

Chapter 7: Testing Kotlin Flows

In the previous chapter, we focused on understanding Kotlin Flow cancellation, learning how to make Flows cancellable, and handling the cancellation. We also learned about retrying tasks with Flows and handling completion and exceptions in your Flows.

Adding tests for the Kotlin Flows in your code is an important part of app development. Tests will ensure that the Flows we add to our projects are free of bugs or errors and that they will work as we intended. They can make developing apps easier and help you refactor and maintain your code confidently.

In this chapter, we will learn how to test Kotlin Flows in Android. First, we will understand how to set up your Android project for testing Flows. We will then proceed with creating and running tests for Kotlin Flows.


This chapter covers the following main topics:

- Setting up an Android project for testing Flows
- Testing Kotlin Flows
- Testing Flows with Turbine


By the end of this chapter, you will have learned about Kotlin Flow testing. You will be able to write and run unit and integration tests for the Flows 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
- A minimum of 4 GB of RAM
- 4 GB of 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/Chapter07> .

Setting up an Android project for testing Flows

In this section, we will start by looking at how to set up our Android project for testing Kotlin Flows. Once we have done that, it will be easy for us to add unit and integration tests for the Flows in our project.

To create a unit test in Android, your project must have the JUnit 4 testing library, a unit testing framework for Java. New projects created in Android Studio should already have this added in the **app/build** dependencies. If your project does not have JUnit yet, you can add it by adding the following in your **app/build.gradle** dependencies:

```
dependencies {  
  
    ...  
  
    testImplementation 'junit:junit:4.13.2'  
  
}
```

Adding this to your dependencies enables you to use the JUnit 4 testing framework to unit-test your code.

It is also a good idea to use mock objects for your tests. Mockito is a popular Java mocking library that you can use on Android. You can also use Mockito-Kotlin to use Mockito with idiomatic Kotlin code. To add Mockito and Mockito-Kotlin to your Android tests, you can add the following in your **app/build.gradle** dependencies:

```
dependencies {  
  
    ...  
  
    testImplementation 'org.mockito:mockito-core:4.0.0'  
  
    testImplementation 'org.mockito.kotlin:mockito-  
  
        kotlin:4.0.0'  
  
}
```

This will allow you to use Mockito to create mock objects for your Android tests using Kotlin-like code. Mockito-Kotlin has a dependency to **mockito-core** so you can simply use the following to import both **mockito-core** and **mockito-kotlin**:

```
dependencies {  
  
    ...  
  
    testImplementation 'org.mockito.kotlin:mockito-  
  
        kotlin:4.0.0'  
  
}
```

As Kotlin Flow is built on top of coroutines, you can use the **kotlinx-coroutines-test** library to help you add tests for both coroutines and Flows. This library contains utility classes to make the writing of tests easier. To add it to your project, you can add the following to your **app/build.gradle** dependencies:

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

Adding this allows you to use the **kotlinx-coroutines-test** library for testing coroutines and flows in your project.

In this section, we have learned about setting up our Android project to test Kotlin Flows. We will learn about testing Kotlin Flows in the next section.

Testing Kotlin Flows

In this section, we will focus on testing Kotlin Flows. We can create unit and integration tests for classes such as **ViewModel** that use Flow in their code.

To test code that collects a Flow, you can use a mock object that can return values which you can do assertion checks. For example, if your **ViewModel** listens to the Flow from a repository, you can create a custom **Repository** class that emits a Flow with a predefined set of values for easier testing.

For example, say you have a **MovieViewModel** class such as the following that has a **fetchMovies** function that collects a Flow:

```
class MovieViewModel(private val movieRepository:  
  
    MovieRepository) {
```

```

    ...

    suspend fun fetchMovies() {

        movieRepository.fetchMovies().collect {

            _movies.value = it

        }

    }

}

```

Here, the **fetchMovies** function collects a Flow from **movieRepository.fetchMovies()**. You can write a test for this **MovieViewModel** by creating **MovieRepository**, which returns a specific set of values, which you will check to see whether it's the same value that will be set to the movies **LiveData** in **MovieViewModel**. An example implementation of this looks like the following:

```

class MovieViewModelTest {

    ...

    @Test

    fun fetchMovies() {

        ...

        val list = listOf(movie1, movie2)

        val expected = MutableLiveData<List<Movie>>()

        expectedMovies.value = list

        val movieRepository: MovieRepository = mock {

```

```

        onBlocking { fetchMoviesFlow() } doReturn

        flowOf(movies)

    }

    val dispatcher = StandardTestDispatcher()

    val movieViewModel =

        MovieViewModel(movieRepository, dispatcher)

    runTest {

        movieViewModel.fetchMovies()

        dispatcher.scheduler.advanceUntilIdle()

        assertEquals(expectedMovies.value,

            movieViewModel.movies.value)

        ...

    }

}

```

In this example, **fetchMoviesFlow** of **MovieRepository** returns a list of movies that has only one item. After calling **movieViewModel.fetchMovies()**, the test checks whether the value in the **MovieViewModel.movies LiveData** was set to this list.

