes `addViewModel` using the mock repository as a constructor parameter.

Now that `viewmodel` and the mocks are initialized, it's time to write some tests!

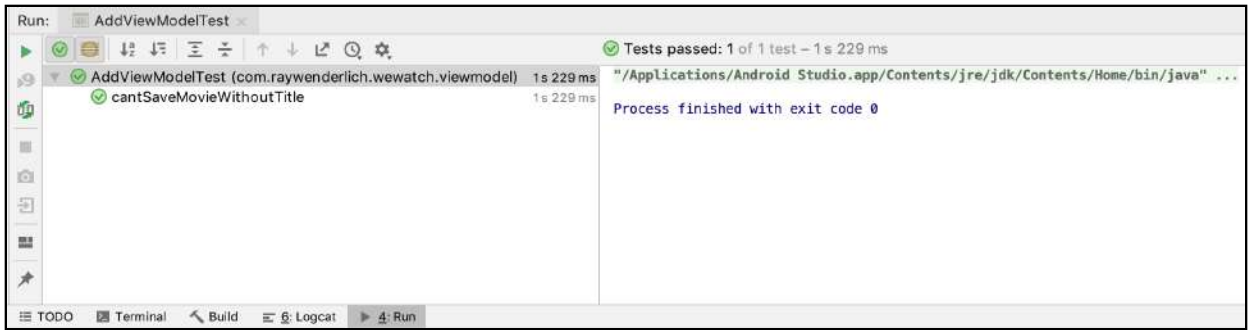
The first thing you need to test is that there are no movies getting saved without a title, so add the following method:

```
//1
@Test
fun cantSaveMovieWithoutTitle() {
  //2
  addViewModel.title.set("")
  addViewModel.releaseDate.set("")
  //3
  val canSaveMovie = addViewModel.canSaveMovie()
  //4
  assertEquals(false, canSaveMovie)
}
```

Here's what's happening:

1. `@Test` informs JUnit that this method needs to get executed as a test. Each time a test runs, JUnit creates a fresh instance of the class and any exceptions thrown are reported as failures.
2. These lines set `title` and `releaseDate` of `addViewModel` to an empty string.
3. This stores the result of `saveMovie()` to `canSaveMovie`.
4. `assertEquals` verifies that the objects passed as parameters are equal. If they aren't, an error is thrown and the test fails. Because `canSaveMovie()` returns `false` when `title` in `AddViewModel` is blank, this call should't throw errors.

Click **Run Test** to the left of `AddViewModelTest` and select **Run 'AddViewModelTest'** to execute the tests:



Hurray! The test passes, which means `canSaveMovie()` is working as expected.

Next, you'll add a feature that verifies whether or not a release date gets set for the movie. Normally, to add an extra feature to an app, you'd write the code for it and then verify that it's working with a passing test. You could do it that way, but there's something called **Test Driven Development** or **TDD**. Here's how it works:

1. Add a new test.
2. Run the test and watch it fail. If it doesn't fail, your job is done.
3. If it fails, you write the necessary code that makes the test pass.
4. Rinse and repeat until the test passes.

Following the TDD approach, you'll implement this new feature by adding a test to verify that a movie can't be saved without a release date and a title.

Add the following method to `AddViewModelTest`:

```
@Test
fun cantSaveMovieWithoutDate() {
  addViewModel.title.set("Awesome Movie I")
  addViewModel.releaseDate.set("")
  val canSaveMovie = addViewModel.canSaveMovie()
  assertEquals(false, canSaveMovie)
}
```

Here, you're supplying a title for the movie with a blank release date. Because `canSaveMovie()` only verifies that `title` isn't blank, it should return `true`, and the test should fail.

Now, execute all of the tests again by clicking **Run Test**, which is located to the left of `AddViewModelTest`. Selecting **Run 'AddViewModelTest'**:

The test fails because now you're expecting `canSaveMovie()` to return `false` when the release date is blank.

Excellent. The first and second steps of the TDD approach are complete: You have a failing test, and you watched it fail. It's time for the third step: Writing the code to make the test pass.

Open **AddViewModel.kt** and modify `canSaveMovie()`:

```
fun canSaveMovie(): Boolean {
  val title = this.title.get()
  val releaseDate = this.releaseDate.get()

  if (title != null && releaseDate != null){
    return title.isNotEmpty() && releaseDate.isNotEmpty()
  }
  return false
}
```

Although the basic logic is still the same, you're now verifying that both `title` and `releaseDate` are populated by using `isNotEmpty()`.

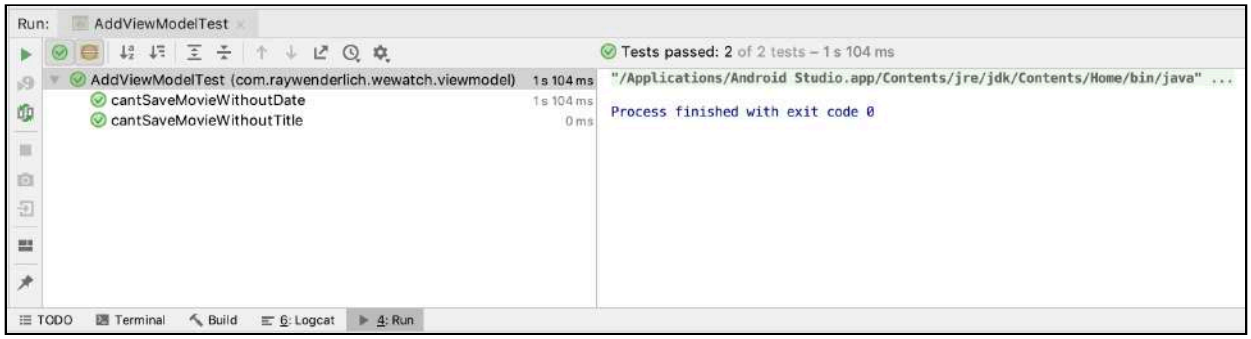
You also need to inform the users that both fields are mandatory. To accomplish this, you can modify the text displayed in the `Snackbar`. Inside **resources/values**, open **strings.xml** and add the following:

```
<string name="title_date_message">You must enter a title and release
date</string>
```

Next, open **AddMovieActivity.kt** and modify `configureLiveDataObservers()` to use the new value:

```
private fun configureLiveDataObservers() {
  viewModel.getSaveLiveData().observe(this, Observer { saved ->
    saved?.let {
      if (saved) {
        finish()
      } else {
        showMessage(getString(R.string.title_date_message))//Only this
line changes
      }
    }
  })
}
```

That's it! Go back to **AddViewModelTest.kt** and execute the test to watch it pass.



Great! You completed all of the necessary steps of TDD to implement a new feature. Build and run the app to see your new feature in action.

Testing LiveData

There's only one method left to test on `AddViewModel`: `saveMovie()`. Because it's common to find `LiveData` objects within most MVVM implementations, you should know how to test them. `saveMovie()` uses `LiveData` objects, so it's perfect for this section.

Add the following method to `AddViewModelTest`:

```
@Test
fun isMovieProperlySaved(){
  addViewModel.title.set("Awesome Movie II")
  addViewModel.releaseDate.set("1994")
  addViewModel.saveMovie()
  assertEquals(true, addViewModel.getSaveLiveData().value)
}
```

This method sets a title and release date for a new movie on `addViewModel`. Because both fields have values, the new movie saves successfully in the database and `getSaveLiveData()` returns `true`, making the test pass.

Run the tests again to see what happens:

Oh no, the test fails!

Take a closer look at the logcat console. Notice the following messages:

```
java.lang.RuntimeException: Method getMainLooper in android.os.Looper not
mocked. See http://g.co/androidstudio/not-mocked for details.

at android.os.Looper.getMainLooper(Looper.java)
at androidx.arch.core.executor.DefaultTaskExecutor.postToMainThread
at androidx.lifecycle.LiveData.postValue(LiveData.java:273)
```

As you might remember, `LiveData` is a `DataHolder` like a list or a hash map; it can contain any type of objects. You can change the value that your `LiveData` instance holds by using `postValue()` and `setValue()`. `setvalue()` immediately changes the value using the main thread, while `postValue()` asynchronously changes the value using a background thread.

When you create JUnit tests, they'll always run on the main thread. But when you're using `postValue()` in `saveMovie()` to update the `LiveData` object, that will run asynchronously.

Refactoring `saveMovie()` to use `setValue()` instead of `postValue()` might seem like an obvious and simple solution to the problem, but since database operations are long-running tasks, that will likely cause issues in your app. For the record, you never want to execute something that takes a long time on the main thread.

Recall back in the chapter about testing the MVP pattern, you created a custom `TestRule` to force all RxJava schedulers to run immediately when run inside of tests. Similarly, you're going to add a new rule to your tests to indicate that all tasks should execute instantly, rather than asynchronously. Remember that `TestRules` only change the behavior of code during tests. In production, the code will run in parallel. This time, instead of creating your own custom `TestRule`, you'll use one created by Google.

Open the app-level **build.gradle** and add the following line under the dependencies block:

```
testImplementation "androidx.arch.core:core-testing:2.0.0"
```

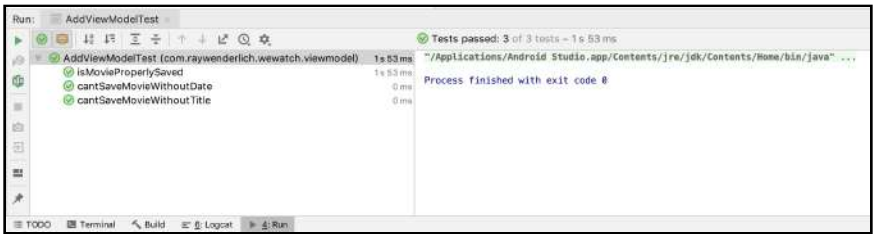
This dependency contains everything you need to perform unit testing for most of the Architecture Components such as `LiveData`.

Sync the project. Open **AddViewModelTest.kt** again and add the following:

```
@get:Rule
var rule: TestRule = InstantTaskExecutorRule()
```

`@get:Rule` indicates that this is a rule that all of your tests must be follow. In this case, the rule you're using is `InstantTaskExecutorRule()`, which swaps the background executor used by the Architecture Components to a different one that executes your tasks synchronously.

Execute your tests one more time to see what happens.



Excellent! Everything passes.

Section III: VIPER and MVI

In this section, you'll rewrite WeWatch two different ways, first using VIPER and then using MVI. VIPER is similar to the MVP architecture but adds another two layers of abstraction to provide the highest level of modularity. On the other hand, MVI follows a different approach by using reactive programming principles with powerful libraries such as RxJava.

- **Chapter 14, VIPER Theory**: Here, you'll learn all of the theory behind each of the layers of the VIPER architecture pattern: **V**iew, **I**nteractor, **P**resenter, **E**ntity and **R**outer.
- **Chapter 15, VIPER Sample**: Now that you know how VIPER works, you'll apply your new knowledge by rewriting WeWatch using this new pattern. You'll also learn how to use Cicerone to create your Routers to navigate through the Views of your app.
- **Chapter 16, Testing VIPER**: In this chapter, you'll learn how to test your VIPER implementation by creating unit tests for each of your Presenters and using Mockito to mock your Views and Interactors.
- **Chapter 17, MVI Theory**: MVI is a special architecture pattern that relies on reactive programming to react to user actions and render your Views. You'll learn how to create Intents, Views and Models that represent a state of your app.
- **Chapter 18, MVI Sample**: Here, you'll rewrite WeWatch using the MVI pattern and the popular RxJava framework for your Observable callbacks.
- **Chapter 19, Debugging MVI**: In this chapter, you'll learn how to debug your MVI implementation using RxJava operators — such as onNext() — to see which was the last Intent emitted and the last View rendered.

Chapter 14: VIPER Theory

By Aldo Olivares Domínguez

After making it to this point in the book, you should already know that there are many options to structure your applications.

In this chapter, you will learn about **VIPER**, an architecture pattern closely related to the MVP architecture. By the end of the chapter you should know:

- The theory behind VIPER, why it is useful and how it fits into the Android framework.
- Each of VIPER's components, their roles and their responsibilities.
- The advantages and disadvantages of VIPER.

What is VIPER?

VIPER stands for **V**iew, **I**nteractor, **P**resenter, **E**ntity and **R**outer. In case you haven't noticed, all the previous patterns have relied on three layers of abstraction, while **VIPER** relies on five. This five-layer structure aims to balance the workload between the different entities to provide the maximum level of modularity.

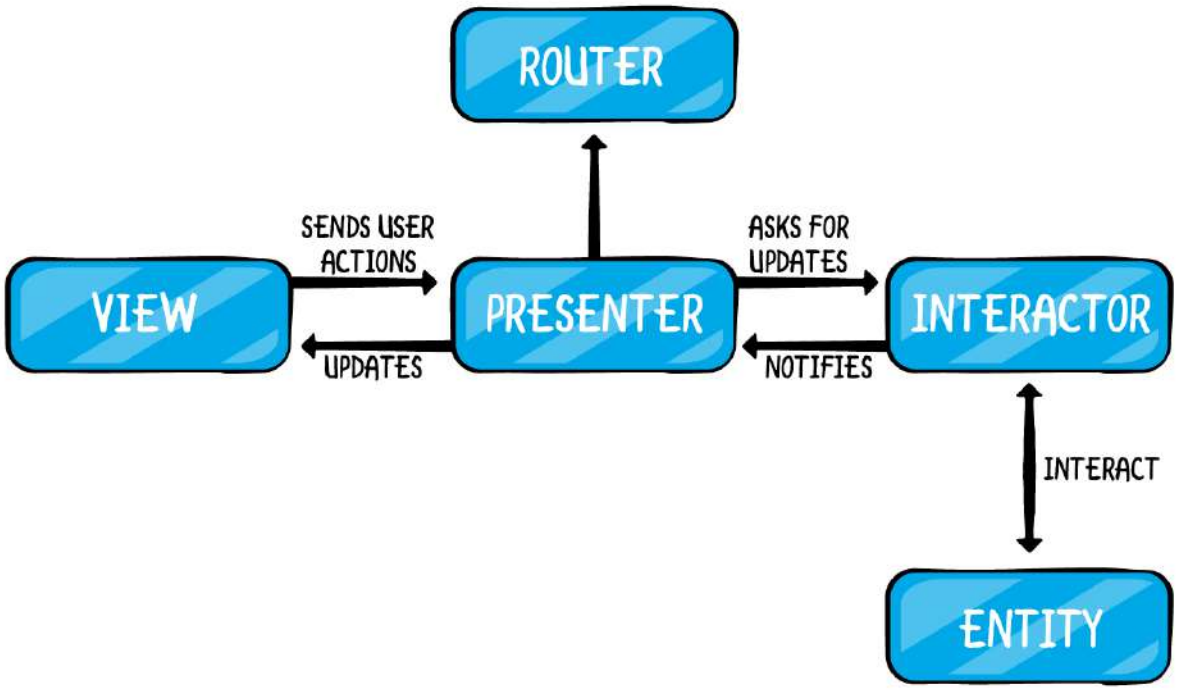
The role of each of the **Entities** are as follows:

- The **View** displays the UI and informs the Presenter of user actions.
- The **Interactor** performs any action and logic related to the entities.
- The **Presenter** acts as a Head of Department. It tells the **View** what to display, tells the router to navigate to other screens and tells the **Interactor** about any necessary updates.

- The **Entity** represents your app's data.
- The **Router** handles the navigation in your app.

Like other architecture patterns, **VIPER's** aims to create a cleaner and more modular structure to isolate your app's dependencies, improving maintainability and testability.

The interaction between the layers can be illustrated by the following diagram:



Each entity corresponds to an object that performs a task and the connections represent how they interact with each other. The communication between each layer is usually handled in Android with interfaces, as you will see in the next chapter.

Let's walk through a real-world example to further illustrate these concepts.

Imagine that you are going on a trip to the mountains and you need to know the weather for the week to ensure that you don't get trapped in a storm.

Normally, you open the weather app on your phone and choose the best day for your journey. When you tap the Refresh button to retrieve the latest weather data, the interaction between the VIPER layers would look like the following:

1. **The View receives the refresh action from the user and sends it to the Presenter.** Whenever you do something in the user interface (like tapping the Refresh button) the View immediately reports to the presenter what you did. It's not the View's job to handle the action directly.

2. **The Presenter asks the Interactor to request new weather data.** Whenever you need to interact with the backend of your app, such as a weather API, the presenter delegates the task to the **Interactor**.

3. **The Interactor calls your Weather API and handles the response.** If the call fails, the **Interactor** immediately returns a fail response to the **Presenter** so that it can be handled appropriately. If the response succeeds, the **Interactor** converts the data into one or more **Entities** and sends them back to the **Presenter**.

4. **The Presenter receives weather data from your Interactor.** If the data needs further processing, the Presenter will call the appropriate functions. If the data is ready to be displayed, the Presenter will send it to the Views along with instructions on how it should be displayed to the User.

5. **The View displays weather information to the user.** The last step of the process is to display new information to the user according to the instructions supplied by the **Presenter**.

And now you are ready to take your long awaited trip to the mountains! Yay!

Now that you have an overview of how each layer interacts with each other. how you'll explore them one by one.

The View

As the name implies, the **View** is in charge of displaying the User Interface (UI) and it is the only thing that the user really interacts with.

In **VIPER**, **Views** can only interact with the presenter to communicate **UI** updates. When the user performs an action that requires business logic, such as pressing a button or tapping a list item, the task is immediately delegated to the **Presenter**, and the **View** waits for a response on what should happen next. **The View knows nothing about how things get done**.

In Android, Views correspond to an **Activity** or **Fragment**, and the goal is to make them as simple as possible. Their sole purpose should be to display what the **Presenter** orders and to handle **UI** interactions.

The Interactor

The **Interactor** takes care of performing any actions that the **Presenter** requests and handles the interaction with your backend modules, such as web services or local databases. Similar to the **ViewModel** from the **MVVM** architecture pattern, the **Interactor** should be completely independent from your Views.

Using the WeWatch app from the previous chapters as an example, the **Interactor** would be in charge of calling the TMDB API to retrieve the list of movies, handle the response and convert the data into the appropriate entities.

The Presenter

As you can see in the previous diagram, the **Presenter** acts as a bridge between the main entities of your **VIPER** implementation.

The **Presenter**:

- Issues navigation-related events to the **Router**.
- Receives data from the **Interactor**, applies any necessary logic and tells the View what to display.
- Sends update events received from the **View** to the **Interactor**.

In Android, a **Presenter** is usually a class that implements one or more interfaces that allow you to define actions in response to what the **Interactor** sends as an output.

