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**

Table of Contents: Overview

Table of Contents: Extended

About This Book Sample

In *Advanced Android App Architectures*, you'll find a diverse and hands-on approach to architecting your apps on Android. Android development can be fun; however, scaling an app can have its fair share of problems.

In this book, you'll learn why a conversation on architecture is the first important step to taking your app to the next level! This book will introduce you to a number of architectures, including Model View Controller, Model View Presenter, Model View Intent, Model-View-ViewModel and VIPER. You'll learn theory, explore samples that you will refactor and learn the fundamentals of testing.

We are pleased to offer you this sample from the full *Advanced Android App Architectures* book. The chapters that follows will introduce you to the Model-View-ViewModel architecture and start you building an app to store your favorite movie titles.

The chapter included:

- **Chapter 9: Model-View-ViewModel Theory**: MVVM (Model-View-ViewModel) is an architecture that's gained a lot of attention. It was first presented to specifically address event-driven programming. In this chapter, you'll gain deep insight into what makes this architecture so exciting.
- **Chapter 11: Model-View-ViewModel Sample with Android Architecture Components**: In this chapter, you'll learn how to further refactor the sample app to conform to MVVM; however, this time, you'll learn how to leverage Google's Architecture Components.

The book is ready for purchase at:

- https://store.raywenderlich.com/products/advanced-android-app-architectures.

Enjoy!

The *Advanced Android App Architectures* Team

Advanced Android App Architectures

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.

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 @vijaysharm or on LinkedIn @vijaysharm.

Manda Frederick is an 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.

What You Need

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

- **Android Studio 3.2.1**, 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.

Book License

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

- You are allowed to use and/or modify the source code in *Advanced Android App Architectures* 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 Architectures* in as many apps as you want, but must include this attribution line somewhere inside your app: "Artwork/images/designs: from *Advanced Android App Architectures*, available at www.raywenderlich.com."
- The source code included in *Advanced Android App Architectures* is for your personal use only. You are NOT allowed to distribute or sell the source code in *Advanced Android App Architectures* 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.

Chapter 9: 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.

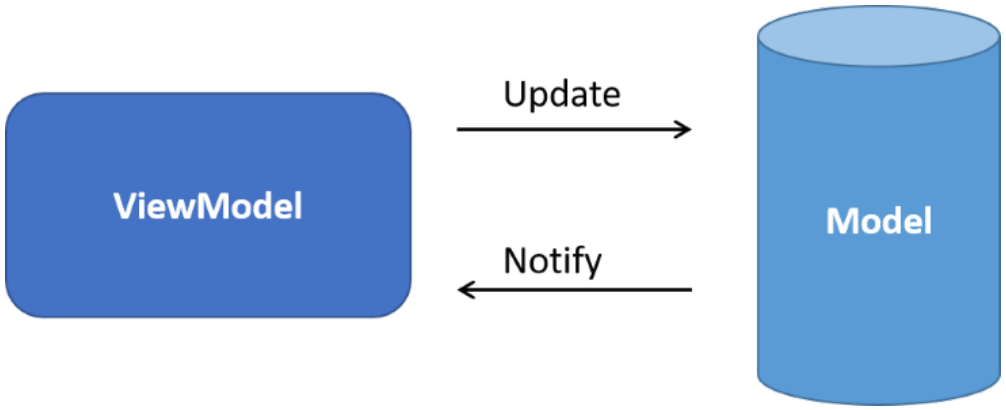
At first glance, MVVM looks a lot like the **MVP** and **MVC** architecture patterns from the last chapters.

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.

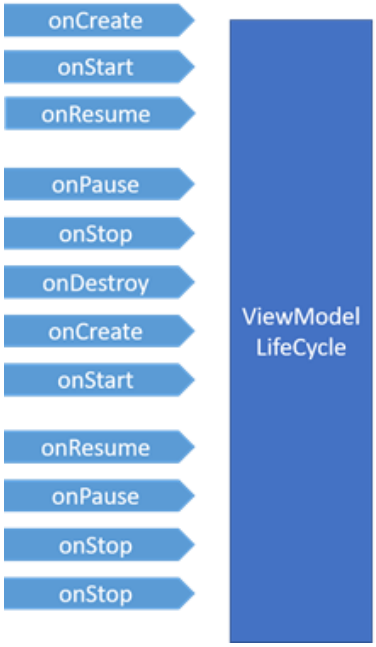
The most common solution to this problem is to save your data in your `onSaveInstanceState()` bundle and restore it later in your `onCreate()` method. But this approach only works for primitive data such as Integers or simple classes that can be serialized and deserialized.

