Chapter 4: Testing Kotlin Coroutines

In the previous chapter, you learned about coroutine cancelations and how you can make your coroutines cancelable. You then learned about coroutine timeouts in milliseconds or **Duration**. Finally, you learned about exceptions and how you can handle them using **`try-catch`** and **`CoroutineExceptionHandler`**.

Creating tests is an important part of app development. The more code you write, the higher the chance that there will be bugs and errors. With tests, you can ensure your application works as you have programmed it. You can quickly discover issues and fix them immediately. Tests can make development easier, saving you time and resources. They can also help you refactor and maintain your code with confidence.

In this chapter, you will learn how to test Kotlin coroutines in Android. First, we will update the Android project for testing. We will then proceed with learning the steps to create tests for Kotlin coroutines.

In this chapter, we are going to cover the following topics:

- Setting up an Android project for testing coroutines
- Unit testing suspending functions
- Testing coroutines

By the end of this chapter, you will understand coroutine testing. You will be able to write and run unit and integration tests for the coroutines in your Android applications.

Technical requirements

You will need to download and install the latest version of Android Studio. You can find the latest version at https://developer.android.com/studio ↗️. For an optimal learning experience, a computer with the following specifications is recommended:

- Intel Core i5 or equivalent or higher
- 4 GB RAM minimum
- 4 GB available space

The code examples for this chapter can be found on GitHub at https://github.com/PacktPublishing/Simplifying-Android-Development-with-Coroutines-and-Flows/tree/main/Chapter04 ↗️.

Setting up an Android project for testing coroutines

In this section, we will start by looking at how you can update your Android app to make it ready for adding and running tests. Once your project is properly set up, it will be easy to add unit and integration tests for your coroutines.

When creating unit tests on Android, you must have the **JUnit 4** testing framework in your project. JUnit is a unit testing framework for Java. It should be automatically included in the **`app/build.gradle`** dependencies when creating a new Android project in Android Studio.

If your Android project does not have JUnit 4 yet, you can add it by including the following to your **`app/build.gradle`** dependencies:

```
dependencies {

    ...

    testImplementation 'junit:junit:4.13.2'
```

```
}
```

This allows you to use the JUnit 4 framework for your unit tests.

To create mock objects for your tests, you can also use mocking libraries. Mockito is the most popular Java mocking library, and you can use it on Android. To add Mockito to your tests, add the following to the dependencies in your **app/build.gradle** file:

```
dependencies {

    ...

    testImplementation 'org.mockito:mockito-core:4.0.0'

}
```

Adding this dependency allows you to use Mockito to create mock objects for your unit tests in your project.

If you prefer to use Mockito with idiomatic Kotlin code, you can use Mockito-Kotlin. Mockito-Kotlin is a Mockito library that contains helper functions to make your code more Kotlin-like.

To use Mockito-Kotlin in your Android unit tests, you can add the following dependency to your **app/build.gradle** file dependencies:

```
dependencies {

    ...

    testImplementation 'org.mockito.kotlin:mockito-

      kotlin:4.0.0'

}
```

This will enable you to use Mockito to create mock objects for your tests, using idiomatic Kotlin code.

If you are using both Mockito (**`mockito-core`**) and Mockito-Kotlin in your project, you can just add the dependency for Mockito-Kotlin. It already has a dependency to **`mockito-core`**, which it will automatically import.

To test Jetpack components such as **`LiveData`**, add the **`androidx.arch.core:core-testing`** dependency:

```
dependencies {

    ...

    testImplementation 'androidx.arch.core:core-

      testing:2.1.0'

}
```

This dependency contains support for testing Jetpack architecture components. It includes JUnit rules such as **`InstantTaskExecutorRule`** that you can use to test the **`LiveData`** objects in your code.

Testing coroutines is a bit more complicated than the usual testing. This is because coroutines are asynchronous, tasks can run in parallel, and tasks can take a while before finishing. Your tests must be fast and consistent.

To help you with testing coroutines, you can use the coroutine testing library from the **`kotlinx-coroutines-test`** package. It contains utility classes to make testing coroutines easier and more efficient. To use it in your Android project, you must add the following to the dependencies in your **`app/build.gradle`** file:

```
dependencies {

    ...
```

```
    testImplementation 'org.jetbrains.kotlinx:kotlinx-
      coroutines-test:1.6.0'
}
```

This will import the **`kotlinx-coroutines-test`** dependency into your Android project. You will then be able to use the utility classes from the Kotlin coroutine testing library to create unit tests for your coroutines.

