Chapter 10: Test Your App with UI and Unit Tests

In the previous chapters, one of our main focuses was to have a testable architecture. We tried to achieve that by decoupling different components from each other.

In this chapter, because of the architecture we put in place, we will see how easy it is to test in isolation different parts of the Restaurants app.

In the *Exploring the fundamentals of testing* section, we will be understanding the benefits of testing and exploring various types of tests. In the *Learning the basics of testing your Compose UI* section, we will learn how to test our Compose UI.

Finally, in the *Covering the basics of unit-testing your core logic* section, we will learn how to test the core functionality of your Restaurants app.

To summarize, in this chapter we will be covering the following sections:

- Exploring the fundamentals of testing
- Learning the basics of testing your Compose UI
- Covering the basics of unit-testing your core logic

Before jumping in, let's set up the technical requirements for this chapter.

Technical requirements

Building Compose-based Android projects for this chapter usually requires your day-to-day tools. However, to follow along smoothly, make sure you also have the following:

- The Arctic Fox 2020.3.1 version of Android Studio. You can also use a newer Android Studio version or even Canary builds but note that the IDE interface and other generated code files might differ from the ones used throughout this book.
- The Kotlin 1.6.10 or newer plugin installed in Android Studio.
- The Restaurants app code from the previous chapter.

The starting point for this chapter is represented by the Restaurants app developed in the previous chapter. If you haven't followed the implementation from the previous chapter, access the starting point for this chapter by navigating to the **`Chapter_09`** directory of the repository and import the Android project entitled **`chapter_9_restaurants_app`**.

To access the solution code for this chapter, navigate to the **`Chapter_10`** directory:https://github.com/PacktPublishing/Kickstart-Modern-Android-Development-with-Jetpack-and-Kotlin/tree/main/Chapter_10/chapter_10_restaurants_app

Exploring the fundamentals of testing

In this section, we will briefly cover the basics of testing. More precisely, we will be doing the following:

- Understanding the benefits of testing
- Exploring the types of tests

Let's start with the benefits of testing!

Understanding the benefits of testing

Testing our code is essential. Through tests, we ensure that our app's functional behavior is correct and as expected, while also making sure

that it's usable, just as it was designed. By performing tests, we can release stable and functional apps to end users.

More importantly, if we develop an app and then test it consistently, we ensure that new updates with new functionality won't break the existing functionality, and no bugs will arise. This is often referred to as **regression testing**.

You can test your app manually by navigating through it on your device or emulator and making sure that every piece of data is displayed correctly, while also being able to interact correctly with every UI component.

However, manual testing is neither efficient nor fast. With manual testing, you must traverse every user flow, generate every user interaction, and verify the integrity of data displayed at any moment. Also, you must do this, consistently, on every application update. Moreover, manual testing scales poorly, as with every new update that contains a new functionality, the manual workload of testing the entire application increases.

Over time, manual testing becomes a burden for medium- and large-sized applications. Also, manual testing involves a human tester, which generates a human factor – this basically means that a tester may or may not in some circumstances overlook some bugs.

To alleviate these issues, in this chapter, we will be writing automated tests. Practically, we will define some scripted tests and then allow tools to run them, automatically. This approach is faster, consistent, and more efficient, as it scales better with the size of the project.

In other words, we will write other chunks of code that will test the code of our application. While this might sound weird, the approach of having automated tests is much more productive and reliable, and less time-consuming than manual testing.

Next up, let's cover the different types of tests that we can write.

Exploring types of tests

To better understand how to write tests, we must first decide *what exactly can be tested* in our apps. From this perspective, let's cover the most important types of tests:

- **Functional tests**: Is the app doing what is expected? We already touched upon functional tests and their benefits in the *Understanding the benefits of testing* section.
- **Compatibility tests**: Is the app working correctly on all devices and Android API levels? The Android ecosystem makes this particularly difficult if you consider the variety of devices and manufacturers.
- **Performance tests**: Is the app fast or efficient enough? Sometimes, apps can suffer from bottlenecks and UI stutters that can be identified via performance benchmarks.
- **Accessibility tests**: Is the app working well with accessibility services? Such services are used to assist users with disabilities in using our Android application.

In this chapter, we will be mainly focusing on functional tests in an attempt to ensure the functional integrity of our application.

Now, apart from deciding what has to be tested, we must also think about the *scope or size of the tests*. The scope indicates the size of the app's portion we're testing. From the perspective of the scope of the tests, we have the following:

- **Unit tests**: Often referred to as small tests, these test the functional behavior of methods, classes, or groups of classes in an isolated environment. Usually, unit tests target small portions of the app without interacting with the real-world environment; hence, they are more reliable than tests that depend on external input.
- **Integration tests**: Often referred to as medium tests, these test whether multiple units interact and function correctly together.

- **End-to-end tests**: Often referred to as big tests, these test large portions of the application, from multiple screens to entire user flows.

Depending on the size of the tests, each type has a degree of isolation. The **degree of isolation** is tightly related to the scope of the tests, as it measures how dependent the component we're testing is on other components. As the size of the test increases, from small to big, the isolation level of the tests decreases.

In this chapter, we will be mainly focusing on unit tests, as they are fast, with the simplest setup, and most reliable in helping us validate the functionality of our application. These traits are tightly related to the higher isolation level of unit tests from external components.

Lastly, we must also classify tests based on the system they will be running on:

- **Local tests**: Run on your workstation or development system (used in practices such as **Continuous Integration (CI)**) without the need of an Android device or emulator. They are usually small and fast, isolating the component under test from the rest of the application. Most of the time, unit tests are local tests.
- **Instrumented tests**: Run on an Android device, be it a physical device or emulator. Most of the time, UI tests are considered instrumented tests, since they allow the automated testing of an application on an Android device.

In this chapter, our unit tests will be local when we will be testing the core logic of some components in isolation and instrumented when we will be performing UI unit tests for a specific screen in isolation.

Let's proceed with local UI tests first!

Learning the basics of testing your Compose UI

UI tests allow us to evaluate the behavior of our Compose code against what is expected to be correct. This way, we can catch bugs early in our UI development process.

To test our UI, we must first decide what we are aiming to evaluate. To keep it simple, in this section, we will unit-test our UI in an isolated environment. In other words, we want to test the following:

- That our composable screens consume the received state as expected. We want to make sure that the UI correctly represents the different state values that it can receive.
- For our composable screens, that user-generated events are correctly forwarded to the caller of the composable.

To keep our tests simple, we will define these tests as unit tests and try to isolate screen composables from their **`ViewModel`** or from other screen composables; otherwise, our test will become an integration or an end-to-end test.

In other words, we will test separately each screen, with total disregard of anything outside of their composable function definition. Even though our tests will run on an Android device, they will be testing only one unit – a screen composable.

NOTE

Some UI tests can also be considered unit tests, as long as they are testing only one part of the UI of your application, as we will do in this section.

For starters, we need to test the first screen of our application, represented by the **`RestaurantsScreen()`** composable. Let's begin!

1. First, add the following testing dependencies inside the **`dependencies`** block of the app-level **`build.gradle`** file:

```
dependencies {
    […]
    androidTestImplementation
"androidx.compose.ui:ui-
        test-junit4:$compose_version"
    debugImplementation "androidx.compose.ui:ui-
test-
        manifest:$compose_version"
}
```

These dependencies will allow us to run our Compose UI tests on an Android device.

After updating the **`build.gradle`** file, make sure to sync your project with its Gradle files. You can do that by pressing on the **File** menu option and then by selecting **Sync Project with Gradle Files**.

2. Before creating a test class, locate the **`androidTest`** package that is suited for instrumented tests:

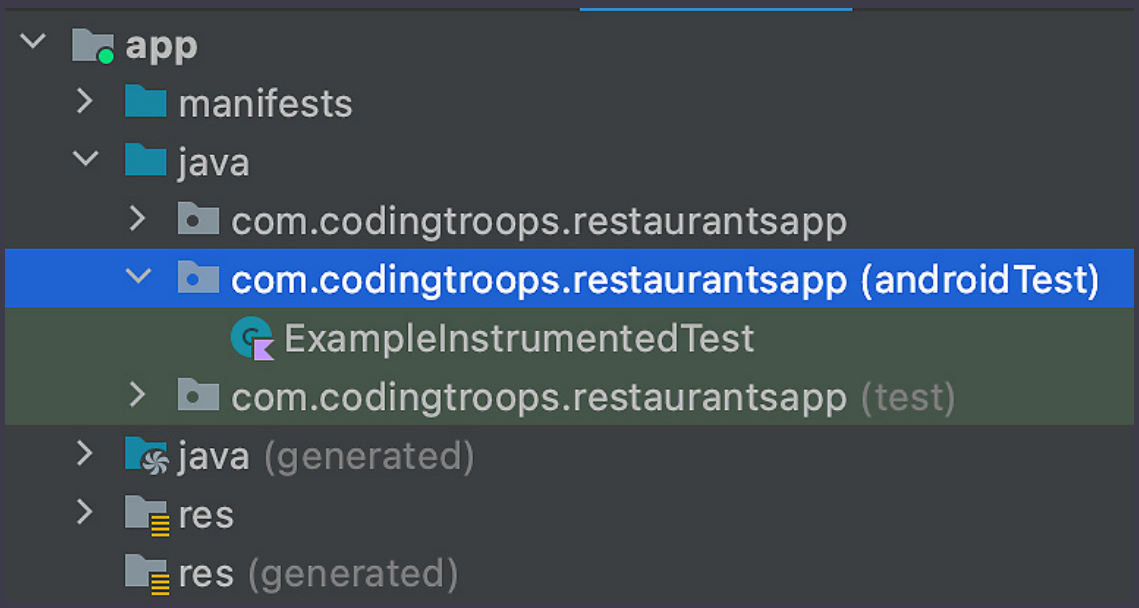

Figure 10.1 – Observing the androidTest package for UI tests

In Android projects, this directory stores source files for UI tests. Also note that the pre-built **`ExampleInstrumentedTest`** class resides in this directory.