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.

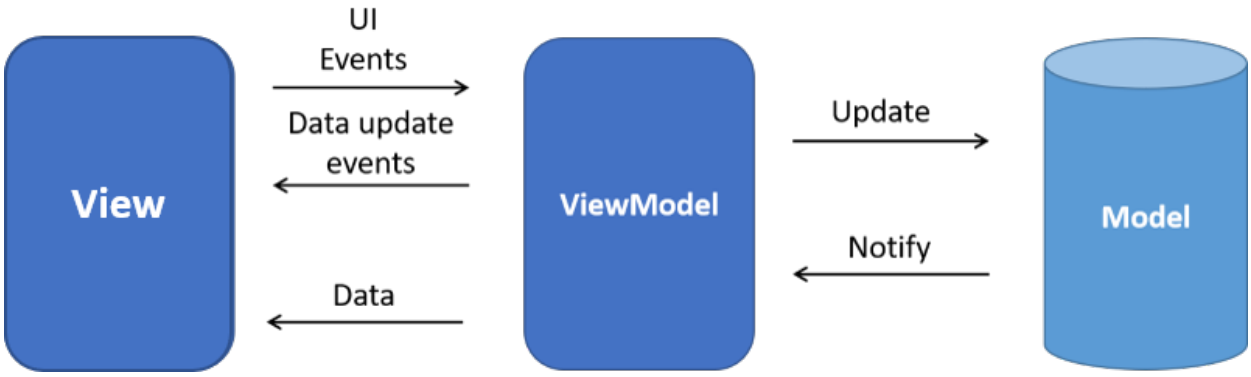
The View

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 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, 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

Open the starter project attached to this chapter in Android Studio 3.1 or greater by going to **File ▸ New ▸ Import Project**, and selecting the **build.gradle** file in the root of the project.

> **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.

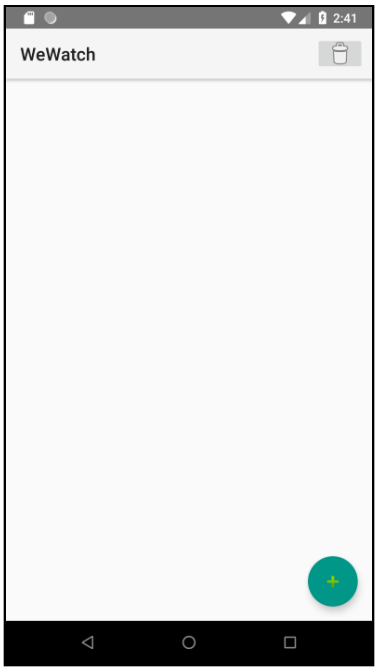
The **data** package has three packages related to the backend of your app:

- 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**.

Before moving on, make sure to replace your **TMDB** API key in the **RetrofitClient.kt** file. For this, you just need to update the value of the **API_KEY** constant at the top of the file.