If you want to use **`kotlinx-coroutines-test`** in your Android instrumented tests that will run on an emulator or physical device, you should add the following to your **`app/build.gradle`** file dependencies:

```
dependencies {
    ...
    androidTestImplementation
      'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.6.0'
}
```

Adding this to your dependencies will allow you to use **`kotlinx-coroutines-test`** in your instrumented tests.

As of version 1.6.0, the coroutine testing library is still labeled as experimental. You may have to annotate the test classes with the **`@ExperimentalCoroutinesApi`** annotation, as shown in the following example:

```
@ExperimentalCoroutinesApi
class MovieRepositoryUnitTest {
    ...
```

```
}
```

In this section, you learned how to set up your Android project to add tests. You will learn how to create unit tests for suspending functions in the next section.

Unit testing suspending functions

In this section, we will focus on how you can unit test your suspending functions. You can create unit tests for classes such as **`ViewModel`** that launch a coroutine or have suspending functions.

Creating a unit test for a suspending function is more difficult to write as a suspending function can only be called from a coroutine or another coroutine. What you can do is use the **`runBlocking`** coroutine builder and call the suspending function from there. For example, say you have a **`MovieRepository`** class like the following:

```
class MovieRepository (private val movieService:
  MovieService) {
    ...
    private val movieLiveData =
      MutableLiveData<List<Movie>>()
    fun fetchMovies() {
        ...
        val movies = movieService.getMovies()
        movieLiveData.postValue(movies.results)
    }
```

```
}
```

This **`MovieRepository`** has a suspending function called **`fetchMovies`**. This function gets the list of movies by calling the **`getMovies`** suspending function from **`movieService`**.

To create a test for the **`fetchMovies`** function, you can use **`runBlocking`** to call the suspending function, like the following:

```
class MovieRepositoryTest {

    ...

    @Test

    fun fetchMovies() {

        ...

        runBlocking {

            ...

            val movieLiveData =

              movieRepository.fetchMovies()

            assertEquals(movieLiveData.value, movies)

        }

    }

}
```

Using the **`runBlocking`** coroutine builder allows you to call suspending functions and do the assertion checks.

The **`runBlocking`** coroutine builder is useful for testing. However, there are times when it can be slow because of delays in the code. Your unit tests must ideally be able to run as fast as possible. The coroutine testing library can help you with its **`runTest`** coroutine builder. It is the same as the **`runBlocking`** coroutine builder except it runs the suspending function immediately and without delays.

Replacing **`runBlocking`** with **`runTest`** in the previous example would make your test look like the following:

```
@ExperimentalCoroutinesApi
class MovieRepositoryTest {
    ...
    @Test
    fun fetchMovies() {
        ...
        runTest {
            ...
            val movieLiveData =
              movieRepository.fetchMovies()
            assertEquals(movieLiveData.value, movies)
        }
    }
}
```

The **`runTest`** function allows you to call the **`movieRepository.fetchMovies()`** suspending function and then check the result of the operation.

In this section, you learned about writing unit tests for suspending functions in your Android project. In the next section, you will learn about testing coroutines.

Testing coroutines

In this section, we will focus on how you can test your coroutines. You can create tests for classes such as **`ViewModel`** that launch a coroutine.

For coroutines launched using **`Dispatchers.Main`**, your unit tests will fail with the following error message:

```
java.lang.IllegalStateException: Module with the Main
dispatcher had failed to initialize. For tests
Dispatchers.setMain from kotlinx-coroutines-test
module can be used
```

This exception happens because **`Dispatchers.Main`** uses **`Looper.getMainLooper()`**, the application's main thread. This main looper is not available in Android for local unit tests. To make your tests work, you must use the **`Dispatchers.setMain`** extension function to change the **`Main`** dispatcher. For example, you can create a function in your test class that will run before your tests:

```
@Before

fun setUp() {

    Dispatchers.setMain(UnconfinedTestDispatcher())

}
```

The **`setUp`** function will run before the tests. It will change the main dispatcher to another dispatcher for your test.

`Dispatchers.setMain` will change all subsequent uses of **`Dispatchers.Main`**. After the test, you must change the **`Main`** dispatcher back with a call to **`Dispatchers.resetMain()`**. You can do something like the following:

```
@After
fun tearDown() {
    Dispatchers.resetMain()
}
```

After the tests have run, the **`tearDown`** function will be called, which will reset the **`Main`** dispatcher.