3. Create an empty Kotlin class named **`RestaurantsScreenTest`** inside the **`androidTest`** package.

Inside this class, we will define a method for each independent test. Behind the scenes, every method will become a standalone UI test that can pass or fail.

4. Before creating our first test method, inside the **`RestaurantsScreenTest`** class, add the following code:

```
import androidx.compose.ui.test.junit4.*
import org.junit.Rule
class RestaurantsScreenTest {
    @get:Rule
    val testRule: ComposeContentTestRule =
        createComposeRule()
}
```

To run our Compose UI tests, we are using the JUnit testing framework that will allow us to write repeatable unit tests in an isolated environment with the help of a test rule. **Test rules** allow us to add functionality to all the tests within a test class.

In our case, we need to test Compose UI in every test method, so we had to use a special **`ComposeContentTestRule`** object. To access this rule, we have previously imported a special JUnit rule dependency so that our test class now defines a **`testRule`** variable and instantiates it by using the **`createComposeRule()`** method.

`ComposeContentTestRule` will not only allow us to set the Compose UI under test but also host tests on an Android device, while also giving us the

ability to interact with the composables under test or perform UI assertions.

Before writing our first test method though, we need to clearly understand what behavior we are trying to test.

Let's have a look at how our **`RestaurantsScreen()`** composable consumes a **`RestaurantsScreenState`** instance from its **`state`** parameter, and how it forwards events to its caller through the **`onItemClick`** and **`onFavoriteClick`** function parameters:

```
@Composable
fun RestaurantsScreen(
    state: RestaurantsScreenState,
    onItemClick: (id: Int) -> Unit,
    onFavoriteClick: (id: Int, oldValue: Boolean) ->
Unit
) {
    Box(…) {
        LazyColumn(…) {
            items(state.restaurants) { restaurant ->
                RestaurantItem(restaurant,
                   onFavoriteClick = { id, oldValue -
>
                        onFavoriteClick(id, oldValue)
},
                   onItemClick = { id ->
                        onItemClick (id) })
            }
        }
        if(state.isLoading)
            CircularProgressIndicator()
        if(state.error != null)
            Text(state.error)
    }
```

```
}
```

By looking at the previous snippet, we see that we can test how the **`onItemClick`** and **`onFavoriteClick`** functions are called, based on different UI interactions, and also that we can test whether the state is consumed correctly or not. Yet we can't infer very well the possible values of the state that our composable is receiving.

To get an overview of the possible states that we want to feed into our **`RestaurantsScreen()`** so that we can test its behavior, we need to have a look at its state producer, **`RestaurantsViewModel`**:

```
class RestaurantsViewModel @Inject constructor(…) :
ViewModel() {
    private val _state = mutableStateOf(
        RestaurantsScreenState(
            restaurants = listOf(),
            isLoading = true
        )
    )
    […]
    private val errorHandler =
CoroutineExceptionHandler
    { ... ->
        exception.printStackTrace()
        _state.value = _state.value.copy(
            error = exception.message,
            isLoading = false)
    }
    init { getRestaurants() }
    fun toggleFavorite(itemId: Int, oldValue:
Boolean) {
        […]
    }
    private fun getRestaurants() {
        viewModelScope.launch(errorHandler) {
```

```
                val restaurants = getRestaurantsUseCase()
                _state.value = _state.value.copy(
                    restaurants = restaurants,
                    isLoading = false)
            }
        }
    }
```

We can say that our screen should have three possible states:

- **Initial loading state**: At this point, we're waiting for restaurants, thus rendering a loading status. You can see this initial state declared at the top of the **`ViewModel`** class in the initialization of the **`_state`** variable, where the **`restaurants`** parameter of **`RestaurantsScreenState`** is set to **`emptyList()`** and the **`isLoading`** parameter is set to **`true`**.
- **State with content**: The restaurants have arrived, so we reset the loading status and render them. You can see how this state is created inside the coroutine launched in the **`getRestaurants()`** method where we mutated the initial state and set the **`isLoading`** parameter to **`false`**, while also passing the list of restaurants to the **`restaurants`** parameter.
- **Error state**: Something went wrong when the app tried to fetch its content – that is, restaurants. You can see how this state is created inside the block of code exposed by **`CoroutineExceptionHandler`** where the **`isLoading`** parameter is set to **`false`** to reset the loading status, while also passing the message of **`Exception`** to the **`error`** parameter.

Now that we know what behavior the **`RestaurantsScreen`** composable should exhibit and what input we should pass to it in order to produce such a behavior, it's time to actually put our composable screen under test.

Let's begin by verifying whether the **`RestaurantsScreen`** composable correctly renders the first state – that is, the initial loading state.

5. Inside the **`RestaurantsScreenTest`** class, add an empty test function named **`initialState_isRendered()`** that will later test whether our

`RestaurantsScreen()` composable properly renders the initial state:

```
class RestaurantsScreenTest {
    @get:Rule
    val testRule: ComposeContentTestRule =
        createComposeRule()

    @Test
    fun initialState_isRendered() {  }
}
```

To tell the JUnit testing library to run an individual test for this method, we've annotated it with the **`@Test`** annotation.

Also, note that we named this method around the specific behavior it's trying to test, going from what we're testing (the initial state) to how it's supposed to behave (to be rendered correctly), while separating these two with an underscore. For unit tests, there are a lot of naming conventions, yet we will try to stick to the simple version mentioned before.

NOTE

Each test method annotated with **`@Test`** *should focus on only one specific behavior, just as* **`initialState_isRendered()`** *will test whether the* **`RestaurantsScreen()`** *properly renders the initial state, and no other pieces. This allows us to focus on only one behavior on each test method so that we can better identify later which specific behavior is no longer working as expected.*

6. Prepare the **`initialState_isRendered()`** method to set the Compose UI by calling **`testRule.setContent()`**, just as our **`MainActivity`** did with its own **`setContent()`** method:

```
    @Test
    fun initialState_isRendered() {
        testRule.setContent { }
    }
```

7. Inside the block of code exposed by the **`setContent()`** method, we must pass the **unit under test**, which is nothing else than the composable we're trying to test.

In our case, we will pass the **`RestaurantsScreen()`** composable, not before wrapping it inside the **`RestaurantsAppTheme()`** theme function so that the Compose UI that is under test mimics what our app is actually displaying in the production code:

```
@Test
fun initialState_isRendered() {
    testRule.setContent {
        RestaurantsAppTheme {
            RestaurantsScreen()
        }
    }
}
```

If you have named your app name differently, then the theme composable might have a different definition.

8. Now, the **`RestaurantsScreen()`** composable is expecting a **`RestaurantsScreenState`** object into its **`state`** parameter and two functions for its **`onFavoriteClick()`** and **`onItemClick()`** parameters. Let's add these while also passing the expected initial state from the screen's **`ViewModel`**:

```
    @Test
    fun initialState_isRendered() {
        testRule.setContent {
            RestaurantsAppTheme {
                RestaurantsScreen(
                    state = RestaurantsScreenState(
                        restaurants = emptyList(),
                        isLoading = true),
                    onFavoriteClick =
```

```
                                {  _: Int, _: Boolean -
> },
                        onItemClick = { })
                }
            }
        }
```

Since we're looking to test whether **`RestaurantsScreen()`** is correctly rendering the initial state, we have passed an instance of **`RestaurantsScreenState`** that had the **`restaurants`** parameter set to **`emptyList()`**, and the **`isLoading`** parameter is set to **`true`** while the **`error`** parameter is by default set to **`null`**.

We have now finished setting up the **`RestaurantsScreen()`** composable and fed it with the expected initial state. It's time to perform the assertion of whether our composable is correctly consuming this initial state or not.

In the **`RestaurantsScreen()`** composable, the initial state is a state mainly defined by the loading indicator that expresses how the app is waiting for content:

```
@Composable
fun RestaurantsScreen(…) {
    Box(…) {
        LazyColumn(…) {…}
        if(state.isLoading)
            CircularProgressIndicator()
        if(state.error != null)
            Text(state.error)
    }
}
```

That's why we can check whether the **`CircularProgressIndicator()`** is visible on the screen. But how can we assert whether this composable is visible or not?

Compose provides us with several testing APIs to help us find elements, verify their attributes, and even perform user actions. For UI tests with Compose, we consider pieces of UI as **nodes** that we can identify with the help of semantics. **Semantics** give meaning to a UI element, and for an entire composable hierarchy, a semantics tree is generated to describe it.

In other words, we should be able to identify anything that is described on the screen with the help of its exposed semantics.

To give an example, a **`Text`** composable that displays a **`String`** object such as **`"Hello"`** will become a node in the semantics tree that we can identify by its **`text`** property value – **`"Hello"`**. Similarly, composables such as **`Image`** expose a mandatory **`contentDescription`** parameter whose value will allow us to identify the corresponding node in the semantics tree inside our tests. Don't worry – we'll see a practical example of this in a second.

NOTE

*While the semantics attributes are mainly used for accessibility purposes (***`contentDescription`***, for example, is a parameter that allows people with disabilities to better understand what the visual element it describes is about), it's also a great tool that exposes semantics used to identify nodes in our tests.*

Now that we have briefly covered how we can use semantics information to identify UI elements as nodes, it's time to get back to our test, which should validate if, upon an initial state consumed by **`RestaurantsScreen()`**, its **`CircularProgressIndicator()`** is visible.

However, if we look again at the usage of **`CircularProgressIndicator()`**, we can see that it exposes no semantics that we can use to identify it later in our test:

```
@Composable
fun RestaurantsScreen(…) {
    Box(…) {
```

```
            LazyColumn(…) {…}
            if(state.isLoading)
                CircularProgressIndicator()
            […]
        }
    }
```

There is no **contentDescription** parameter and no visual text displayed. To be able to identify the node of **CircularProgressIndicator()** we must manually add a semantics **contentDescription** property.