You can also test a Flow by collecting it to another object. You can do that by converting the Flow to a list with **toList()** or to a set with **toSet()**, getting the first item with **first**, taking items with **take()**, and other terminal

operators. Then, you can check the values returned with the expected values.

For example, say you have **MovieViewModel**, which has a function that returns a Flow, such as the following class:

```
class MovieViewModel(private val movieRepository:
    MovieRepository) {
    ...
    fun fetchFavoriteMovies(): Flow<List<Movie>> {
        ...
    }
}
```

Here, the **fetchFavoriteMovies** function returns a Flow of **List<Movie>**. You can write a test for this function by converting **Flow<List<Movie>>** into a list, as shown in the following example:

```
class MovieViewModelTest {
    ...
    @Test
    fun fetchFavoriteMovies() {
        ...
        val expectedList = listOf(movie1, movie2)
        val movieRepository: MovieRepository = mock {
            onBlocking { fetchFavoriteMovies() } doReturn

```

```
        flowOf(expectedList)

    }

    val movieViewModel =

        MovieViewModel(movieRepository)

    runTest {

        ...

        assertEquals(expectedList,

            movieViewModel.fetchFavoriteMovies().toList())

    }

}
```

In this example, you converted the Flow of the list of movies from `movieViewModel.fetchFavoriteMovies()` to a list of movies and compared it with the expected list.

To test error-handling in Flow, you can mock your test objects to throw an exception. You can then check the exception thrown or the code that handles it. The following example shows how you can write tests for a Flow's failure case:

```
class MovieRepositoryTest {

    ...

    @Test
```



```
fun fetchMoviesFlowWithError() {  
  
    val movieService: MovieService = mock {  
  
        onBlocking { getMovies(anyString()) } doThrow  
  
        IOException(exception)  
  
    }  
  
    val movieRepository = MovieRepository(movieService)  
  
    runTest {  
  
        movieRepository.fetchMoviesFlow().catch {  
  
            assertEquals(exception, it.message)  
  
        }  
  
    }  
  
}
```

In this test class, every time **MovieService.getMovies()** is called, it will throw **IOException**. We then call **movieRepository.fetchMoviesFlow()** and use the **catch** operator to handle the exception. Then, we compare the exception message with the expected string.

We can also test Flow retries by mocking our class to return a specific exception that would trigger a retry. For retries that still fail afterward, you can check the exception or the exception handling. To test retries that

succeed, you can mock your class to either throw an exception or return a Flow that you can compare with the expected values.

The following example shows how you can test a Flow that has a retry for **IOException** and any number of attempts:

```
class MovieViewModelTest {  
  
    ...  
  
    @Test  
  
    fun fetchMoviesWithError() {  
  
        ...  
  
        val movies = listOf(Movie(title = "Movie"))  
  
        val exception = "Exception"  
  
        val hasRetried = false  
  
        val movieRepository: MovieRepository = mock {  
  
            onBlocking { fetchMoviesFlow() } doAnswer {  
  
                flow {  
  
                    if (hasRetried) emit(movies) else throw  
  
                        IOException (exception)  
  
                }  
  
            }  
  
        }  
  
    }  
  
}
```

```

        }

    }

    ...

}

}

```


Here, we used a **hasRetried** variable to determine whether to return a Flow of movies or to throw an exception that can trigger a retry. It is **false** by default to allow a retry. Later in the code, we can change this value to **true** to return a Flow of movies, which we can then compare to the expected values.

In this section, we learned how to create and run tests for Kotlin Flows in our Android project. We will learn about testing hot flows with Turbine in the next section.

Testing Flows with Turbine

In this section, we will learn how to test Flows using Turbine, which is a third-party library that we can use to test flows in our project.

Hot flows such as **SharedFlow** and **StateFlow**, as you learned in the previous chapter, emit values even if there are no listeners. They also keep emitting values and do not complete. Testing them is a bit more complicated. You won't be able to convert these flows to a list and then compare it to the expected values.

To test hot flows and make testing other Flows easier, you can use a library from Cash App called Turbine (<https://github.com/cashapp/turbine> ). Turbine is a small testing library for Kotlin Flow that you can use in Android.

You can use the Turbine testing library in your Android project by adding the following to your **app/build.gradle** dependencies:

```
dependencies {  
  
    ...  
  
    testImplementation 'app.cash.turbine:turbine:0.8.0'  
  
}
```

Adding this will allow you to use the Turbine testing library in your project to test the Flow in your code.