If you have many coroutines to test, copying and pasting this boilerplate code in each test class is not ideal. You can make a custom JUnit rule instead that you can reuse in your test classes. This JUnit rule must be in the root folder of your test source set, as shown in *Figure 4.01*:

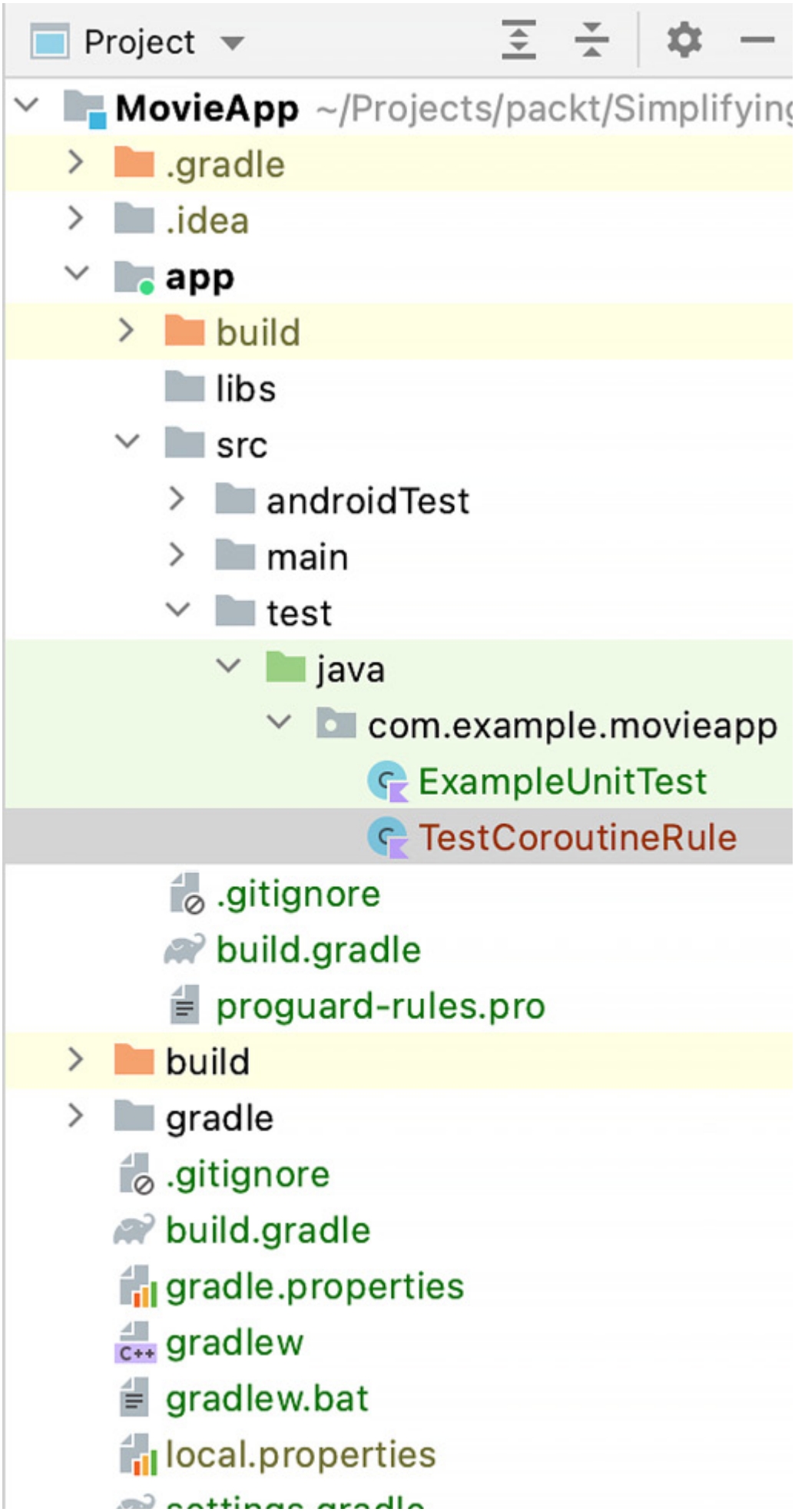
Project
MovieApp ~/Projects/packt/Simplifying
.gradle
.idea
app
build
libs
src
androidTest
main
test
java
com.example.movieapp
ExampleUnitTest
TestCoroutineRule
.gitignore
build.gradle
proguard-rules.pro
build
gradle
.gitignore
build.gradle
gradle.properties
gradlew
gradlew.bat
local.properties