9. For a moment, let's head out of the **androidTest** directory and go back inside the main package where our production code resides. Inside the **presentation** package, create a new **object** class named **Description** and define a constant description **String** variable for our loading composable:

```
object Description {
    const val RESTAURANTS_LOADING =
            "Circular loading icon"
}
```

10. Inside the **RestaurantsScreen()** composable, pass a **semantics** modifier to the **CircularProgressIndicator()** composable and set its **contentDescription** property to the previously defined **RESTAURANTS_LOADING**:

```
@Composable
fun RestaurantsScreen(…) {
    Box(…) {
        LazyColumn(…) { … }
        if (state.isLoading)
            CircularProgressIndicator(
                Modifier.semantics {
                    this.contentDescription =

Description.RESTAURANTS_LOADING
                })
        […]
    }
```

```
}
```

Now, we will be able to identify the node represented by the **`CircularProgressIndicator()`** composable inside our UI tests by using the **`contentDescription`** semantics property.

11. Now, go back inside the **`androidTest`** directory and navigate to the **`RestaurantsScreenTest`** class, and in the **`initialState_isRendered()`** test method, use the **`testRule`** variable to identify the node with the **`RESTAURANT_LOADING`** content description with the help of the **`onNode-WithContentDescription()`** method, and finally, verify that the node is displayed with the **`assertIsDisplayed()`** method:

```
@Test
fun initialState_isRendered() {
    testRule.setContent {
        RestaurantsAppTheme { RestaurantsScreen(…)
}
    }
    testRule.onNodeWithContentDescription(
        Description.RESTAURANTS_LOADING
    ).assertIsDisplayed()
}
```

NOTE

As you can see with the **`initialState_isRendered()`** *method, every test method has two parts – the setup of the expected behavior and then the assertions that verify that the resultant behavior is correct.*

12. Inside the **Project** tab on the left, right-click on the **`RestaurantsScreenTest`** class and select **Run RestaurantsScreenTest**.

This command will run all the tests inside this class (only one in our case) on an Android device (either your physical Android device or your emulator).

If we switch to the **Run** tab, we will see that our test where we checked whether the initial state of **`RestaurantsScreen()`** was rendered correctly

has passed:

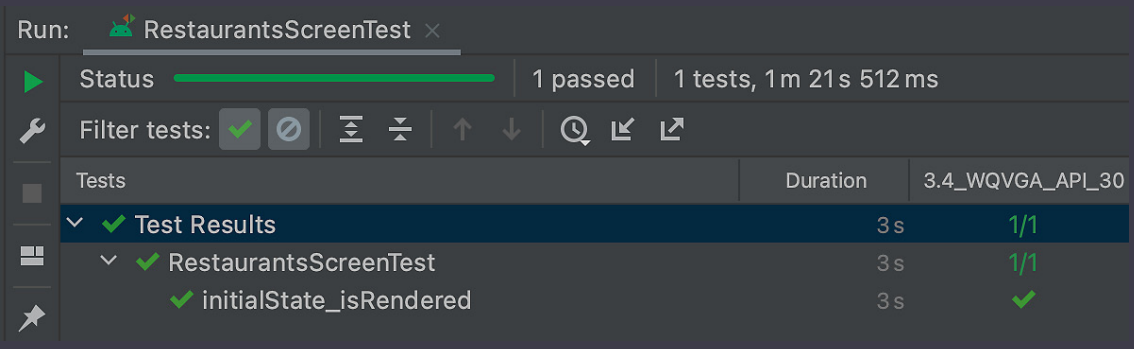

Figure 10.2 – Observing the UI tests that have passed

NOTE

Although we have defined a test where UI elements are identified via semantic properties, it's also possible to match a piece of UI by making it incorporate a **`testTag`** *modifier that is later identified via the* **`hasTestTag()`** *matcher. However, you should avoid this practice, as you will be polluting your Compose UI production code with testing identifiers used only for tests.*

While your test ran on an Android device or emulator, you might have noticed that no UI was shown on its screen. This happens because the UI tests are really fast. If you want to see the UI that you're testing, you can add a **`Thread.sleep()`** call at the end of the test method; however, you should avoid such a practice in your production test code.

Now, it's time to test whether the **`RestaurantsScreen()`** composable is rendering another state correctly – the state with content. In this state, the restaurants have arrived, so we reset the loading status to **`false`** and render the restaurants.

13. Inside the **`RestaurantsScreenTest`** class, add another test function named **`stateWithContent_isRendered()`**, which should test whether the state with content is rendered correctly:

```
@Test
fun stateWithContent_isRendered() {
```

```
    testRule.setContent {
        RestaurantsAppTheme {
            RestaurantsScreen(
                state = RestaurantsScreenState(
                    restaurants =,
                    isLoading = false),
                onFavoriteClick =
                    { _: Int, _: Boolean -> },
                onItemClick = { }
            )
        }
    }
}
```

Inside this test method, we have set the **`RestaurantsScreen()`** composable with a state whose **`isLoading`** field is **`false`** (as the restaurants have arrived) but haven't passed a list of restaurants yet. We need to create a dummy list of restaurants to mimic some restaurants from our data layer.

14. For a moment, let's head out of the **`androidTest`** directory and go back inside the main package where our production code resides. Inside the **`restaurants`** package, create a new **`object`** class named **`DummyContent`**, and inside this class, add a **`getDomainRestaurants()`** method that will return a dummy array list of **`Restaurant`** objects:

```
object DummyContent {
    fun getDomainRestaurants() = arrayListOf(
        Restaurant(0, "title0", "description0",
false),
        Restaurant(1, "title1", "description1",
false),
        Restaurant(2, "title2", "description2",
false),
        Restaurant(3, "title3", "description3",
false))
}
```

15. Now, go back inside the **androidTest** directory and navigate to the **RestaurantsScreenTest** class. Inside the **stateWithContent_isRendered()** method, declare a **restaurants** variable that will hold the dummy restaurants from the **DummyContent** class and pass it to the **restaurants** parameter of **RestaurantsScreenState**:

```
@Test
fun stateWithContent_isRendered() {
    val restaurants =
DummyContent.getDomainRestaurants()
    testRule.setContent {
        RestaurantsAppTheme {
            RestaurantsScreen(
                state = RestaurantsScreenState(
                    restaurants = restaurants,
                    isLoading = false), […])
        }
    }
}
```

Now that we have finished the setup part of this test method, it's time to perform our assertions. Since we are testing that **RestaurantsScreen()** is correctly rendering the state that contains restaurants, let's have another quick look at the composable under test:

```
@Composable
fun RestaurantsScreen(state: RestaurantsScreenState,
[…]) {
    Box(…) {
        LazyColumn(…) {
            items(state.restaurants) { restaurant ->
                RestaurantItem(…)
             }
        }
        if(state.isLoading)
            CircularProgressIndicator()
```

```
            if(state.error != null)
                Text(state.error)
        }
    }
```

We can deduct that the two conditions we can assert are as follows:

- The restaurants from **`RestaurantsScreenState`** are displayed on the screen.
- The **`CircularProgressIndicator()`** composable is not rendered, so its node is not visible on the screen.

Let's start off with the first assertion. Instead of relying on the **`contentDescription`** semantic property, we can use another semantic property that is more obvious – the text displayed on the screen. Since **`LazyColumn`** will render a list of **`RestaurantItem()`** composables, each one will call a **`Text`** composable that will render the title and the description of the restaurant passed to its **`text`** parameter. With the help of our **`ComposeContentTestRule`**, we can identify a node with a certain text value by calling the **`onNodeWithText()`** method.

16. Back in the **`stateWithContent_isRendered()`** method, let's assert that **`title`** of the first **`Restaurant`** object from our dummy list is visible.

Do that by passing the title of the first element from the **`restaurants`** variable to the **`onNodeWithText()`** method, thereby identifying its corresponding node. Finally, call the **`assertIsDisplayed()`** method to verify whether this node is displayed:

```
    @Test
    fun stateWithContent_isRendered() {
        val restaurants =
    DummyContent.getDomainRestaurants()
        testRule.setContent {
            RestaurantsAppTheme {
                RestaurantsScreen(
```

```
                state = RestaurantsScreenState(
                    restaurants = restaurants,
                    isLoading = false
                ),
                […])
        }
    }
    testRule.onNodeWithText(restaurants[0].title)
        .assertIsDisplayed()
}
```

17. Similarly, to assert whether the node of the title of the first restaurant from our dummy list is displayed, verify whether the node of the description of the first restaurant is displayed:

```
@Test
fun stateWithContent_isRendered() {
    val restaurants =
DummyContent.getDomainRestaurants()
    testRule.setContent { ... }
    testRule.onNodeWithText(restaurants[0].title)
        .assertIsDisplayed()
    testRule.onNodeWithText(restaurants[0].descript
ion)
        .assertIsDisplayed()
}
```

You might be wondering why we aren't asserting whether **`title`** or **`description`** of all the elements from the **`DummyContent`** class is visible. It's important to understand that our test is asserting whether some nodes are displayed on the screen.

That's why, if our **`restaurants`** list contained 10 or 15 elements and we tested whether all titles and description nodes are visible, we could have had this test method pass on tall devices, since all the restaurants would fit and would be *composed* on the screen, but it could have failed if the

test device was small and only some of the restaurants fitted on the screen and were composed.

This would have made our test flaky. To prevent our test from being flaky, we only asserted whether the first restaurant is visible, therefore minimizing the chance of having the test run and fail on an incredibly small screen.

Another interesting tactic that you can employ in order to test that content is correctly rendered would be to emulate a scroll action inside your test to the bottom of the list and check whether the last element is visible. This, however, is more complex, so we will proceed with the simpler version that we have implemented.