Turbine has a **test** extension function on Flow. It has a suspending validation block, where you can consume items from the Flow one by one and compare them with the expected values. It will then cancel the Flow at the end of the validation block.

An example of using Turbine and the **test** extension function to test Flows is shown in the following code block:

```
class MovieViewModelTest {  
  
    ...  
  
    @Test  
  
    fun fetchMovies() {  
  
        ...  
  
        val expectedList = listOf(movie1, movie2)  
  
        val movieRepository: MovieRepository = mock {  
  
            onBlocking { fetchMovies() } doReturn
```

```
        flowOf(expectedList)

    }

    val movieViewModel =

        MovieViewModel(movieRepository)

    runTest {

        movieViewModel.fetchMovies().test {

            assertEquals(movie1, awaitItem())

            assertEquals(movie2, awaitItem())

            awaitComplete()

        }

    }

}
```

Here, the test used an **awaitItem()** function to get the next item emitted by the Flow and compared it with the expected items. Then, it used an **awaitComplete()** function to assert that the Flow had completed.

To test for exceptions thrown by the Flow, you can use the **awaitError()** function that returns **Throwable**. You can then compare this **Throwable** to the one you expected to be thrown. The following example shows how you can use this to test your Flow:

```
class MovieViewModelTest {

    ...

}
```

```
@Test

fun fetchMoviesError() {

    ...

    val exception = "Test Exception"

    val movieRepository: MovieRepository = mock {

        onBlocking { fetchMovies() } doAnswer

            flow {

                throw RuntimeException(exception)

            }

    }

    }//mock

    val movieViewModel =

        MovieViewModel(movieRepository)

    runTest {

        movieViewModel.fetchMovies().test {

            assertEquals(exception,

                awaitError().message)

        }

    }

}
```

```
}
```

In this example, we used the `awaitError()` function to receive the exception and compare its message with the expected exception.

To test hot flows, you have to emit values inside the `test` lambda. You can also use the `cancelAndConsumeRemainingEvents()` function or the `cancelAndIgnoreRemainingEvents()` function to cancel any remaining events from the Flow.

The following shows an example of using the `cancelAndIgnoreRemainingEvents()` function after checking the first item from the Flow:

```
class MovieViewModelTest {

    ...

    @Test

    fun fetchMovies() {

        ...

        val expectedList = listOf(movie1, movie2)

        val movieRepository: MovieRepository = mock {

            onBlocking { fetchMovies() } doReturn

                flowOf(expectedList)

        }

        val movieViewModel =

            MovieViewModel(movieRepository)

        runTest {
```

```
        movieViewModel.fetchMovies().test {  
  
            assertEquals(movie1, awaitItem())  
  
            cancelAndIgnoreRemainingEvents()  
  
        }  
  
    }  
  
}
```

Here, the test will check the first item from the Flow, ignore any remaining items, and cancel the Flow.

In this section, you have learned how to test Flows with Turbine. Let's try what we have learned so far by adding some tests to Flows in an Android project.

Exercise 7.01 – Adding tests to Flows in an Android app

For this exercise, you will be continuing the movie app you worked on in *Exercise 6.01 – Handling Flow exception in an Android app*. This application displays the movies that are currently playing in movie theatres. You will be adding tests for the Kotlin Flows in the project by following these steps:

1. Open in Android Studio the movie app you worked on in *Exercise 6.01 – Handling Flow exception in an Android app*.
2. Go to the **MovieViewModelTest** class. Run the test class, and the **fetchMovies()** test function will fail. That is because we changed the implementation to use Flow in the previous chapter.

3. Remove the content of the **fetchMovies()** test function and replace it with the following content:

```
@Test
fun fetchMovies() {
    val dispatcher = StandardTestDispatcher()
    val movies = listOf(Movie(title = "Movie"))
    val expectedMovies =
        MutableLiveData<List<Movie>>()
    expectedMovies.postValue(movies)
    val movieRepository: MovieRepository = mock {
        onBlocking { fetchMoviesFlow() } doReturn
            flowOf(movies)
    }
    val movieViewModel =
        MovieViewModel(movieRepository, dispatcher)
}
```

With this code, we will be mocking **MovieRepository** to return a Flow of a list of movies, **movies**, which contains a single movie.

4. At the end of the **fetchMovies()** function, add the following code to test the **fetchMovies()** function of **MovieViewModel**:

```
@Test
fun fetchMovies() {
    ...
    runTest {
        movieViewModel.fetchMovies()
        dispatcher.scheduler.advanceUntilIdle()
        assertEquals(expectedMovies.value,
            movieViewModel.movies.value)
    }
}
```

This will call the **fetchMovies()** function from **movieViewModel**. We will then compare the returned **movieViewModel.movies** to see whether they

are the same as the expected **movies** list (with a single **Movie** item).