Figure 4.1 – Custom TestCoroutineRule in the root test folder

An example of a custom JUnit rule that you can write to reuse for automatically setting **`Dispatchers.setMain`** and **`Dispatchers.resetMain`** is this **`TestCoroutineRule`**:

```
@ExperimentalCoroutinesApi
class TestCoroutineRule(val dispatcher: TestDispatcher =
  UnconfinedTestDispatcher()):
   TestWatcher() {
   override fun starting(description: Description?) {
       super.starting(description)
       Dispatchers.setMain(dispatcher)
   }
   override fun finished(description: Description?) {
       super.finished(description)
       Dispatchers.resetMain()
   }
}
```

This custom JUnit rule will allow your test to automatically call **`Dispatchers.setMain`** before the tests and **`Dispatchers.resetMain`** after the tests.

You can then use this **`TestCoroutineRule`** in your test classes by adding the **`@get:Rule`** annotation:

```
@ExperimentalCoroutinesApi
class MovieRepositoryTest {
    @get:Rule
    var coroutineRule = TestCoroutineRule()
    ...
}
```

With this code, you will not need to add the **`Dispatchers.setMain`** and **`Dispatchers.resetMain`** function calls every time in your test classes.

When testing your coroutines, you must replace your coroutine dispatchers with a **`TestDispatcher`** for testing. To be able to replace your dispatchers, your code should have a way to change the dispatcher that will be used for the coroutines. For example, this **`MovieViewModel`** class has a property for setting the dispatcher:

```
class MovieViewModel(private val dispatcher:
  CoroutineDispatcher = Dispatchers.IO): ViewModel() {
    ...
    fun fetchMovies() {
        viewModelScope.launch(dispatcher) {
            ...
        }
```

```
        }

    }
```

`MovieViewModel` uses the dispatcher specified in its constructor or the default value (**`Dispatchers.IO`**) for launching the coroutine.

In your test, you can then set a different **`Dispatcher`** for testing purposes. For the preceding **`ViewModel`**, your test could initialize **`ViewModel`** with a different dispatcher, as shown in the following example:

```
    @ExperimentalCoroutinesApi
    class MovieViewModelTest {

        ...

        @Test
        fun fetchMovies() {

            ...

            runTest {

                ...

                val viewModel =

                  MovieViewModel(UnconfinedTestDispatcher())

                viewModel.fetchMovies()

                ...

            }

        }
```

```
}
```

The **viewModel** in **MovieViewModelTest**'s **fetchMovies** test was initialized with **UnconfinedTestDispatcher** as the coroutine dispatcher for testing purposes.

In the previous examples, you used **UnconfinedTestDispatcher** as the **TestDispatcher** for the test. There are two available implementations of **TestDispatcher** in the **kotlinx-coroutines-test** library:

- **StandardTestDispatcher**: Does not run coroutines automatically, giving you full control over execution order
- **UnconfinedTestDispatcher**: Runs coroutines automatically; offers no control over the order in which the coroutines will be launched

Both **StandardTestDispatcher** and **UnconfinedTestDispatcher** have constructor properties: **scheduler** for **TestCoroutineScheduler** and **name** for identifying the dispatcher. If you do not specify the scheduler, **TestDispatcher** will create a **TestCoroutineScheduler** by default.

The **TestCoroutineScheduler** of the **StandardTestDispatcher** controls the execution of the coroutine. **TestCoroutineScheduler** has three functions you can call to control the execution of the tasks:

- **runCurrent()**: Runs the tasks that are scheduled until the current virtual time
- **advanceUntilIdle()**: Runs all pending tasks
- **advanceTimeBy(milliseconds)**: Runs pending tasks until current virtual advances by the specified milliseconds

TestCoroutineScheduler also has a **currentTime** property that specifies the current virtual time in milliseconds. When you call functions such as **advanceTimeBy**, it will update the **currentTime** property of the scheduler.

The **runTest** coroutine builder creates a coroutine with a coroutine scope of **TestScope**. This **TestScope** has a **TestCoroutineScheduler** (**testScheduler**)

that you can use to control the execution of tasks.

This **`testScheduler`** also has extension property called **`currentTime`** and the **`runCurrent`**, **`advanceUntilIdle`**, and **`advanceTimeBy`** extension functions, which simplifies calling these functions from the **`testScheduler`** of the **`TestScope`**.