18. Lastly, let's assert that the node corresponding to the **`CircularProgressIndicator()`** composable does not exist, therefore ensuring that the app is not loading anything anymore. Do that by calling the **`assertDoesNotExist()`** method on the node with the **`RESTAURANTS_LOADING`** content description:

```
@Test
fun stateWithContent_isRendered() {
    val restaurants =
DummyContent.getDomainRestaurants()
    testRule.setContent { … }
    testRule.onNodeWithText(restaurants[0].title)
        .assertIsDisplayed()
    testRule.onNodeWithText(restaurants[0].descript
ion)
        .assertIsDisplayed()
    testRule.onNodeWithContentDescription(
        Description.RESTAURANTS_LOADING
    ).assertDoesNotExist()
}
```

19. Now that we have finished writing our second test method asserting whether the **`RestaurantsScreen()`** composable is correctly rendering

the state with content, inside the **Project** tab on the left, right-click on the `RestaurantsScreenTest` class and select **Run RestaurantsScreenTest**.

The tests should run and pass.

ASSIGNMENT

Try writing a test method on your own that asserts whether the `RestaurantsScreen()` *composable renders the error state correctly. As a hint, you should be passing an error text to the* `error` *parameter of the* `RestaurantsScreen()`*, and then you should be asserting whether a node with that particular text is visible, while also verifying that the node corresponding to the* `CircularProgressIndicator()` *composable does not exist.*

Finally, let's write a test method where we can verify whether upon clicking on a restaurant element from our dummy list, the correct callback is exposed by the parent `RestaurantsScreen()` composable:

20. Inside the `RestaurantsScreenTest` class, add another test function named `stateWithContent_ClickOnItem_isRegistered()`. Inside this method, store the dummy list inside a `restaurants` variable, and then store the first restaurant that we will click upon inside the `targetRestaurant` variable:

```
@Test
fun stateWithContent_ClickOnItem_isRegistered() {
    val restaurants =
DummyContent.getDomainRestaurants()
    val targetRestaurant = restaurants[0]
}
```

21. Then, set `RestaurantsScreen()` under test and feed it with a state with content by passing the contents of the `restaurants` variable to the `restaurants` parameter of `RestaurantsScreenState`:

```
@Test
fun stateWithContent_ClickOnItem_isRegistered() {
```

```
    val restaurants =
DummyContent.getDomainRestaurants()
    val targetRestaurant = restaurants[0]
    testRule.setContent {
        RestaurantsAppTheme {
            RestaurantsScreen(
                state = RestaurantsScreenState(
                    restaurants = restaurants,
                    isLoading = false),
                onFavoriteClick = { _, _ -> },
                onItemClick = { id ->  })
        }
    }
}
```

22. Then, identify the node that contains the **`title`** text of **`targetRestaurant`** and then simulate a user click on this node by calling the **`performClick()`** method:

```
@Test
fun stateWithContent_ClickOnItem_isRegistered() {
    val restaurants =
DummyContent.getDomainRestaurants()
    val targetRestaurant = restaurants[0]
    testRule.setContent {
        RestaurantsAppTheme {
            RestaurantsScreen(
                state = RestaurantsScreenState(
                    restaurants = restaurants,
                    isLoading = false),
                onFavoriteClick = { _, _ -> },
                onItemClick = { id -> })
        }
    }
    testRule.onNodeWithText(targetRestaurant.title)
        .performClick()
}
```

23. Now that we have simulated a user-click interaction, let's assert that the **`id`** value from the **`onItemClick`** callback exposed by the **`RestaurantsScreen()`** composable matches with the **`id`** value of the restaurant we have clicked on:

```
@Test
fun stateWithContent_ClickOnItem_isRegistered() {
    val restaurants =
DummyContent.getDomainRestaurants()
    val targetRestaurant = restaurants[0]
    testRule.setContent {
        RestaurantsAppTheme {
            RestaurantsScreen(
                state = RestaurantsScreenState(
                    restaurants = restaurants,
                    isLoading = false),
                onFavoriteClick = { _, _ -> },
                onItemClick = { id ->
                    assert(id ==
targetRestaurant.id)
                }
            )
        }
    }
    testRule.onNodeWithText(targetRestaurant.title)
        .performClick()
}
```

24. Inside the **Project** tab on the left, right-click on the **`RestaurantsScreenTest`** class and select **Run RestaurantsScreenTest**.

The three tests should run and pass.

NOTE

You might have noticed that we haven't given any attention to testing how the UI updates when a user toggles a restaurant as a favorite or not a fa-

vorite. The only way we could have done that is by adding a dedicated semantic property to the heart icon of each restaurant from the list and then testing the value of that property. However, we would have tested a semantic property value and not the UI – for such cases, it's better to look into **screenshot testing** *strategies. Screenshot testing is a UI testing practice that generates screenshots of your app, which are then compared to the initially defined correct versions.*

Now that we briefly covered UI testing with Compose, it's time to unit-test our app's behind-the-scenes functionality!

Covering the basics of unit-testing your core logic

Apart from testing our UI layer, we must also test the core logic of our application. This means that we should try to verify as much behavior as possible in terms of presentation logic (testing `ViewModel` classes), business logic (testing `UseCase` classes), or even data logic (testing `Repository` classes).

The easiest way of validating such logic is by writing unit tests for each class or group of classes whose behavior we're trying to verify.

In this section, we will be writing unit tests for the `RestaurantsViewModel` class and the `ToggleRestaurantUseCase` class. Since these components don't interact directly with the UI, their unit tests will run directly on your local workstation's **Java Virtual Machine (JVM)**, rather than running on an Android device, as our UI tests did.

To summarize, in this section, we will be doing the following:

- Testing the functionality of a `ViewModel` class
- Testing the functionality of a `UseCase` class

Let's begin by testing the **`RestaurantsViewModel`** class!

Testing the functionality of a ViewModel class

We want to test the functionality of our **`RestaurantsViewModel`** so that we can make sure that it's correctly performing the role of state producer for the **`RestaurantsScreen()`** composable.

To achieve that, we will write unit tests for this **`ViewModel`** class in isolation. Let's begin:

1. First, locate the **`test`** package that is suited for regular unit tests:

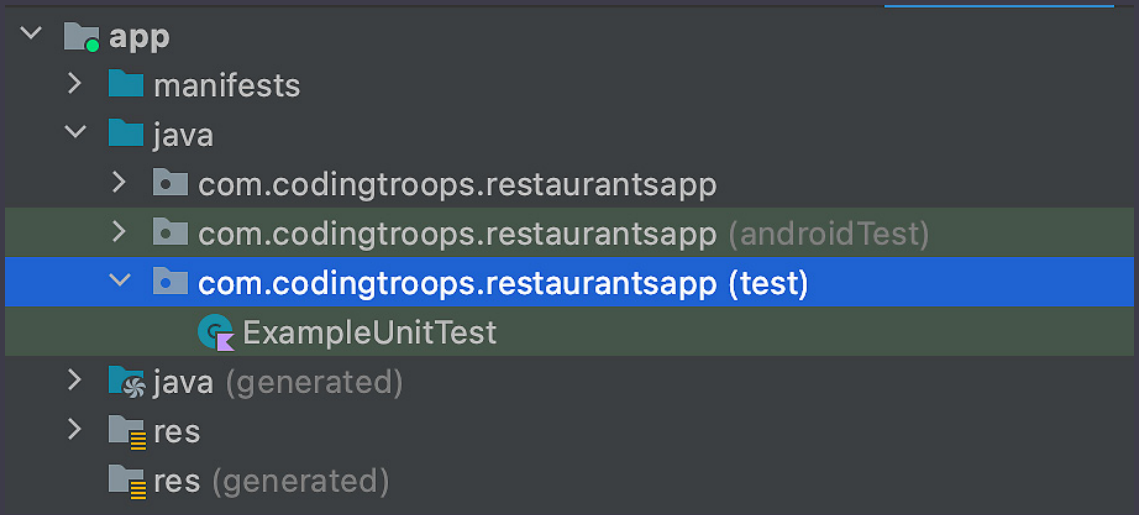

Figure 10.3 – Observing the test package used for regular unit tests

Also, note that the pre-built **`ExampleUnitTest`** class resides inside this package.

2. Create an empty Kotlin class named **`RestaurantsViewModelTest`** inside the **`test`** package. Inside this class, we will define a method for each independent test. Behind the scenes, every method will become a standalone unit test that can pass or fail.

Before starting to write our first test method, let's have another look at our **`RestaurantsViewModel`** class so that we can remind ourselves which

cases we're looking to test:

```
class RestaurantsViewModel @Inject constructor(…) :
ViewModel() {
    private val _state = mutableStateOf(
        RestaurantsScreenState(
            restaurants = listOf(),
            isLoading = true
        )
    )
    […]
    private val errorHandler =
CoroutineExceptionHandler
    { … ->
        exception.printStackTrace()
        _state.value = _state.value.copy(
            error = exception.message,
            isLoading = false)
    }
    init { getRestaurants() }
    fun toggleFavorite(itemId: Int, oldValue:
Boolean) {
        […]
    }
    private fun getRestaurants() {
        viewModelScope.launch(errorHandler) {
            val restaurants = getRestaurantsUseCase()
            _state.value = _state.value.copy(
                restaurants = restaurants,
                isLoading = false)
        }
    }
}
```

We can say that our **`RestaurantsViewModel`** should produce the exact three states that we fed to the **`RestaurantsScreen()`** composable in its own UI

tests:

- **Initial loading state**: At this point, we're waiting for restaurants, thus rendering a loading status. You can see this initial state declared at the top of the **`ViewModel`** class in the initialization of the **`_state`** variable, where the **`restaurants`** parameter of **`RestaurantsScreenState`** is set to **`emptyList()`** and the **`isLoading`** parameter is set to **`true`**.
- **State with content**: The restaurants have arrived. This state is produced inside the coroutine launched in the **`getRestaurants()`** method, where we mutated the initial state and set the **`isLoading`** parameter to **`false`** while also passing the list of restaurants to the **`restaurants`** parameter.
- **Error state**: Something went wrong when the app tried to fetch its content. You can see how this state is created inside **`CoroutineExceptionHandler`**, where the **`isLoading`** parameter is set to **`false`** to reset the loading status while also passing the message of **`Exception`** to the **`error`** parameter.