5. In the **loading()** test function, replace the assertions with the following:

```
assertTrue(movieViewModel.loading.value)
dispatcher.scheduler.advanceUntilIdle()
assertFalse(movieViewModel.loading.value)
```

The **loading** variable is no longer nullable, so this simplifies the assertion statements.

6. Run the **MovieViewModelTest** class again. It should successfully run, and all the tests will pass.
7. Open the **MovieRepositoryTest** class. We will be adding tests for the **fetchMoviesFlow()** function of **MovieRepository**. First, add the following function to test the successful case of the function:

```
@Test
fun fetchMoviesFlow() {
    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)
    runTest {
        movieRepository.fetchMoviesFlow().collect {
            assertEquals(movies, it)
        }
    }
}
```

This will mock **MovieRepository** to always return the list of movies that we will later compare with the movies from the **fetchMoviesFlow()** function.

8. Add the following function to add a test for the case when the **fetchMoviesFlow()** function throws an exception:

```
@Test
fun fetchMoviesFlowWithError() {
    val exception = "Test Exception"
    val movieService: MovieService = mock {
        onBlocking { getMovies(anyString()) }
    }
    doThrow
        RuntimeException(exception)
    }
    val movieRepository =
        MovieRepository(movieService)
    runTest {
        movieRepository.fetchMoviesFlow().catch {
            assertEquals(exception, it.message)
        }
    }
}
```

This test will use a fake **MovieRepository** that will always throw an error when calling **fetchMoviesFlow**. We will then test whether the exception thrown will be the same as the one that we expect.

9. Run the **MovieRepositoryTest** class. All the tests in **MovieRepositoryTest** should run and pass without an error.
10. Now, we will use the Turbine testing library to test the Flow from the **fetchMoviesFlow()** function of **MovieRepository**. Add the following in the **app/build.gradle** dependencies:

```
testImplementation 'app.cash.turbine:turbine:0.8.0'
```

This will allow us to use the Turbine testing library to create unit tests for Flows in our Android project.

11. Add a new test function to test the success case of the **fetchMoviesFlow()** function by adding the following:

```
@Test
fun fetchMoviesFlowTurbine() {
    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)
    runTest {
        movieRepository.fetchMoviesFlow().test {
            assertEquals(movies, awaitItem())
            awaitComplete()
        }
    }
}
```

With this, we will be mocking **MovieRepository** to return a list of movies. We will later compare that with the list from **movieRepository.fetchMoviesFlow()** using **awaitItem()**. The **awaitComplete()** function will then check that the Flow has terminated.

12. Add another function to test using Turbine in the case when **fetchMoviesFlow** throws an exception by adding the following:

```
@Test
fun fetchMoviesFlowWithErrorTurbine() {
    val exception = "Test Exception"
    val movieService: MovieService = mock {
        onBlocking { getMovies(anyString()) }
    }
    doThrow
```

```
        RuntimeException(exception)
    }
    val movieRepository =
        MovieRepository(movieService)
    runTest {
        movieRepository.fetchMoviesFlow().test {
            assertEquals(exception,
                awaitError().message)
        }
    }
}
```

This will use a **MovieRepository** mock class that will throw **RuntimeException** when calling **fetchMoviesFlow()**. We will then test that the exception message is the same one that was fetched, using the **awaitError()** call.

13. Run the **MovieRepositoryTest** class again. All the tests in **MovieRepository Test** should run and pass without an error.

In this exercise, we have worked on an Android project that uses Kotlin Flow, and we have created tests for these Flows.

Summary

This chapter focused on testing Kotlin Flows in our Android project. We started by setting up the project for adding tests for the Flows. The coroutines testing library (**kotlinx-coroutines-test**) can help you in creating tests for coroutines and Flows.



We learned how to add tests for the Flows in your Android application. You can use a mock class that returns a Flow of values and then compare it with the returned values. You can also convert a Flow into **List** or **Set**,

or take values from the Flow; you can then compare them with the expected values.

Then, we learned about testing hot Flows with Turbine, a third-party testing library for testing Kotlin Flows. Turbine has a **test** extension on Flow where you can consume and compare values one by one.

Finally, we worked on an exercise where we created tests for the Kotlin Flows in an existing Android project. We also used the Turbine testing library to make the writing of tests for Flows easier.

Throughout the book, we have gained knowledge and skills about asynchronous programming in Android. We learned how to use Kotlin coroutines and Flow to simplify asynchronous programming in our Android projects.

Everything in Android is always evolving. There are also more advanced topics about coroutines and Flow that we have not covered. It is good to keep yourself up to date with the latest updates about Android, Kotlin coroutines, and Kotlin Flow. You can find out the latest about coroutines on Android at <https://developer.android.com/kotlin/coroutines>  and the latest about Kotlin Flow on Android at <https://developer.android.com/kotlin/flow> .

Support

Sign Out