Using **`runTest`** with a **`TestDispatcher`** allows you to test cases when there are time delays in the coroutine and you want to test a line of code before moving on to the next ones. For example, if your **`ViewModel`** has a **`loading`** Boolean variable that is set to **`true`** before a network operation and then is reset to **`false`** afterward, your test for the **`loading`** variable could look like this:

```
@Test
fun loading() {
    val dispatcher = StandardTestDispatcher()
    runTest() {
        val viewModel = MovieViewModel(dispatcher)
        viewModel.fetchMovies()
        dispatcher.scheduler.advanceUntilIdle()
        assertEquals(false, viewModel.loading.value)
    }
}
```

This test uses **`StandardTestDispatcher`** so you can control the execution of the tasks. After calling **`fetchMovies`**, you call **`advanceUntilIdle`** on the dispatcher's **`scheduler`** to run the task, which will set the **`loading`** value to **`false`** after completion.

In this section, you learned about adding tests for your coroutines. Let's test what we have learned so far by adding some tests to existing coroutines in an Android project.

Exercise 4.01 – adding tests to coroutines in an Android app

For this exercise, you will be continuing the movie app that you worked on in *Exercise 2.01*, *Using coroutines in an Android app*. This application displays the movies that are currently playing in cinemas. You will be adding unit tests for the coroutines in the project by following these steps:

1. Open the movie app you worked on in *Exercise 2.01*, *Using coroutines in an Android app*, in Android Studio.
2. Go to the **`app/build.gradle`** file and add the following dependencies, which will be used for the unit test:

```
testImplementation 'org.mockito.kotlin:mockito-
  kotlin:4.0.0'
testImplementation 'androidx.arch.core:core-
  testing:2.1.0'
testImplementation 'org.jetbrains.kotlinx:kotlinx-
  coroutines-test:1.6.0'
```

The first line will add Mockito-Core and Mockito-Kotlin, the second line will add the architecture testing library, and the last line will add the Kotlin coroutine testing library. You will be using these for the unit tests you will add to the Android project.

3. In **`app/src/test/resources`**, create a **`mockito-extensions`** directory. In that directory, create a new file named **`org.mockito.plugins.MockMaker`**, as shown in *Figure 4.2*:

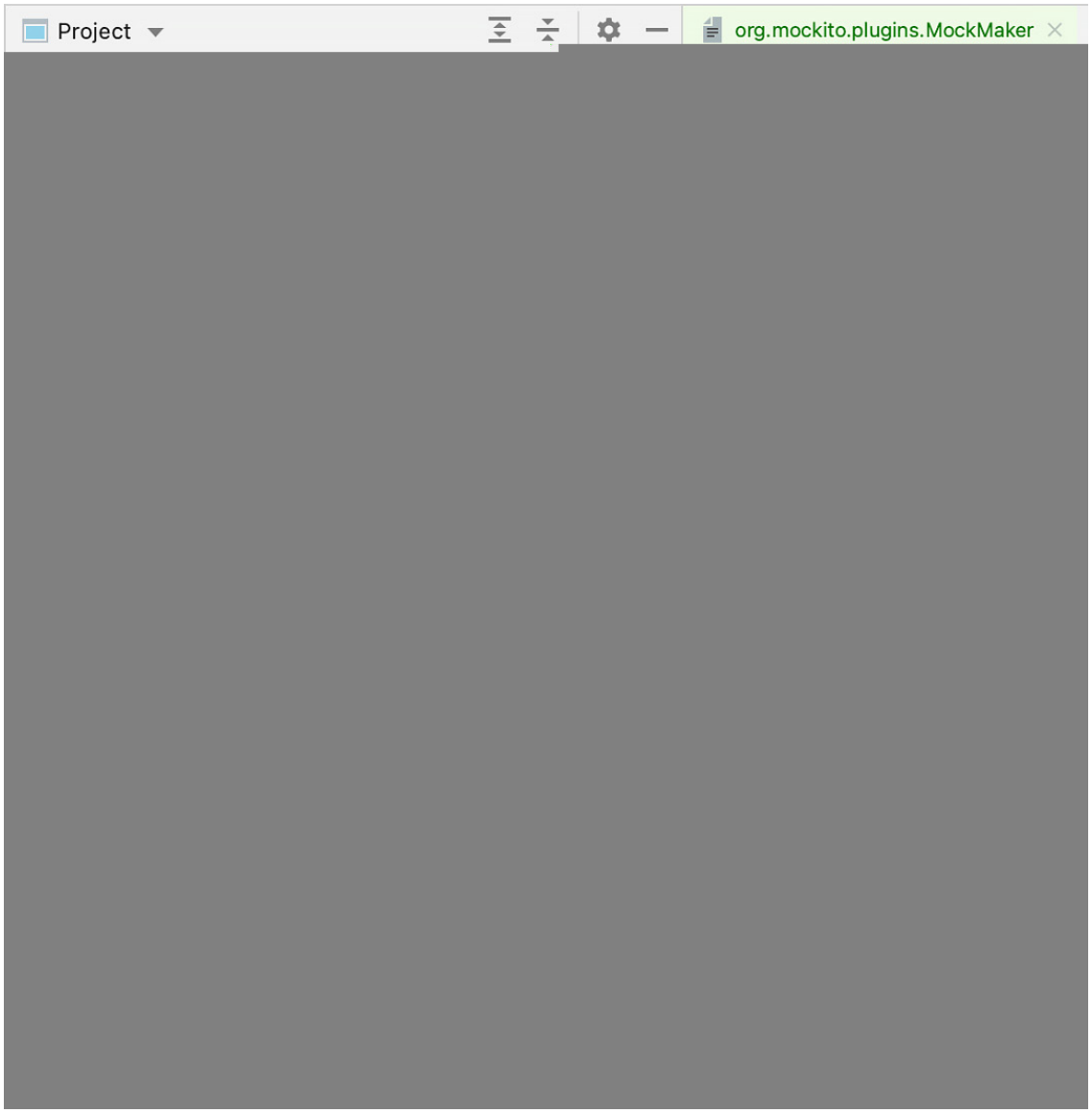

Figure 4.2 – The file you need to add to the app/src/test/mockito-extensions directory

4. In the **`app/src/test/mockito-extensions/org.mockito.plugins.MockMaker`** file, add the following content:

```
mock-maker-inline
```

This will allow you to create mocks using Mockito for final classes in your code. Without this, your test will fail with the following error message:

```
Mockito cannot mock/spy because : final class
```

5. You will first add a unit test for the **MovieRepository** class. In **app/src/test**, create a test class called **MovieRepositoryTest** and add the **@OptIn(ExperimentalCoroutinesApi::class)** annotation to this class:

```
@OptIn(ExperimentalCoroutinesApi::class)
class MovieRepositoryTest {
    ...
}
```

This will be the test class for **MovieRepository**. The **ExperimentalCoroutinesApi OptIn** annotation was added as some of the classes in the **kotlinx-coroutines-test** library are still marked as experimental.

6. Inside the **MovieRepositoryTest** class, add a JUnit test rule for **InstantTaskExecutorRule**:

```
@get:Rule
val rule = InstantTaskExecutorRule()
```

InstantTaskExecutorRule allows the test to execute the tasks synchronously. This is needed for the **LiveData** objects in **MovieRepository**.

7. Create a test function called **fetchMovies** to test the **fetchMovies** suspending function from **MovieRepository**, successfully retrieving a list of movies:

```
@Test
fun fetchMovies() {
    ...
}
```

This will be the first test for **MovieRepository.fetchMovies**: a success scenario that displays a list of movies.

8. In the **MovieRepositoryTest** class' **fetchMovies** function, add the following code to mock **MovieRepository** and **MovieService**:

```
@Test
fun fetchMovies() {
    val movies = listOf(Movie(id = 3), Movie(id =
4))
    val response = MoviesResponse(1, movies)
    val movieService: MovieService = mock {
        onBlocking { getMovies(anyString()) }
doReturn
          response
    }
    val movieRepository =
      MovieRepository(movieService)
}
```

This will mock **`MovieService`** so that when its **`getMovies`** function is called, it will always return the **`movies`** list we provided.

9. At the end of the **`fetchMovies`** function of **`MovieRepositoryTest`**, add the following to test that calling **`fetchMovies`** from the **`MovieRepository`** class returns the list of movies we expect it to return:

```
@Test
fun fetchMovies() {
    ...
    runTest {
        movieRepository.fetchMovies()
        val movieLiveData = movieRepository.movies
        assertEquals(movies, movieLiveData.value)
    }
}
```

This will call the **`fetchMovies`** function from the **`MovieRepository`** class, which will call **`getMovies`** from **`MovieService`**. We are checking whether it indeed returns the list of movies that we set in the mocked **`MovieService`** earlier.