In the end, what we basically have to do is assert that the value of the **`state`** variable (of type **`RestaurantsScreenState`**), which is exposed to the UI, evolves correctly over time, from the initial state to all possible states.

Let's begin with a test method that asserts whether the initial state is produced as expected:

1. Inside the **`RestaurantsViewModelTest`** class, add an empty test function named **`initialState_isProduced()`** that will later test whether our **`RestaurantsViewModel`** class properly produces the initial state:

```
@Test
fun initialState_isProduced() {  }
```

As with the UI tests, we will make use of the JUnit testing library to define and run individual unit tests for each method annotated with the **`@Test`** annotation.

Again similar to the UI tests, we named this method around the specific behavior it's trying to test, going from what we're testing (the initial state) to what should happen (the state being correctly produced).

2. Inside the **`initialState_isProduced()`** method, we must create an instance of the subject under test – that is, **`RestaurantsViewModel`**. Define a **`viewModel`** variable and instantiate it with the value returned by the **`getViewModel()`** method, which we will define in a second:

```
@Test
fun initialState_isProduced() {
    val viewModel = getViewModel()
}
```

3. Still inside the **`RestaurantsViewModelTest`** class, define the **`getViewModel()`** method, which will return an instance of **`RestaurantsViewModel`**:

```
private fun getViewModel(): RestaurantsViewModel {
    return RestaurantsViewModel()
}
```

The problem now is that the **`RestaurantsViewModel`** constructor needs an instance of **`GetInitialRestaurantsUseCase`** and **`ToggleRestaurantsUseCase`**. In turn, these two classes also have other dependencies that we must instantiate. Let's have a clearer look at what classes we need to instantiate:

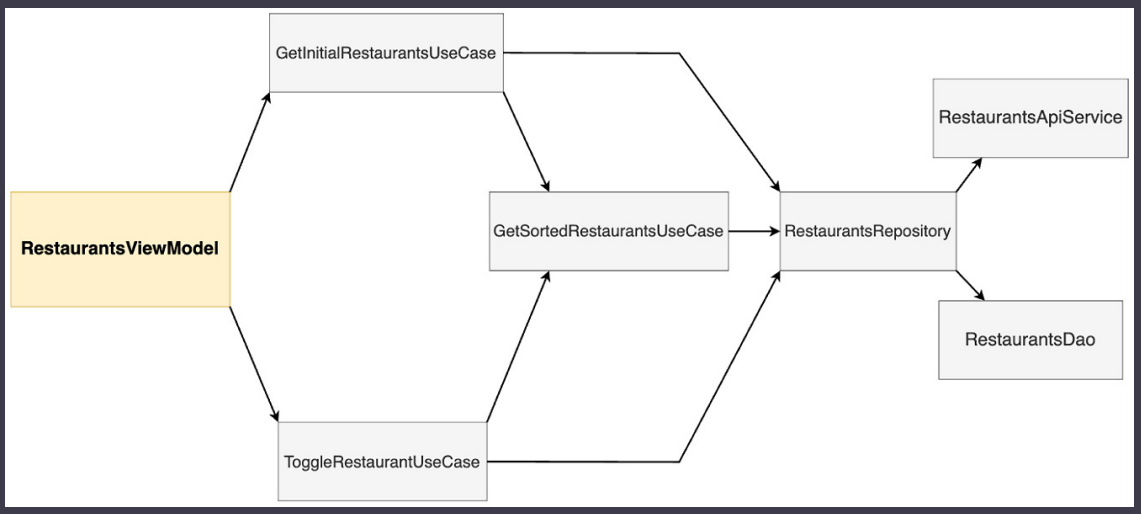

Figure 10.4 – Observing the direct and transitive dependencies of RestaurantsViewModel

We can see that both **`GetInitialRestaurantsUseCase`** and **`ToggleRestaurantsUseCase`** depend on **`GetSortedRestaurantsUseCase`** and **`RestaurantsRepository`**. The latter then depends on two library interfaces – **`RestaurantsApiService`** and **`RestaurantsDao`**.

Essentially, we must instantiate all these classes to test our **`RestaurantsViewModel`**.

4. Inside the **`RestaurantsViewModelTest`** class, refactor the **`getViewModel()`** method to construct all the necessary dependencies of **`RestaurantsViewModel`**:

```
private fun getViewModel(): RestaurantsViewModel {
    val restaurantsRepository =
        RestaurantsRepository(?, ?)
    val getSortedRestaurantsUseCase =

GetSortedRestaurantsUseCase(restaurantsRepository)
    val getInitialRestaurantsUseCase =
        GetInitialRestaurantsUseCase(
            restaurantsRepository,
            getSortedRestaurantsUseCase)
    val toggleRestaurantUseCase =
        ToggleRestaurantUseCase(
            restaurantsRepository,
            getSortedRestaurantsUseCase
        )
    return RestaurantsViewModel(
        getInitialRestaurantsUseCase,
        toggleRestaurantUseCase
    )
}
```

If you read the previous snippet from bottom to top, you will notice that we were able to construct all the dependencies of the **`RestaurantsViewModel`**, and their dependencies, and so on until we hit

RestaurantsRepository. This depends on two library interfaces, **RestaurantsApiService** and **RestaurantsDao**, whose implementations are provided by the Retrofit and Room libraries.

In our production code, these two interfaces cross the boundary to the *real world* because their implementations, provided by the Retrofit and Room libraries, communicate with a real Firebase REST API and a real Room local database:

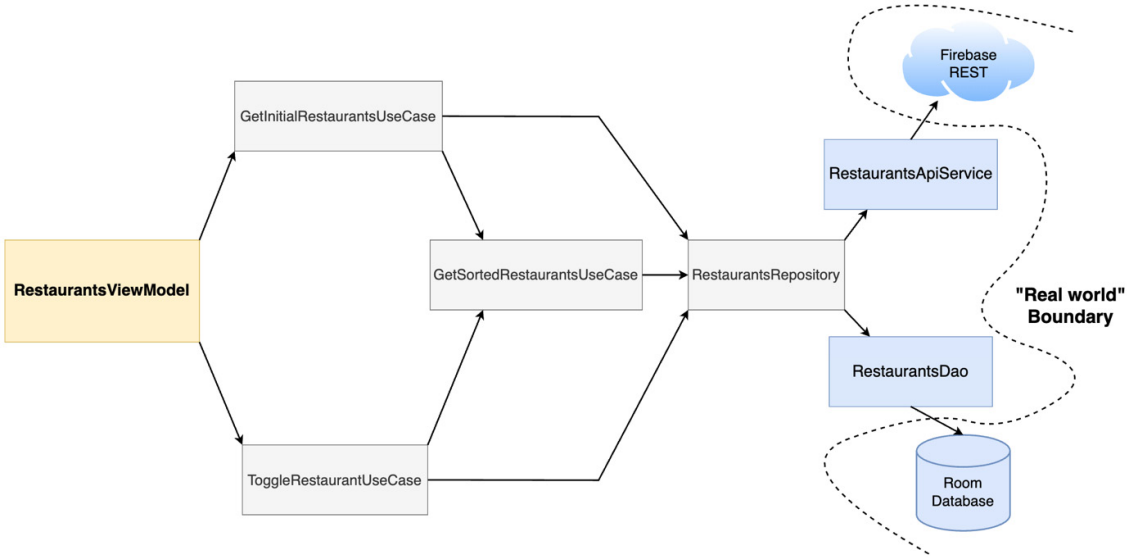

Figure 10.5 – Observing the real-world boundary crossed by the transitive dependencies of RestaurantsViewModel

If we were to use the existing implementations of these two interfaces provided by Retrofit and Room in our test code, the **RestaurantsViewModel** instance will communicate with the external world and our tests won't be isolated. Instead, our test code will be slow and not reliable because it will be dependent on our web REST API and a real local database.

Yet how can we make our **RestaurantsViewModel** tests isolated, fast, and reliable? We can simply make sure that instead of having Retrofit and Room provide the implementations for **RestaurantsApiService** and **RestaurantsDao**, we define dummy implementations for these interfaces that won't communicate with the real world.

These dummy implementations are often called fakes. **Fakes** are simplified implementations of the interfaces that we're looking to interact with in our tests. Such implementations mimic the behavior of the production implementations in a very simplified manner, often by returning dummy data. Fakes will only be used in our tests, so we can ensure that our testing environment is isolated.

Apart from fakes, to mimic the functionality of components that cross the boundary to the real world, you can also use mocks. **Mocks** are objects that also simulate the behavior of a real object; however, you can configure their output on the fly without any additional classes.

In this chapter, we will only focus on fakes, since most of the time, to create mocks, you need to use special mocking frameworks. Also, fakes tend to be more practical and can be reused across tests, whereas mocks tend to clutter your tests, as they bring a lot of boilerplate code.

NOTE

Whenever you have a component that interacts with the real world, be it a web API, local database, or other production systems, you should define an interface for it. This way, in your production code, your other components interact with a real implementation of that interface, while your tests interact with a fake implementation of it.