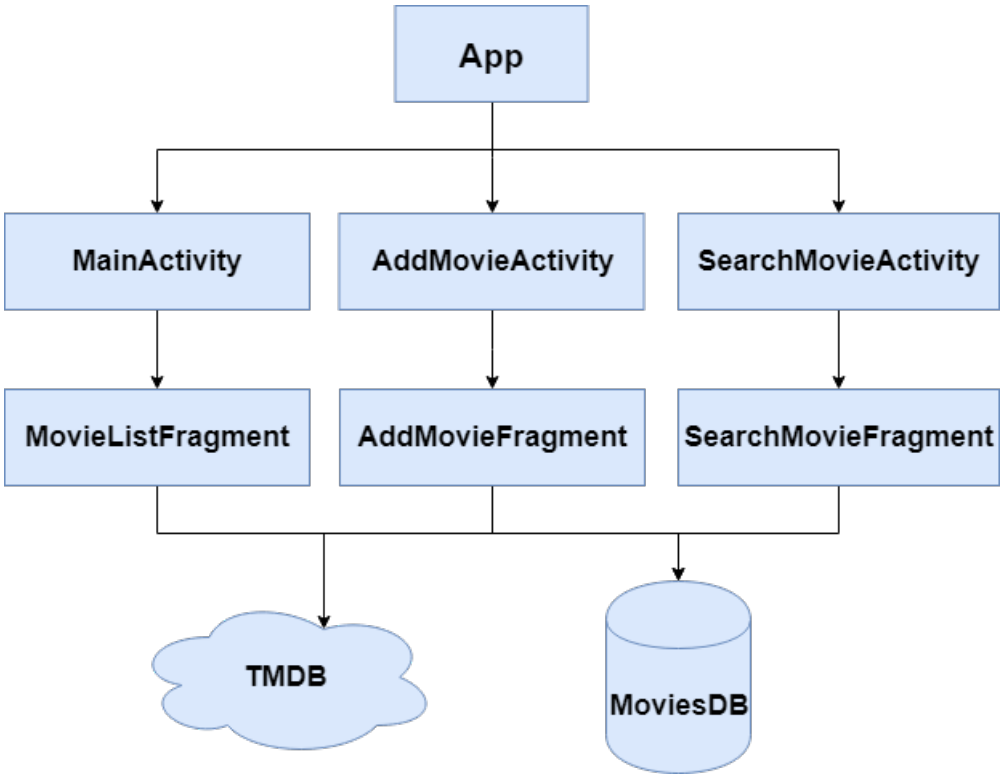
The **ui** package contains three packages related to the front end of your app such as the Fragments and the Activities.

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 despite it working 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.

Under the **data** package, create a new Kotlin class and name it **MovieRepository**.

Replace the autogenerated class with the following:

```
class MovieRepository(application: Application) {
  //1
  private val movieDao: MovieDao
  private val retrofitClient = RetrofitClient()
  //2
  init {
    val movieDatabase = MovieDatabase.getInstance(application)
    movieDao = movieDatabase.movieDao()
  }
  //3
  fun getSavedMovies(): List<Movie> {
    return movieDao.getAll()
  }
  //4
  fun saveMovie(movie: Movie) {
    thread {
      movieDao.insert(movie)
    }
  }
  //5
  fun updateMovie(movie: Movie) {
    thread {
      movieDao.updateMovie(movie)
    }
  }
  //6
  fun deleteWatchedMovies() {
    thread {
      movieDao.deleteMovies(true)
    }
  }
}
```

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 your ViewModels

While it is possible to access the movie repository from your views, it is generally considered a 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.

Start by opening **build.gradle** in your **app** directory. Add the following line inside the dependencies block:

```
// Lifecycle Components
def lifecycleVersion = "1.1.1"
implementation "android.arch.lifecycle:extensions:$lifecycleVersion"
```

The code above adds the lifecycle part of the Android Architecture Components to your dependencies.

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.

Now that you have your dependencies ready it is time to create some sweet **ViewModel** classes

Create a new class under the **ui ▸ add** package and name it **AddMovieViewModel**.

Replace the autogenerated class with the following code:

```
//1
class AddMovieViewModel(application: Application) :
AndroidViewModel(application) {
  //2
  private val movieRepository =
getApplication<App>().getMovieRepository()
  //3
  fun saveMovie(movie: Movie) {
    movieRepository.saveMovie(movie)
  }
}
```

Taking each commented section in turn:

1. `AddMovieViewModel` extends from the `AndroidViewModel` class, which is a subclass of 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 to configuration changes. Everything is already taken care of for you.
2. Here, you get a reference to your movie repository using the `getMovieRepository()` method that you created before.
3. `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 **AddMovieFragment.kt** and add the following attribute to store a reference to an instance of `AddMovieViewModel`:

```
private lateinit var viewModel: AddMovieViewModel
```

Once you have your attribute add the following code to override the `onCreate()` method:

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

`ViewModelProviders` is a special class that returns an existing ViewModel or creates a new one while the scope of a given Fragment is alive. In this case, since you are passing a reference to your `AddMovieFragment`, it will create a new `AddMovieViewModel` that will stay alive during the whole lifecycle of your `AddMovieFragment` fragment.