10. Run the **`MovieRepositoryTest`** class. **`MovieRepositoryTest`** should pass and there should be no errors.
11. Create another test function called **`fetchMoviesWithError`** in the **`MovieRepositoryTest`** class to test the **`fetchMovies`** suspending function from the **`MovieRepository`** failing to retrieve a list of movies:

```
@Test
fun fetchMoviesWithError() {
    ...
}
```

This will test the case when **`MovieRepository`** fails while retrieving the list of movies.

12. In the **`MovieRepositoryTest`** class' **`fetchMoviesWithError`** function, add the following:

```
@Test
fun fetchMoviesWithError() {
    val exception = “Test Exception”
    val movieService: MovieService = mock {
        onBlocking { getMovies(anyString()) }
doThrow
          RuntimeException(exception)
    }
    val movieRepository =
      MovieRepository(movieService)
}
```

This will mock **`MovieService`** so that when its **`getMovies`** function is called, it will always throw an exception with the message **`Test Exception`**.

13. At the end of the **`fetchMoviesWithError`** function of **`MovieRepositoryTest`**, add the following to test that calling **`fetchMovies`** from the **`MovieRepository`** class returns the list of movies we expect it to return:

```
@Test
```

```
fun fetchMovies() {
    ...
    runTest {
        movieRepository.fetchMovies()
        val movieLiveData = movieRepository.movies
        assertNull(movieLiveData.value)
        val errorLiveData = movieRepository.error
        assertNotNull(errorLiveData.value)
        assertTrue(errorLiveData.value.toString()
          .contains(exception))
        }
}
```

This will call the **`fetchMovies`** function from the **`MovieRepository`** class, which will call the **`getMovies`** from the **`MovieService`** that will always throw an exception when called.

In the first assertion, we are checking that **`movieLiveData`** is null as there were no movies fetched. The second assertion checks that **`errorLiveData`** is not null as there was an exception. The last assertion checks that **`error-LiveData`** contains the **`Test Exception`** message we set in the previous step.

14. Run the **`MovieRepositoryTest`** test. Both the **`fetchMovies`** and **`fetch-MoviesWithError`** tests should have no errors and both should pass.
15. We will then create a test for **`MovieViewModel`**. First, we would need to update **`MovieViewModel`** so that we can change the dispatcher that the coroutine runs on. Open the **`MovieViewModel`** class and update its constructor by adding a dispatcher property to set the coroutine dispatcher:

```
class MovieViewModel(private val movieRepository:
  MovieRepository, private val dispatcher:
  CoroutineDispatcher = Dispatchers.IO) :
ViewModel()
  {
    ...
```

```
}
```

This will allow you to change the dispatcher of **`MovieViewModel`** with another dispatcher, which you will be doing in the tests.

16. In the **`fetchMovies`** function, change the **`launch`** coroutine builder to use the **`dispatcher`** from the constructor instead of the hardcoded dispatcher:

```
viewModelScope.launch(dispatcher) {
    ...
}
```

This updates the code to use the **`dispatcher`** set from the constructor or the default dispatcher (**`Dispatchers.IO`**). You can now create a unit test for the **`MovieViewModel`** class.

17. In the **`app/src/test`** directory, create a test class named **`MovieViewModelTest`** for **`MovieViewModel`** and add the **`@OptIn(ExperimentalCoroutinesApi::class)`** annotation to the class:

```
@OptIn(ExperimentalCoroutinesApi::class)
class MovieViewModelTest {
    ...
}
```

This will be the test class for **`MovieViewModel`**. The **`ExperimentalCoroutinesApi`** annotation was added as some of the classes in the **`kotlinx-coroutines-test`** library are still experimental.

18. Inside the **`MovieViewModelTest`** class, add a JUnit test rule for **`InstantTaskExecutorRule`**:

```
@get:Rule
val rule = InstantTaskExecutorRule()
```

The **`InstantTaskExecutorRule`** in the unit test executes the tasks synchronously. This is for the **`LiveData`** objects in **`MovieViewModel`**.

19. Create a test function called **`fetchMovies`** to test the **`fetchMovies`** suspending function from **`MovieViewModel`**:

```
@Test
fun fetchMovies() {
    val expectedMovies =
      MutableLiveData<List<Movie>>()
    expectedMovies.postValue(listOf(Movie
      (title = “Movie”)))
    val movieRepository: MovieRepository = mock {
        onBlocking { movies } doReturn
expectedMovies
    }
}
```

This will mock **`MovieRepository`** so that its **`movies`** property will always return the **`expectedMovies`** as its value.