Let's see how we can implement fakes. In our case, **`RestaurantsRepository`** needs fake implementations of the **`RestaurantsApiService`** and **`RestaurantsDao`** interfaces. Let's begin with a fake implementation of the **`RestaurantsApiService`** interface:

1. To create a fake for the **`RestaurantsApiService`** interface, we must define a class that will implement the interface and simulate the functionality of a REST API. Inside the **`test`** package, create a Kotlin class named **`FakeApiService`** that implements the **`RestaurantsApiService`** interface and add the following code inside:

```
class FakeApiService : RestaurantsApiService {
    override suspend fun getRestaurants()
            : List<RemoteRestaurant> {
        delay(1000)
        return DummyContent.getRemoteRestaurants()
    }
    override suspend fun getRestaurant(id: Int)
            : Map<String, RemoteRestaurant> {
        TODO("Not yet implemented")
    }
}
```

Our **`FakeApiService`** overrides the required methods and returns some dummy restaurants from the **`DummyContent`** class. In the **`getRestaurants()`** method, we also call a coroutine-based **`delay()`** function of 1,000 milliseconds to better simulate an asynchronous response. Since we will not be using the **`getRestaurant()`** method in our tests right now, we haven't added any implementation inside it.

Going back to the dummy content that is returned, note that the **`getRestaurants()`** method must return a list of **`RemoteRestaurant`** objects, so we called a non-existent **`getRemoteRestaurants()`** method on the **`DummyContent`** class. Let's define this method up next.

2. Head back inside the main source set where our production code resides. Inside the **`DummyContent`** class, add a new method called **`getRemoteRestaurants()`** that maps the list of **`Restaurant`** objects returned by the **`getDomainRestaurants()`** method to **`RemoteRestaurant`** objects:

```
object DummyContent {
    fun getDomainRestaurants() = arrayListOf(…)
    fun getRemoteRestaurants() =
getDomainRestaurants()
        .map {
            RemoteRestaurant(
                it.id,
```

```
                    it.title,
                    it.description
                )
            }
    }
```

3. Now, head back inside the **`test`** package. We've created a fake for the **`RestaurantsApiService`** interface, but we must also create one fake for the **`RestaurantsDao`** interface that will implement the interface and simulate the functionality of a local database. Inside the **`test`** package, create a Kotlin class named **`FakeRoomDao`** that implements the **`RestaurantsDao`** interface and add the following code inside:

```
class FakeRoomDao : RestaurantsDao {
    private var restaurants =
              HashMap<Int, LocalRestaurant>()
    override suspend fun getAll()
            : List<LocalRestaurant> {
        delay(1000)
        return restaurants.values.toList()
    }
    override suspend fun addAll(
        restaurants: List<LocalRestaurant>
    ) {
        restaurants.forEach {
            this.restaurants[it.id] = it
        }
    }
    override suspend fun update(
        partialRestaurant: PartialLocalRestaurant
    ) {
        delay(1000)
        updateRestaurant(partialRestaurant)
    }
    override suspend fun updateAll(
        partialRestaurants:
List<PartialLocalRestaurant>
```

```
        ) {
            delay(1000)
            partialRestaurants.forEach {
                updateRestaurant(it)
            }
        }
        override suspend fun getAllFavorited()
                : List<LocalRestaurant> {
            return restaurants.values.toList()
                .filter { it.isFavorite }
        }
    }
```

Our **`FakeRoomDao`** class mimics the functionality of a real Room database, yet instead of storing restaurants in the local SQL database, it stores them in memory in the **`restaurants`** variable. We will not cover each method implementation of **`FakeRoomDao`**.

However, we will conclude that each method simulates the interaction with a persistent storage service. Additionally, as our **`FakeRoomDao`** simulates interaction with a real local database, each of its actions will cause a delay triggered by the pre-built suspending **`delay()`** function.

However, our **`FakeRoom`** class makes use of an **`updateRestaurant()`** method that we haven't defined so far. Let's do that now.

4. At the end of the body of the **`FakeRoom`** class, add the missing **`updateRestaurant()`** method that toggles the value of the **`isFavorite`** field:

```
    class FakeRoomDao : RestaurantsDao {
        [...]
        override suspend fun getAllFavorited()
                : List<LocalRestaurant> { ... }
        private fun updateRestaurant(
            partialRestaurant: PartialLocalRestaurant
        ) {
```

```
            val restaurant =
                this.restaurants[partialRestaurant.id]
            if (restaurant != null)
                this.restaurants[partialRestaurant.id] =
                    restaurant.copy(
                        isFavorite =
                            partialRestaurant.isFavorite
                    )
        }
    }
```

5. Now that we have finished implementing the fakes for our **`RestaurantsApiService`** and **`RestaurantsDao`** interfaces, it's time to pass them where we need them in our tests. Remember that the last missing piece was to provide fake implementations of the **`RestaurantsRepository`** dependencies so that our test is isolated.

Head back inside the **`RestaurantsViewModelTest`** class and update the **`getViewModel()`** function to pass instances of the **`FakeApiService`** and **`FakeRoomDao`** classes to **`RestaurantsRepository`**:

```
private fun getViewModel(): RestaurantsViewModel {
    val restaurantsRepository =
RestaurantsRepository(
        FakeApiService(), FakeRoomDao())
    […]
    return RestaurantsViewModel(…)
}
```

Now that the **`getViewModel()`** method is able to return an instance of **`RestaurantsViewModel`** that we can easily test, let's get back to our **`initialState_isProduced()`** test method, which currently looks like this:

```
@Test
fun initialState_isProduced() {
    val viewModel = getViewModel()
}
```

Remember that the scope of this test method is to verify that when our **`RestaurantsViewModel`** is initialized, it produces a correct initial state. Let's do that now.

6. First, inside the **`initialState_isProduced()`** test method, store the initial state inside an **`initialState`** variable:

```
@Test
fun initialState_isProduced() {
    val viewModel = getViewModel()
    val initialState = viewModel.state.value
}
```

7. Next, by using the built-in **`assert()`** function, verify whether the content of **`initialState`** is as expected:

```
@Test
fun initialState_isProduced() {
    val viewModel = getViewModel()
    val initialState = viewModel.state.value
    assert(
        initialState == RestaurantsScreenState(
            restaurants = emptyList(),
            isLoading = true,
            error = null)
    )
}
```

In this test method, we're asserting whether the value of the **`initialState`** variable is a **`RestaurantsScreenState`** object with a **`false`** **`isLoading`** field and an **`emptyList()`** value inside the **`restaurants`** field. Additionally, we're testing that there is no value stored inside the **`error`** field.

8. Now that we have defined our first test method, it's time to run the test!

Inside the **Project** tab on the left, right-click on the **`RestaurantsViewModelTest`** class and select **Run**

RestaurantsViewModelTest. This command will run all the tests inside this class (only one in our case) directly on your local JVM, rather than running on an Android device, as our UI tests did.

If you switch to the **Run** tab, you will see that our test has failed:

```
java.lang.IllegalStateException Create breakpoint : Module with the Main dispatcher had failed to
 initialize. For tests Dispatchers.setMain from kotlinx-coroutines-test module can be used
    at kotlinx.coroutines.internal.MissingMainCoroutineDispatcher.missing(MainDispatchers.kt:113)
    at kotlinx.coroutines.internal.MissingMainCoroutineDispatcher.isDispatchNeeded(MainDispatchers
.kt:94)
```

Figure 10.6 – Observing how the test inside the RestaurantsViewModelTest class has failed

This exception is thrown because our Restaurants app handles asynchronous work with the help of coroutines, and our test code doesn't know how to interact with them.

For instance, our **`RestaurantsViewModel`** launches coroutines that call several suspend functions, and all of these happen on **`viewModelScope`**, which has the **`Dispatchers.Main`** dispatcher set by default:

```
@HiltViewModel
class RestaurantsViewModel @Inject constructor(...) :
[…] {
    [...]
    fun toggleFavorite(itemId: Int, oldValue:
Boolean) {
        viewModelScope.launch(errorHandler) { ... }
    }
    private fun getRestaurants() {
        viewModelScope.launch(errorHandler) { ... }
    }
}
```

The main issue here is that our coroutines are launched on the Main thread on our local JVM, which can't work with the UI thread.

NOTE

The **`Dispatchers.Main`** *dispatcher uses the Android* **`Looper.getMainLooper()`** *function to run code in the UI thread. That method is available in UI tests but not in the regular unit tests that run on our JVM.*

To make our testing code compliant with the usage of coroutines, we need to use the Kotlin coroutines testing library, which will provide us with scopes and dispatchers that are dedicated to testing coroutines. If our test code is run from coroutines that are built with these dedicated scopes and dispatchers, our test will no longer fail.

Let's add the Kotlin coroutines testing library!

1. In the app-level **`build.gradle`** file, add a **`testImplementation`** dependency to the Kotlin coroutines testing package:

```
dependencies {
    […]
    testImplementation
"com.google.truth:truth:1.1.2"
    testImplementation
'org.jetbrains.kotlinx:kotlinx-
        coroutines-test:1.6.1'
}
```

2. Synchronize your project with its Gradle files by clicking on the **Sync your project with Gradle files** button in Android Studio or by pressing on the **File** menu option and then by selecting **Sync Project with Gradle Files**.
3. Head back inside the **`RestaurantsViewModelTest`** class and define a variable for a **`StandardTestDispatcher`** object and a variable for a **`TestScope`** object based on the previously defined dispatcher:

```
@ExperimentalCoroutinesApi
class RestaurantsViewModelTest {
    private val dispatcher =
StandardTestDispatcher()
```

```
    private val scope = TestScope(dispatcher)
    @Test
    fun initialState_isProduced() {…}
    private fun getViewModel():
RestaurantsViewModel {…}
}
```

Additionally, we've added the **`@ExperimentalCoroutinesApi`** annotation to the **`RestaurantsViewModelTest`** class, since these testing APIs are still experimental.