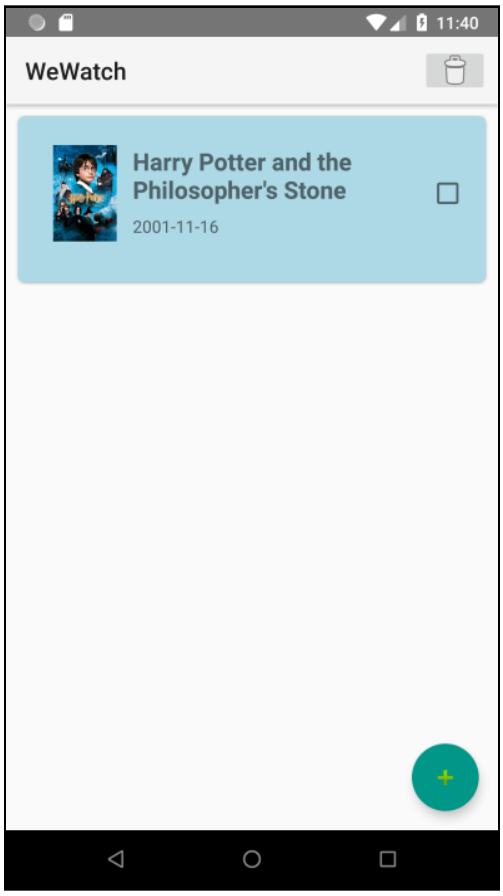
The only step left is to use your **ViewModel** to save a movie when the user presses the **Save Movie** button.

Locate the `addMovie()` method and replace the `thread` block and its contents with the following call:

```
viewModel.saveMovie(movie)
```

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:



Easy, right?

Before creating the next ViewModels, you will need to learn about LiveData.

Using LiveData with your ViewModels

In the past section, you created a ViewModel for your `AddMovieFragment` with a single method to save a movie.

LiveData was not absolutely necessary since the movie was going to be saved to your local database immediately, and there was no need to wait for a response.

But what happens if you need to retrieve a list of movies from a local database, such as Room, or from a webservice, such as TMDB API?

In this case, you need to wait for a response to update your UI, and this is where LiveData comes in handy.

LiveData is a data holder class, just like a List or a HashMap, that can be observed for any 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, change your app to make use of all the advantages of LiveData. Open **MovieDao.kt** and change the `getAll()` method to return a list of LiveData movies:

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

Then, open **MovieRepository.kt** and update the `getSavedMovies()` method like below:

```
fun getSavedMovies(): LiveData<List<Movie>> {
  return movieDao.getAll()
}
```

As you can see, it is very easy to use **LiveData** along with any objects. Since `LiveData` is a holder class you just need to wrap your return values with the **LiveData** component.

Finally, create a new class under the **ui ▸ list** package and name it **MovieListViewModel**.

Replace the code inside with the following:

```
class MovieListViewModel(application: Application) :
AndroidViewModel(application) {
  //1
  private val movieRepository =
getApplication<App>().getMovieRepository()
  private val movieList = MediatorLiveData<List<Movie>>()
  //2
  init {
    getAllMovies()
  }
  //3
  fun getSavedMovies(): LiveData<List<Movie>> {
    return movieList
  }
  //4
  fun getAllMovies() {
    movieList.addSource(movieRepository.getSavedMovies()) { movies ->
      movieList.postValue(movies)
    }
  }
  //5
  fun deleteSavedMovies() {
    movieRepository.deleteWatchedMovies()
  }
  //6
  fun updateMovie(movie: Movie) {
    movieRepository.updateMovie(movie)
  }
}
```

Step by step:

1. First, you create the `movieRepository` and the `movieList` properties. You may have noticed that the `movieList` 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 `MovieListViewModel` class is initialized.
3. `getSavedMovies()` returns a LiveData list of movies stored in your `movieList` property.
4. `getAllMovies()` sets the datasource of movieList from `MovieRepository`. It fetches the list of movies by executing `movieRepository.getSavedMovies()` and posting the value to `movieList`.
5. `deleteSavedMovies()` uses the `deleteWatchedMovies()` method from your ViewModel to delete movies whose `watched` field is set to `true`.
6. `updateMovie()` updates the database representation of the passed-in movie.