A **Presenter**, in Android, would look like this:

```
class MainPresenter()
    : MainContract.Presenter, MainContract.InteractorOutput {   // 1

  private var interactor: MainContract.Interactor? =
MainInteractor()   // 2

  override fun onViewCreated() {   // 3
  }

  override fun onQuerySuccess(data: List<Item>) {   // 4
  }
}
```

Taking each commented section in turn:

1. This class implements two interfaces: a **Presenter** interface and an **InteractorOutput** interface. The **Presenter** interface tells your **Presenter** how to behave, and the **InteractorOutput** interface defines actions in response to what the **Interactor** returns. You can create as many interfaces as you need to define how your own **Presenters** will behave.
2. You will usually need an **Interactor** instance in your Presenter to perform certain actions.
3. This callback defines what will happen when the View is loaded. Normally, you want to override this method to define things that happen automatically without user interaction. For example, you might want to load a list of items to the main screen.
4. This is a method defined in your **InteractorOutput** interface and it helps you define what to do with the Interactor's output.

Needless to say, this is an oversimplified example, but the point remains: The **Presenter** receives output from the **Interactor**, sends data to the UI and tells the router when to navigate to other screens.

The Entity

The **Entity** corresponds to the **Model** in other architecture patterns such as **MVP** or **MVVM**. Its main responsibility is to expose relevant data to your **Interactors** using properties or methods.

The entity should not contain any logic and is usually represented in Kotlin as **Data Classes**.

In most apps, the **Entity** represents the payload coming from a server, such as a Movie from a Movie API or a Pokemon from a Pokemon API.

An **Entity** in an Android app would look like this:

```
data class Movie(
  var voteCount: Int? = null,
  var id: Int? = null,
  var video: Boolean? = null,
  var voteAverage: Float? = null,
  var title: String? = null,
  var popularity: Float? = null,
  var posterPath: String? = null,
  var originalLanguage: String? = null,
  var originalTitle: String? = null,
  var genreIds: List<Int>? = null,
  var backdropPath: String? = null,
  var adult: Boolean? = null,
  var overview: String? = null,
  var releaseDate: String? = null,
  var watched: Boolean = false
)
```

The code above represents a Movie object retrieved from a web service.

As you can see, you can take advantage of Kotlin's data classes to declare your object's attributes inside the constructor, making your code clean and simple.

The Router

The last component of the **VIPER** Architecture pattern is the **Router**. The **Router** handles the navigation in your app and tells the **Views** how they should be presented to the user.

The **Router** is also in charge of passing data from one **View** to another, such as **View** to **View** or **Fragment** to **Fragment**.

In Android, you have many options to implement your Routers with libraries such as **Cicerone** (github.com/terrakok/Cicerone) or the **Navigation Component** from **Google's Architecture Components**.

VIPER Advantages and Concerns

VIPER is a great architecture pattern focused on providing each class in your App a unique responsibility. It provides the maximum level of abstraction amongst the patterns that we have seen until now such as **MVC**, **MVP** or **MVVM**.

When should you use VIPER?

- You expect your code base to increase dramatically over a short period of time.
- You need an architecture pattern that makes adding new features easier and faster.
- You need to write Unit Tests consistently during your development.
- You often find yourself debugging big classes with even bigger methods.

When should you not use VIPER?

As you learned in previous chapters, there is not a single solution to every problem.

Having five different layers for the **VIPER** architecture pattern might cause you to write too much boilerplate code when starting a new project, slowing your development at the beginning.

Also, **VIPER** can be overkill when working on projects in which the code base is rather simple. In this case, other patterns such as **MVC** or **MVP** might be more appropriate.

Remember that architecture patterns are like a Swiss army knife, you must choose the best one according to your needs.

Questions you didn't think to ask

Q. VIPER looks very similar to MVP... are they the same?

Some developers think about VIPER as a *fancy* version of MVP but with an additional two layers of abstraction: the **Interactor** and the **Router**.

Q. Are five layers of abstraction necessary (even on big apps)?

As with any other tool, architecture patterns are only as good as the developer implementing them. There have been numerous apps with features implemented using the traditional MVC pattern and they work great. However, when you are starting a new project that you expect to scale, it is better to just use a pattern suited for your own needs. If you had to dig a hole, what would you choose — a spoon or a shovel? Well, that depends on you and size of the hole needed.

Q. What framework do you recommend for implementing the Router in my apps?

Cicerone is a good framework to use for implementing navigation in your apps. It has broad adoption amongst the community and it is very easy to implement. Google's

Navigation Component is another nice option that is becoming popular, but, at the time of this writing, the first stable version has just been released and there might be some bugs that may slow down your development, so proceed with caution if you are planning to use it on big projects.

Q. Will the models always be the same amongst all the architecture patterns?

Models are a common approach to representing data in your app since they tend to be concise and easy to define. However, the way they communicate to other layers is different across **MVC**, **MVP**, **MVVM** and **VIPER**, so you must be careful to not confuse their implementation or you might end up with a weird **MVP/VIPER** hybrid. For instance, you don't want your **Entities (Models)** to communicate directly with your **Presenters** in **VIPER**, but that is a fine approach in **MVP**.

Key points

- VIPER stands for **V**iew, **I**nteractor, **P**resenter, **E**ntity and **R**outer.
- The **View** displays the User Interface.
- The **Interactor** performs actions related to the Entities.
- The **Presenter** acts as a command center to manage your Views, Interactors and Routers.
- The **Entity** represents data in your app.
- The **Router** handles the navigation between your Views.
- **VIPER** is a great architecture pattern for projects that are expected to scale fast but might be overkill for simple apps.

Where to go from here?

In this chapter, you have learned the building blocks of a completely new architecture pattern: **VIPER**.

This pattern is unique in the sense that it relies on five layers of abstraction rather than three. You might be wondering how this could be applied on a real-world project. In the next chapter, you will apply your knowledge by re-writing the Movies app using **VIPER**.

Chapter 15: VIPER Sample

By Aldo Olivares

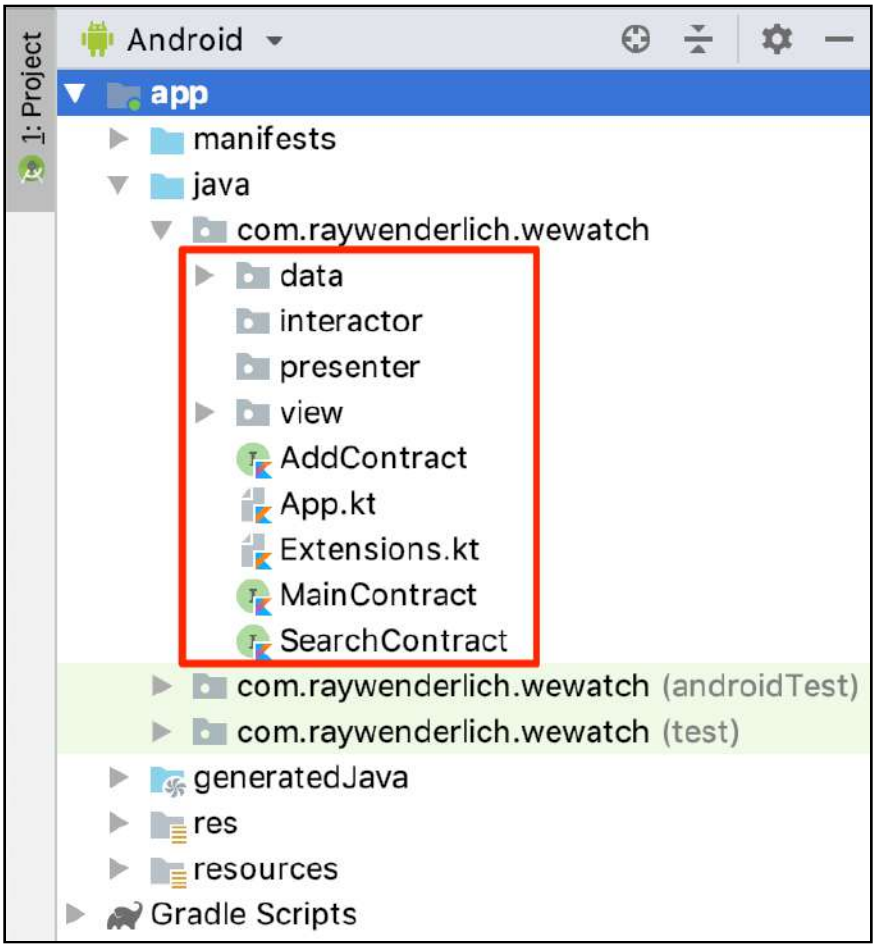
In the last chapter, you learned the theory behind VIPER. You learned how all of its layers work and why this architecture is an excellent option for building maintainable and scalable Android apps.

In this chapter, you'll apply that knowledge to rebuild the WeWatch app using VIPER. Specifically, you'll learn how to:

- Create a Presenter that acts as a bridge between all of the components in your app.
- Create an Interactor that communicates with your app's backend.
- Create a Router that handles the navigation between your Views.
- Use Cicerone to implement your Routers.
- Use Interfaces as contracts to implement the layers of VIPER.

Getting started

Using Android Studio, open the starter project for this chapter project by going to **File ▸ New ▸ Import Project** and selecting **build.gradle** in the root of the project.



The starter project contains the basic structure you'll use for this app, and it contains the following packages:

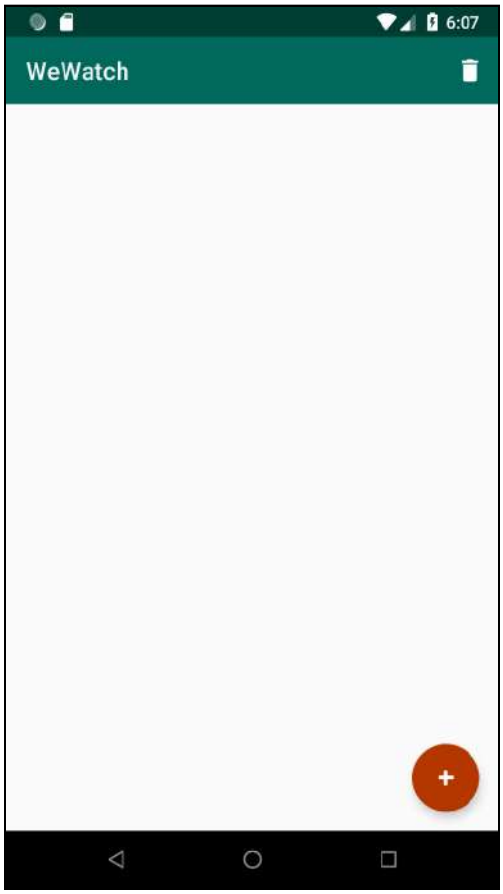
- **Data**: All of the backend code that your app needs to work correctly, including the Entities, a Repository and the Room database components.
- **Interactor**: The interactors of your app. Package should currently be empty.
- **View**: The activities, fragments and adapters.
- **Presenter**: The Presenters of your app. Package should currently be empty.

> **Note**: If you don't see the interactor or presenter packages, then go ahead and just add them to the project by right clicking on the `com.raywenderlich.wewatch` package and selecting **New ▸ Package**.

For didactic purposes, this code was organized with a structure according to VIPER layer names, but in a real-world app you often want to structure your project in packages according to the different modules in your app such as **MainModule**, **AddModule** or **SearchModule**.

Take some time to familiarize yourself with the rest of the starter project and the features included out-of-the-box, like the adapters and layouts.

Once the starter project finishes loading and building, run the app on a device or emulator.



Currently, it's an empty canvas, but that's about to change!

Note: Before moving on to the next section, remember to get an API Key from TMDB API (https://developers.themoviedb.org/3/getting-started/introduction) and to substitute the value inside **RetrofitClient.kt**.

Defining your app's contract

For this project, every module is represented as an Interface contract that needs to be implemented by several associated classes. The contract represents which VIPER layers you must implement in every module and the actions the layers need to perform.

Open **MainContract.kt** and review the contract that corresponds to Main Module:

```
interface MainContract {

  interface View {
    fun showLoading()
    fun hideLoading()
    fun showMessage(msg: String)
    fun displayMovieList(movieList: List<Movie>)
    fun deleteMoviesClicked()
  }

  interface Presenter {
    fun deleteMoviesClick(selectedMovies: HashSet<*>)
    fun onViewCreated()
    fun onDestroy()
    fun addMovieClick()
  }

  interface Interactor {
    fun loadMovieList(): LiveData<List<Movie>>
    fun delete(movie: Movie)
    fun getAllMovies()
  }

  interface InteractorOutput {
    fun onQuerySuccess(data: List<Movie>)
    fun onQueryError()
  }
}
```

Notice that a few interfaces represent some of the main components of the VIPER architecture pattern like View, Presenter and Interactor. In the next few sections, you'll implement each module with the corresponding Interactors, Presenters and Views.

Implementing the Main Module

Now that you have the contract defined for the main module, it's time to apply the contract to each individual component. You'll start with the View.

The View

Open **MainActivity.kt** inside **view/activities** and make it so `MainActivity` implements `MainContract.View`:

```
class MainActivity : BaseActivity(), MainContract.View {
```

Android Studio immediately displays a warning informing you of some members that your class needs to implement. Press **Control-I** to get a complete list of the missing members. Select all of them and click **OK**.

Start by implementing `showLoading()` and `hideLoading()`:

```
override fun showLoading() {
  moviesRecyclerView.isEnabled = false
  progressBar.visibility = View.VISIBLE
}

override fun hideLoading() {
  moviesRecyclerView.isEnabled = true
  progressBar.visibility = View.GONE
}
```

> **Note**: Remember to use **Alt-Enter** on Windows or **Option-Return** on Mac to import any missing classes.

The above code hides or shows the `progressBar` and the `moviesRecyclerView` whenever the Presenter instructs it to do so.

Now, add the following inside `showMessage()`:

```
toast(msg)
```

This method displays a simple message to the user using a toast message.

Add the following properties to `MainActivity` to hold a reference to the Presenter, which you'll implement later, and the adapter for `moviesRecyclerView`:

```
lateinit var presenter: MainContract.Presenter
private lateinit var adapter: MovieListAdapter
```

Add this inside `displayMoviesList()`:

```
adapter = MovieListAdapter(movieList)
moviesRecyclerView.adapter = adapter
```

This creates a new `MovieListAdapter` and assigns it to `moviesRecyclerView` once the data is received.

Now, add the following inside `deleteMoviesClicked()`:

```
presenter.deleteMovies(adapter.selectedMovies)
```

This function tells the Presenter when the user wants to delete all of the movies marked as watched by passing the selected movies as a parameter to `deleteMovies()`.

Add this code inside `goToSearchActivity()`:

```
presenter.addMovie()
```

This tells the Presenter that you tapped + and you intend to add a new movie to your list.

Next, override the activity's `onResume()` and `onDestroy()` to tell the Presenter when the View is created and when it gets destroyed. This way the Presenter can perform any initial set up tasks and remove all unnecessary references.

```
override fun onResume() {
  super.onResume()
  presenter.onViewCreated()
}

override fun onDestroy() {
  super.onDestroy()
  presenter.onDestroy()
}
```

Finally, add the following code to override `onOptionsItemSelected()`:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
  when (item.itemId) {
    R.id.action_delete -> this.deleteMoviesClicked()
    else -> toast(getString(R.string.error))
  }
  return super.onOptionsItemSelected(item)
}
```

This method calls `deleteMoviesClicked()` when the user taps the delete icon at the top.

With the View ready, it's time to implement the Router.

The Router

As you learned in the previous chapter, the Router layer is in charge of the navigation across the multiple Views in your app. In a traditional VIPER implementation you would have a Presenter that receives actions from the View that are then translated as commands for the Interactor or the Router.

With this architecture in mind, it makes a lot of sense to create a Router that knows about all of the Views but at the same time lets the Presenter command the navigation. However, due to how Android is designed, you'll always need an `Intent` and a call to `startActivity()` to move between different modules in your app. Therefore, it's challenging to move the navigation logic from the Views to another component.

There are many workarounds to solve this problem, including the new **Jetpack Navigation Controller**. At the time of writing this chapter, the Navigation Controller was just released as a stable version, so there might still be some bugs. For that reason, you'll use a different tool: **Cicerone**.

So, what is Cicerone? According to the documentation, Cicerone is a lightweight library for MVP designed to help you manage the navigation in your Android app.

It offers several benefits:

- Unlike other libraries, Cicerone is not tied to fragments, which means that you can also use it with activities.
- It's easy to test compared to other libraries and frameworks.
- It's lifecycle-safe which makes it great to use with other lifecycle aware components such as the ViewModels from Google's Architecture Components.

To use Cicerone in your app, you need to perform some initial setup tasks. First, add the required dependency to **app/build.gradle** in the dependencies block:

```
def ciceroneVersion = "2.1.0"
implementation "ru.terrakok.cicerone:cicerone:$ciceroneVersion"
```

Sync the app Gradle files and wait until the project finishes building.

Then, open **App.kt** inside the app's root package and add the following property and method:

```
lateinit var cicerone: Cicerone<Router>

private fun initCicerone() {
  this.cicerone = Cicerone.create()
}
```

This is a simple function that initializes a Cicerone instance that should be available for all your classes.

Now, add the following line inside `onCreate()` to initialize your `cicerone` property when the app launches:

```
this.initCicerone()
```

That's it! That's everything you need to use Cicerone in your app. Now, you only need to make your Views aware of this new component.

Open **MainActivity.kt** and add the following properties:

```
//1
companion object {
  val TAG: String = "MainActivity"
}
//2
private val navigator: Navigator? by lazy {
  object : Navigator {
    //3
    override fun applyCommand(command: Command) {   // 2
      if (command is Forward) {
        forward(command)
      }
    }
    //4
    private fun forward(command: Forward) {
      when (command.screenKey) {
        AddMovieActivity.TAG -> startActivity(Intent(this@MainActivity,
AddMovieActivity::class.java))
        else -> Log.e("Cicerone", "Unknown screen: " + command.screenKey)
      }
    }
  }
}
```

Taking each comment section in turn:

1. Declare a companion object with a TAG to identify this View.
2. Declare a `Navigator` instance that let's your Router process a series of navigation commands usually coming from a Presenter.
8. Maps the `Forward` command to a specified View in your app. In this case, you are specifying that if the `forward` command contains `AddMovieActivity.TAG`, then start `AddMovieActivity`.

You now need to set your navigator in the activity's `onResume()` by adding the following line:

```
App.INSTANCE.cicerone.navigatorHolder.setNavigator(navigator)
```

You also need to remove it when your activity is no longer visible by overriding `onPause()`. Add the following line:

```
App.INSTANCE.cicerone.navigatorHolder.removeNavigator()
```

Now, add the following property to get an instance of your `router`:

```
private val router: Router? by lazy { App.INSTANCE.cicerone.router }
```

That's it! Your Router is ready for use. Now it's time to implement the Presenter.

The Presenter

Create a new package named **presenter**, and then add a new file named **MainPresenter.kt**. Replace everything inside with the following:

```
//1
class MainPresenter(private var view: MainContract.View?,
                    private var interactor: MainContract.Interactor?,
                    private val router: Router?) :
MainContract.Presenter, MainContract.InteractorOutput {
  //2
  override fun addMovie() {
    router?.navigateTo(AddMovieActivity.TAG)
  }
  //3
  override fun deleteMovies(selectedMovies: HashSet<*>) {
    for (movie in selectedMovies) {
      interactor?.delete(movie as Movie)
    }
  }
  //4
  override fun onViewCreated() {
    view?.showLoading()
    interactor?.loadMovieList()?.observe((view as MainActivity), Observer
{ movieList ->
      if (movieList != null) {
        onQuerySuccess(movieList)
      } else {
        onQueryError()
      }
    })
  }
  //5
  override fun onDestroy() {
    view = null
    interactor = null
  }
```

```
    //6
    override fun onQuerySuccess(data: List<Movie>) {
      view?.hideLoading()
      view?.displayMovieList(data)
    }
    //7
    override fun onQueryError() {
      view?.hideLoading()
      view?.showMessage("Error Loading Data")
    }
  }
```

Here's the breakdown:

1. `MainPresenter` implements `MainContract.Presenter`. This class takes a `View`, `Router` and an `Interactor` as constructor parameters.
2. When the user taps + to add a new movie, you call `addMovie()`. This method uses `router` to navigate to `AddMovieActivity`.
3. When the user presses the delete button to delete watched movies, you call `deleteMovies()`. You'll use the `interactor` to remove movies from the Room database.
4. You call `onViewCreated()` when the View is visible to the user. This method uses the `loadMovieList()` from the Interactor to retrieve a list of Movies. If the response is successful, you call `onQuerySuccess()`, otherwise you use `onQuerryError()` to send an error message to the user.
5. Here, you use `onDestroy()` to remove the references to the View and Interactor.
6. When the Interactor successfully returns a response, you call `onQuerySuccess()` to instruct the View to hide the progress bar and display a list of movies.
7. When the Interactor is unable to retrieve the list of movies, you call `onQueryError()` to instruct the View to hide the progress bar and show an error message.

> **Note**: In a real-world app you'd typically use a dependency injection library such as **Dagger**, **Kodein** or **Koin** to manage the dependencies.

The Interactor

Next, you need to implement the Interactor for the Main Module.

Create another new package named **interactor** and add a file named **MainInteractor.kt**. Replace everything inside with the following:

```
//1
class MainInteractor : MainContract.Interactor {
  //2
  private val movieList = MediatorLiveData<List<Movie>>()
  private val repository: MovieRepositoryImpl = MovieRepositoryImpl()
  //3
  init {
    getAllMovies()
  }
  //4
  override fun loadMovieList() = movieList
  //5
  override fun delete(movie: Movie) = repository.deleteMovie(movie)
  //6
  override fun getAllMovies() {
    movieList.addSource(repository.getSavedMovies()) { movies ->
      movieList.postValue(movies)
    }
  }
}
```

Here's how it works:

1. `MainInteractor` implements `MainContract.Interactor`.
2. Here, you declare a `movieList` property that contains a `LiveData` list of movies and a `repository` property to hold a reference to `MovieRepository`.
3. When the class is initialized, you immediately call `getAllMovies()`.
4. `loadMovieList()` returns a reference to `movieList`.
5. `delete()` calls your repository's `deleteMovie()` to delete the movie passed as a parameter.
6. `getAllMovies()` adds your repository's `getSavedMovies()` as a data source for `movieList`. When the data changes, the list automatically gets updated.

You may have noticed there's an unresolved reference to `AddMovie.TAG` in both `MainPresenter` and `MainActivity`. For Cicerone to work correctly, you need to define `TAG`s on each activity.

Open **AddMovieActivity.kt** and add the following companion object to `AddMovieActivity`:

```
companion object {
  val TAG: String = "AddMovieActivity"
}
```

Then, open **SearchMovieActivity.kt** and add this one to `SearchMovieActivity`:

```
companion object {
  val TAG: String = "SearchMovieActivity"
}
```

The last step to finish the Main Module is to initialize `MainPresenter` inside `MainActivity`'s `onCreate()`:

```
presenter = MainPresenter(this, MainInteractor(), router)
```

Now, call `onViewCreated()` inside `onResume()`:

```
presenter.onViewCreated()
```

It's time to implement the **AddMovie** module.

Implementing the AddMovie module

With the main module out the way, it's time to apply the same architecture to the add movie module. You'll again start with the View. Be sure to review the contract for the add movie module in the **AddContract.kt** file found at the root package.

The View

Open **AddMovieActivity.kt** inside **view/activities**, and make `AddMovieActivity` implement `AddContract.View`:

```
class AddMovieActivity : BaseActivity(), AddContract.View {
```

Press **Control-I** to implement the missing members.

Now that the View is ready, add the following properties for `presenter`, `router` and `navigator`:

```
var presenter: AddContract.Presenter? = null
private val router: Router? by lazy { App.INSTANCE.cicerone.router }

private val navigator: Navigator? by lazy {
  object : Navigator {
```

```
    override fun applyCommand(command: Command) {
      if (command is Back) {
        back()
      }
      if (command is Forward) {
        forward(command)
      }
    }

    private fun forward(command: Forward) {
      when (command.screenKey) {
        SearchMovieActivity.TAG ->
startActivity(Intent(this@AddMovieActivity,
SearchMovieActivity::class.java)
            .putExtra("title", titleEditText.text.toString()))
        MainActivity.TAG -> startActivity(Intent(this@AddMovieActivity,
MainActivity::class.java)
            .setFlags(Intent.FLAG_ACTIVITY_NEW_TASK or
Intent.FLAG_ACTIVITY_CLEAR_TASK))
        else -> Log.e("Cicerone", "Unknown screen: " + command.screenKey)
      }
    }

    private fun back() {
      finish()
    }
  }
}
```

Just like the Main module, you need a `navigator` to process the commands that the `router` sends. The only difference is that, in this case, you have two possibilities for the forward command: `SearchMovieActivity` and `MainMovieActivity`.

To display a `Snackbar` with a message, replace the code inside `showMessage()`:

```
addLayout.snack((msg), Snackbar.LENGTH_LONG) {
  action(getString(R.string.ok)) {
  }
}
```

To inform the Presenter that the user wants to add a new movie, add the following inside `addMovieClicked()`:

```
presenter?.addMovies(titleEditText.text.toString(),
yearEditText.text.toString())
```

Now, add the following code inside `searchMovieClicked()` to tell the Presenter that the user wants to query the TMDB API:

```
presenter?.searchMovies(titleEditText.text.toString())
```

Finally, override `onResume()`, `onDestroy()` and `onPause()`:

```
override fun onResume() {
  super.onResume()
  App.INSTANCE.cicerone.navigatorHolder.setNavigator(navigator)
}

override fun onDestroy() {
  super.onDestroy()
  presenter?.onDestroy()
}

override fun onPause() {
  super.onPause()
  App.INSTANCE.cicerone.navigatorHolder.removeNavigator()
}
```

Similar to what you previously did for `MainView`, you need to set the navigator in `onResume()` and remove it in `onPause()`.

The Presenter

Create a new file inside the **presenter** package, name it **AddPresenter.kt** and replace everything inside with the following:

```
//1
class AddPresenter(private var view: AddContract.View?,
                   private var interactor: AddContract.Interactor?,
                   private val router: Router?) : AddContract.Presenter {
  //2
  override fun onDestroy() {
    view = null
    interactor = null
  }
  //3
  override fun addMovies(title: String, year: String) {
    if (title.isNotBlank()) {
      val movie = Movie(title = title, releaseDate = year)
      interactor?.addMovie(movie)
      router?.navigateTo(MainActivity.TAG)
    } else {
      view?.showMessage("You must enter a title")
    }
  }
  //4
  override fun searchMovies(title: String) {
    if (title.isNotBlank()) {
      router?.navigateTo(SearchMovieActivity.TAG)
    } else {
      view?.showMessage("You must enter a title")
    }
  }
}
```

Here's the step-by-step:

1. `AddPresenter` implements `AddContract.Presenter` and accepts a `Router`, an `Interactor` and a `View` as constructor parameters.
2. `onDestroy()` removes the reference to the `View` and the `Interactor`.
3. `addMovies()` uses the `Interactor`'s `addMovie()` to add a new movie to the Room database, passing in the title and year as a parameter. If `title` is blank, the View displays an error message to the user.
4. `searchMovies()` uses `router` to navigate to `SearchMovieActivity` and passes `title` as an Extra value in the Intent. If `title` is empty, the View displays an error message as a `Snackbar` to the user.

The Interactor

Create a new file inside **interactor**, name it **AddInteractor** and replace everything inside with the following:

```
class AddInteractor : AddContract.Interactor {

  private val repository: MovieRepositoryImpl = MovieRepositoryImpl()

  override fun addMovie(movie: Movie) = repository.saveMovie(movie)

}
```

`AddInteractor` contains a single method, `addMovie()`, that uses the repository to add a new record in the Room database.

To create a new `AddMoviePresenter` instance, open **AddMovieActivity.kt** and add the following code inside `onCreate()`:

```
presenter = AddPresenter(this, AddInteractor(), router)
```

Implementing SearchMovie

There's only one module remaining: `SearchMovie`.

You've completed the main module and the add movie module, you'll cap things off with the search module. Again, be sure to review the contract for the search module in the **SearchContract.kt** file found in the root package of the project. As usual, you'll start by making changes to the View.

The View

Open **SearchMovieActivity.kt** inside **view/activities** and make `SearchMovieActivity` implement `SearchContract.View`:

```
class SearchMovieActivity : BaseActivity(), SearchContract.View {
```

Remember to implement all of the missing members by using the **Control-I** shortcut. Add the following properties for `presenter`, `router` and `navigator`:

```
private var presenter: SearchContract.Presenter? = null
private val router: Router? by lazy { App.INSTANCE.cicerone.router }

private val navigator: Navigator? by lazy {
  object : Navigator {
    override fun applyCommand(command: Command) {
      if (command is Back) {
        back()
      }
      if (command is Forward) {
        forward(command)
      }
    }

    private fun forward(command: Forward) {
      when (command.screenKey) {
        MainActivity.TAG ->
startActivity(Intent(this@SearchMovieActivity, MainActivity::class.java)
            .setFlags(Intent.FLAG_ACTIVITY_NEW_TASK or
Intent.FLAG_ACTIVITY_CLEAR_TASK))
        else -> Log.e("Cicerone", "Unknown screen: " + command.screenKey)
      }
    }

    private fun back() {
      finish()
    }
  }
}
```

Just like the Main and AddMovie modules, you'll need a `Navigator` for the `Router` and a `SearchPresenter` instance.

Update `showLoading()` and `hideLoading()`, like so:

```
override fun showLoading() {
  searchProgressBar.visibility = View.VISIBLE
  searchRecyclerView.isEnabled = false
}

override fun hideLoading() {
  searchProgressBar.visibility = View.GONE
  searchRecyclerView.isEnabled = true
}
```

These methods toggle the display property pf the progress bar and `RecyclerView` list of movies. When one is shown, the other is hidden. These are usually called before and after making a network call to retrieve the movies.

Add the following code inside `showMessage()`:

```
searchLayout.snack(getString(R.string.network_error),
Snackbar.LENGTH_INDEFINITE) {
  action(getString(R.string.ok)) {
    val title = intent.extras.getString("title")
    presenter?.searchMovies(title)
  }
}
```

This method displays a `Snackbar` with a network error message. When the user clicks **OK**, the Presenter tries to retrieve the list of movies again.

For `displayMovieList()`, add this code to attach a new adapter with the movie list data passed as a parameter:

```
adapter = SearchListAdapter(movieList) { movie ->
presenter?.movieClicked(movie) }
searchRecyclerView.adapter = adapter
```

Add the following lines to `displayConfirmation()`:

```
searchLayout.snack("Add ${movie?.title} to your list?",
Snackbar.LENGTH_LONG) {
  action(getString(R.string.ok)) {
    presenter?.addMovieClicked(movie)
  }
}
```

Here, you're displaying a confirmation message before asking the presenter to add a new movie to the database.

Finally, update `onDestroy()`, `onPause()` and `onResume()`:

```
override fun onResume() {
  super.onResume()
  val title = intent.extras.getString("title")
  presenter?.searchMovies(title)
  App.INSTANCE.cicerone.navigatorHolder.setNavigator(navigator)
}

override fun onPause() {
  super.onPause()
  App.INSTANCE.cicerone.navigatorHolder.removeNavigator()
}

override fun onDestroy() {
  super.onDestroy()
```

```
    presenter?.onDestroy()
  }
```

Like you did in `MainActivity` and `AddMovieActivity`, you use these methods to set the Navigator and to tell the Presenter when the View is visible to the user.

The Presenter

Create a new file inside **presenter**, name it **SearchPresenter.kt** and replace everything inside with the following:

```
//1
class SearchPresenter(private var view: SearchContract.View?, private var
interactor: SearchContract.Interactor?, val router: Router?) :
SearchContract.Presenter, SearchContract.InteractorOutput {
  //2
  override fun searchMovies(title: String) {
    view?.showLoading()
    interactor?.searchMovies(title)?.observe(view as SearchMovieActivity,
Observer { movieList ->
      if (movieList == null) {
        onQueryError()
      } else {
        onQuerySuccess(movieList)
      }
    })
  }
  //3
  override fun addMovieClicked(movie: Movie?) {
    interactor?.addMovie(movie)
    router?.navigateTo(MainActivity.TAG)
  }
  //4
  override fun movieClicked(movie: Movie?) {
    view?.displayConfirmation(movie)
  }
  //5
  override fun onDestroy() {
    view = null
    interactor = null
  }
  //6
  override fun onQuerySuccess(data: List<Movie>) {
    view?.hideLoading()
    view?.displayMovieList(data)
  }
  //7
  override fun onQueryError() {
    view?.hideLoading()
    view?.showMessage("Error")
  }
}
```

1. `SearchPresenter` implements `SearchContract.Presenter` and accepts a `Router`, a `View` and an `Interactor` as constructor parameters.
2. `searchMovies()` uses the Interactor to retrieve a list of movies and observes it until there's a response. If the response is successful, you pass the new list of movies as a parameter to`onQuerySuccess()`, otherwise you call `onQueryError()`.
3. `addMovieClicked()` uses the Interactor to save a new movie to the database and the Router to navigate to `MainActivity`.
4. `movieClicked()` is a simple method that asks for confirmation from the user using the View.
5. `onDestroy()` removes the references to the View and Interactor.
6. `onQuerySuccess()` tells the View to display a new movie list to the user.
7. `onQueryError()` displays an error message to the user.

The Interactor

Create a new file inside **interactor**, name it **SearchInteractor.kt** and replace everything inside with the following:

```
//1
class SearchInteractor : SearchContract.Interactor {
  //2
  private val repository: MovieRepositoryImpl = MovieRepositoryImpl()
  //3
  override fun searchMovies(title: String): LiveData<List<Movie>?> =
repository.searchMovies(title)
  //4
  override fun addMovie(movie: Movie?) {
    movie?.let {
      repository.saveMovie(movie)
    }
  }
}
```

1. `SearchInteractor` implements `SearchContract.Interactor`.
2. Here, you create a new `repository` property for `MovieRepository`.
3. `searchMovies()` uses `repository` to get a list of movies that match the title passed as a parameter.
4. `addMovie()` uses `repository` to save a movie in the Room database.

Open **SearchMovieActivity.kt** and instantiate a new instance of `SearchPresenter` by adding the following line to `onCreate()`:

```
presenter = SearchPresenter(this, SearchInteractor(), router)
```

That's it! You're ready to test your app and see it in action. Build and run, and you'll see a fully functioning app.

Key points

- View is the component that receives UI actions from the user and sends them to the Presenter.
- Interactor is the component in charge of interacting with your backend such as your databases and web services.
- Presenter is like the commander of your architecture. It's in charge of coordinating your Views, Interactors and Routers.
- Entity is the component that represents your app's data. It's usually represented as data classes in Kotlin.
- Router is the component that manages the navigation between the Views in your app.

Where to go from here?

VIPER is a great architecture pattern that focuses on providing the maximum level of modularity for your Android projects. It allows you to create app's that are maintainable, scalable and easy to test.

In the next chapter, you'll learn how to test the architecture of your app by creating unit tests for your Presenters and using mockito to mock your Views and Interactors.

Chapter 16: Testing VIPER

Aldo Olivares

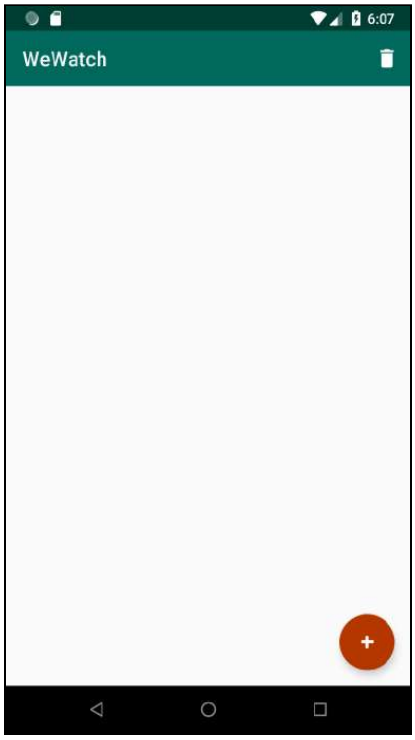
In the last chapter, you refactored WeWatch to use the View-Interactor-Presenter-Entity-Router (better know as VIPER) architecture pattern. In this chapter, you'll learn how to test your VIPER architecture by creating unit tests and mocking classes using Mockito.

Getting started

Start by opening the starter project and selecting the app **build.gradle** from the project.

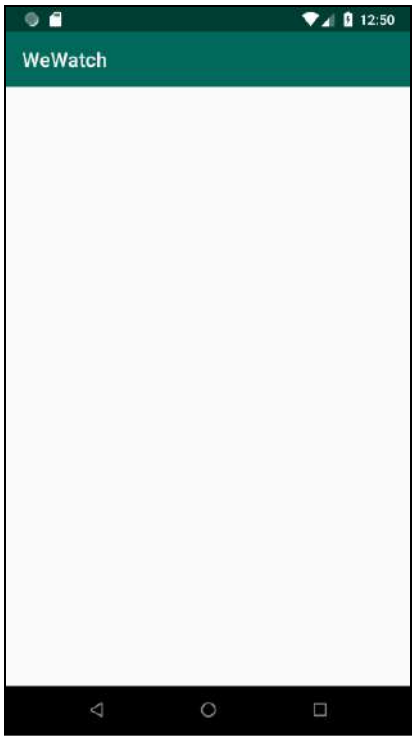
> **Note**: In order to search for movies in the WeWatch app, you must first get access to an API key from the Movie DB. To get your API own key, sign up for an account at www.themoviedb.org. Then, navigate to your account settings on the website, view your settings for the API, and register for a developer API key. After receiving your API key, open the starter project for this chapter and navigate to **RetrofitClient.kt**. There, you can replace the existing value for `API_KEY` with your own.

After the starter project finishes loading and building, run the app on a device or emulator and try adding a new movie by tapping the + float action button.

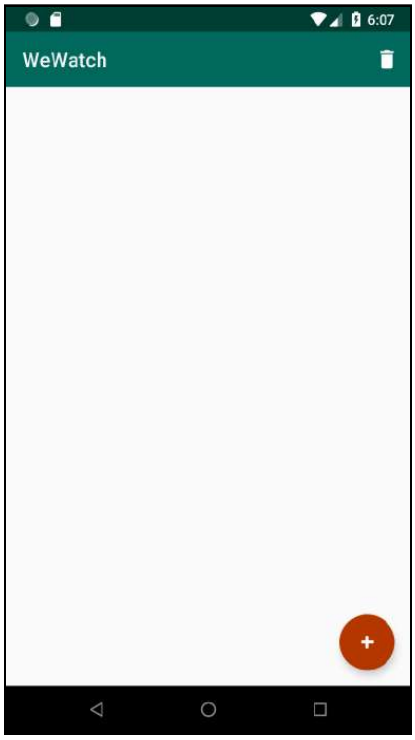


Type in a title and press the icon to the right to search for a movie.

Nothing shows up, indicating there's some kind of problem.



At this point, it's unclear whether there's an issue with the search function or the TMDB API. Try adding a movie manually using **Add Movie**. Notice the main screen is still empty.



Something is clearly wrong — and thanks to unit testing, you'll be able to find out what it is by creating unit tests for each of your Presenters.

Testing your Main presenter

For this project, you'll use Mockito to mock the necessary dependencies for your app, and you'll use JUnit to perform the tests.

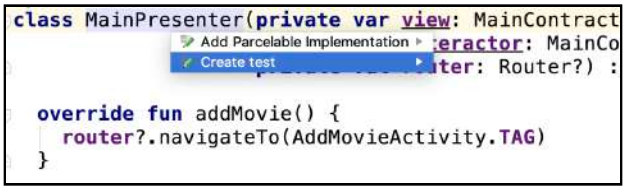
Open **App/build.gradle** and add the Mockito and JUnit dependencies to the dependencies block:

```
testImplementation 'junit:junit:4.12'
testImplementation 'com.nhaarman:mockito-kotlin-kt1.1:1.5.0'
```

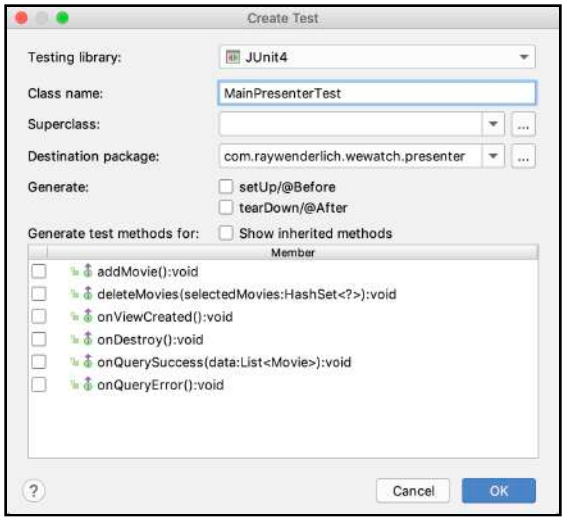
Sync the project Gradle files and run the app.

Because the problem with the app might be that the View is not calling the right method from the Presenter, the first thing you need to do is test `MainPresenter`, which is in charge of commanding everything in `MainActivity`.

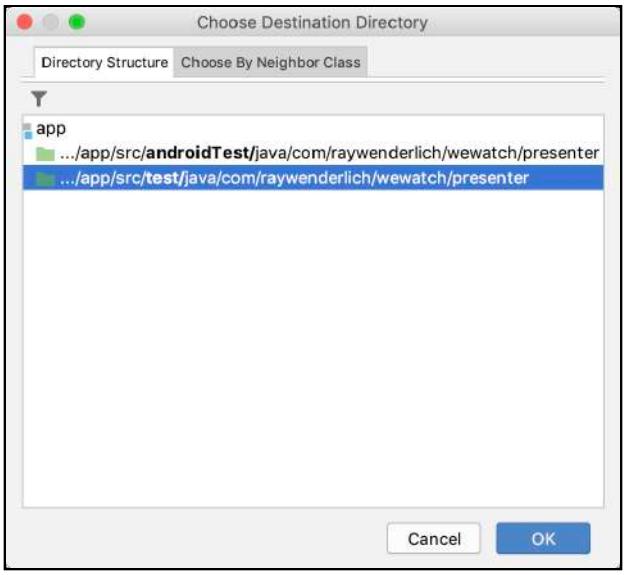
Open **MainPresenter.kt** and select the class name. Press **Alt-Enter**, or **Option-Return** on a Mac, and select **Create test**.



Then, click **OK**.



In the next window, select **test** and click **OK**.



Android Studio creates a new `MainPresenterTest` empty test class for you to test `MainPresenter`.

To test `MainPresenter`, you need four things: a `MainPresenter` instance, a `Router`, a `View` and an `Interactor`.

Add the following code to `MainPresenterTest`:

```
@RunWith(MockitoJUnitRunner::class)
class MainPresenterTest {
    private lateinit var presenter: MainPresenter
}
```

You'll use the presenter property to create a `MainPresenter` instance to test.

Add the following properties immediately below `presenter`:

```
@Mock
private lateinit var view: MainContract.View
@Mock
private lateinit var interactor: MainContract.Interactor
```

You need a View and an Interactor to test `MainPresenter`. You'll use Mockito to mock your View and your Interactor. The `@Mock` annotation is an abbreviated way to inform Mockito that these dependencies are mocked and will be initialized later.

Add the following method:

```
//1
@Before
fun setup() {
```

```
    //2
    val cicerone = Cicerone.create()
    //4
    presenter = MainPresenter(view, interactor, cicerone.router)
  }
```

Here's what's happening:

1. `@Before` is an annotation that informs Mockito that this function should be executed before any tests. When testing a class with Mockito, it's common to have one `setup()` function that initializes all of the mocks and classes you need for your tests.
2. Here, you create a `Cicerone` instance to get a Router for `MainPresenter`.
3. You create a `MainPresenter` instance by passing the mocks you just created.

Typically, you should have at least one unit test for each function in your class. Since `MainPresenter` has six methods, you need at least six tests. However, to keep things short and sweet, you'll only create tests for a few of these methods.

Getting back to your mission, you need to investigate why the list of movies isn't showing up in your View. It's not obvious where the bug might be, so you'll start by testing the View's `displayMovieList` method. Add the following test:

```
  @Test
  fun displayMoviesOnQuerySuccess() {
    //1
    val movieList = listOf(Movie())
    //2
    presenter.onQuerySuccess(movieList)
    //3
    verify(view).displayMovieList(movieList)
  }
```

Here's what's happening:

1. This creates a new list containing a single movie.
2. Here, you call the Presenter's `onQuerySuccess()`.
3. This verifies that your View's `displayMovieList()` is called with the appropriate list when `onQuerySuccess()` is invoked.

Now that your test is ready, click the green arrow to the left.

```
@Test
fun displayMoviesOnQuerySuccess() {
    //2
    val movieList = listOf(Movie())
    //3
    presenter.onQuerySuccess(movieList)
    //4
    verify(view).displayMovieList(movieList)
}
```

Look at the console, and you'll see the test passes:

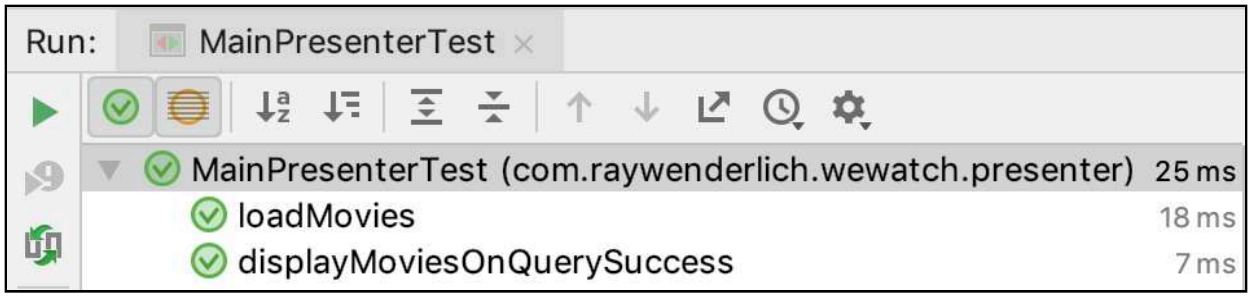
Because this test passes, you now know that `displayMovieList()` is properly getting called with a movie list when `onQuerySuccess()` is invoked in `MainPresenter`.

Next you'll check if the View's `loadMovieList` might be the problem, so add the following test:

```
@Test
fun loadMovies() {
  presenter.onViewCreated()
  verify(interactor).loadMovieList()
}
```

This test verifies that the Presenter is calling the Interactor's `loadMovieList()`.

To run all of the tests at the same time, click the green arrow next to the test class name.



Interesting. It looks like the problem is someplace else in the code. Perhaps it's in the `AddMoviePresenter`?

Testing the AddMovie presenter

In this section, you'll create a test class for `AddMoviePresenter`.

Open **AddPresenter.kt** and add a test for this class. Remember to add the test to the **test** directory when prompted.

Next, add the following properties and method to `AddPresenterTest`:

```
private lateinit var presenter: AddPresenter
@Mock
private lateinit var view: AddContract.View
@Mock
private lateinit var interactor: AddContract.Interactor

@Before
fun setup() {

  val cicerone = Cicerone.create()
  val router = cicerone.router

  presenter = AddPresenter(view, interactor, router)
}
```

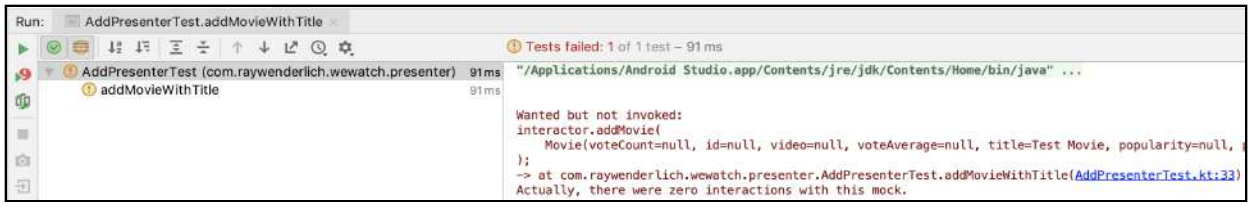
Similar to `MainPresenterTest`, you need the `AddPresenter` instance and the mocks for the View and Interactor.

Now, add the following test:

```
@Test
fun addMovieWithTitle() {
  val movie = Movie(title = "Test Movie", releaseDate = "1991")
  presenter.addMovies(movie.title!!, movie.releaseDate!!)
  verify(interactor).addMovie(movie)
}
```

Here, you create a new `Movie` instance and invoke `addMovies()` of `AddPresenter` to tell the Interactor that you need to save the movie.

Execute the test.



That's right, the test fails! Look at the console, and you'll see the following messages:

```
Wanted but not invoked:
interactor.addMovie

There were zero interactions with this mock.
```

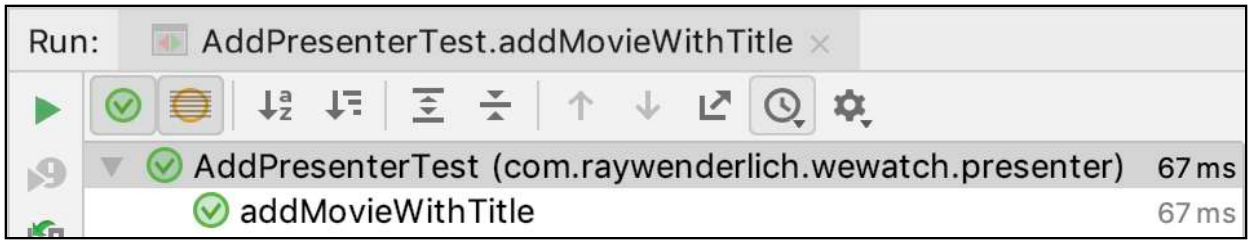
The first message tells you that `addMovie()` was never invoked and the second message is warning you that there were zero interactions with the Interactor, which means you never used any of the Interactor's methods.

Excellent, now you know what's causing the problem — `addMovies()` is not getting called in `AddPresenter` — and you can fit it.

Open **AddPresenter.kt** and add the following lines to `addMovies()` immediately below the call to `router.navigateTo()`:

```
val movie = Movie(title = title, releaseDate = year)
interactor?.addMovie(movie)
```

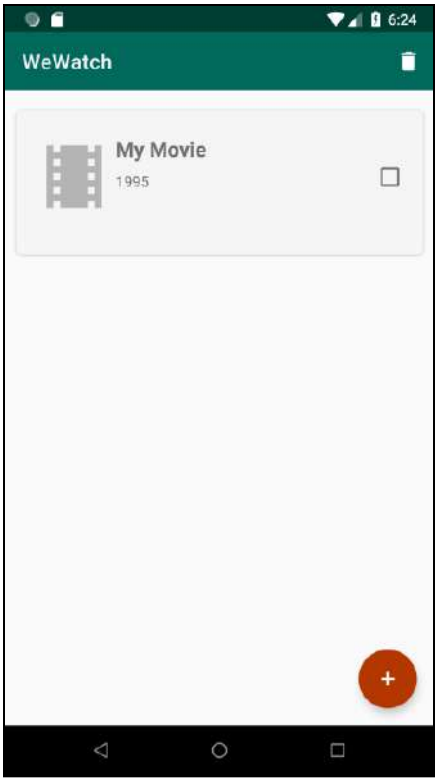
Run the test again.



The test passes!

Build and run the app again, and try to *manually* add a movie. Don't forget to give it a title. When you're done, tap **Add Movie**.

So, now your new movie is visible on the screen.



There's only one step remaining: To test `SearchMoviePresenter`.

Testing SearchMovie Presenter

It's time to create a test class for `SearchPresenter` following the usual steps.

After adding the usual boilerplate to `SearchPresenterTest` (remembering to add an instance of `SearchPresenter`), add a test to verify the View hides the progress bar and displays the movie list received from the Presenter:

```
@Test
fun displayMovieListAndHideLoadingOnQuerySuccess() {
  val movieList = listOf(Movie(title = "Best Movie"))
  presenter.onQuerySuccess(movieList)
  verify(view).hideLoading()
  verify(view).displayMovieList(movieList)
}
```

Click the green arrow to the left of your method to run your test.

Notice the test fails. Inspect the logs, and you'll see something interesting:

```
Wanted but not invoked:
view.displayMovieList

However, there was exactly 1 interaction with this mock:
view.hideLoading();
```

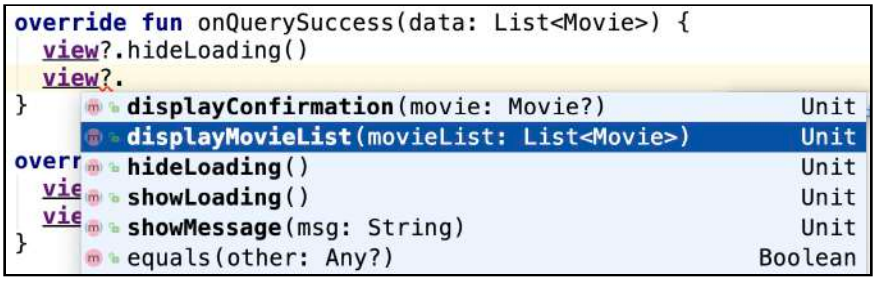
The logs tell you that `AddPresenter` interacted with the View but only with `hideLoading()`, not with `displayMovieList()`.

Open **SearchPresenter.kt** and look at `onQuerySuccess()`:

```
override fun onQuerySuccess(data: List<Movie>) {
  view?.hideLoading()
}
```

Yes, there's something wrong here. This method is telling the View to hide the progress bar. The trouble is, it never actually invokes any method to display the movie list passed as a parameter.

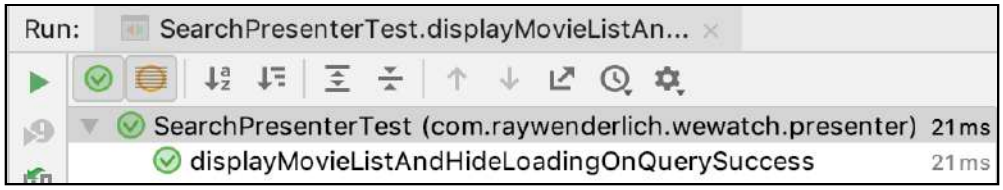
If you inspect the View's methods, you'll see that there's only one method that accepts a movie list as a parameter: `displayMovieList()`.



Inside `onQuerySuccess()`, add the following immediately below `hideLoading()`:

```
view?.displayMovieList(data)
```

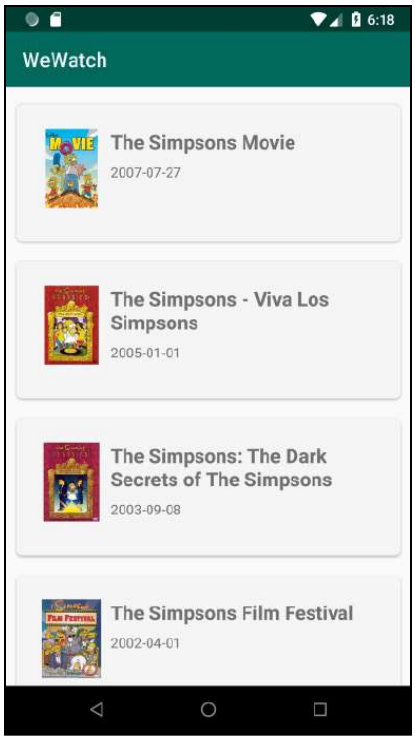
Run your test again to see if this solves the problem.



Excellent, the test succeeds! But now you need to make sure your app works.

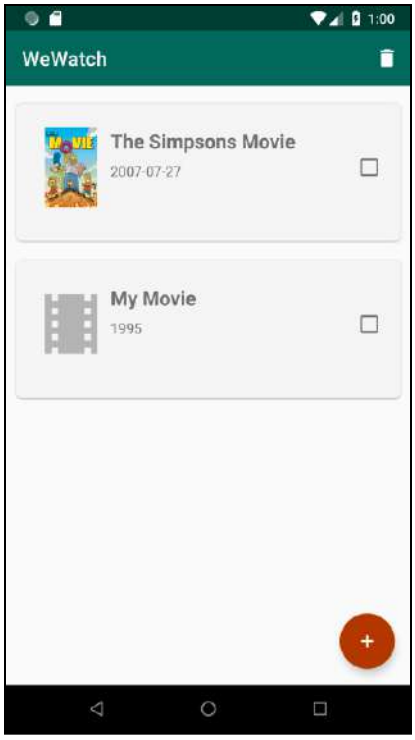
Build and run the app, and try searching for a movie of your choice.

The movies from the TMDB API are properly displayed:



Select any movie and click **OK** to save it.

The movie is also displayed in your main screen. Well done!



Chapter 17: MVI Theory

By Aldo Olivares Domínguez

You have learned about many different architecture patterns at this point, including **MVVM**, **MVI** and **MVC**.

In this chapter, you are going to learn about a very different architecture pattern for building Android apps called MVI.

Along the way, you will learn:

- What MVI is and how it works.
- The different layers of the MVI architecture pattern.
- How a unidirectional flow of an Android app works.
- How MVI improves the testability of your app by providing predictable and testable states.
- MVI advantages and concerns vs other architecture patterns.

What is MVI?

MVI stands for **Model-View-Intent**. MVI is one of the newest architecture patterns for Android. The architecture was inspired by the unidirectional and cyclical nature of the **Cycle.js** framework and brought to the Android world by **Hannes Dorfaman**.

MVI works in a very different way compared to its distant relatives such as **MVC, MVP or MVVM.** The role of each of its components goes as follows:

- **Model**: Represents a state. Models in MVI should be immutable to ensure a unidirectional data flow between them and the other layers in your architecture.

- **Intent**: Represents an intention or a desire to perform an action by the user. For every user action, an intent will be received by the View, which will be observed by the Presenter and translated into a new state in your Models.
- **View**: Just like in MVP they are represented by Interfaces, which are then implemented in one or more Activities or Fragments.

Next, you'll explore each of the layers one by one.

Model

In other architecture patterns such as MVVM or MVP, the Models act as a rather simple layer to hold your data and act as a bridge to the backend of your app such as your databases or your APIs. However, in MVI, Models have a much more important role; they not only hold data, but also represent the state of your app.

But... What is the state of your app?

As you learned in the **RxJava** chapter, reactive programming is a paradigm in which you react to a change, such as the value of a variable or a button click in your UI. Well, when your apps react to this change, they enter into a new **state**. The new state is usually, but not always, represented as a UI change with something like a progress bar, a new list of movies or a completely different screen.

To illustrate how Models work in MVI, imagine that you want to retrieve a list of the most popular movies from a web service such as the *IMDB*. In an app built with the usual MVP pattern, Models are a rather simple class that represent the data in your app such as this:

```
data class Movie(
  var voteCount: Int? = null,
  var id: Int? = null,
  var video: Boolean? = null,
  var voteAverage: Float? = null,
  var title: String? = null,
  var popularity: Float? = null,
  var posterPath: String? = null,
  var originalLanguage: String? = null,
  var originalTitle: String? = null,
  var genreIds: List<Int>? = null,
  var backdropPath: String? = null,
  var adult: Boolean? = null,
  var overview: String? = null,
  var releaseDate: String? = null
)
```

Then, the Presenter would be in charge of using the above-mentioned Model to display a list of movies with code like this:

```
class MainPresenter(private var view: MainContract.View?) :
MainContract.Presenter, MainContract.InteractorOutput {

  override fun onViewCreated() {
    view.showLoading()
    interactor
        .loadMovieList()
        .observe(view as MainActivity, Observer) {movieList ->
          movieList.let {
            this.onQuerySuccess(movieList)
          }
        }
  }

  override fun onQuerySuccess(data: List<Movie>) {
    view.hideLoading()
    view.displayMovieList(data)
  }
}
```

On the other hand, with an app built with the MVVM architecture pattern, something similar happens. The difference is that, instead of the Presenter, your ViewModel uses **RxJava** or **LiveData** to bind Observables to your UIs and display the data contained in your Models.

While this above approach is not bad, there are still a couple of issues that MVI attempts to solve:

- **Multiple Inputs**: In MVP and MVVM the Presenter and the ViewModel usually end up with a large number of inputs and outputs that have to be managed very carefully. This becomes a huge problem on big apps with a large number of background tasks tied to multiple Observables.
- **Multiple States**: With patterns such as MVP or MVVM, the business logic and the Views may have a different state at any point. You often synchronize the state with Observable/Observer callbacks but this may lead to a conflicting behavior if the synchronization is not handled properly. It becomes difficult to decide which is the correct state of your app at any given point... *Should I display a progress bar? Should I display a list of movies? What should I do?*

How do you solve the above issues? By making your Models represent a state rather than plain old data.

This is how you could create a Model that represents a state from the previous example:

```
sealed class MovieState {
  object LoadingState : MovieState()
  data class DataState(val data: List<Movie>) : MovieState()
  data class ErrorState(val data: String) : MovieState()
  data class ConfirmationState(val movie: Movie) : MovieState()
  object FinishState : MovieState()
}
```

When you model your Models like this, you no longer have to manage the state in multiple places such as your Views and the Presenters/ViewModel. They will indicate when your app should display a progress bar, an error message or a list of items.

Then, the Presenter for the above example would look like this:

```
class MainPresenter(private var view: MainContract.View?) :
MainContract.Presenter, MainContract.InteractorOutput {

  override fun onViewCreated() {
    view.render(MovieModel(true, null, null))
    interactor
        .loadMovieList()
        .observe(view as MainActivity, Observer) { movieList ->
          movieList.let {
            this.onQuerySuccess(movieList)
          }
        }
  }

  override fun onQuerySuccess(data: List<Movie>) {
    view.render(MovieModel(false, data, null))
  }
  private fun observeMovieDisplay() = movieInteractor.getMovieList()
      .observeOn(AndroidSchedulers.mainThread())
      .doOnSubscribe { view.render(MovieState.LoadingState) }
      .doOnNext { view.render(it) }
      .subscribe()
}
```

Your Presenter now only has one output: **the state of your View**. This is done with the View's `render()` method that accepts as an argument the current state for your app.

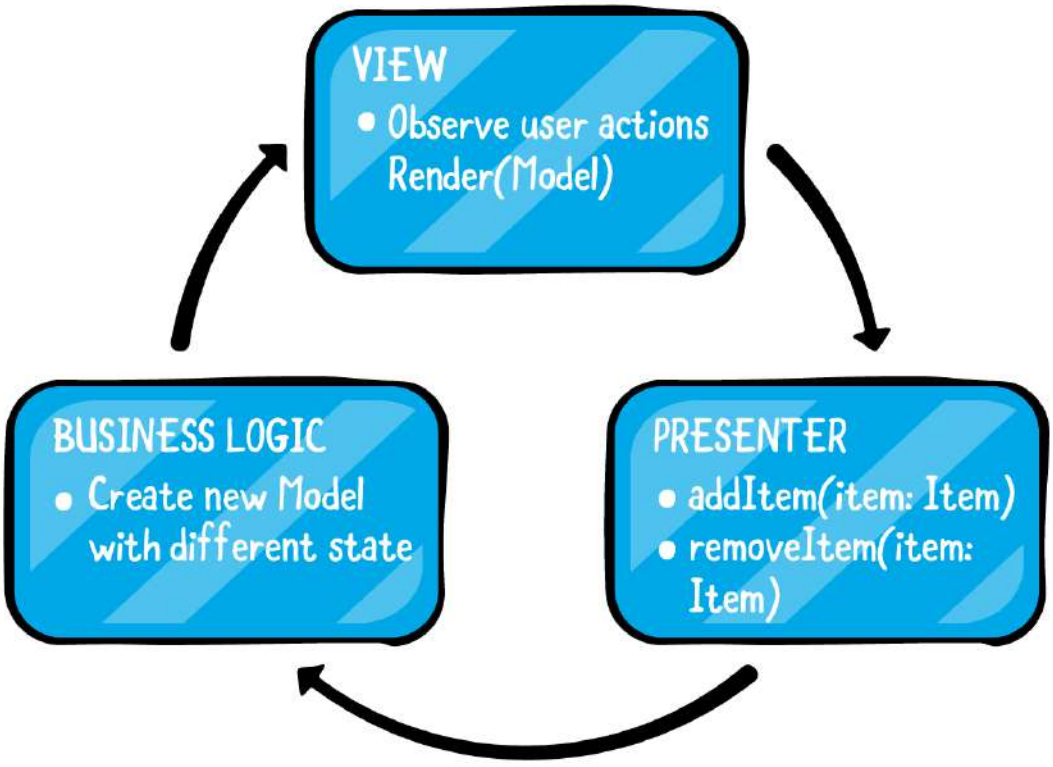
Another important and distinctive characteristic of the Models in MVI is that they should be immutable to maintain your business logic as the single source of truth.

This way, you are sure that your Models won't be modified in multiple places thus maintaining a single state during the whole lifecycle of your app.

To make things clearer, imagine that you want to add a new item to a list of todo items. This is how a typical implementation of MVI will work under the hood:

1. An Observable will send a notification to its subscribers about a new item being added to the list. Usually, this means adding a new record in a local and/or remote database.
2. A method in your Presenter, such as `presenter.addNewItem(item)`, will be called.
3. Your business logic will use your current Model to create a new Model which contains the new set of items. It is very important that you remember the immutability of your Models at this point since you can't just use the current Model to add the new item, you need to create a new one.
4. Your Presenter will then be notified about a new state in your app with an Observer.
5. Your Presenter will then call a `render()` method in your View and pass the new Model with the updated information as an argument.
6. Your View will display the new list of items based on the Model received from the Presenter.

The interaction between the different layers can be illustrated by the following diagram:



Do you notice something in particular about this diagram? If you said **cyclical flow**, you

are correct!

Thanks to the immutability of your Models, and the cyclical flow of your layers, you get other benefits:

- **Single State**: Since immutable data structures are very easy to handle and can only be managed in one place, you can be sure there will only be a single state between all the layers in your app.
- **Thread Safety:** This is specially useful while working with reactive apps that make use of libraries such as **RxJava** or **LiveData**. Since no methods can modify your Models they will always need to be recreated and kept in a single place, with this you make sure that there will be no other side effects such as different objects modifying your Models from different threads.

Of course the above are just hypothetical examples and you could model your Models and Presenters in a very different way, but the main premise is the same.

Now, take a look at the Views and Intents.

Views & Intents

In MVI, just like in in MVP, the Views are defined with the help of an Interface that acts as a contract which is implemented by a Fragment or an activity. The difference lies in the fact that Views in MVI tend to have a single `render()` method that accepts a state to render to the screen and different `intent()` methods as Observables that respond to user actions.

The intents in MVI don't represent the usual `android.content.Intent` class that is used for things like starting a new class. Intents in MVI represent an action to be performed that is translated to a change of the state in your app. For this simple example you only have one intent, the `getItemsIntent()`, but you can have any number of intents in your Views depending on the number of actions.

This is how an Activity would implement the `MainView` interface from above:

```
class MainActivity : MainView {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
  }

  //1
  override fun getItemsIntent() = button.clicks()

  //2
  override fun render(state: ViewState) {
```

```
        when(state) {
            is ViewState.DataState -> renderDataState(state)
            is ViewState.LoadingState -> renderLoadingState()
            is ViewState.ErrorState -> renderErrorState(state)
        }
    }

    //4
    private fun renderDataState(dataState: ViewState.DataState) {
        //Render Data on Screen
    }

    //3
    private fun renderLoadingState() {
        //Render Loading indicator on screen
    }

    //5
    private fun renderErrorState(errorState: ViewState.ErrorState) {
        //Render Error on Screen
    }
}
```

Taking each commented section in turn:

1. `getItemsIntent()`: Binds UI actions to the appropriate intents. In this case, we are binding a button click Observable to the getItemsIntent() method.
2. `render()`: Maps your ViewState to the correct methods in your View.
3. `renderDataState()` Renders the data contained in your Model to your View. This data can be anything such as weather data, a list of movies or an error.
4. `renderLoadingState()`: Renders a loading screen in your View.
5. `renderErrorState()`: Renders an error message in your View.

As you can see in the example above, you only have one `render()` method that receives the state of your app from your Presenter and an Intent that is triggered by a button click. This state is then translated into a UI change such as an error message or a loading screen.

State Reducers

With your usual mutable Models it is very easy to change the state of your app. Whenever you need to add, remove or update some underlying data you just need to call a method in your Models such as this:

```
myModel.addItems(newItems)
```

But you already learned that, since your Models are immutable, they need to be recreated each time the state of your app changes. If you just want new data to be displayed you can just create a new Model, but what do you do when you need information from a previous state?

This is where **State Reducers** come to save the day.

State Reducers are a concept derived from the reducer functions in reactive programming. Reducer functions, to put it simply, are functions that provide steps to properly merge things into a single component called the accumulator.

Since reducer functions are such a handy tool for developers most standard libraries have a similar method already implemented for their data structures. Kotlin's Lists, for example, include a `reduce()` method that accumulates a value starting with the first element of the list and applying the operation passed as an argument:

```
val myList = listOf(1, 2, 3, 4, 5)
var result = myList.reduce { accumulator, currentValue ->
  println("accumulator = $accumulator, currentValue = $currentValue")
  accumulator + currentValue
}
println(result)
```

Output:

```
accumulator = 1, currentValue = 2
accumulator = 3, currentValue = 3
accumulator = 6, currentValue = 4
accumulator = 10, currentValue = 5
15
```

The above code iterates over each element of `myList` using the `reduce` method and adds each element to the current accumulator value.

Reducer functions basically consist of two main components:

- **Accumulator** The total value accumulated so far in each iteration of your reducer function. It is usually the first argument.
- **Current Value** The current value passing through each iteration of your reducer function. It is usually the second argument.

Easy right?

But what does all of this have to do with State Reducers and MVI?

Well, State Reducers work in a very similar way to reducer functions, the main difference lies in that State Reducers create a new state for your app based on a previous state and a current state that holds the new changes.

The overall process goes as follows:

- You create a new state that represents the new changes in your app. This state is usually called **PartialState**.
- When there is a new Intent that requires a previous state of your app as a starting point you will create a new **PartialState** rather than a *complete* state.
- You create a new `reduce()` method that takes a previous state and a **PartialState** as arguments and defines how to merge both into a new state to be displayed.
- You will then use the **RxJava** `scan()` method to apply your `reduce()` method to the initial state of your app and return the new state.

Naturally, it is up to each developer to implement a reducer function to properly merge two states of the current app. However, it is a common approach to use **RxJava** scan/merge operators to help with this task.

MVI Advantages and Concerns

Just like the previous patterns, **Model-View-Intent** is just an additional tool that you have at your disposal to create maintainable and scalable apps.

The main advantages of MVI are:

- A unidirectional and cyclical data flow for your app.
- Consistent state during the whole lifecycle of your Views.
- Immutable Models that provide a reliable behavior and thread safety on big apps.

Probably the only downside of using MVI rather than other architecture patterns for Android is that the learning curve for this pattern tends to be a bit higher since you need to have a decent amount of knowledge of other intermediate/advanced topics such as reactive programming, multi-threading and RxJava. Therefore, other architecture patterns such as MVC or MVP might be easier to grasp for beginner Android developers.

Frequently Not Asked MVI Questions

Q. MVI and MVP look very similar...What is the main difference between the two patterns?

Both patterns rely on similar components such as Presenter, Views and Models. The main difference lies in the way those components are implemented and interact with each other in your app. For instance, the Models in MVI represent a state, rather than just data and the Views in MVI tend to have a single render() method that receives the state from the Presenter which is then mapped to the appropriate actions.**

Q. Is there an actual Intent layer?

It depends on what you mean by layer. As explained in this chapter, Intent in MVI represents an intention to do something like a database update or a web service call. You won't typically find an Intent package or class in your MVI apps.

Q. Is it completely necessary to use RxJava in MVI?

A. No, it is not necessary to use a reactive programming library such as RxJava to create apps with the MVI architecture pattern. However, they will make your life much easier when you need to react to UI actions and observe for state changes in your Models.

Q. Has anyone actually asked you these questions before?

A. Nope, but someone might, and I want to be ready!

Key points

- MVI stands for Model-View-Intent.
- Models in MVI represent a state of your app.
- The state represents how your app behaves or reacts at any given moment such as a loading screen, new data about to be displayed on a list or even a network error.
- Views in MVI can have one or more `intent()` methods that handle user actions and a single `render()` method that renders the state of your app.
- The Intent represents an intention to perform an action by the user like an API call or a new query in your database. It does NOT represent the usual `android.content.Intent`.

- Reducer functions are functions that provide steps to properly merge things into a single component called the **accumulator**.
- MVI provides a unidirectional and cyclical data flow for your app.
- MVI relies on intermediate/advanced android topics such as reactive programming, multi-threading and **RxJava**. Therefore, it might be harder to learn for beginner developers compared to other patterns such as MVC or MVP.

Where to go from here?

MVI is a powerful architecture pattern that relies on a unidirectional data flow and immutable Models to solve common concerns across Android development such as the state problem and thread safety.

Since understanding **RxJava** is a very important prerequisite to MVI you might want to take a look at the **RxJava** chapters if you haven't done so already. Also, make sure to checkout the **MVP Theory** and **MVP Sample** chapters if you haven't used that pattern in the past.

In the next chapter, you will apply your newly acquired knowledge to rewrite the **WeWatch** app to use the MVI architecture pattern.

Chapter 18: MVI Sample

By Aldo Olivares

In the previous chapter you learned all the theory behind the **M**odel-**V**iew-**I**ntent architecture pattern. You learned how **MVI** works and how each of its layers interact with each other.

Now you are going to use your new knowledge to rebuild the movies app using **MVI**. Along the way, you will also learn:

- How to create an **RxJava** `Observable` from scratch.
- How to create **RxJava** `Observables` from previously created objects with methods such as `just()`.
- How to use a `PublishSubject`.
- How to use a `Disposable` and a `CompositeDisposable`.
- How to integrate **Room** with **RxJava**.
- How to integrate **Retrofit** with **RxJava.**

Ready? Lets get started.

Getting started

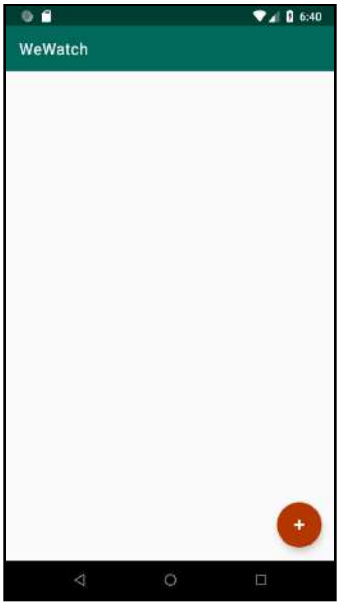
Start by opening the starter project for this chapter. The project contains the following packages:

- **data**: contains the **Room** database components such as your **DAOS** and your **MovieDatabase**, your **Models** and your **RetrofitClient**.
- **domain**: contains the **MovieState** class which you will use to represent the state of your **App**.
- **view**: contains the activities and adapters.

Take some time to familiarize yourself with the code because you will be spending a lot of time with each file during this chapter.

> **Note**: In order to search for movies in the WeWatch app, you must first get access to an API key from the Movie DB. To get your API own key, sign up for an account at www.themoviedb.org. Then, navigate to your account settings on the website, view your settings for the API, and register for a developer API key. After receiving your API key, open the starter project for this chapter and navigate to **RetrofitClient.kt**. There, you can replace the existing value for `API_KEY` with your own.

Build and **Run** the **App** to verify that everything is working properly.



Right now it is just an empty canvas, but that is about to change.

Going Reactive

One of the advantages of working with popular libraries developed by reliable sources such as Google is that they often include support for other popular libraries from the Android community such as **RxJava**.

For the **WeWatch** App you are using two main libraries for your backend: **Retrofit** and **Room**.

Room was developed by Google as the data persistence solution for their new **Architecture Components** suite of libraries for **Jetpack**. Since the developers behind **Room** are well aware that most Android Developers will use **Room** alongside **RxJava** they took the time to develop an extension that lets you return `Observable` callbacks when the user **creates**, **reads**, **updates** or **deletes** a record in the database.

On the other hand, since **Retrofit** is probably the most popular library to interact with web services, the developers at Square created an adapter github.com/JakeWharton/retrofit2-rxjava2-adapter to integrate Retrofit with **RxJava** a long time ago. However, thanks to the popularity of his adapter, when **Retrofit** 2.2 came out it included first party support for **RxJava** as well.

In both cases, you only need to add a few lines of code to your app level **build.gradle** file to take advantage of the reactive capabilities of **RxJava**.

Add the following to dependencies block of your app level **build.gradle** file:

```
implementation "com.squareup.retrofit2:adapter-rxjava2:$retrofitVersion"
implementation "androidx.room:room-rxjava2:$roomVersion"
```

Now that **Room** and **Retrofit** are ready to work with **RxJava** you need to modify your classes to return `Observables`.

Let's start with **Retrofit**. Open the **RetrofitClient.kt** file under the **data/net** package and modify your `Retrofit.Builder()` call:

```
val retrofit = Retrofit.Builder()
    .baseUrl(TMDB_BASE_URL)
    .addConverterFactory(GsonConverterFactory.create())
    .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
    .client(okHttpClient)
    .build()
```

This adds a `RxJava2CallAdapterFactory` which let's you return an `Observables` from your service methods.

Now open the **MovieApi.kt** file and modify the `searchMovies()` method inside the same file to return an `Observable`:

```
@GET("search/movie")
fun searchMovie(@Query("api_key") api_key: String, @Query("query") q:
String): Observable<MoviesResponse>
```

Next, return to the **RetrofitClient.kt** file and update the `searchMovies()` method there to return an `Observable` as well:

```
fun searchMovies(query: String): Observable<MoviesResponse> {
  return moviesApi.searchMovie(API_KEY, query)
}
```

That is all you need to do to make **Retrofit** return an **Observable** each time your user calls the `searchMovies()` method. It is time for **Room** to be reactive too!

The **Room** integration with **RxJava** allows different return types according to the operation being performed in your Database like **Insert**, **Delete**, **Update** or **Query**.

Insert operations in **Room** allow you three different return types:

- `Completable` where `onComplete()` is called as soon as the insertion is successful. This is useful when you don't need to know the ID of the new item inserted and there are no extra actions that need to be performed.
- `Single<Long>` or `Maybe<Long>` where `onSuccess()` is called as soon as the insertion is successful and the `Long` value represents the ID of the item inserted. This is useful when you want to perform an additional operation immediately after the item is added to your database.
- `Single<List<Long>>` or `Maybe<List<Long>` same as above, but the value emitted on `onSuccess()` is the list of rows inserted.

Update/Delete operations allow you two different return types:

- `Completable` where `onComplete()` is called as soon as the update/delete is completed with no additional value returned.
- `Single<Integer>` or `Maybe<Integer>` where `onSuccess()` is called as soon as the insertion/update is completed and the `Integer` value emitted represents the number of rows affected.

Query operations allow you three different return types:

- `Maybe` where the value emitted on `onSuccess()` is the object returned from your database. If there are no records in your database that match your query `Maybe` will complete. If the object is later updated nothing will happen.
- `Single` where the value emitted on `onSuccess()` is the object returned from your database. If there are no records in your database that match your query `Single` will call `onError(EmptyResultSetException.class)`. If the object is updated nothing will happen.
- `Flowable/Observable` where the value emitted on `onNext()` is the object returned from your database. If there are no records in your database that match your query `Flowable/Observable` won't emit anything, neither `onNext()`, nor `onError()`. If the object is later updated `Observable/Flowable` will automatically emit the updated object on `onNext()`, allowing you to take the appropriate action in your App.

> **Note**: Pay special attention to the **Query** operation because depending on the return type the behavior will be **very different**. For example, if you only need to retrieve a single value from your database and display it to your user `Maybe` might be the best choice. On the other hand, if you need to keep your UI updated `Flowable/Observable` might be the way to go since your `Observable` will keep emitting the new objects being added or modified in your database.

If you want to learn more about the return types of your Room database this article medium.com/androiddevelopers/understanding-migrations-with-room-f01e04b07929 from Google's **Android Developers** blog might be very useful for you.

Open the **MovieDao.kt** file and modify the class methods like below:

```
//1
@Insert(onConflict = OnConflictStrategy.REPLACE)
fun insert(movie: Movie): Single<Long>
//2
@Query("select * from movie")
fun getAll(): Observable<List<Movie>>
//3
@Delete
fun delete(movie: Movie): Completable
```

Let's walk through the above methods:

1. `insert()` helps you create a new movie record in your Database and returns a `Single` with the id of the new record created.

2. `getAll()` returns all the movies stored in your database inside an `Observable`. Since this is a `Query` operation every time a new record is inserted, you will receive a notification through `onNext()` and you can update your list of movies accordingly.

3. `delete()` deletes the movie object passed as a parameter from your database and calls `onComplete()` as soon as the row has been deleted.

Now that **Room** and **Retrofit** are reactive and ready to be used it is time to create your **Interactor**.

Creating Interactors and State

Room and **Retrofit** are now returning `Observable` callbacks, but they are only emitting plain old objects such as `Movie` or `Long`. In the last chapter you learned that one of the main concepts of **MVI** is that you need to represent the **State** of your App such as **Loading**, **Error** or **Data** based on your models. You will use an **Interactor** as a way to interact with **Retrofit** and **Room** and to transform the responses into the corresponding **State**.

Open the **MovieState.kt** file under the **data/domain** package and add the following inside your `MovieState` class:

```
sealed class MovieState {
  object LoadingState : MovieState() //1
  data class DataState(val data: List<Movie>) : MovieState() //2
  data class ErrorState(val data: String) : MovieState() //3
  data class ConfirmationState(val movie: Movie) : MovieState() //4
  object FinishState : MovieState() //5
}
```

Here you are using Kotlin's sealed classes to represent 5 different and mutually exclusive **States** of your App:

1. **LoadingState** as the name implies, represents the **State** in which your App is loading data, either from Room or from the TMDB API using **Retrofit**.

2. **DataState** represents the **State** in which your App is ready to display some data to the user, in this case a list of movies.

3. **ErrorState** represents an error in your App such as a Network Error.

4. **ConfirmState** is used to confirm that the user wants to save a new movie in your **Room** database.

5. **FinishState** used when a **View** is done with its tasks.

Now that your `MovieState` class is ready, it's time to create your **Interactor**.

Create a new class under the **data** package, name it **MovieInteractor** and make it implement the `Interactor` interface like below:

```
class MovieInteractor : Interactor
```

Make sure to implement all the missing members using **Ctrl+ I**.

Add the following properties:

```
private val retrofitClient = RetrofitClient()
private val movieDao = db.movieDao()
```

This creates instances of your `RetrofitClient` and your `MovieDao` classes to interact with your **Room** database and the TMDB API.

Now implement the overridden methods like this:

```
//1
override fun getMovieList(): Observable<MovieState> {
  return movieDao.getAll()
      .map<MovieState> { MovieState.DataState(it) }
      .onErrorReturn {
        MovieState.ErrorState("Error")
      }
}
//2
override fun deleteMovie(movie: Movie): Observable<Unit> =
movieDao.delete(movie).toObservable()
//3
override fun searchMovies(title: String): Observable<MovieState> =
retrofitClient.searchMovies(title)
    .observeOn(Schedulers.io())
    .map<MovieState> { it.results?.let { MovieState.DataState(it) } }
    .onErrorReturn { MovieState.ErrorState("Error") }
//4
override fun addMovie(movie: Movie): Observable<MovieState> =
movieDao.insert(movie)
    .map<MovieState> {
      MovieState.FinishState
    }.toObservable()
```

Step by step:

1. `getMovieList()` uses your `movieDao.getAll()` method to retrieve the list of movies from your database. The `map()` method helps you transform the list of movies to a `MovieState.DataState`. In case of an error you return a `MovieState.ErrorState`.

2. `deleteMovie()` uses your `movieDao.delete()` method to delete a `Movie` object from your database and transforms the `Complete` callback to an `Observable` callback.

3. `searchMovies()` uses your `RetrofitClient`'s `searchMovies()` method to retrieve a list of movies from the TMDB API that match the given title passed as a parameter. The `map()` operator transforms the response into a MovieState. Just like `getMovieList()`, you return a `MovieState.ErrorState` in case of a problem with the web service.

4. `addMovie()` uses your `movieDao.insert()` method to insert a new movie in your database. As soon as the insertion is successful you return an `Observable` with a `MovieState.FinishState`.

And that's it! You now have a `MovieInteractor` that transforms your callbacks into the appropriate **State** for your App. The next step is to create your **Presenters**.

Creating the Presenters

To connect the **View** with an **Interactor** you are going to need a **Presenter** for each of your activities.

Creating the MainPresenter

Create a new package under the root directory named **presenter**. Inside this package create a new Kotlin class named `MainPresenter`.

`MainPresenter` should accept a `MovieInteractor` as a constructor argument like below:

```
class MainPresenter(private val movieInteractor: MovieInteractor)
```

Then, add the following properties:

```
private lateinit var view: MainView
private val compositeDisposable = CompositeDisposable()
```

In order to render the `MovieState`, you'll need a reference to a **MainView**. The `CompositeDisposable` is needed to dispose of your `Disposables` once your `MainPresenter` is destroyed.

What is a Disposable?

You will be reading the word `Disposable` a lot over the next sections, but what is a Disposable? A `Disposable` is the object returned by the `subscribe()` method each time you subscribe to an `Observable`. This object represents a reference to a disposable resource with which the caller can use to stop receiving events.

To put it simply, you can use this object to tell your subscribers that they can stop receiving items even if the `Observable` is still emitting objects.

Next, add the following code to your `MainPresenter`:

```
//1
fun bind(view: MainView) {
  this.view = view
  compositeDisposable.add(observeMovieDeleteIntent())
  compositeDisposable.add(observeMovieDisplay())
}
//2
fun unbind() {
  if (!compositeDisposable.isDisposed) {
    compositeDisposable.dispose()
  }
}
//3
private fun observeMovieDeleteIntent() = view.deleteMovieIntent()
    .subscribeOn(AndroidSchedulers.mainThread())
    .observeOn(Schedulers.io())
    .flatMap<Unit> { movieInteractor.deleteMovie(it) }
    .subscribe()
//4
private fun observeMovieDisplay() = movieInteractor.getMovieList()
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { view.render(MovieState.LoadingState) }
    .doOnNext { view.render(it) }
    .subscribe()
```

Taking each commented section in turn:

1. `bind()` lets you bind the corresponding `MainView` to this `MainPresenter` in your Activity's `onCreate()` method. This method is also used to add your `Disposable`'s to your `CompositeDisposable`.

2. `unbind()` is called when your `MainView` is destroyed. This method disposes of all the `Disposable`'s added to your `CompositeDisposable`.

3. `observeMovieDeleteIntent()` is called when the `MainView` sends an **Intent** to delete a movie. This method uses **RxJava** `flatMap()` operator to call the `movieInteractor.deleteMovie()` method to delete a movie from the database. As you can see, you used the `observeOn()` operator to specify that delete operation of your database has to be done on a worker thread rather than the UI thread.

4. `observeMovieDisplay()` calls the `mainInteractor.getMovieList()` method to retrieve the list of saved movies from your database. When you subscribe to this `Observable` you will use the `doOnSubscribe()` method to tell the View to render the `MovieState.LoadingState`. As soon as there is a response with a `MovieState` you will call the `view.render()` method. Since your `MainView` has to render the **State** on the main thread you have to specify it using the `observeOn()` operator.

> **Note:** You might notice that you didn't have to specify a worker thread for the `movieInteractor.getMovieList()` method. This is because the `getAll()` method of your `MovieDao` returns an `Observable` and according to the documentation **all Observable queries are done off the main thread**. On the other hand for **Insert**, **Update** and **Delete** operations that return `Single`, `Completable` or `Maybe` you need to use the `observeOn()` operator to specify the Scheduler on which the operation will take place.

Creating the AddPresenter

Create a new class under the **presenter** package and name it **AddPresenter**. Replace everything inside with the following:

```
class AddPresenter(private val movieInteractor: MovieInteractor) {
  //1
  private val compositeDisposable = CompositeDisposable()
  private lateinit var view: AddView
  //2
  fun bind(view: AddView) {
    this.view = view
    compositeDisposable.add(observeAddMovieIntent())
  }
  //3
  fun unbind() {
    if (!compositeDisposable.isDisposed) {
      compositeDisposable.dispose()
    }
  }
  //
  fun observeAddMovieIntent() = view.addMovieIntent()
      .observeOn(Schedulers.io())
      .flatMap<MovieState> { movieInteractor.addMovie(it) }
      .observeOn(AndroidSchedulers.mainThread())
      .subscribe { view.render(it) }
}
```

The code for the `AddPresenter` class is very similar to the `MainPresenter`:

1. You need an instance of your `AddView` to render the state of your App and a `CompositeDisposable` to dispose of your `Disposables` when your `AddView` is destroyed.
2. `bind()` is used to get an instance of your `AddView` and to add your `Disposables` to your `CompositeDisposable`.
3. `unbind()` disposes of all your active `Disposables` when your `AddView` is destroyed.
4. `observeAddMovieIntent()` is called when `AddView` sends an **Intent** to save a movie. The `flatMap()` operator is used to call the `addMovie()` method of your `MovieInteractor` to add a record to your database. `observeOn()` is used to specify that `addMovie()` should be called on a worker thread while `view.render()` should be executed on the UI thread.

Creating the SearchPresenter

Create a new class under the **presenter** package and name it **SearchPresenter**. Replace everything inside with the following:

```
class SearchPresenter(private val movieInteractor: MovieInteractor) {
  //1
  private val compositeDisposable = CompositeDisposable()
  private lateinit var view: SearchView
  //2
  fun bind(view: SearchView) {
    this.view = view
    compositeDisposable.add(observeMovieDisplayIntent())
    compositeDisposable.add(observeAddMovieIntent())
    compositeDisposable.add(observeConfirmIntent())
  }
  //3
  fun unbind() {
    if (!compositeDisposable.isDisposed) {
      compositeDisposable.dispose()
    }
  }
  //4
  private fun observeConfirmIntent() = view.confirmIntent()
      .observeOn(Schedulers.io())
      .flatMap<MovieState> { movieInteractor.addMovie(it) }
      .observeOn(AndroidSchedulers.mainThread())
      .subscribe { view.render(it) }

  //5
  private fun observeAddMovieIntent() = view.addMovieIntent()
      .map<MovieState> { MovieState.ConfirmationState(it) }
      .subscribe { view.render(it) }
```

```
    //6
    private fun observeMovieDisplayIntent() = view.displayMoviesIntent()
        .flatMap<MovieState> { movieInteractor.searchMovies(it) }
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .doOnSubscribe { view.render(MovieState.LoadingState) }
        .subscribe { view.render(it) }
}
```

1. You need an instance of your `SearchView` to render the **State** and a `CompositeDisposable`.
2. `bind()` attaches your `SearchView` instance to this class and adds all your `Disposables` to your `CompositeDisposable`.
3. `unbind()` is called when `SearchView` is destroyed and uses your `CompositeDisposable` to dispose of all the `Disposables` added to it.
4. `observeConfirmIntent()` is called when the `SearchView` sends a confirmation Intent. The `flatMap()` operator is used to call the `addMovie()` method of your `MovieInteractor` and map the result to a `MovieState`.
5. `observeAddMovieIntent()` is called when your `SearchView` sends an Intent that the user wants to add a new movie to the database. The `map()` operator is used to transform the result into a `ConfirmationState` .
6. `observeMovieDisplayIntent()` is called when your `SearchView` wants to display a list of movies from the TMDB API that match the title passed as a parameter. The `flatmap()` operator is used to call the `searchMovies()` method of your `MovieInteractor` and map the result to a `MovieState`.

Creating the Views

Now that the **Presenter** layer of your **MVI** architecture is ready, creating the **Views** should be a piece of cake :]

Creating the MainView

Open the **MainActivity.kt** file under the **view/activity** package. Right now this is an empty activity with a **toolbar** and a simple method to navigate to the `AddMovieActivity` once the user presses the + button, but that is about to change.

Start by making your `MainActivity` class implement the `MainView` interface like below:

```
class MainActivity : BaseActivity(), MainView {
```

The `MainView` has a simple contract given below:

```
interface MainView {
  fun render(state: MovieState)
  fun deleteMovieIntent(): Observable<Movie>
}
```

Make sure to implement all missing members by pressing **Ctrl + I** on your keyboard.

Add the following property to store a reference to your `MainPresenter`:

```
private lateinit var presenter: MainPresenter
```

And initialize it inside your `onCreate()` method:

```
presenter = MainPresenter(MovieInteractor())
presenter.bind(this)
```

The above code creates a new `MainPresenter` instance and uses its `bind()` method to send a reference to this **View**.

Next you'll add some interactive controls to the list. You'll make use of the left or right swipe gestures to delete a movie. Add the following code inside the `deleteMovieIntent()` method:

```
//1
val observable = Observable.create<Movie> { emitter ->
  //2
  val helper = ItemTouchHelper(
      object : ItemTouchHelper.SimpleCallback(
          0,
          ItemTouchHelper.LEFT or ItemTouchHelper.RIGHT
      ) {
        //3
        override fun onMove(
            recyclerView: RecyclerView,
            viewHolder: RecyclerView.ViewHolder,
            target: RecyclerView.ViewHolder
        ): Boolean {
          return false
        }
        //4
        override fun onSwiped(viewHolder: RecyclerView.ViewHolder,
direction: Int) {
          val position = viewHolder.getAdapterPosition()
          val movie = (moviesRecyclerView.adapter as
MovieListAdapter).getMoviesAtPosition(position)
          emitter.onNext(movie)
        }
```

```
        })
    //5
    helper.attachToRecyclerView(moviesRecyclerView)
  }
  return observable
```

The above code creates a new `Observable` and emits the movie item that needs to be deleted every time the user swipes left or right. Step by step:

1. You use the `create()` method to create a new `Observable` instance that emits a `Movie` object.
2. Creates a new `ItemTouchHelper` object that can be attached to a `RecyclerView` to add swipe and move capabilities to each one of the items being displayed.
3. For this app you are not implementing the `onMove()` method.
4. Here you are overriding the `onSwiped()` method that lets you know when a certain item is being swiped left or right in your `RecyclerView`. In this case you use the `Emitter` `onNext()` method to send the item being deleted to your observers.
5. Finally, you attach the `ItemTouchHelper` to your `MoviesRecyclerView` using the `attachToRecyclerView()` method.

Add the following code inside your `render()` method:

```
when (state) {
  is MovieState.LoadingState -> renderLoadingState()
  is MovieState.DataState -> renderDataState(state)
  is MovieState.ErrorState -> renderErrorState(state)
}
```

For the `MainView` you are handling three different **States**: **Loading**, **Data** and **Error**. Each one of them will be handled by a different function that you will implement.

You might notice that Android Studio is telling you that the methods called on `render()` have not been created. Lets do that right now by adding the following code:

```
//1
private fun renderLoadingState() {
  moviesRecyclerView.isEnabled = false
  progressBar.visibility = View.VISIBLE
}
//2
private fun renderDataState(dataState: MovieState.DataState) {
  progressBar.visibility = View.GONE
  moviesRecyclerView.apply {
    isEnabled = true
    (adapter as MovieListAdapter).setMovies(dataState.data)
  }
}
```

```
//3
private fun renderErrorState(dataState: MovieState.ErrorState) =
longToast(dataState.data)
```

1. `renderLoadingState()` will display a `progressBar` and hide your `MoviesRecyclerView`.
2. `renderDataState()` will display your `MoviesRecyclerView` and hide the `progressBar`.
3. `renderErrorState()` will display a toast with the error message received from your `MainPresenter`.

The only step left to finish implementing your `MainView` is to call your `MainPresenter`'s `unbind()` method when this activity is no longer going to be used. To do this override your `MainActivitys` `onStop()` method by adding the following code:

```
override fun onStop() {
  presenter.unbind()
  super.onStop()
}
```

This will tell your `MainPresenter` that your `Disposable`'s can be disposed since your `Observables` will no longer emit any new items.

Creating the AddView

Open the **AddMovieActivity.kt** file under the **view/activity** package.

Similar to what you did for your `MainActivity`, make your `AddMovieActivity` class implement the `AddView` interface and implement the missing members by pressing **Ctrl + I**. The `AddView` should already be given in the starter project, but is repeated below for clarity:

```
interface AddView {
    fun render(state: MovieState)
    fun addMovieIntent(): Observable<Movie>
}
```

Add the following property to store a reference to your `AddPresenter`:

```
private lateinit var presenter: AddPresenter
```

And initialize your `AddPresenter` inside `onCreate()`:

```
presenter = AddPresenter(MovieInteractor())
presenter.bind(this)
```

The previous code just initializes the Presenter by creating an instance of your `MovieInteractor`, and binds it to this `AddView`.

Add this code inside `render()`:

```
when (state) {
  is MovieState.FinishState -> renderFinishState()
}
```

For `AddView` you are only handling one **State**, the **Finish State**.

Implement the **Finish** state by adding the following method:

```
private fun renderFinishState() =
startActivity(intentFor<MainActivity>().newTask().clearTask())
```

This code simply returns to the `MainActivity` after your new movie has been inserted into the database.

Now you only need to implement the **Intent** to add a new movie to the database. Start by adding the following property at the top of your class:

```
private val publishSubject: PublishSubject<Movie> =
PublishSubject.create()
```

What is a `PublishSubject`? Well according to the documentation:

> A Subject is a sort of bridge or proxy that is available in some implementations of ReactiveX that acts both as an observer and as an Observable. Because it is an observer, it can subscribe to one or more Observables, and because it is an Observable, it can pass through the items it observes by reemitting them, and it can also emit new items.

A `PublishSubject` in **RxJava** is basically the fusion of an `Observable` with an observer, it can subscribe to another Observable and emit items to its own subscribers. It is very useful in some circumstances where you can't easily modify the return type of certain functions and it can also help you simplify the code to create a new `Observable`.

In this case, your `PublishSubject` will emit `Movie` objects that have to be inserted in your **Room** database.

Use your new `PublishSubject` by adding the following code:

```
fun addMovieClick(view: View) {
  if (titleEditText.text.toString().isNotBlank()) {
    publishSubject.onNext(Movie(title = titleEditText.text.toString(),
releaseDate = yearEditText.text.toString()))
```

```
    } else {
      showMessage("You must enter a title")
    }
  }
```

This method is executed when the user clicks the `AddMovie` button in your UI. If `titleEditText` is not blank it will tell your `PublishSubject` to emit the new `Movie` object on `onNext()` to its subscribers, otherwise it will display an error message.

Your `PublishSubject` is ready to emit new `Movie` objects when the user presses the **Add Movie** button. Now you only need to return your `PublishSubject` by adding the following code to your `addMovieIntent()` method:

```
override fun addMovieIntent() = publishSubject
```

Finally, just like you did for your `MainView`, you need to call your `AddPresenter`'s `unbind()` method inside `onStop()`:

```
override fun onStop() {
  super.onStop()
  presenter.unbind()
}
```

Creating the SearchActivity

Open the **SearchMovieActivity.kt** file under the **view/activity** package. Make your `SearchMovieActivity` class implement the `SearchView` interface and implement all the missing members.

Add the following properties:

```
private lateinit var presenter: SearchPresenter
private val publishSubject: PublishSubject<Movie> =
PublishSubject.create<Movie>()
```

This creates a `SearchPresenter` instance and a `PublishSubject` that emits `Movie` objects.

Add this code inside `onCreate()` to initialize your `SearchPresenter` and bind your `SearchView`:

```
presenter = SearchPresenter(MovieInteractor())
presenter.bind(this)
```

Now, add the following inside `render()`:

```
when (state) {
  is MovieState.LoadingState -> renderLoadingState()
  is MovieState.DataState -> renderDataState(state)
  is MovieState.ErrorState -> renderErrorState(state)
```

```
    is MovieState.ConfirmationState -> renderConfirmationState(state)
    is MovieState.FinishState -> renderFinishState()
  }
```

For the `SearchView` you are handling 5 different **States**: **Loading**, **Data**, **Error**, **Confirm** and **Finish**.

Implement the functionality of each of the above **States** by adding the following methods:

```
  //1
  private fun renderFinishState() =
  startActivity(intentFor<MainActivity>().newTask().clearTask())
  //2
  private fun renderLoadingState() {
    searchRecyclerView.isEnabled = false
    searchProgressBar.visibility = View.VISIBLE
  }
  //3
  private fun renderConfirmationState(confirmationState:
  MovieState.ConfirmationState) {
    searchLayout.snack("Add ${confirmationState.movie.title} to your
  list?", Snackbar.LENGTH_LONG) {
      action(getString(R.string.ok)) {
        publishSubject.onNext(confirmationState.movie)
      }
    }
  }
  //4
  private fun renderDataState(dataState: MovieState.DataState) {
    searchProgressBar.visibility = View.GONE
    searchRecyclerView.apply {
      isEnabled = true
      (adapter as SearchListAdapter).setMovies(dataState.data)
    }
  }
  //5
  private fun renderErrorState(errorState: MovieState.ErrorState) {
    searchProgressBar.visibility = View.GONE
    longToast(errorState.data)
  }
```

1. `renderFinishState()` terminates the current activity and returns to the `MainActivity`.
2. `renderLoadingState()` displays the `progressBar` and hides the `searchRecyclerView`.
3. `renderConfirmationState()` displays a `SnackBar` asking the user to confirm the selected movie. If the user clicks the **OK** button in your `Snackbar` your `PublishSubject` emits a `Movie` object on `onNext()`.
4. `renderDataState()` hides the `progressBar` and displays the list of movies retrieved from the TMDB API in your `SearchRecyclerView`.

5. `renderErrorState()` displays a toast to the user with an error message.

The only step remaining to complete the `SearchView` is to create the Intents with the `Observables` for your subscribers.

Add the following methods to implement all the Intents for this **View**:

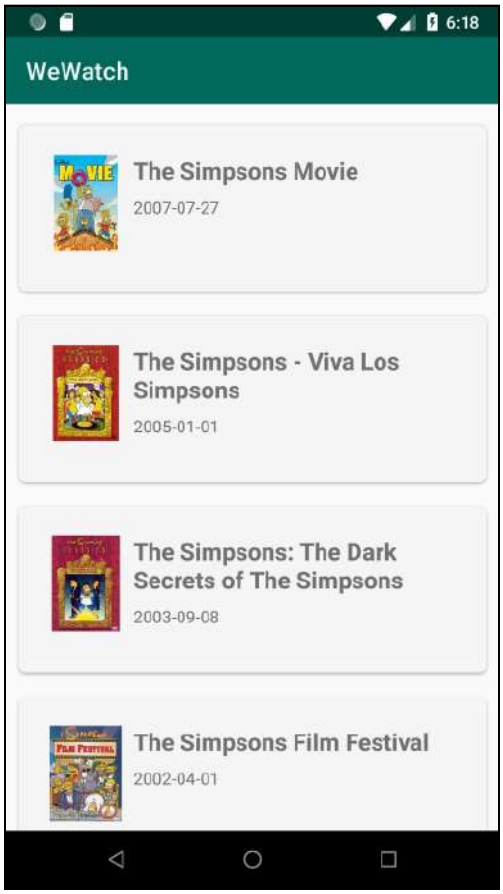
```
//1
override fun displayMoviesIntent(): Observable<String> =
Observable.just(intent.extras.getString("title"))

//2
override fun addMovieIntent(): Observable<Movie> =
(searchRecyclerView.adapter as
SearchListAdapter).getViewClickObservable()

//3
override fun confirmIntent(): Observable<Movie> = publishSubject
```

1. `displayMoviesIntent()` creates an `Observable` that emits the movie title passed to this Activity using the `just()` method.
2. `addMovieIntent()` creates an `Observable` that emits a `Movie` object each time an item is clicked in your `SearchRecyclerView`.
3. `confirmIntent()` returns your `PublishSubject`.

And finally, override your Activity's `onStop()` method to call your `SearchPresenter`'s `unbind()` just like you did for the previous Views:

```
override fun onStop() {
  super.onStop()
  presenter.unbind()
}
```

Your App is finally ready! **Build** and **Run** the App to see it in action:



Final thoughts

Take a closer look at all your Activities. As you can see, there is only one `render()` method that receives the current **State** from the **Presenter** and multiple Intents that represent the actions taken in your **UI**.

Also notice the unidirectional flow of information that this architecture provides by taking a look at your `SearchView` and the execution of `displayMoviesIntent()`:

Having a structure like this makes it very easy to debug your App and guarantees that there is only one **State** at any given moment.

Also, you might remember from the previous chapter that **State Reducers** are an important concept of the **MVI** architecture. However, we didn't have to use any State Reducers because the **WeWatch** App is rather simple and there were no **States** that relied on a previous **State** to be built.

Key points

- **Retrofit** and **Room** include native **RxJava** support. This allows you to create `Observable` callbacks to which you can subscribe and react.
- The **Room** integration with **RxJava** allows different return types according to the operation being performed in your Database.
- Insert operations in **Room** allow you three different return types: `Completable`, `Single<Long>` and `Single<List<Long>>`.
- Update and Delete operations in **Room** allow you two different return types: `Completable` and `Single<Integer>`.
- Query operations in **Room** allow you three different return types: `Maybe`, `Single` and `Observable`.
- Views in **MVI** have a single `render()` method that receives a state and renders it to the UI.
- Views in **MVI** have multiple Intent methods that represent an intention to do something.

Where to go from here?

For didactic purposes, you created most of your `Observables` for this App from scratch, but there are many libraries out there that help you create `Observables` from almost any kind of object.

One of the most popular libraries to create `Observables` from **Views** in Android is RxBinding github.com/JakeWharton/RxBinding. This library is very useful to create **RxJava** bindings for Android UI widgets such as buttons or textviews.

For example, if you wanted to create an `Observable` from a button click with traditional

RxJava code you would do it like this:

```
Observable.create { emitter ->
  button.setOnClickListener {
    emitter.onNext(queryEditText.text.toString())
  }
}
```

Instead, with **RxBinding** it is much simpler and easy to read:

```
button.clicks()
```

Another popular library is RxKotlin github.com/ReactiveX/RxKotlin, which adds convenient extension functions to **RxJava**. For example, take a look at the following code taken from the official repository:

```
val list = listOf("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
list.toObservable() // extension function for Iterables
    .filter { it.length >= 5 }
    .subscribeBy(  // named arguments for lambda Subscribers
            onNext = { println(it) },
            onError =  { it.printStackTrace() },
            onComplete = { println("Done!") }
    )
```

You can instantly convert any **Iterable** (like a List) to an `Observable` using the `toObservable()` extension function!

Why don't you take some time to rewrite some of the code in the App to use both libraries? It might be a fun challenge and you will learn a lot.

Chapter 19: MVI Debugging

By Aldo Olivares

In the previous chapter, you learned how to implement the MVI architecture pattern by rebuilding WeWatch. In this chapter, we'll skip the usual unit testing with JUnit and Mockito and instead you'll learn some helpful techniques for manually testing and debugging MVI and reactive code.

Along the way, you'll:

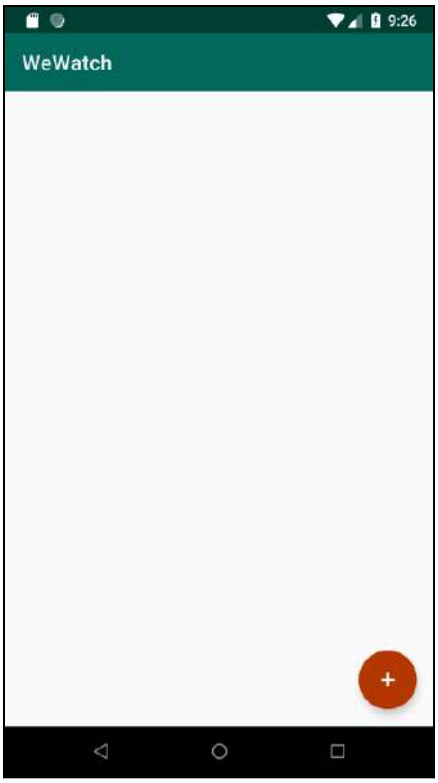
- Verify the execution of your Intents.
- Verify the flow of your architecture.
- Use Timber to log statements in Android.
- Verify your Observables.
- Use RxJava's `startWith()`.

Getting started

Start by opening the starter project for this chapter.

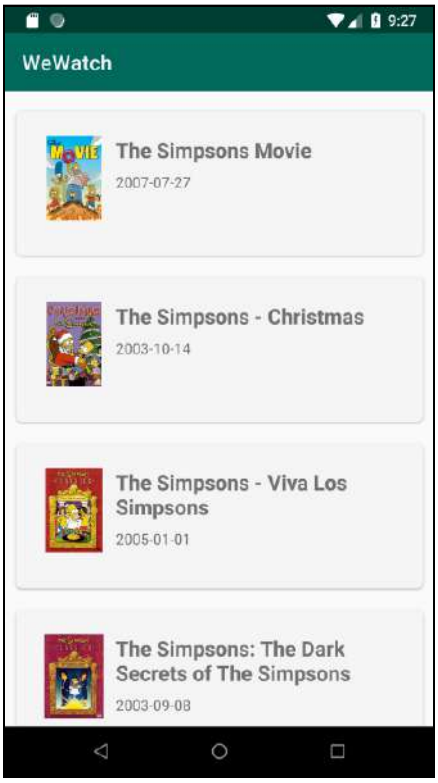
> **Note**: In order to search for movies in the WeWatch app, you must first get access to an API key from the Movie DB. To get your API own key, sign up for an account at www.themoviedb.org. Then, navigate to your account settings on the website, view your settings for the API, and register for a developer API key. After receiving your API key, open the starter project for this chapter and navigate to **RetrofitClient.kt**. There, you can replace the existing value for `API_KEY` with your own.

After Android Studio finishes building the project, run the app to see it in action.



Try adding a movie by pressing the + floating action button.

Enter a title and click the **search button**:



Select a movie and click **OK** on the `Snackbar` that appears:

So far, the app seems to be working fine, but you need to verify that the right Intents are getting sent and that the appropriate states are being returned.

Introducing Timber

Most developers use logs to debug their apps and test their code. To create a log statement, you typically use the `Log` class that comes with the Android SDK.

A typical log statement looks like this:

```
Log.d(TAG, "msg")
```

This code creates a log statement and displays it to the logcat console. `TAG` is typically a constant value that holds the class name that responsible for printing the statement. You can also set different priority levels like *verbose*, *debug* or *error* depending on your needs.

The problem with the traditional `Log` class is that when you release your app to the Play Store, you'll need to remove these statements so that no sensitive information, such as passwords or authentication tokens, are visible as plain text. A possible solution is to use **Control-F** to find every line that starts with `Log`, and then delete what you find. However, if your app contains thousands of lines of code, this might be a difficult and timely task. Besides, you might actually need those statements for debugging purposes later.

To solve this problem, some developers from Square created a handy library for **conditional based logging** named Timber.

Timber lets you display log statements only when they meet certain conditions, for example, if your app's current build is a **DEBUG** build. With Timber, you define the behavior of your logs by creating `Tree` instances, and use `Timber.plant()` to add them. You can use the default `DebugTree` that automatically determines which class is calling it and uses that classes name for the `TAG`.

To start using Timber, add the following line to your app-level **build.gradle**:

```
//Timber
def timberVersion = "4.7.1"
implementation "com.jakewharton.timber:timber:$timberVersion"
```

Keeping in line with the Timber documentation, you should create your Tree instances as soon as possible, preferably in the `onCreate()` of your `Application` class. You'll do that now.

Open **App.kt** and add the following code inside `onCreate()`:

```
if (BuildConfig.DEBUG) {
  Timber.plant(Timber.DebugTree())
}
```

That's all you need to do to start using Timber's enhanced log statements. So now, instead of doing something like this:

```
Log.d(TAG, "message")
```

You can use Timber to print statements — without having to worry about them showing up in production:

```
Timber.d("message")
```

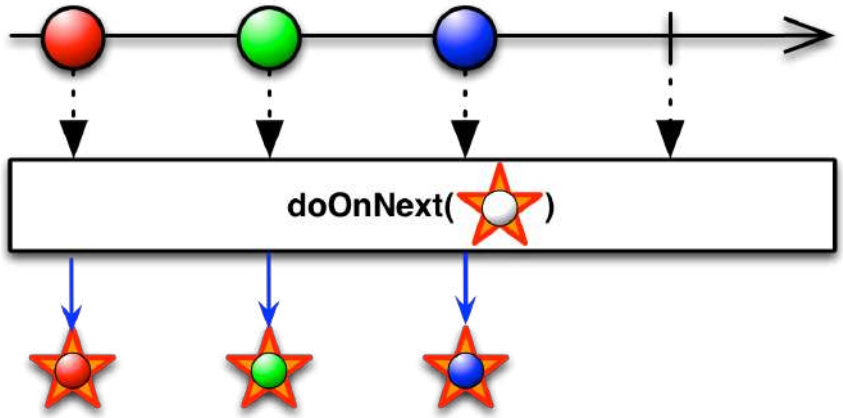
Now that timber is set up, you'll learn how to test your MVI architecture.

> **Note**: If you want to keep using the traditional `Log` class for this chapter — or in your own projects for that matter — that's up to you, but I highly recommend using this or another logging library of your choice. Log messages in production environments pose a significant security risk, and it's easy to forget to delete one of your logs, especially if your app has thousands of lines of code.

Testing the MVI architecture

Having an MVI architecture means you have predictable states that are triggered based on Intents. In other words, you have a unidirectional and cyclical flow for your app's data, and this makes it easier to detect errors because you'll know the last Intent that triggered as well as the state rendered before an exception occurs. However, to detect errors with this type of architecture you first need to make sure your app's states are flowing as expected.

To test your Intents, you'll use RxJava's `doOnNext()`, which modifies your `Observable` source to perform a certain action when it calls `onNext()`.



`doOnNext()` is the perfect choice to debug your app and add a log each time an Intent is triggered.

Inside **view/activity**, open **MainPresenter.kt** and modify `observeMovieDisplay()` and `observeMovieDelete()`, like so:

```
private fun observeMovieDeleteIntent() = view.deleteMovieIntent()
    .doOnNext { Timber.d("Intent: delete movie") }//Add this line
    .subscribeOn(AndroidSchedulers.mainThread())
    .observeOn(Schedulers.io())
    .flatMap<Unit> { movieInteractor.deleteMovie(it) }
    .subscribe()

private fun observeMovieDisplay() = movieInteractor.getMovieList()
    .doOnNext { Timber.d("Intent: display movie") }//Add this line
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { view.render(MovieState.LoadingState) }
    .doOnNext { view.render(it) }
    .subscribe()
```

Whenever there's an intent to display or delete a movie, `MainPresenter` will print a log message.

Now you need to know which states get displayed at any given point in your `MainView`. Open **MainActivity.kt** and modify `render()` so it matches this:

```
override fun render(state: MovieState) {
  Timber.d("State: ${state.javaClass.simpleName}")//Add this line
  when (state) {
    is MovieState.LoadingState -> renderLoadingState()
    is MovieState.DataState -> renderDataState(state)
    is MovieState.ErrorState -> renderErrorState(state)
  }
}
```

This code prints a log with the `MovieState` received from the `MainPresenter`.

Build and run the app. Look at the logcat console and you'll see what's happening under the hood:

```
D/MainActivity: State: LoadingState
D/MainPresenter$observeMovieDisplay: Intent: display movie
D/MainActivity: State: DataState
```

Notice something weird? The `LoadingState` is immediately displayed even before the display movies Intent gets triggered. Although this isn't horrible — because you're still achieving the desired behavior of displaying the loading state before the data state — the `LoadingState` should only be displayed after receiving the display movie's Intent.

If this type of problem exists here, it might also exist elsewhere. To sort out why this is happening, you'll add a log statement to the display intent.

Inside the **presenter** package, open **SearchPresenter.kt** and modify it, like so:

```
private fun observeMovieDisplayIntent() = view.displayMoviesIntent()
    .doOnNext { Timber.d("Intent: display movies") }//Add this line
    .flatMap<MovieState> { movieInteractor.searchMovies(it) }
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { view.render(MovieState.LoadingState) }
    .subscribe { view.render(it) }
```

Next, inside the **view/activity** package, open **SearchMovieActivity.kt** and add a log statement to `render()`:

```
override fun render(state: MovieState) {
  Timber.d("State: ${state.javaClass.simpleName}")
  when (state) {
    is MovieState.LoadingState -> renderLoadingState()
    is MovieState.DataState -> renderDataState(state)
    is MovieState.ErrorState -> renderErrorState(state)
    is MovieState.ConfirmationState -> renderConfirmationState(state)
    is MovieState.FinishState -> renderFinishState()
  }
}
```

Every time `render()` is called, this code prints a log with the `MovieState`.

Build and run the app. Try searching for a movie.

```
D/SearchMovieActivity: State: LoadingState
D/SearchPresenter$observeMovieDisplayIntent: Intent: display movies
D/SearchMovieActivity: State: DataState
```

As suspected, the same thing is happening to `SearchPresenter` and `SearchView`: The `LoadingState` is immediately rendered before the Intent is sent.

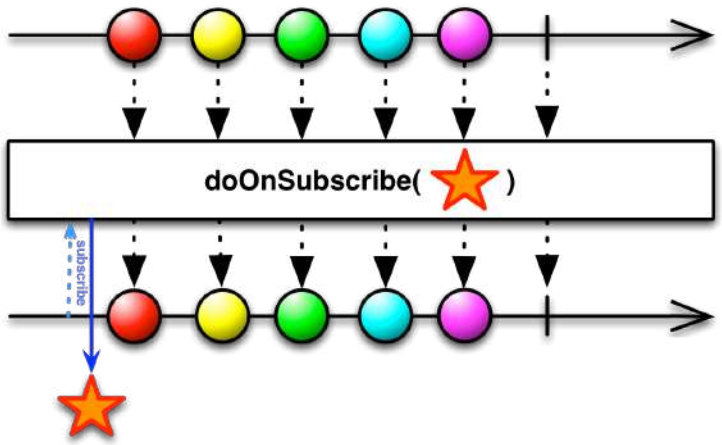
It seems there's a bug in the app that's rendering the `LoadingState` before emitting Intents. This is why testing is so crucial.

Open **MainPresenter.kt** again and look at `observeMovieDisplay()`:

```
private fun observeMovieDisplay() = movieInteractor.getMovieList()//1
    .doOnNext { Timber.d("Intent: display movie") }//2
    .observeOn(AndroidSchedulers.mainThread())//3
    .doOnSubscribe { view.render(MovieState.LoadingState) }//4
    .doOnNext { view.render(it) }//5
    .subscribe()//6
```

Take a moment to review this code:

1. `getMovieList()` retrieves the list of saved movies from the Room database and returns the result on your observable's `onNext()`.
2. `doOnNext()` adds a log message every time there's a response from the previous statement.
3. `observeOn()` changes the thread of all operators further downstream. This means that `doOnSubscribe()`, `doOnNext()` and `subscribe()` will get called from the main thread.
4. `doOnSubscribe()` executes the action passed as a parameter as soon as you subscribe to the `Observable` even before items are emitted. Look at the diagram following this explanation to see how.
5. In this case, you tell the View that you want to render the loading State **before** emitting an item.
6. Finally, `subscribe()` makes your subscriber start observing your observable's emissions.



Do you see the problem here? You're using `doOnSubscribe()` to make your View render the `LoadingState` before an item is emitted.

The second issue is that you're not observing any of the `MainView` Intents; you're only observing `getMovieList()` in the `MainInteractor`.

This may not be a big deal right now, because you don't need any information from the `MainView`, but in a traditional MVI architecture you want to react to your View's Intents before executing any actions.

Because the second problem is the easiest to solve, you'll start there. In the **root**, open **MainView.kt** and add the following method signature to your Interface:

```
fun displayMoviesIntent(): Observable<Unit>
```

You'll implement this method in the `MainView` to send an Intent that you want to display a list of movies.

Open **MainActivity.kt** and press **Control-I** to implement the missing members and select `displayMoviesIntent()`.

Now, add the following code to `displayMoviesIntent()`:

```
return Observable.just(Unit)
```

There are several ways to create an Observable that does not emit any items; one of them is to call `Observable.empty()`. The problem with `empty()` is that it immediately terminates and calls `onComplete()` without calling `onNext()`; this is not what you want because you won't be able to perform any additional actions.

On the other hand, `just()` is better in this case because it converts any item (including `Unit`) into an Observable and emits it on `onNext()`.

Open **MainPresenter.kt** and modify `observeMovieDisplay()`, like so:

```
private fun observeMovieDisplay() = view.displayMoviesIntent()//1
    .doOnNext { Timber.d("Intent: display movie") }
    .flatMap<MovieState> { movieInteractor.getMovieList() }//2
    .observeOn(AndroidSchedulers.mainThread())
    .doOnSubscribe { view.render(MovieState.LoadingState) }
    .doOnNext { view.render(it) }
    .subscribe()
```

1. Instead of directly reacting to `getMovieList()` from the `MovieInteractor`, you're going to react to `displayMoviesIntent()` of your `MainView`.
2. Since `displayMoviesIntent()` is immediately calling `onNext()`, you're going to use the `flatMap()` operator to call `getMovieList()` of the `MovieInteractor`.

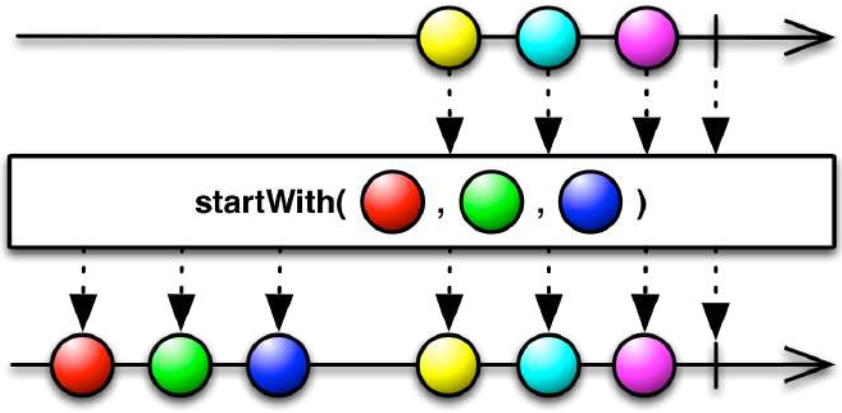
Build and run the app to verify everything is still working as expected.

Look at your logs to see if `LoadingState` is still getting rendered before `displayMoviesIntent()`.

```
D/MainActivity: State: LoadingState
D/MainPresenter$observeMovieDisplay: Intent: display movie
D/MainActivity: State: DataState
```

Sure enough, the problem is still there. So, how can you solve it? Using `startWith()`.

`startWith()` is a useful RxJava operator that lets you emit a specified sequence of items **before** emitting the items from the `Observable` source.



Because of this, `startWith()` is an excellent choice to solve the problem your app is currently facing.

Modify `observeMovieDisplay()`, like so:

```
private fun observeMovieDisplay() = view.displayMoviesIntent()
    .doOnNext { Timber.d("Intent: display movies intent") }
    .flatMap<MovieState> { movieInteractor.getMovieList() }
    .startWith(MovieState.LoadingState)
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe { view.render(it) }
```

Instead of calling `doOnSubscribe()`, you're using `startWith()` to emit the `LoadingState` before any other State is emitted from the `MovieInteractor`.

Build and run the app. Check the logs to see if `startWith()` is working as intended.

```
D/MainPresenter$observeMovieDisplay: Intent: display movies intent
D/MainActivity: State: LoadingState
D/MainActivity: State: DataState
```

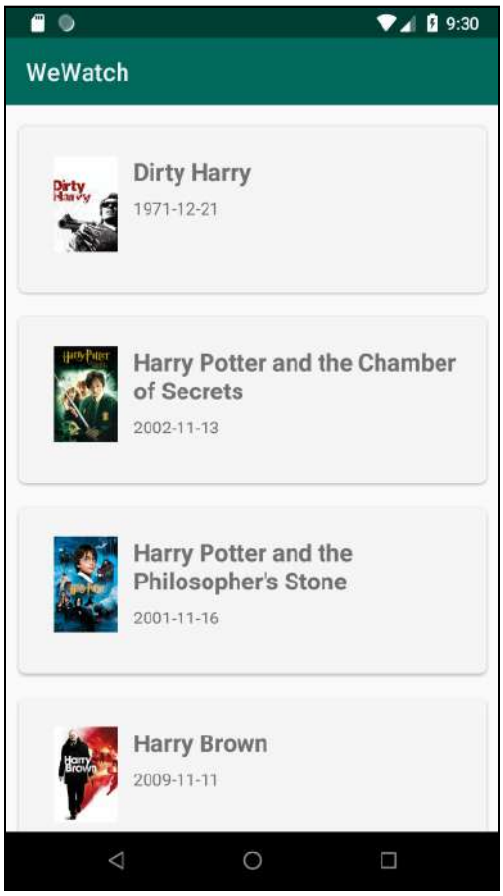
Great! Your app's `LoadingState` and `DataState` are displayed in the proper order. In other words, **after** receiving an Intent.

Now you need to fix `SearchPresenter` by replacing `doOnNext()` with `startWith()`.

Open **SearchPresenter.kt** and modify `observeMovieDisplayIntent()` so it matches this:

```
private fun observeMovieDisplayIntent() = view.displayMoviesIntent()
    .doOnNext { Timber.d("Intent: display movies") }
    .flatMap<MovieState> { movieInteractor.searchMovies(it) }
    .startWith(MovieState.LoadingState)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe { view.render(it) }
```

Build and run the app. Navigate to the `SearchMovieActivity` by searching for a movie.



Check the logs to verify that the `LoadingState` is called after the Intent and before the `DataState`.

```
D/SearchPresenter$observeMovieDisplayIntent: Intent: display movies
D/SearchMovieActivity: State: LoadingState
D/SearchMovieActivity: State: DataState
```

Great!

Now that your MVI architecture is working as expected, you know precisely what the last Intent emitted was before a crash occurs, making it easier to trace and fix errors.

Key points

- Timber is a handy library for conditional based logging that lets you print log statements only when they meet certain conditions.
- `doOnNext()` modifies your `Observable` source to perform a certain action when it calls `onNext()`.
- `doOnSubscribe()` executes the action passed as a parameter as soon as you subscribe to the `Observable`.
- `startWith()` makes an `Observable` emit a specific sequence of items before it begins emitting the items normally expected from it.

Where to go from here?

If you want to learn more about RxJava and MVI, look at the following resources:

- The official ReactiveX website contains an introduction to the most important RxJava concepts such as `Observable`, Operator, Subject and Scheduler: http://reactivex.io/
- The RxJava javadoc: http://reactivex.io/RxJava/2.x/javadoc/.
- This decision tree that can help you find the appropriate ReactiveX operator according to your needs: http://reactivex.io/documentation/operators.html#tree

Conclusion

Congratulations! You've completed your tour of Advanced Android Architecture. Choosing architectures is often a matter of taste rather than objective reasoning. Taking into account team preferences, skill level and interest are equally important when considering what architecture to select for your project. This book aimed to present you with the architectural options, leaving you with the power to decide which architecture works best for you. The skills and knowledge that you have gained throughout these chapters will act as your foundation for many future projects and creative endeavors.

Take time to enjoy the success that you've found in completing the material provided in this book, and then look forward. A developer's world is a blank canvas, just waiting for the stroke of the creator. Endless possibilities await; projects that need the special touch of your talent, creativity, and unique ideas. Take your next step into Android development, and press onward with confidence.

If you have any questions or comments as you work through this book, please stop by our forums at http://forums.raywenderlich.com and look for the particular forum category for this book.

Thank you again for purchasing this book. Your continued support is what makes the tutorials, books, videos, conferences and other things we do at raywenderlich.com possible, and we truly appreciate it!

Wishing you all the best in your continued Android adventures,

– Yun, Aldo, Nick, Matei, Tammy, Vijay and Manda

The *Android App Architecture* team

Learn App Architectures for Android!

Android has such a rich development ecosystem, which allows developers to develop apps in any way they please. However, with this level of freedom, scaling apps can be a mess. You've heard developers talk about "architecting" apps to help scale, but it's never clear which architecture to use.

This is where Advanced Android App Architectures comes to the rescue! In this book, you'll learn all the popular architectures the quick and easy way: by following fun and easy-to-read tutorials.

Who This Book Is For

This book is for intermediate Android developers who already know the basics of Android and Kotlin development but want to learn how best to organize code for scale.

Topics Covered in Advanced Android App Architecture:

- **Model, View, Controller (MVC):** Learn what the most common pattern found in Android Architectures really means.
- **Model, View, Presenter (MVP):** Learn to separate concerns better than MVC including better ways to test your business logic.
- **Model, View, View-Model (MVVM):** Learn to get the most of out of Android architecture components to structure your app in a way that will let you scale!
- **Testing Strategies:** Ensure your code works optimally by testing your newly implemented architectures.
- **Modern UI Architecture:** Learn theories such as unidirectional theory, VIPER, MVI and RXJava.
- **Clean Architecture:** Understand UI vs. system architecture, learning the theory, exploring a sample and seeing how to test clean architecture.

One thing you can count on: After reading this book, you'll be prepared to dive right in to any of the most popular Android app architectures out there!

About the iOS Tutorial Team

The Tutorial Team is a group of app developers and authors who write tutorials at the popular website **raywenderlich.com.** We take pride in making sure each tutorial we write holds to the highest standards of quality. We want our tutorials to be well written, easy to follow, and fun.

If you've enjoyed the tutorials we've written in the past, you're in for a treat. The tutorials we've written for this book are some of our best yet — and this book contains detailed technical knowledge you simply won't be able to find anywhere else.