4. Next up, make sure that all the code from within the body of the **`initialState_isProduced()`** test method is run inside a test-specific coroutine. To do that, launch a coroutine that wraps this method's body by calling the **`runTest()`** coroutine builder on our **`scope`** variable of type **`TestScope`**:

```
@Test
fun initialState_isProduced() = scope.runTest {
    val viewModel = getViewModel()
    val initialState = viewModel.state.value
    assert(
        initialState == RestaurantsScreenState(
            restaurants = emptyList(),
            isLoading = true,
            error = null))
}
```

5. Run the **`RestaurantsViewModelTest`** class. If you switch to the **Run** tab, you will see that our test has failed again, with the same exception as before. Since we wrapped the body of our test method inside a test coroutine, our test code still throws an exception, telling us that we still need to change the dispatcher of the coroutine.

If we have another look at **`RestaurantsViewModel`**, we can note that both the coroutines launched with **`viewModelScope`** have no dispatcher set, so they're using **`Dispatchers.Main`** behind the scenes:

```
@HiltViewModel
class RestaurantsViewModel @Inject constructor(...) :
[…] {
    [...]
    fun toggleFavorite(itemId: Int, oldValue:
Boolean) {
        viewModelScope.launch(errorHandler) { ... }
    }
    private fun getRestaurants() {
        viewModelScope.launch(errorHandler) { ... }
    }
}
```

However, in our test, all coroutines that are launched should use **`StandardTestDispatcher`** defined in our test class. So, how can we pass a test dispatcher to the coroutines launched in our **`RestaurantsViewModel`**?

We can inject the dispatcher inside the **`RestaurantsViewModel`** class by using its constructor and making it accept a **`CoroutineDispatcher`** object, which will be passed to all the coroutines that are launched.

This way, in our production code, **`RestaurantsViewModel`** will receive and use the **`Dispatchers.Main`** dispatcher, and inside our test code, it will receive and use the **`StandardTestDispatcher`** dispatcher.

NOTE

The practice of injecting dispatchers into our coroutine-based classes is encouraged, as it allows us to have better isolation and control over the testing environment of our unit tests.

6. Head back inside the main source set where our production code resides. Inside **`RestaurantsViewModel`**, add a **`dispatcher`** constructor parameter of type **`CoroutineDispatcher`** and pass it to the **`viewModelScope()`** calls:

```
@HiltViewModel
```

```
class RestaurantsViewModel @Inject constructor(
   private val getRestaurantsUseCase: […],
   private val toggleRestaurantsUseCase: […],
   private val dispatcher: CoroutineDispatcher) :
       ViewModel(){
    [...]
    fun toggleFavorite(itemId: Int, oldValue:
Boolean) {
        viewModelScope.launch(errorHandler
            + dispatcher) { … }
    }
    private fun getRestaurants() {
        viewModelScope.launch(errorHandler
            + dispatcher) { … }
    }
}
```

However, if we build the project now, we will get an error because Hilt doesn't know how to provide an instance of **`CoroutineDispatcher`** to **`RestaurantsViewModel`**.

To instruct Hilt on how to provide our **`ViewModel`** with the dispatcher it needs (that is, **`Dispatchers.Main`**), we must create a Hilt module.

7. Inside the **`di`** package, create a new class called **`DispatcherModule`** and add the following code that tells Hilt how to provide any **`CoroutineDispatcher`** dependencies with **`Dispatchers.Main`**:

```
@Module
@InstallIn(SingletonComponent::class)
object DispatcherModule {
    @Provides
    fun providesMainDispatcher():
CoroutineDispatcher
        = Dispatchers.Main
}
```

However, right now, Hilt will always provide **`Dispatchers.Main`** to any **`CoroutineDispatcher`** dependencies. What if we need later to obtain a dispatcher different than **`Dispatchers.Main`**? Let's see how we can prepare for that.

8. At the top of the body of **`DispatcherModule`**, define an annotation class called **`MainDispatcher`** annotated with the **`@Qualifier`** annotation:

```
@Qualifier
@Retention(AnnotationRetention.BINARY)
annotation class MainDispatcher
@Module
@InstallIn(SingletonComponent::class)
object DispatcherModule {…}
```

The **`@Qualifier`** annotation allows us to provide different dispatchers to the **`CoroutineDispatcher`** dependencies. In our case, we defined that a **`@MainDispatcher`** annotation will provide the **`Dispatchers.Main`** dispatcher.

9. Add the **`@MainDispatcher`** annotation to the **`providesMainDispatcher()`** method so that Hilt will know what dispatcher to provide when such an annotation is used on a dependency:

```
@Qualifier
@Retention(AnnotationRetention.BINARY)
annotation class MainDispatcher
@Module
@InstallIn(SingletonComponent::class)
object DispatcherModule {
    @MainDispatcher
    @Provides
    fun providesMainDispatcher():
CoroutineDispatcher
        = Dispatchers.Main
}
```

10. Then, inside **`RestaurantsViewModel`**, annotate the **`dispatcher`** parameter with the newly created **`@MainDispatcher`** annotation so that Hilt will

provide us with the **`Dispatchers.Main`** dispatcher:

```
@HiltViewModel
class RestaurantsViewModel @Inject constructor(
   private val getRestaurantsUseCase: […],
   private val toggleRestaurantsUseCase: […],
   @MainDispatcher private val dispatcher:
       CoroutineDispatcher
) : ViewModel() { ... }
```

11. Now that the **`RestaurantsViewModel`** uses the **`Dispatcher.Main`** dispatcher in our production code, head back inside the **`test`** source set and inside the **`RestaurantsViewModelTest`** class, update its **`getViewModel()`** method by passing the **`dispatcher`** member field to the **`RestaurantsViewModel`** constructor call:

```
@ExperimentalCoroutinesApi
class RestaurantsViewModelTest {
    private val dispatcher =
StandardTestDispatcher()
    private val scope = TestScope(dispatcher)
    @Test
    fun initialState_isProduced() = scope.runTest
{…}
    private fun getViewModel():
RestaurantsViewModel {
        […]
        return RestaurantsViewModel(
            getInitialRestaurantsUseCase,
            toggleRestaurantUseCase,
            dispatcher)
    }
}
```

Now, in our test code, **`RestaurantsViewModel`** will use the **`StandardTestDispatcher`** dispatcher for all the launched coroutines.

12. Now, run the **`RestaurantsViewModelTest`** class again. If you switch to the **Run** tab, you will see that our test has now passed:

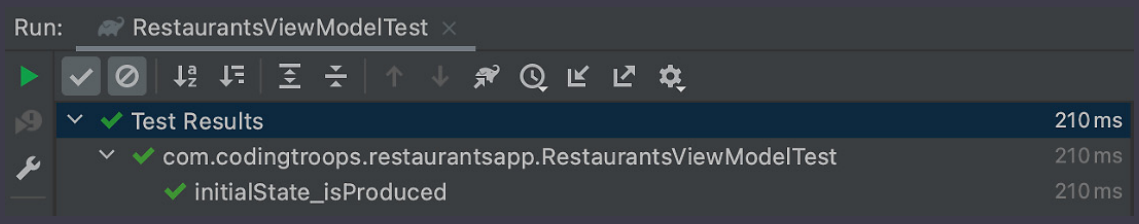

Figure 10.7 – Observing how the test inside the RestaurantsViewModelTest class has succeeded

Since our test uses the **`runTest()`** coroutine builder, any **`delay()`** calls in our fake implementations are skipped, making our test run fast, in just a few hundred milliseconds.

Now that we have tested whether our **`ViewModel`** produces a correct initial state, it's time to test the state that comes after this initial state – the state with content. This state is produced when the restaurants have arrived (from our data layer), so the **`isLoading`** field should be reset to **`false`**, while the **`restaurants`** field should contain a list of restaurants.

13. Add a new testing method inside **`RestaurantsViewModelTest`** called **`stateWithContent_isProduced()`** that asserts whether the state with restaurants is produced as expected:

```
@Test
fun stateWithContent_isProduced() = scope.runTest {
    val testVM = getViewModel()
    val currentState = testVM.state.value
    assert(
        currentState == RestaurantsScreenState(
            restaurants =

DummyContent.getDomainRestaurants(),
            isLoading = false,
            error = null)
    )
```

```
}
```

Since the **FakeApiService** returns the dummy list of **RemoteRestaurant** from the **DummyContent** class, it's only natural that we're expecting to get the same content in **ViewModel** but in the shape of the **Restaurant** objects – so we're asserting that the **restaurants** field of **currentState** contains the restaurants from **DummyContent**.

Unfortunately, if we run the **RestaurantsViewModelTest** class, the **stateWithContent_isProduced()** test will fail, telling us that **currentState** has the **isLoading** field's value of **true** and there are no restaurants inside the **restaurants** field.

This issue makes sense because we're basically obtaining the initial state and expecting it to be the state with content, which, in fact, comes later on. Because there are several **delay()** calls in our **FakeApiService** and **FakeRoomDao** implementations, we must allow time to pass so that **ViewModel** produces the second state – the one with restaurants. But how can we do that?

Inside a test, to immediately execute all pending tasks (such as the launched coroutine to get restaurants in our **ViewModel**) and to advance the virtual clock until after the last delay, we can call the **advanceUntilIdle()** function exposed by the coroutines test library.

14. Inside the **stateWithContent_isProduced()** test method, after **RestaurantsViewModel** is instantiated but before our assertion, add the **advanceUntilIdle()** method call:

```
@Test
fun stateWithContent_isProduced() = scope.runTest {
    val testVM = getViewModel()
    advanceUntilIdle()
    val currentState = testVM.state.value
    assert(
        currentState == RestaurantsScreenState(
```

```
            restaurants =
                DummyContent.getDomainRestaurants()
,
            isLoading = false,
            error = null)
    )
}
```

When we call **`advanceUntilIdle()`** inside our **`scope`** variable of type **`TestScope`**, we're advancing the virtual clock of **`TestCoroutineScheduler`** featured in the **`StandardTestDispatcher`** that we've initially passed to our **`scope`**.

15. Now, run the **`RestaurantsViewModelTest`** class again. If you switch to the **Run** tab, you will see that the **`stateWithContent_isProduced()`** test still failed.