20. At the end of the **`fetchMovies`** test of **`MovieViewModelTest`**, add the following to test that **`MovieViewModel`**'s movies will be equal to **`expectedMovies`**:

```
@Test
fun fetchMovies() {
    ...
    val movieViewModel =
      MovieViewModel(movieRepository)
    assertEquals(expectedMovies.value,
      movieViewModel.movies.value)
}
```

This creates a **`MovieViewModel`** using the mocked **`MovieRepository`**. We are checking that the value of **`MovieViewModel`**'s **`movies`** is equal to the **`expectedMovies`** value we set to the mocked **`MovieRepository`**.

21. Run **`MovieViewModelTest`** or all the tests (**`MovieRepositoryTest`** and **`MovieViewModelTest`**). All tests should pass.

22. Create another test function called **`loading`** in **`MovieViewModelTest`** to test the **`loading LiveData`** in **`MovieViewModel`**:

```
@Test
fun loading() {
    ...
}
```

This will test the **`loading LiveData`** property of **`MovieViewModel`**. The loading property is **`true`** while fetching the movies and displays the **`ProgressBar`**. It becomes **`false`** and hides the **`ProgressBar`** after successfully fetching the movies or when an error is encountered.

23. In the **`loading`** test function, add the following to mock **`MovieRepository`** and initialize a dispatcher that will be used for **`MovieViewModel`**:

```
@Test
fun loading() {
    val movieRepository: MovieRepository = mock()
    val dispatcher = StandardTestDispatcher()
    ...
}
```

This will mock **`MovieRepository`** and create a dispatcher of the **`StandardTestDispatcher`** type that will be used for the **`MovieViewModel`** test. This will allow you to control the execution of the task, which will be used later to check the value of **`MovieViewModel`**'s **`loading`** property.

24. At the end of the **`loading`** test function, add the following to test the loading **`MovieViewModel`**'s **`loading`** property:

```
@Test
fun loading() {
    ...
    runTest {
        val movieViewModel =
          MovieViewModel(movieRepository,
dispatcher)
```

```
            movieViewModel.fetchMovies()
            assertTrue( movieViewModel.loading.value ==
              true)
            dispatcher.scheduler.advanceUntilIdle()
            assertFalse(movieViewModel.loading.value ==
              true)
        }
    }
```

This will create a **`MovieViewModel`** with the mock **`MovieRepository`** and **`dispatcher`** you created in the previous step. Then, **`fetchMovies`** will be called from **`MovieViewModel`** to fetch the list of movies.

The first assertion checks whether the **`loading`** value is **`true`**. We then used **`advanceUntilIdle`** from the dispatcher's **`scheduler`** to execute all the tasks. This should change the **`loading`** value to **`false`**. The last line checks this indeed happens.

25. Run both **`MovieRepositoryTest`** and **`MovieViewModelTest`**. All the tests should pass.

In this exercise, you worked on an Android project that uses coroutines and you have added unit tests for these coroutines.

Summary

This chapter focused on testing coroutines in your Android app. You started with learning how to set up your Android project in preparation for adding tests for your coroutines. The coroutines testing library (**`kotlinx-coroutines-test`**) helps you to create tests for your coroutines.

You learned how to add unit tests for your suspending functions. You can use **`runBlocking`** and **`runTest`** to test code that calls suspending functions. **`runTest`** runs the code immediately, without delays.

Then, you learned how to test coroutines. You can change the dispatcher in your test with a **`TestDispatcher`** (**`StandardTestDispatcher`** or **`UnconfinedTestDispatcher`**). **`TestCoroutineScheduler`** allows you to control the execution of the coroutine task.

Finally, you worked on an exercise where you added unit tests for coroutines in an existing Android project.

In the next chapter, you will explore Kotlin Flows and learn how you can use them for asynchronous programming in Android.

Further reading

This book assumes that you already have knowledge of testing Android applications. If you would like to learn more about Android testing, you can read *Chapter 9*, *Unit Tests and Integration Tests with JUnit, Mockito, and Espresso*, from the book *How to Build Android Apps with Kotlin (Packt Publishing, 2021, ISBN 9781838984113)*. You can also check the Android testing documentation at https://developer.android.com/training/testing ↗️.

As of the time of writing, the coroutine testing library is still marked as experimental. Before the library becomes stable later, there might be some code-breaking changes to the classes. You can check the latest version of the library on GitHub at https://github.com/Kotlin/kotlinx.coroutines/tree/master/kotlinx-coroutines-test ↗️ to find the latest information about the coroutine testing library.