Your ViewModel is now ready to be used inside your Fragment.

Open **MovieListViewFragment.kt** and add the following attribute to hold a reference to your MovieListViewModel:

```
  private lateinit var viewModel: MovieListViewModel
```

Initialize it inside the `onCreate()` method just like you did in `AddMovieViewModel`:

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

Now that your viewModel has been initialized, it's time to use it.

Replace the `thread` block call inside `onViewCreated()` with the following:

```
  viewModel.getSavedMovies().observe(this, Observer { movies ->
    movies?.let {
      moviesRecyclerView.adapter = MovieListAdapter(movies,
this@MovieListFragment)
    }
  })
```

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.

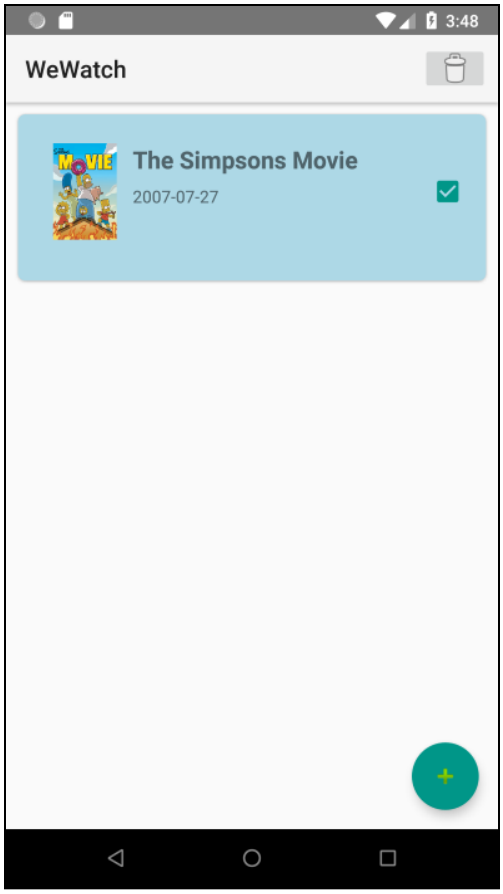
You might also notice that the `observe` method passes your Fragment, an instance of `LifecycleOwner`, as the first parameter. By doing so, the observer is bound to the `Lifecycle` object associated. This basically means three things:

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, replace all the code inside `deleteMovies()` with the following:

```
viewModel.deleteSavedMovies()
```

The MovieList screen of your app is now ready! Build and run to see it in action:



Creating the ViewModel

The last step is to create the ViewModel for the **Search Movies** screen of your app.

Open **MovieRepository.kt** and add the following method:

```
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) {
      Log.d(javaClass.simpleName, "Remember to add your KEY in your
RetrofitClient.kt file")
    }
  //4
    override fun onResponse(call: Call<MoviesResponse>, response:
Response<MoviesResponse>) {
      if (response.isSuccessful) {
        val movies = response.body()?.results
        data.value = movies
      }
    }
  })
  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 **ui ▸ search** package and name it `SearchViewModel`.

Replace the auto-generated code inside the class with the following:

```
class SearchViewModel(application: Application) :
AndroidViewModel(application) {

  private val movieRepository =
getApplication<App>().getMovieRepository()

  fun searchMovie(query: String): LiveData<List<Movie>> {
    return movieRepository.searchMovies(query)
  }

  fun saveMovie(movie: Movie) {
    movieRepository.saveMovie(movie)
  }
}
```

The code above gets a reference to your movieRepository 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 Fragment.

Open **SearchFragment.kt** and add a property to store an instance of your `SearchViewModel` class:

```
lateinit var viewModel: SearchViewModel
```

Initialize your attribute inside `onCreate()`:

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

Replace the code inside `getMovieList()` with the following:

```
viewModel.searchMovie(query).observe(this, Observer { movies ->
  movies?.let {
    searchResultsRecyclerView.adapter = SearchAdapter(movies,
this@SearchFragment)
  }
})
```

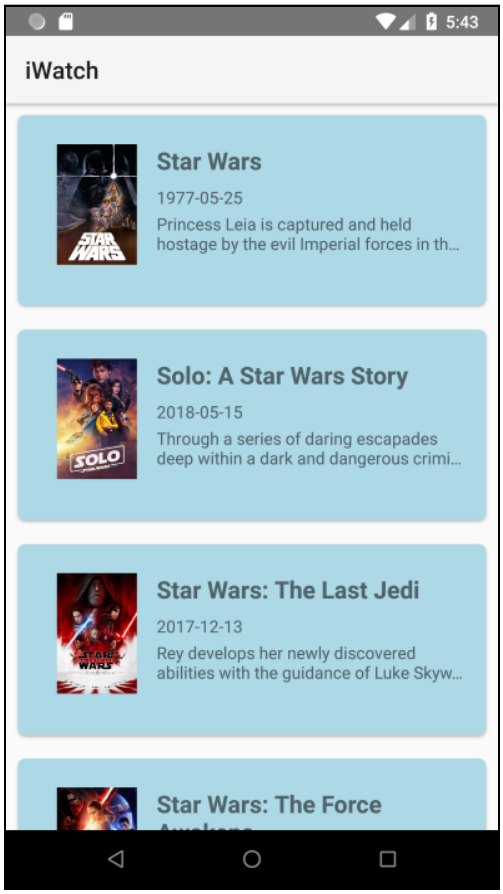
The above uses the `searchMovie()` method of your **ViewModel** to retrieve the list of movies from the TMDB API. Once the list of movies have been retrieved, your observer's callback is executed and the adapter is attached to your **RecyclerView**.

Finally, go to the `onItemClick()` method and replace the code inside the snackbar action with the following:

```
viewModel.saveMovie(movie)
goToMainActivity()
```

The code above uses the `saveMovie()` method of your `MovieRepository` to save the movie passed as a parameter and returns to the `MainActivity`.

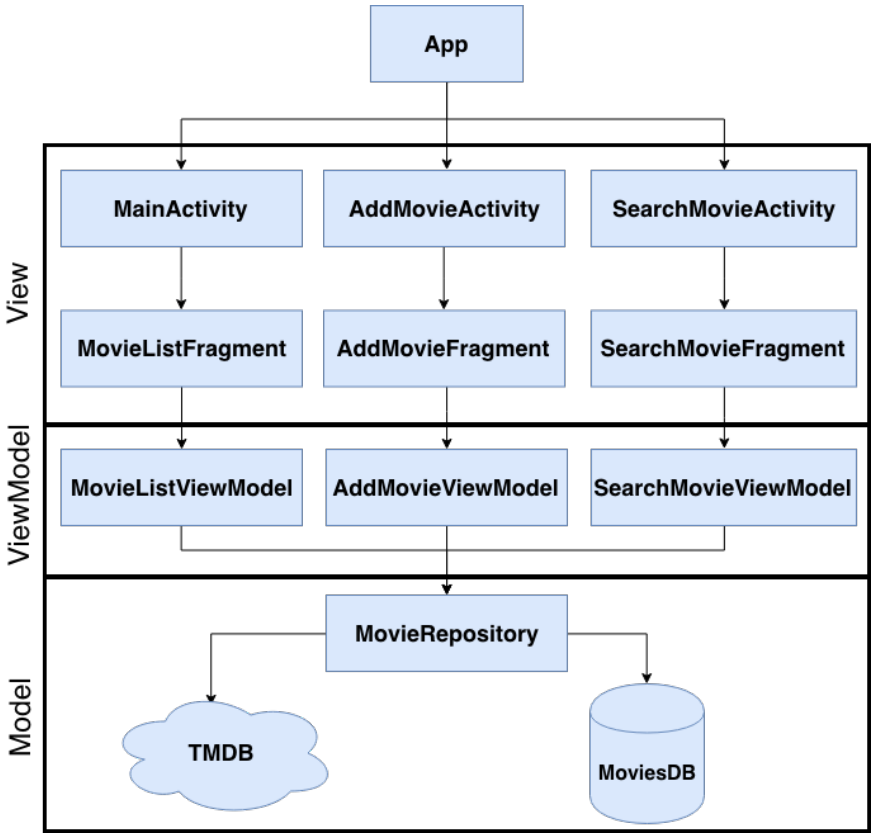
Build and Run your app one last time to test your changes:



Awesome!

The MVVM architecture

At the beginning of this chapter, you reviewed the current architecture of your app without MVVM. Take a look at what it looks like, now:



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.

Although refactoring your 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.

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?

We hope you enjoyed this sample of *Advanced Android App Architectures*!

If you enjoyed this sample, be sure to check out the full book, which will contain the following chapters:

1. **Introduction**: Android development can be fun; however, scaling an app can have its fair share of problems. In this book and in this chapter, you'll learn why a conversation on architecture is the first important step to taking your app to the next level!

2. **Model View Controller Theory**: You'll start by learning about MVC (Model View Controller) architecture. MVC is the baseline architecture when it comes to Android development. You'll learn how to identify this architecture and how it became so commonplace. You'll also be introduced to the sample app, which you'll refactor into each of the learned architectures.

3. **Testing Model View Controller**: Learn how to take the most common architecture and how to make it testable. Learn the pain points of using this architecture.

4. **Android Architecture Components**: All the way back in May of 2017, Google introduced its take on how best to architect an Android app. You'll learn all about the major components that make up the AAC since many of them can be fit to work with other architectures.

5. **Dependency Injection**: Dependency Injection is a simple idea that often finds itself as one of the cornerstones to many app architectures. You'll learn how you can structure your code to use dependency injection to help with all architectures presented in this book.

6. **Model View Presenter Theory**: The first of many architectures you'll learn, MVP (Model View Presenter) is often considered the most natural progression out of MVC. In this chapter, you'll learn what advantages MVP might have over an MVC architecture.
7. **Model View Presenter Sample**: You've learned about MVP, now it's time to conform the sample app to use it.
8. **Testing MVP**: Once you've refactored your app to leverage MVP, learn strategies for MVP architected components.
9. **Model-View-ViewModel Theory**: MVVM (Model-View-ViewModel) is an architecture that's gained a lot of attention. It was first presented to specifically address event-driven programming. In this chapter, you'll gain deep insight into what makes this architecture so exciting.
10. **Model-View-ViewModel Sample with Data Binding**: MVVM, at its core, requires data binding in order to be effective. In this chapter, you'll refactor the sample app to use MVVM along with Data Binding.
11. **Model-View-ViewModel Sample with Android Architecture Components**: In this chapter, you'll learn how to further refactor the sample app to conform to MVVM; however, this time, you'll learn how to leverage Google's Architecture Components.
12. **Testing Model-View-ViewModel**: You've refactored your app, now it's time to find ways to test it. In this chapter, you'll learn how an MVVM architecture can make this easier.
13. **Unidirectional Theory**: Unidirectional architectures build on the idea of data binding, and turns it up to 11! Most famously, Unidirectional data flow was popularized by Flux, the app architecture most commonly found in React apps.
14. **VIPER Theory**: Don't be scared, you won't be wrangling snakes in this chapter. Instead, you'll learn how the letters of VIPER form the architectural pieces to this architecture.
15. **VIPER Sample**: In this chapter, you'll refactor the sample app in to each of the VIPER components. You'll draw distinct lines of responsibility for each component and how this architecture can be used to scale your app.
16. **Testing VIPER**: What good is an architecture if you can't test it. In this chapter, you'll learn to test each of the VIPER components.

17. **Model View Intent Theory**: MVI (Model View Intent) is commonly known as the architecture that prevents developers from misusing patterns like MVP or MVVM. In this chapter, you'll learn the last architecture explored in this book and how Android makes this architecture really stand out.
18. **Model View Intent Sample**: You've learned what MVI is, but now you'll learn how to refactor the sample app into each of the MVI components.
19. **Testing Model View Intent**: After refactoring your app into MVI, you'll write some tests to make sure each component is doing exactly as you expect.

You can find the book on the raywenderlich.com store here: https://store.raywenderlich.com/products/advanced-android-app-architectures

We hope you enjoy the book!

— The *Advanced Android App Architectures* 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.