The main issue here is that while we're trying to advance the virtual clock of our test by leveraging the fact that our test instance of **`RestaurantsViewModel`** now launches its coroutines on the **`StandardTestDispatcher`** instance that we've passed to it, we have another class that is passing its own **`CoroutineDispatcher`**.

If we have a closer look inside our **`RestaurantsRepository`**, we can see that it's passing a production-use **`Dispatchers.IO`** dispatcher to all its **`withContext()`** calls:

```
@Singleton
class RestaurantsRepository @Inject constructor(…) {
    suspend fun toggleFavoriteRestaurant(…) =
        withContext(Dispatchers.IO) {…}
    suspend fun getRestaurants() : List<Restaurant> {
        return withContext(Dispatchers.IO) {…}
    }
    suspend fun loadRestaurants() {
```

```
            return withContext(Dispatchers.IO) {…}
        }
        private suspend fun refreshCache() {…}
    }
```

Because the **`RestaurantsRepository`** instance that our **`RestaurantsViewModel`** indirectly depends on uses the **`Dispatchers.IO`** dispatcher and not the **`StandardTestDispatcher`** one, the virtual clock of our test is not advanced as expected. Let's fix this issue by injecting the dispatcher in the **`RestaurantsRepository`**, just as we did for **`RestaurantsViewModel`**.

16. However, before performing the injection, we must first define a new type of **`CoroutineDispatcher`** that Hilt should know how to inject – the **`Dispatchers.IO`** dispatcher.

Head inside the **`DispatchersModule`** class and, just as we did for the **`Dispatchers.Main`** dispatcher, instruct Hilt on how to provide us with the **`Dispatchers.IO`** dispatcher:

```
    @Qualifier
    @Retention(AnnotationRetention.BINARY)
    annotation class MainDispatcher
    @Qualifier
    @Retention(AnnotationRetention.BINARY)
    annotation class IoDispatcher
    @Module
    @InstallIn(SingletonComponent::class)
    object DispatcherModule {
        @MainDispatcher
        @Provides
        fun providesMainDispatcher(): CoroutineDispatcher
    =
            Dispatchers.Main
        @IoDispatcher
        @Provides
```

```
    fun providesIoDispatcher(): CoroutineDispatcher =
        Dispatchers.IO
}
```

17. Head back inside the main source set where our production code resides. Inside the **`RestaurantsRepository`** class, inject **`CoroutineDispatcher`**, annotate it with the **`@IoDispatcher`** qualifier, and then pass the injected **`dispatcher`** to all the **`withContext()`** calls:

```
@Singleton
class RestaurantsRepository @Inject constructor(
    private val restInterface:
RestaurantsApiService,
    private val restaurantsDao: RestaurantsDao,
    @IoDispatcher private val dispatcher:
        CoroutineDispatcher
) {
    suspend fun toggleFavoriteRestaurant(…) =
        withContext(dispatcher) {…}
    suspend fun getRestaurants() : List<Restaurant>
{
        return withContext(dispatcher) {…}
    }
    suspend fun loadRestaurants() {
        return withContext(dispatcher) {…}
    }
    private suspend fun refreshCache()
{…}           }
```

18. Then, heading back inside our **`test`** package, inside the **`RestaurantsViewModelTest`** class, update the **`getViewModel()`** method to pass our **`dispatcher`** field of type **`StandardTestDispatcher`** to the **`RestaurantsRepository`** constructor:

```
private fun getViewModel(): RestaurantsViewModel {
    val restaurantsRepository =
RestaurantsRepository(
        FakeApiService(),
```

```
            FakeRoomDao(),
            dispatcher)
        […]
        return RestaurantsViewModel(
            getInitialRestaurantsUseCase,
            toggleRestaurantUseCase,
            dispatcher)
    }
```

19. Now, run the **RestaurantsViewModelTest** class again. If you switch to the **Run** tab, you will see that both our tests have now passed.
    ASSIGNMENT
    Try testing on your own that **RestaurantsViewModel** *is correctly producing an error state. As a tip, make sure to throw an instance of the* **Exception** *class inside* **FakeApiService** *but just for this specific test method where you're verifying the error state. To achieve that, you can configure a constructor parameter in* **FakeApiService** *so that it can throw an exception if needed.*

Now that we tested how **RestaurantsViewModel** is producing the UI state, let's briefly have a look at how we could test a business component.

Testing the functionality of a UseCase class

Aside from unit-testing the presentation layer of our application, it's very important to also test the business rules present in the app. In our Restaurants app, the business logic is encapsulated in **UseCase** classes.

Let's see say that we want to test **ToggleRestaurantUseCase**. Essentially, we want to make sure that when we execute this **UseCase** class for a specific restaurant, the business logic of negating the **isFavorite** field of the **Restaurant** is working.

In other words, if one restaurant was not marked as favorite, after executing **ToggleRestaurantUseCase** for that specific restaurant, its **isFavorite** field should become **true**. While this business logic is indeed slim, in

medium to large-sized applications, such business logic can become much more complex.

Let's see how a unit test for **`ToggleRestaurantUseCase`** would look:

```
@ExperimentalCoroutinesApi
class ToggleRestaurantUseCaseTest {
    private val dispatcher = StandardTestDispatcher()
    private val scope = TestScope(dispatcher)
    @Test
    fun toggleRestaurant_IsUpdatingFavoriteField() =
            scope.runTest {
        // Setup useCase
        val restaurantsRepository =
RestaurantsRepository(
            FakeApiService(),
            FakeRoomDao(),
            dispatcher)
        val getSortedRestaurantsUseCase =
            GetSortedRestaurantsUseCase(restaurantsRe
pository)
        val useCase = ToggleRestaurantUseCase(
            restaurantsRepository,
            getSortedRestaurantsUseCase)
        // Preload data
        restaurantsRepository.loadRestaurants()
        advanceUntilIdle()
        // Execute useCase
        val restaurants =
DummyContent.getDomainRestaurants()
        val targetItem = restaurants[0]
        val isFavorite = targetItem.isFavorite
        val updatedRestaurants = useCase(
            targetItem.id,
            isFavorite
```

```
            )
            advanceUntilIdle()
            // Assertion
            restaurants[0] = targetItem.copy(isFavorite =
                !isFavorite)
            assert(updatedRestaurants == restaurants)
        }
    }
```

This unit test is similar to the ones we wrote for **`RestaurantsViewModel`** in the sense that it's also using **`StandardTestDispatcher`** and **`TestScope`**, simply because the **`invoke()`** operator of **`ToggleRestaurantUseCase`** is a **`suspending`** function.

The structure of the test is split into three parts, delimited by the suggestive comments:

- **Setup**: In this first phase, we have constructed a **`ToggleRestaurantUseCase`** instance and its direct and transitive dependencies, while passing our test dispatcher to the dependencies that need it.
- **Preload data**: For **`ToggleRestaurantUseCase`** to be able to execute its business logic on a specific restaurant, we first had to make sure that our **`RestaurantsRepository`** instance had loaded the dummy restaurants. We then called **`advancedUntilIdle()`**, allowing any suspending (blocking) work related to obtaining and caching dummy restaurants to finish.
- **Execute Use Case**: We defined the restaurant whose **`isFavorite`** field we want to toggle as **`targetItem`**, obtained its current **`isFavorite`** field value, and executed **`ToggleRestaurantUseCase`**, storing the resultant restaurants inside the **`updatedRestaurants`** variable. Since this operation refreshes and re-obtains the restaurants from the fake local database, we then called **`advancedUntilIdle()`**, allowing any suspending work to finish.
- **Assertion**: We've first updated the dummy list we're expecting to be correct by manually toggling the first restaurant's **`isFavorite`** field.

Finally, we asserted that the resultant **`updatedRestaurants`** list is the same as the one we would expect to be correct – that is, **`restaurants`**.

If you run this test, it should pass.

ASSIGNMENT

Try testing the behaviour of other Use Case classes such as **`GetSortedRestaurantsUseCase`** *or even classes from the data layer such as* **`RestaurantsRepository`**.

Summary

In this chapter, we first had a look at the benefits of testing and classified tests based on different aspects. Afterward, we took a shot at testing our Compose-based UI and learned how to write UI unit tests by leveraging the power of the semantics modifiers.

Finally, we learned how to write regular – non-UI – unit tests in order to validate the core functionality of our application. In this part, we learned how to test our coroutine-based code and how important it is to inject the **`CoroutineDispatcher`** objects.

In the next chapter, we're steering away from the architectural side of Android development, and we will be incorporating data pagination with the help of yet another interesting library called Jetpack Paging.

Further reading

In this chapter, we've briefly covered the basics of UI and unit testing, so the core concepts taught here should give you a solid starting point. However, there are several other topics that you might need while you continue your testing adventure:

- For UI tests, we used semantics modifiers to identify UI elements from our node hierarchy. When testing Compose UI, you should also be aware of the merged and unmerged semantics tree. Learn more about this topic by reading the official docs: https://developer.android.com/jetpack/compose/semantics.
- With UI tests, we only scratched the surface in terms of testing APIs. Make sure to check out this official testing cheat sheet: https://developer.android.com/jetpack/compose/testing-cheatsheet.
- Our unit tests are based on the JUnit testing framework. To discover the power and flexibility of Junit, check out its official docs: https://junit.org/junit4/.
- In your coroutine-based tests, apart from the `advanceUntilIdle()` API, you can also use the `advancetimeBy()` API to fast-forward the virtual clock of the test by a certain amount. Learn more about this function from the official Coroutines docs: https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-test/kotlinx.coroutines.test/-delay-controller/advance-time-by.html.
- Your unit tests must be deterministic in the sense that every run of one test for the same revision of code should always yield the same result. Learn more about deterministic and non-deterministic tests from Martin Fowler: https://martinfowler.com/articles/nonDeterminism.html.